16. Design pattern III

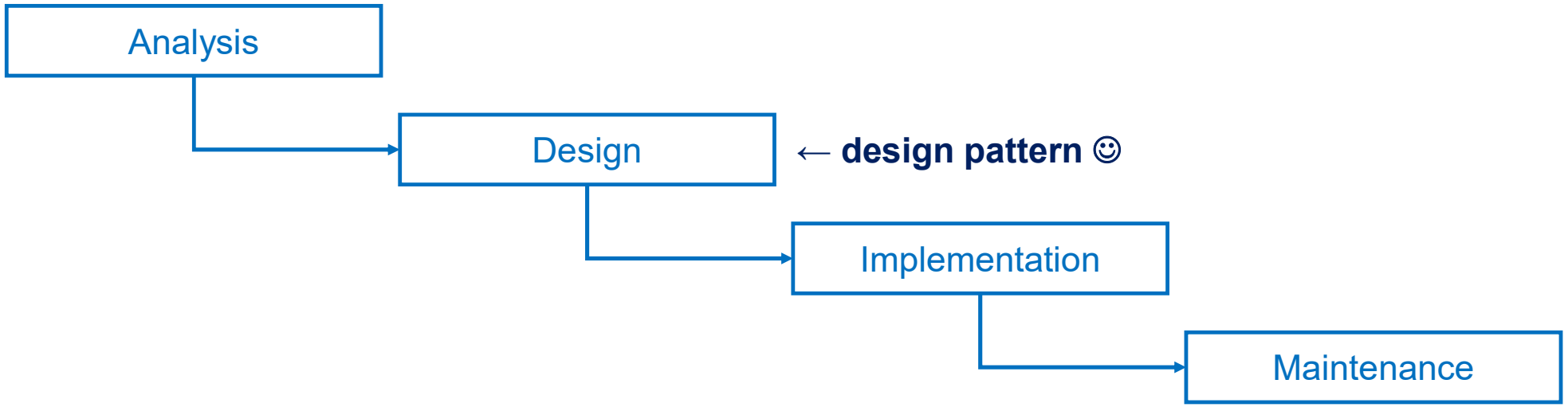
IS 2024-2025

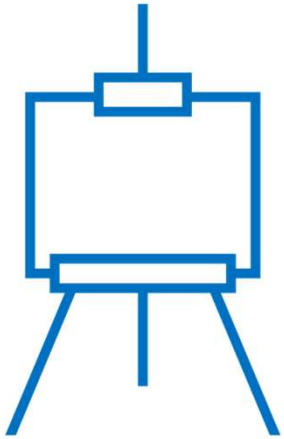
Laura Semini, Jacopo Soldani
Corso di Laurea in Informatica
Dipartimento di Informatica, Università of Pisa

REMINDER

«Each pattern **describes a problem** which occurs over and over again in our environment, and then **describes the core of the solution** to that problem, in such a way that **you can use this solution a million times over**, without ever doing it the same way twice»
(Christopher Alexander, A Pattern Language, 1977)

Pattern per ogni fase del processo software





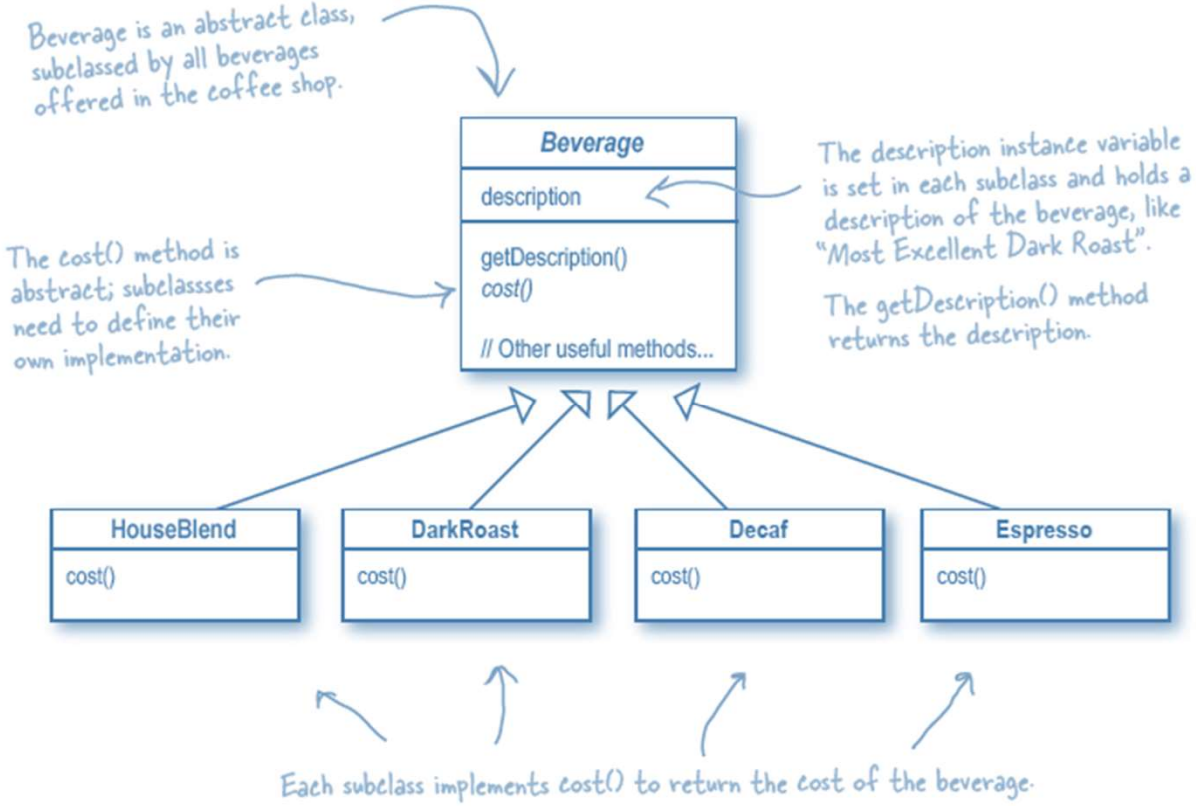
DECORATOR

ESEMPIO

Cafecito è una catena di caffetterie che ha avuto successo e si è diffusa rapidamente

- La rapida espansione rende difficile mantenere aggiornato il loro sistema di ordini
- Difficile includere tutte le bevande offerte nei punti vendita

Ad inizio business, avevano organizzato il sistema come mostrato a fianco

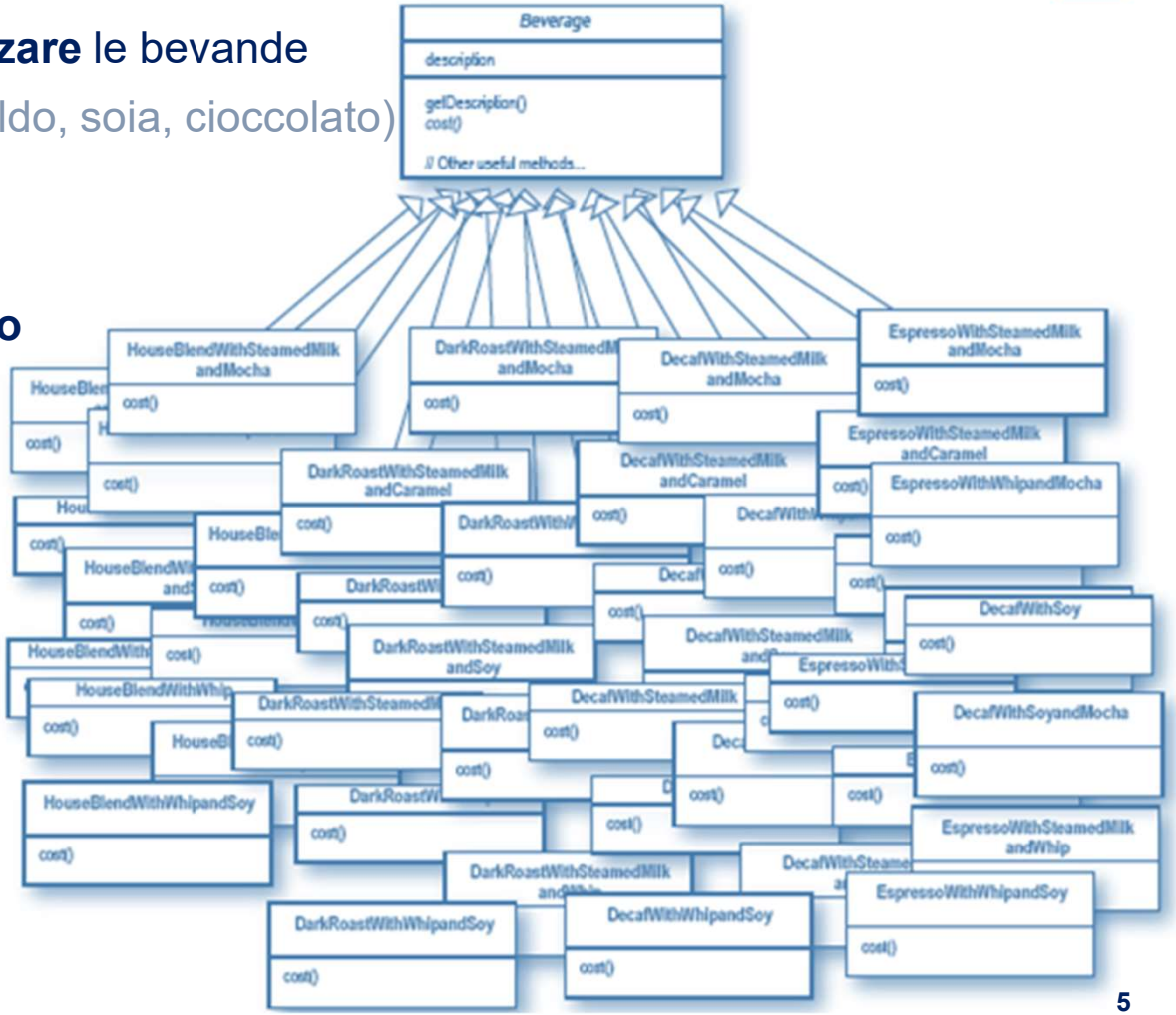


ESEMPIO (CONT.)

In realtà, **Cafecito** consente di **personalizzare** le bevande

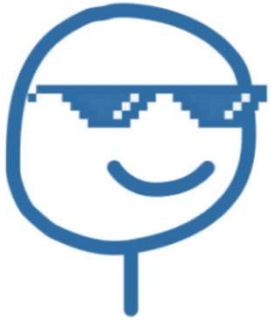
- aggiungendo condimenti (come latte caldo, soia, cioccolato)
- panna montata

Ciascuna aggiunta risulta in un **incremento** del **costo** della bevanda, richiedendo quindi un aggiornamento del sistema (per il calcolo del costo complessivo)



~~Not~~
Bad

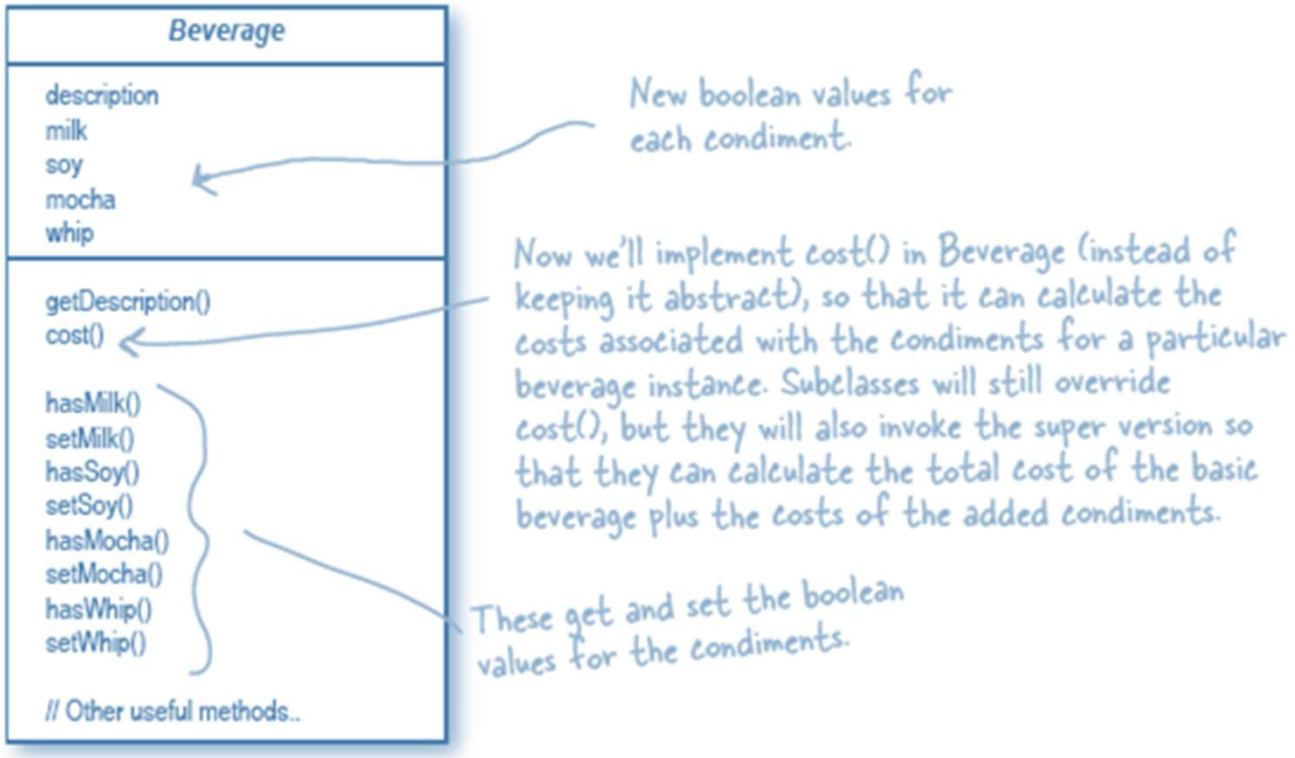
ESEMPIO (CONT.)



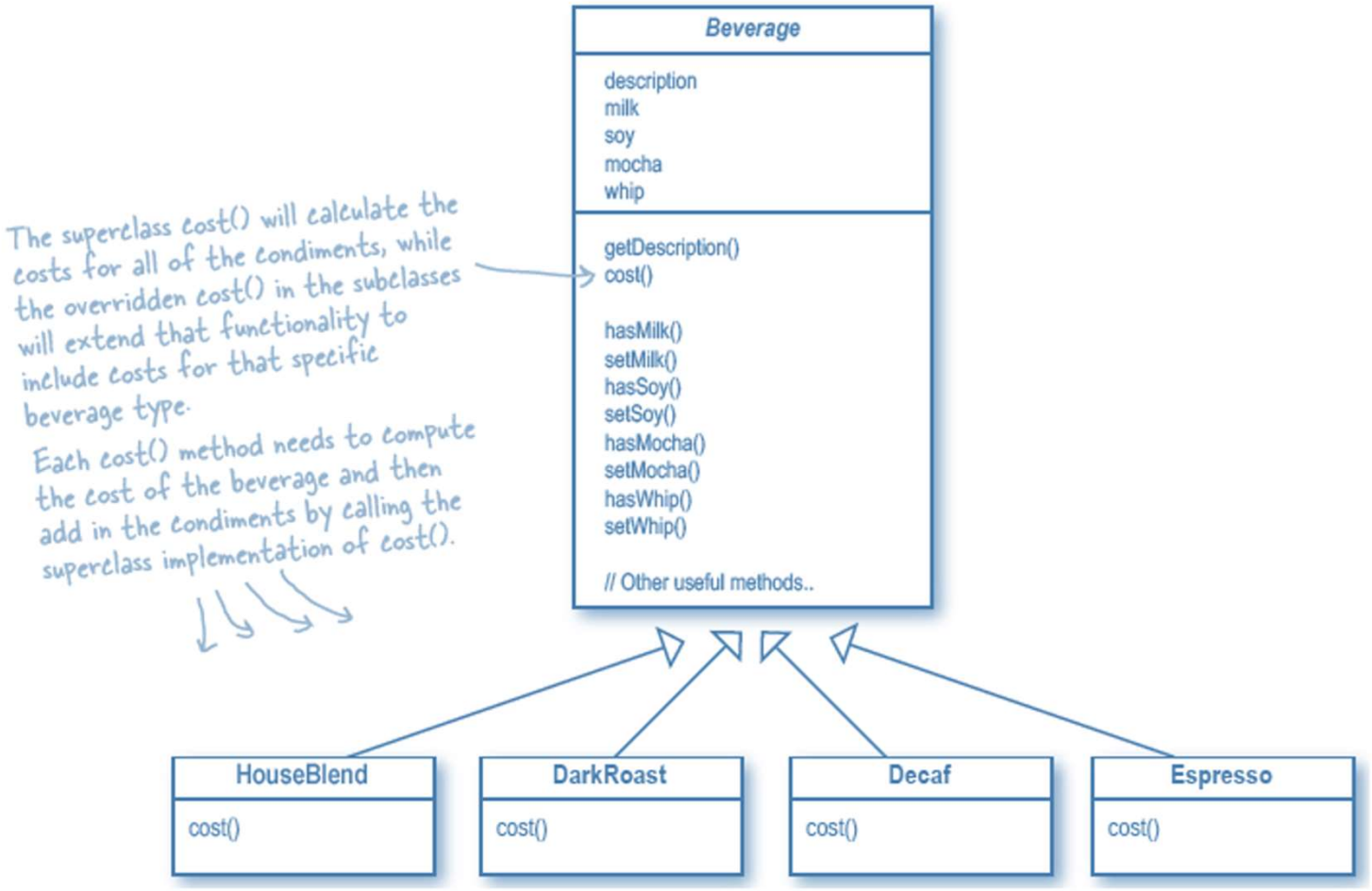
Ma che sono del mestiere questi?

Perché tutte queste classi?

Bastano delle variabili d'istanza nella superclasse e abbiamo risolto…



ESEMPIO (CONT.)



Poco manutenibile!

ESEMPIO (CONT.)

Problematiche:

- Eventuali **cambiamenti di prezzo** richiederebbero di aggiornare il codice esistente
- L'aggiunta di **nuovi condimenti** richiederebbe di aggiungere nuovi metodi e alterare il metodo `cost` della superclasse
- Potremmo voler offrire **nuove bevande**, per le quali i condimenti esistenti potrebbero non essere appropriati
- Cosa succede se un cliente vuole un **espresso doppio**?

MEMENTO: PRINCIPIO OPEN CLOSED

Le entità software devono essere **aperte per estensione** ma **chiuse per modifiche**

Disegnare classi/moduli che **non cambiano**

- Quando i requisiti cambiano, estenderne il comportamento aggiungendo nuovo codice (senza cambiare quello esistente e funzionante)
- Disegnare classi in modo che sia possibile estenderle ma senza cambiarle
- Possibile, per esempio, sfruttando
  - classi astratte e classi concrete
  - delega
  - plugin (aggiunta di codice senza ricompilare l'esistente)

FAQ

Q: Mi conviene garantire che TUTTO il mio progetto segua il principio open-closed?

A: Solitamente, non è possibile – o meglio, è **troppo costoso**. Seguire il principio open-closed richiede l'introduzione di nuovi livelli di astrazione, che aggiungono complessità al codice. Vale quindi la pena farlo in quelle aree in cui i cambiamenti sono più probabili.

Q: Come individuo quali sono le aree più importanti/prone al cambiamento?

A: Non esiste una regola d'oro – le scelte tipicamente si fanno in base all'**esperienza** maturata e al **dominio** dell'applicazione. Guardare altri esempi simili può essere d'aiuto.

IL PATTERN DECORATOR (AKA. WRAPPER)

Problemi delle versioni precedenti:

- proliferazione delle classi e design rigido, oppure
- funzionalità aggiunte alla classe base ma inappropriate per alcune sottoclassi

Idea: Supponiamo che un cliente voglia un `DarkRoast` con `Mocha` e `Whip`

- Prendere un oggetto `DarkRoast`
- Decorare con un oggetto `Mocha`
- Decorare (ulteriormente) con un oggetto `Whip`
- Chiamare il metodo cost e affidarsi alla delega per aggiungere i costi dei condimenti

Il pattern **decorator** aggiunge dinamicamente responsabilità ad un oggetto

alternativa flessibile all'ereditarietà per l'aggiunta di funzionalità

ESEMPIO

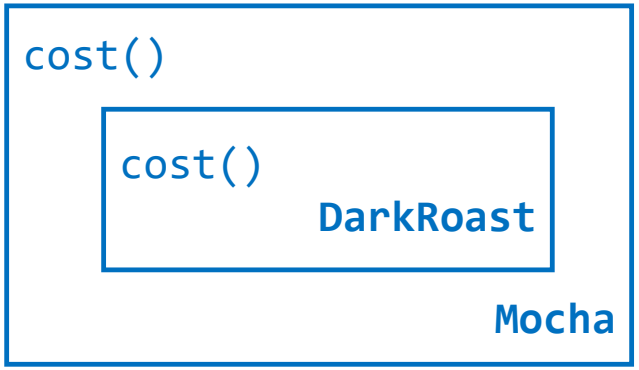
Cominciamo con l'oggetto `DarkRoast`

// Memento: `DarkRoast` estende `Beverage`, ereditando il metodo `cost` per il calcolo del costo

Usiamo un oggetto `Mocha` come «wrap» per l'oggetto `DarkRoast`

// l'oggetto `Mocha` è un decorator

// il tipo di `Mocha` è lo stesso di `DarkRoast` (entrambi sottotipi di `Beverage`)

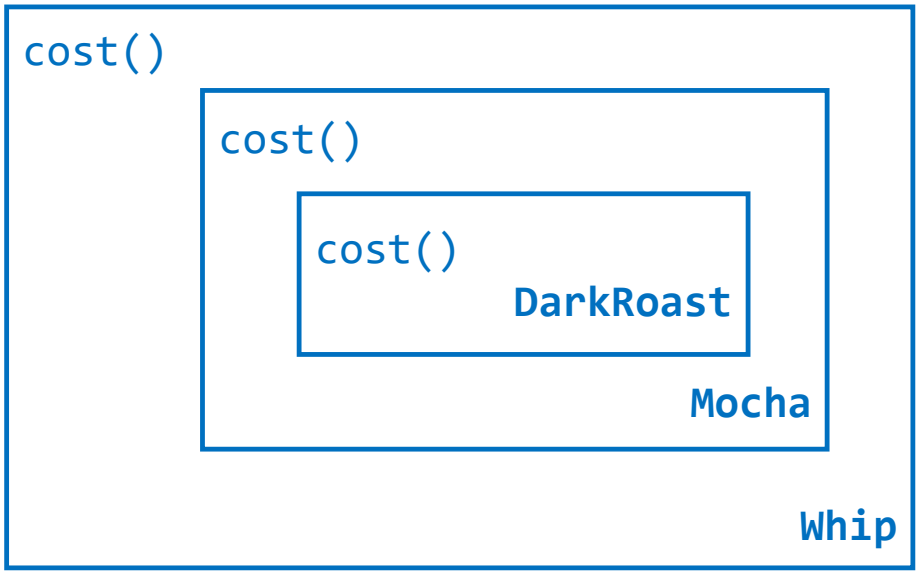


ESEMPIO (CONT.)

Creiamo un ulteriore decorator (un oggetto `Whip` come «wrap» per l'oggetto `Mocha`)

// il tipo di `Whip` è lo stesso di `Mocha` (entrambi sottotipi di `Beverage`)

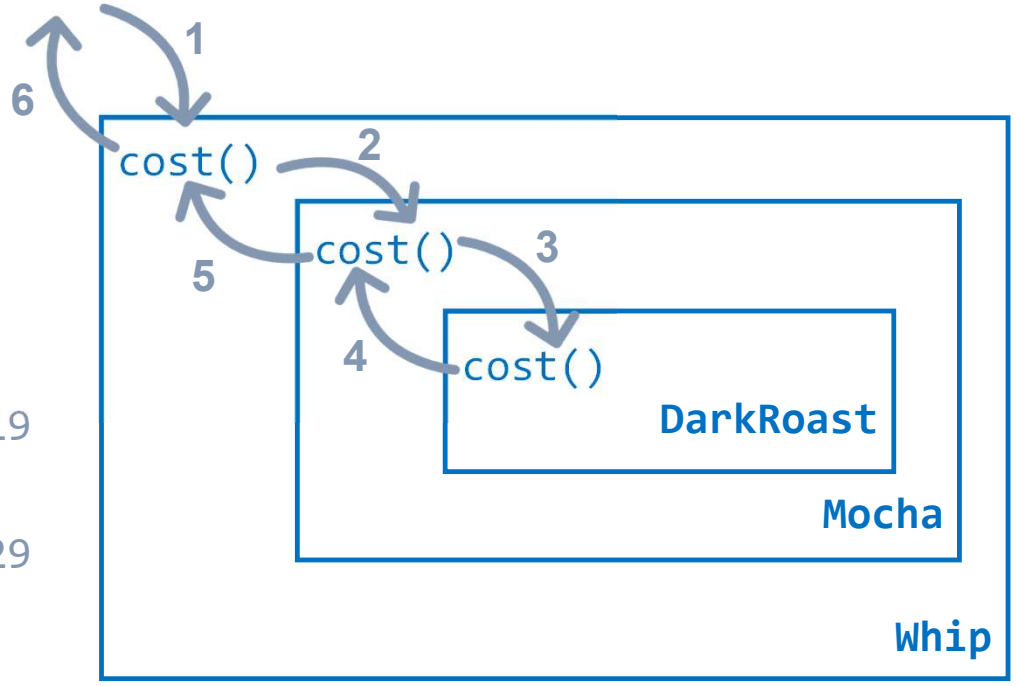
// un `DarkRoast` decorato con `Mocha` e `Whip` è sempre un `Beverage` e possiamo quindi fare le stesse cose – incluso invocare il metodo `cost`



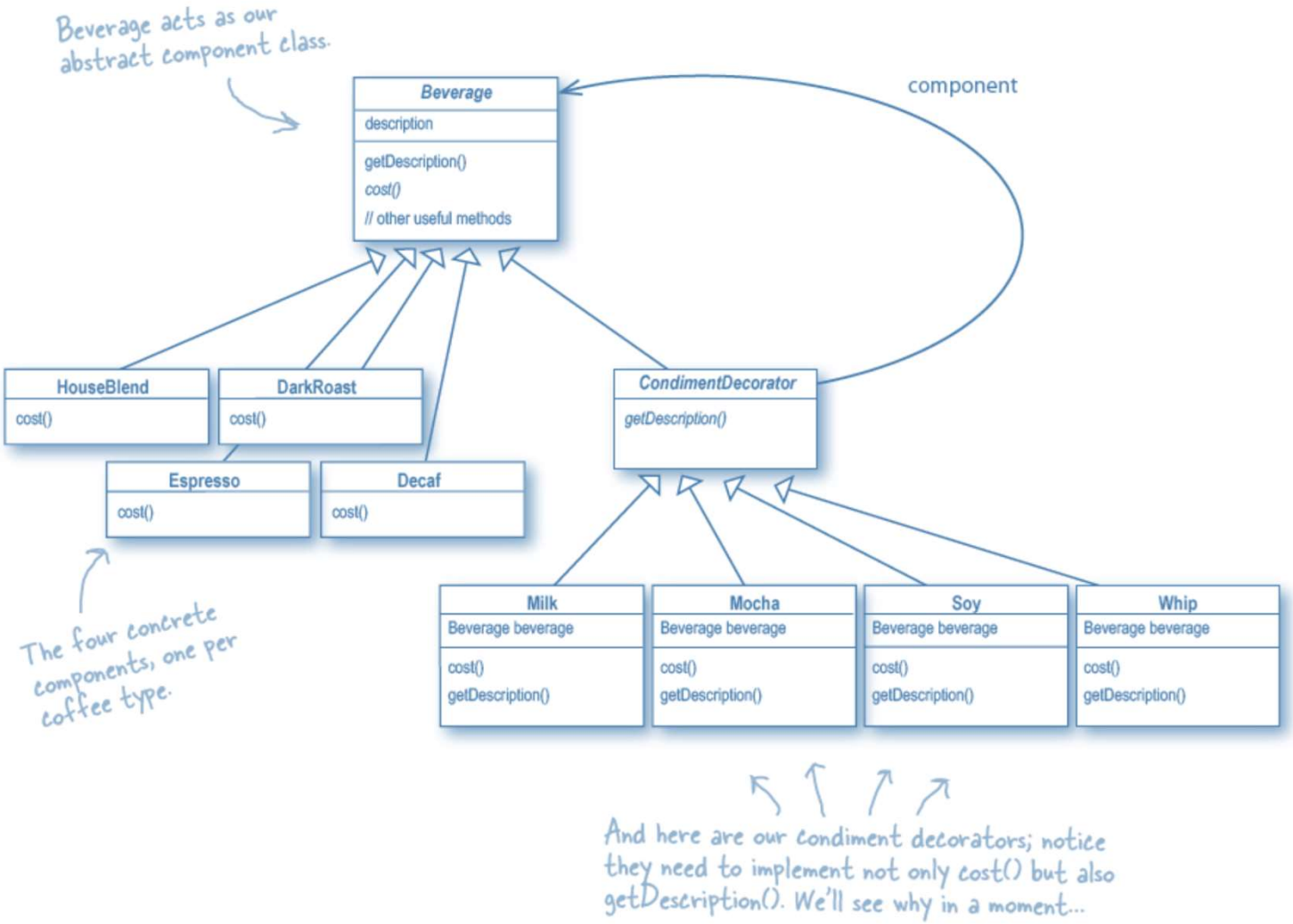
ESEMPIO (CONT.)

Possiamo calcolare il costo della bevanda ottenuta chiamando il metodo `cost`

1. Chiamiamo cost sul decoratore più esterno (`Whip`)
2. L'oggetto `Whip` chiama `cost` su `Mocha`
3. L'oggetto `Mocha` chiama `cost` su `DarkRoast`
4. `DarkRoast` restituisce `0.99`
5. `Mocha` aggiunge il suo costo e restituisce `1.19`
6. `Whips` aggiunge il suo costo e restituisce `1.29`



ESEMPIO (CONT.)



PIÙ IN GENERALE

Component

- Interfaccia degli oggetti decorati

ConcreteComponent

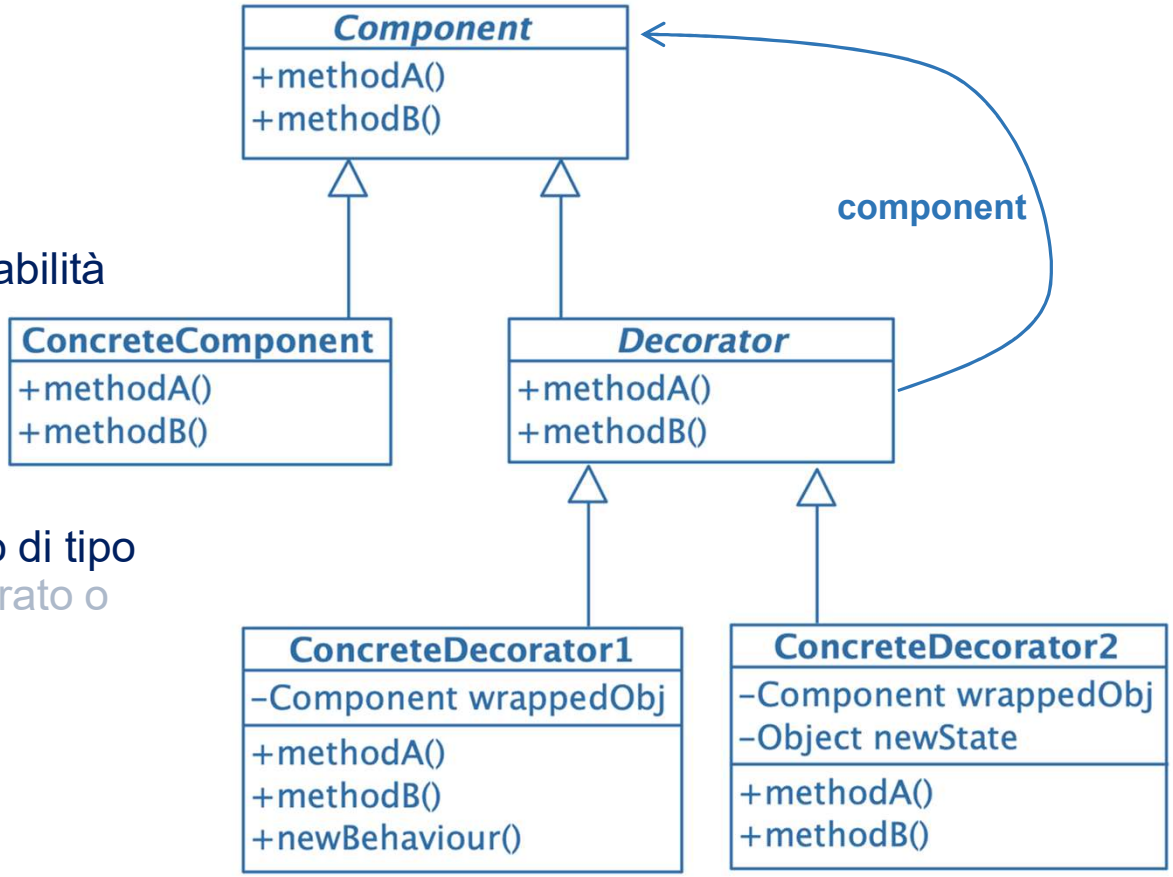
- Oggetti decorabili con nuove responsabilità

Decorator

- Interfaccia conforme a quella comune
- Mantiene un riferimento ad un oggetto di tipo Component (che può essere già decorato o non decorato)

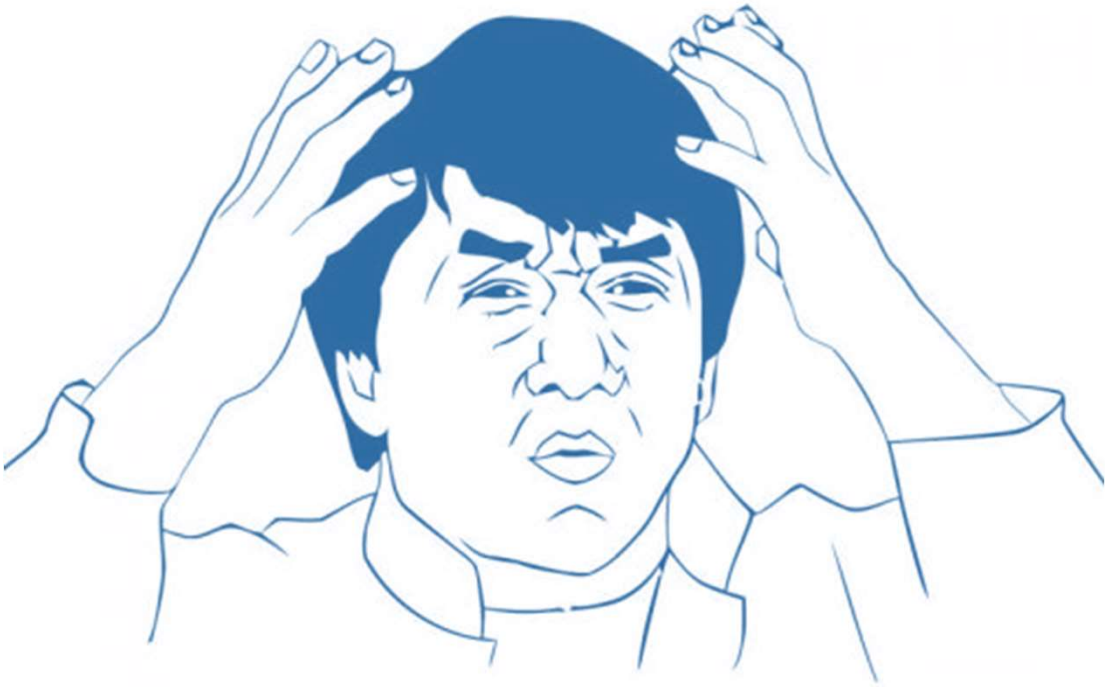
ConcreteDecorator

- Aggiunge nuove responsabilità

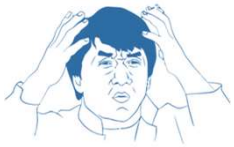


MA NON DOVEVAMO EVITARE L'EREDITARIETÀ?

Dovevamo usare la composizione invece dell'ereditarietà, ma ci sono un sacco di classi estese!

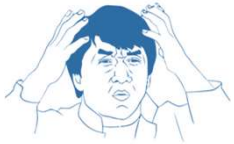


DECORATOR, EREDITARIETÀ, COMPOSIZIONE



Guarda il diagramma delle classi dell'esempio. Il `CondimentDecorator` estende `Beverage`: questa è ereditarietà, giusto?

Vero: i decorator hanno lo stesso tipo degli oggetti che vanno a decorare. Usiamo l'ereditarietà per ottenere **type matching** ma non per ereditare comportamento

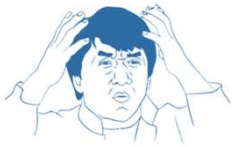


Ok. Quindi i decorator hanno bisogno della stessa «interfaccia» dei componenti che decorano per poterli sostituire. Ma dove entra in gioco il comportamento, allora?

Quando componiamo un decorator con un componente, aggiungiamo nuovo comportamento. Lo otteniamo per **composizione**, invece che per estensione

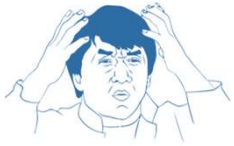
Ok. Quindi estendiamo la classe astratta Beverage per avere il tipo corretto e non per ereditarne il comportamento. Il comportamento lo introduciamo componendo i decorator con i componenti base o altri decorator

DECORATOR, EREDITARIETÀ, COMPOSIZIONE (CONT.)

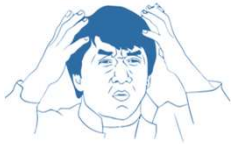


E usando la composizione di oggetti otteniamo molta più flessibilità per combinare bevande e condimenti. WOW!

Si, se ci affidassimo solo all'ereditarietà, allora il nostro comportamento potrebbe essere determinato staticamente a tempo di compilazione. Con la composizione, possiamo combinare i decoratori come vogliamo e soprattutto.. **a runtime**!



E possiamo implementare nuovi decorator quando vogliamo per aggiungere nuovo comportamento. Se ci affidassimo all'ereditarietà, dovremmo cambiare il codice esistente ogni volta. WOW!



Un'ultima domanda: se l'unico obiettivo è ereditare il tipo `Beverage`, perché non usiamo un'interfaccia invece di una classe astratta?

Abbiamo ereditato il codice di Caffecito che già includeva una classe astratta. Quindi, per minimizzare le modifiche, l'abbiamo mantenuta. In Java, in generale, avremmo potuto realizzare il pattern Decorator usando un'interfaccia

IL CODICE DELL'ESEMPIO

```
// Component abstract class
public abstract class Beverage {
  String description = "Unknown Beverage";
  public String getDescription() { return description; }
  public abstract double cost();
}
```

```
// Decorator abstract class
public abstract class CondimentDecorator extends Beverage {
  // getDescription to be re-defined by decorators
  public abstract String getDescription();
}
```

IL CODICE DELL'ESEMPIO

```java
// A concrete base class (beverage)
public class Espresso extends Beverage {
  public Espresso() { this.description = "Espresso"; }
  public double cost() { return 1.99; }
}
```

```java
// A concrete decorator (condiment)
public class Mocha extends CondimentDecorator {
  private Beverage beverage;
  public Mocha(Beverage beverage) { this.beverage = beverage; }
  public String getDescription() {
    return beverage.getDescription() + ", Mocha";
  }
  public double cost() { return 0.20 + beverage.cost(); }
}
```

IL CODICE DELL'ESEMPIO (CONT.)

```
// Creating beverages from decorator classes dynamically
public class Caffecito {
  public static void main(String args[]) {
    // Beverage only
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription() + "$" + beverage.cost());
    // Beverage with condiments
    Beverage beverage2 = new DarkRoast();
    beverage2 = new Mocha(beverage2);
    beverage2 = new Mocha(beverage2);
    beverage2 = new Whip(beverage2);
    System.out.println(beverage2.getDescription() + "$" + beverage2.cost());
  }
}
```

LA «TRASPARENZA» DEI DECORATOR

I decorator possono essere inseriti in modo trasparente

- il cliente non si accorge (e non deve sapere) che sta interagendo con un decorator

Se però scriviamo codice dipendente da tipi specifici, possono succedere cose brutte :-)

```
// The good way
Beverage b = new DarkRoast();
b = new Mocha(b);
b = new Whip(b);
System.out.println(b.getDescription() + "$" + b.cost());
```

```
// The poor way
Beverage b = new DarkRoast();
b = new Mocha(b);
b = new Mocha(b);
Whip b2 = new Whip(b);
System.out.println(b2.getDescription() + "$" + b.cost());
```

OSSERVAZIONI SUL PATTERN DECORATOR

Pro

- Maggiore **flessibilità** rispetto all'ereditarietà statica
- Più **semplice** da usare rispetto all'ereditarietà multipla
- Può essere usato per combinare feature e aggiungere la stessa proprietà due volte
- Favorisce l'**aggiunta incrementale** di feature

Contro

- Se il decorator è complesso, diventa **costoso** da usare
- Decorator e Component **non sono identici**
    - Dal punto di vista dell'identità dell'oggetto, un componente decorato non è identico al componente stesso
    - Meglio **non affidarsi all'identità degli oggetti** quando si usano i decorator
- Si possono ottenere sistemi composti da **molti oggetti «piccoli»**
    - **Difficili** da comprendere e debuggare

DA CONSIDERARE DURANTE L'IMPLEMENTAZIONE

- L'interfaccia di un oggetto decorator deve essere **conforme all'interfaccia** del componente che decora

- Se è necessaria una sola «decorazione», non serve definire la classe astratta Decorator (le cui responsabilità possono essere fuse con quelle di ConcreteDecorator)

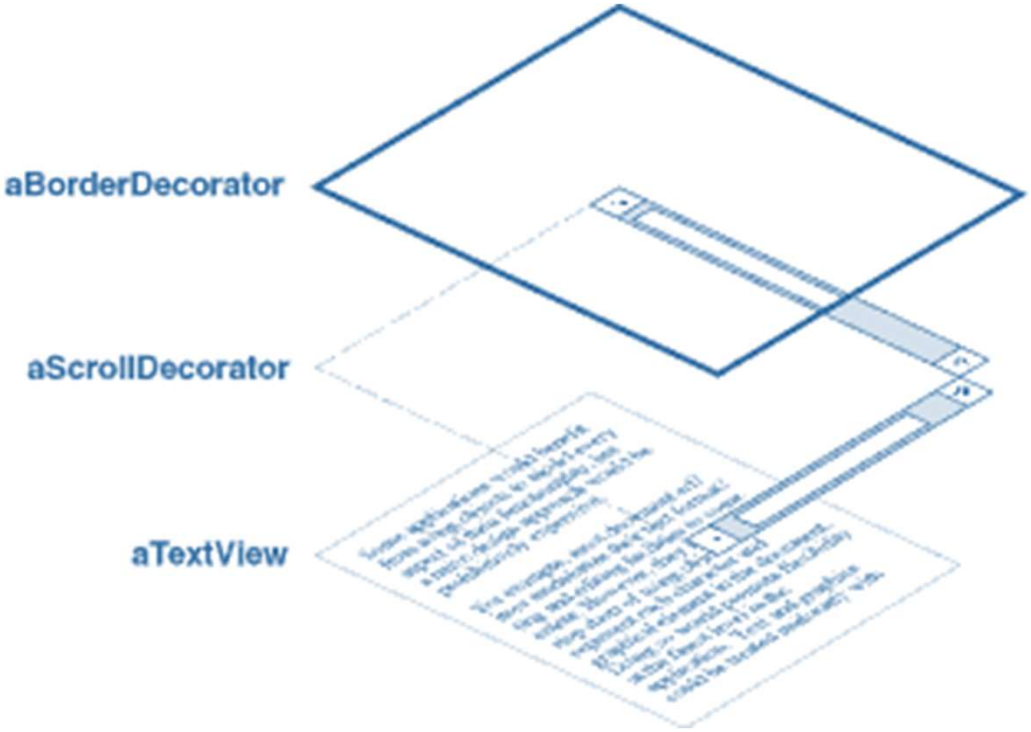
- Mantenere **leggere** le classi astratte **component** che sono estese da componenti e decorator
  - Devono definire solo l'interfaccia comune (e nessun'altra funzione)
  - Se diventano complesse, potrebbero rendere i decorator troppo costosi da usare

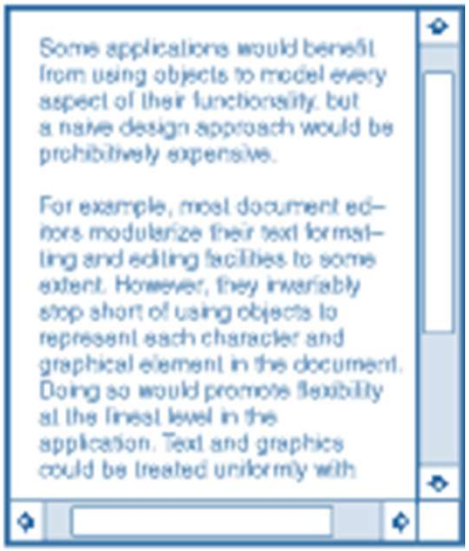
- Le classi decorator dovrebbero funzionare come «strati» sopra un oggetto (se gli oggetti devono cambiare internamente, meglio usare lo Strategy pattern)

UN ALTRO ESEMPIO

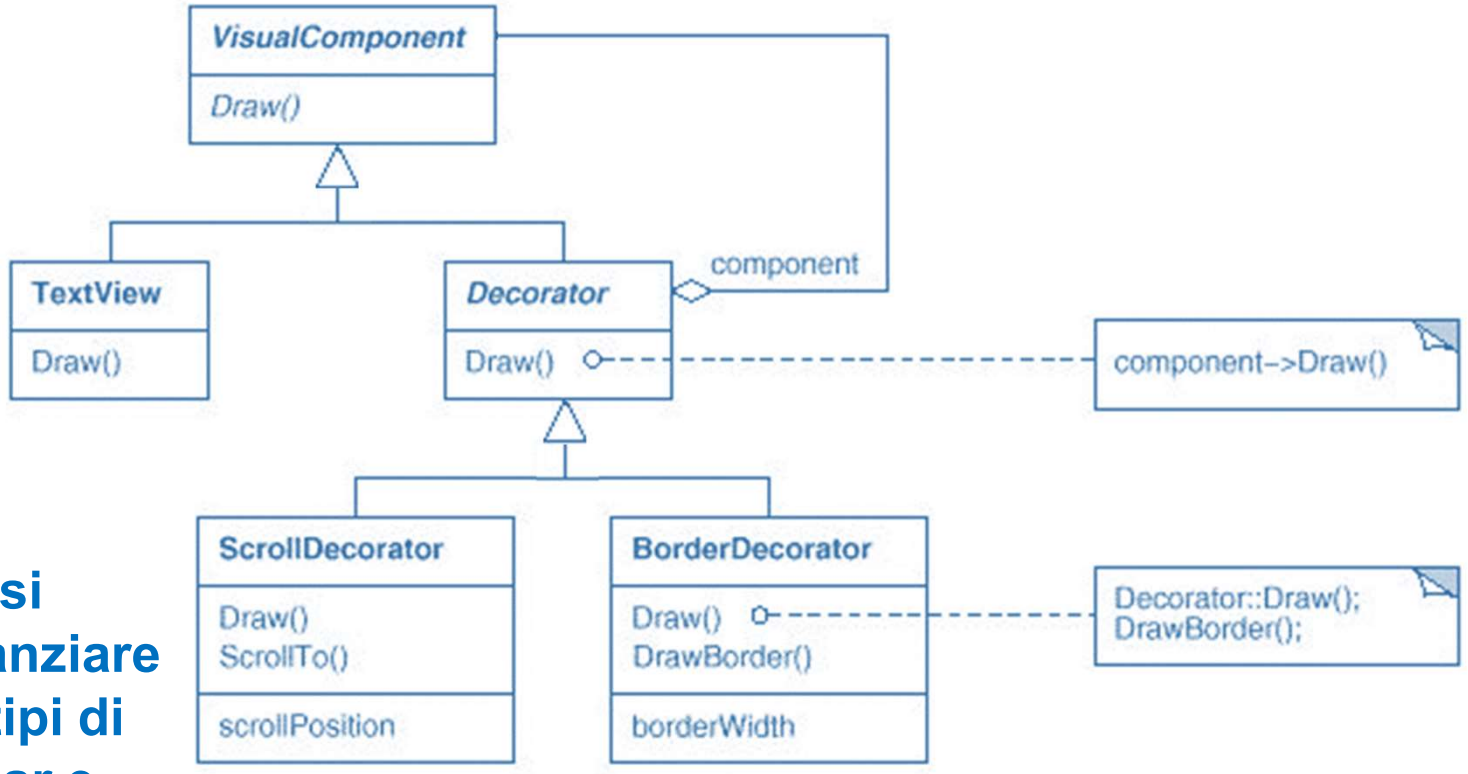
Vogliamo aggiungere proprietà (come bordi e scrollbar) ad un componente di una GUI

- Possiamo farlo usando l'ereditarietà, ma questo limita la nostra flessibilità
- Meglio usare la composizione!





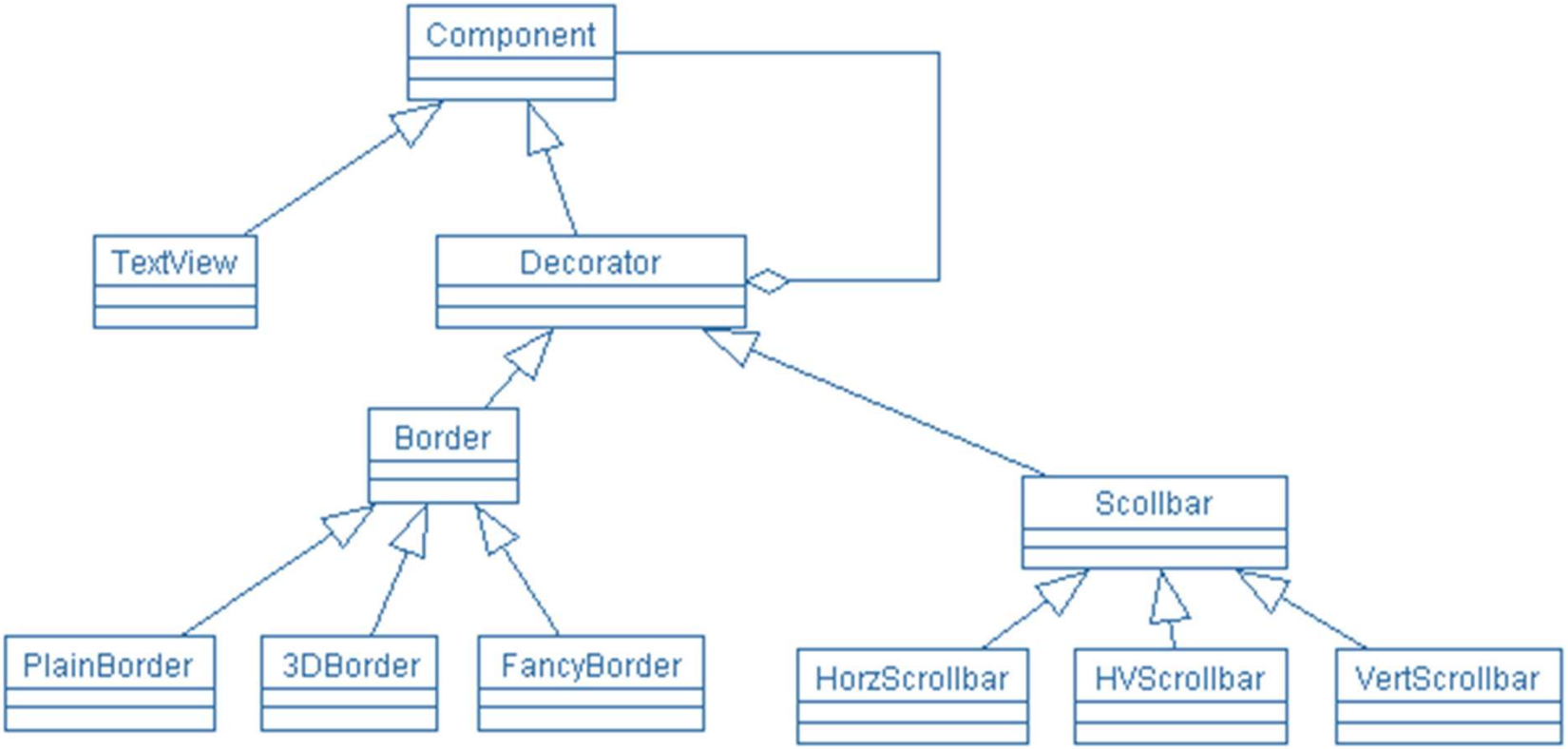
UN ALTRO ESEMPIO (CONT.)



E se volessi poter istanziare diversi tipi di scrollbar e border a runtime?

UN ALTRO ESEMPIO (CONT.)

Combiniamo i pattern decorator e strategy!



HOMEWORK

Seguendo l’esempio di Cafecito, usare Decorator per costruire un ponce alla livornese :-)

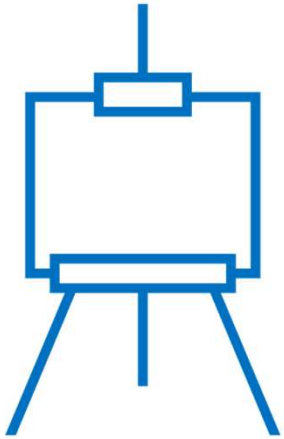
HOMEWORK

Il Natale è in arrivo! Essendo persone organizzate, abbiamo già un piano per l'albero. Vogliamo quindi implementare un sistema software che ci permetta di calcolare il prezzo di qualsiasi albero con qualsiasi combinazione di decorazioni. Il sistema deve essere facilmente estendibile, nel senso che ogni volta che vengono aggiunte nuove decorazioni nel negozio, dovrai aggiungere al massimo una classe.

Di seguito alcuni esempi di costo per alberi e decorazioni

Trees	Cost
Fraser Fir	12
Colorado Blue Spruce	20

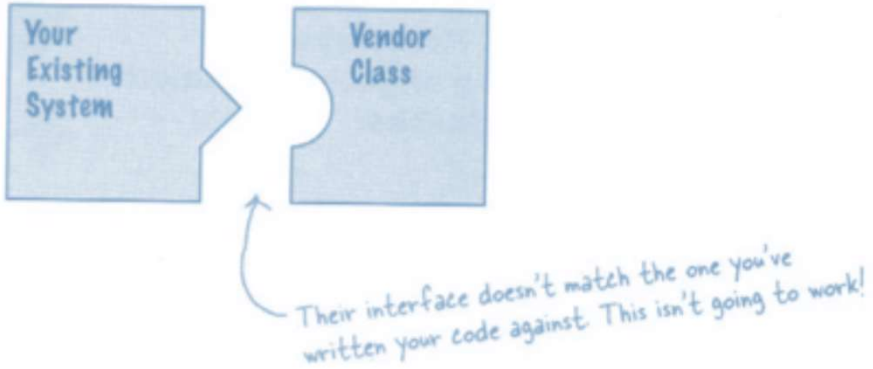
Decorations	Cost
Star	4
Balls Red	1
Balls Silver	3
Lights	5

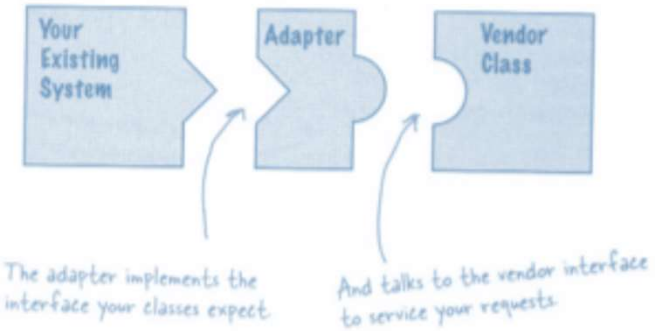


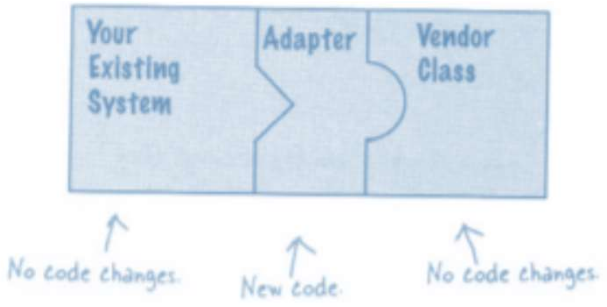
ADAPTER

«ADAPTER» NELLA VITA REALE

«ADAPTER» NELLA PROGRAMMAZIONE OO







ESEMPIO, CON LE PAPERE

```java
public interface Duck {
  public void display();
  public void swim();
}
```

```java
public class Duckling implements Duck {
  public void display() {
    System.out.println("I’m a pretty duckling");
  }
  public void swim() {
    System.out.println("I'm learning…");
  }
}
```

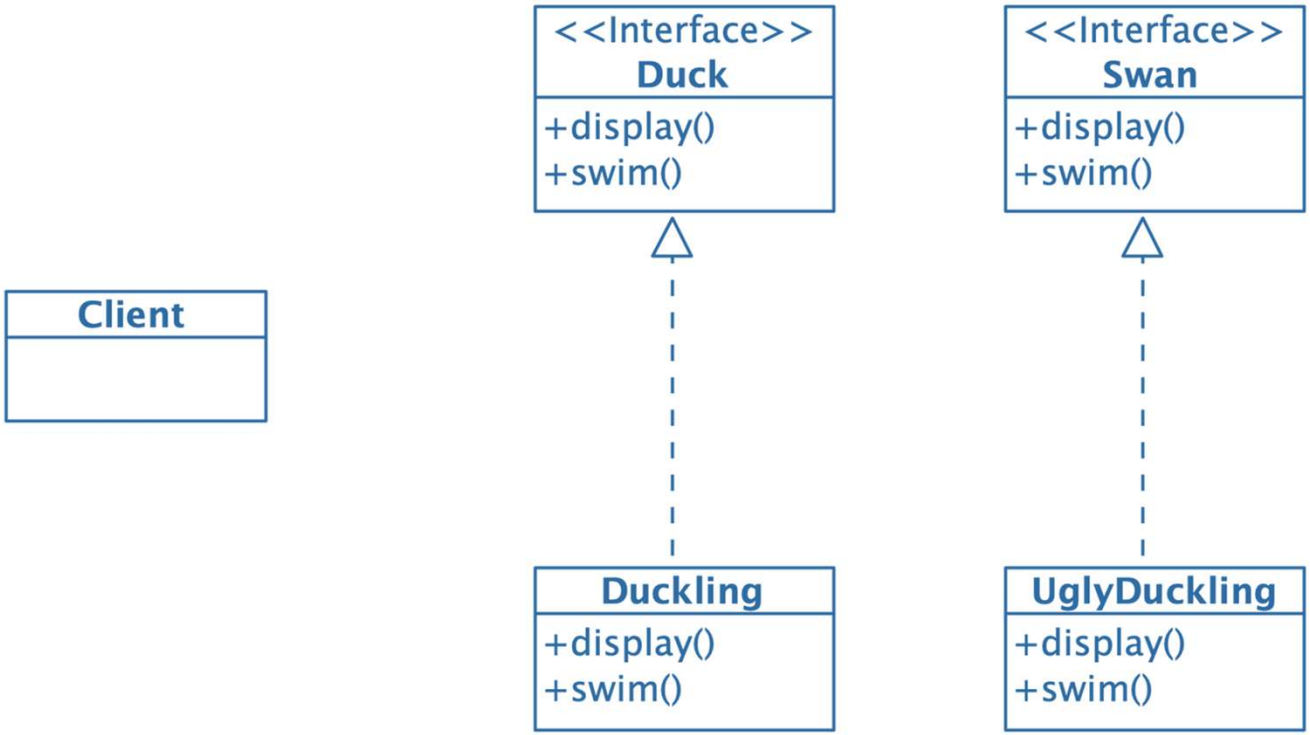
```java
public interface Swan {
  public void show();
  public void swim();
}
```

```java
public class UglyDuckling implements Swan {
  public void display() {
    System.out.println("I’m large and ugly");
  }
  public void swim() {
    System.out.println("I’m swimming");
  }
}
```

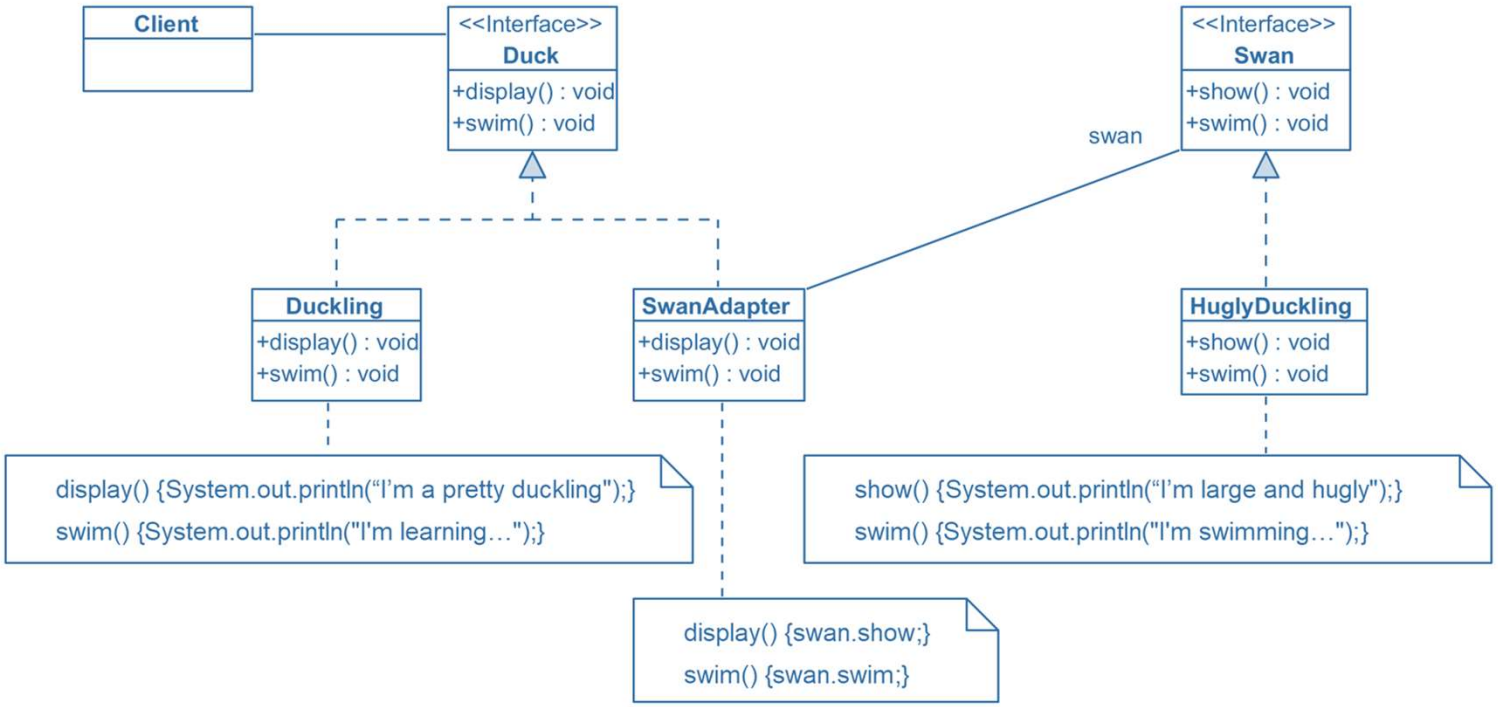
ESEMPIO (CONT.)

Due gerarchie diverse, ma abbiamo la necessità di gestire gli oggetti in modo uniforme

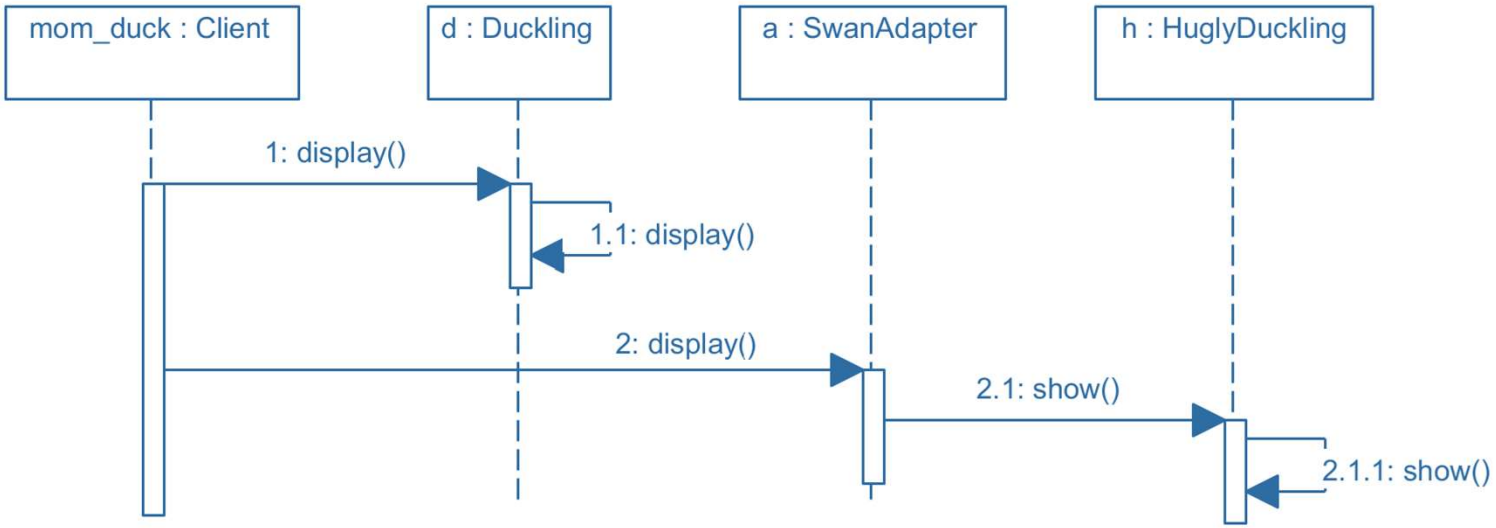
→ Come fare?



ESEMPIO (CONT.)



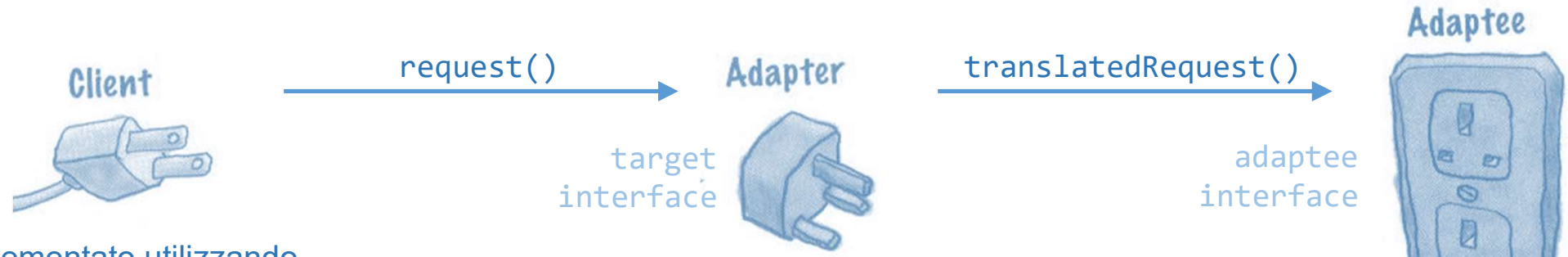
ESEMPIO (CONT.)



IL PATTERN ADAPTER

Converte l'interfaccia di una classe nell'interfaccia attesa dal cliente

Consente l'integrazione di classi che altrimenti non potrebbero essere integrate (a causa di interfacce incompatibili)



Implementato utilizzando un'interfaccia diversa da quella di `Adaptee`

Implementa l'interfaccia richiesta da `Client` e mantiene un riferimento ad un'istanza di `Adaptee`

IL PATTERN ADAPTER (CONT.)

Target

- Definisce l'interfaccia application-specific (usata da Client)

Client

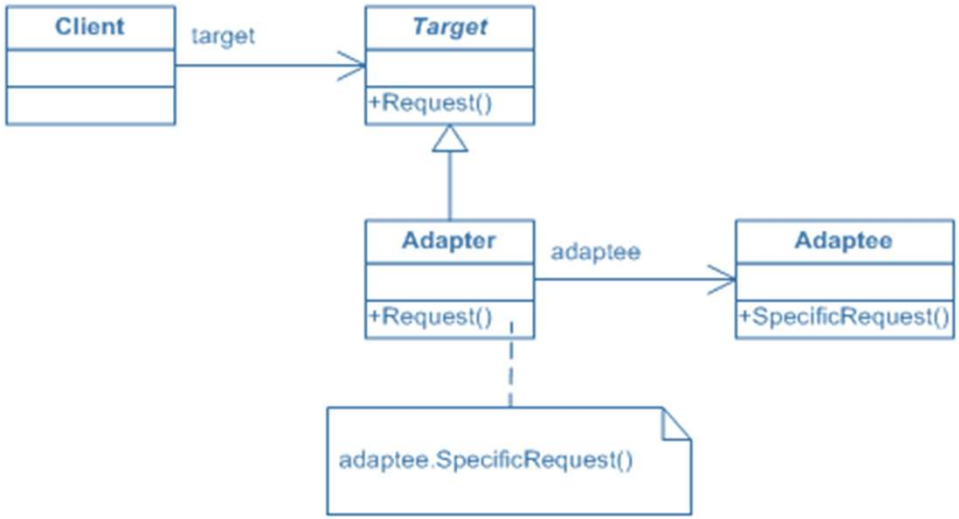
- Interagisce con oggetti conformi all'interfaccia Target

Adaptee

- Definisce un'interfaccia esistente (da adattare)

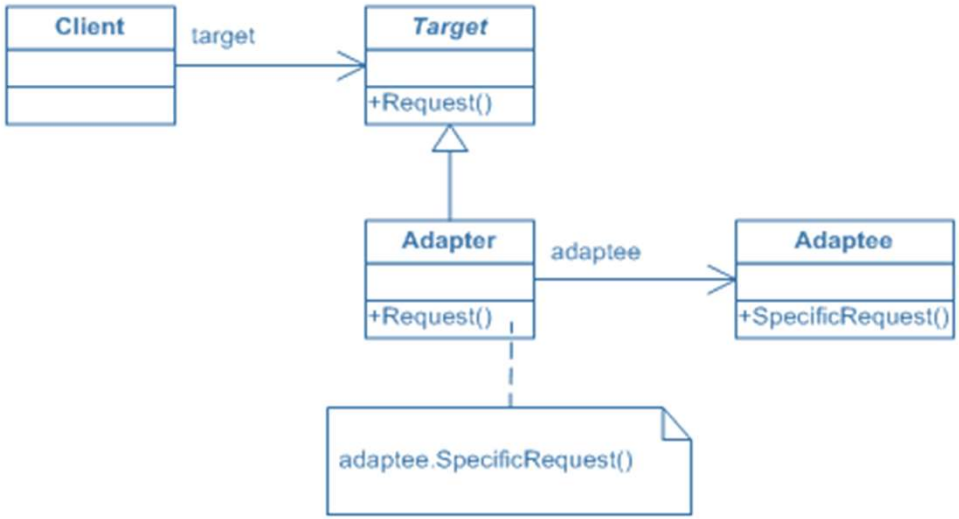
Adapter

- Adatta l'interfaccia di Adaptee per renderla conforme a quella Target

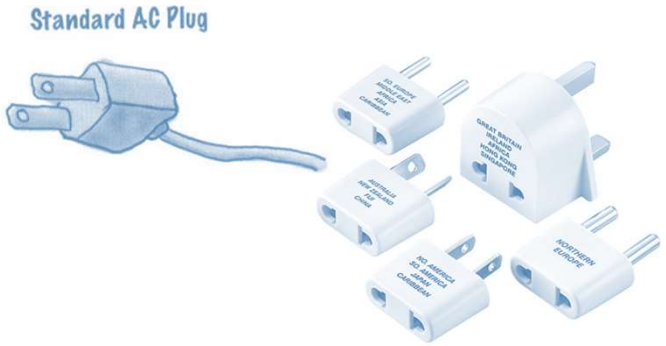


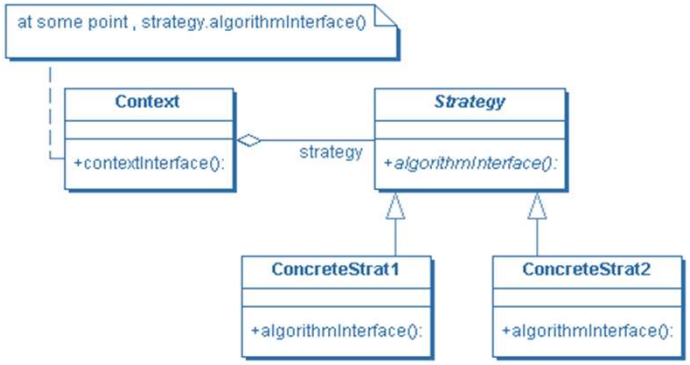
IL PATTERN ADAPTER (CONT.)

- Target e Adaptee esistono prima di Adapter (es. sistemi legacy)
- In Java, Target può essere realizzato come interfaccia
- Il binding tra Adapter e Adaptee è realizzato con il meccanismo di **delega**
- Ereditarietà come meccanismo di specifica dell'interfaccia della classe Adapter



ADAPTER E STRATEGY





Gli adapter possono essere realizzati come strategie

- se abbiamo diversi moduli che implementano la stessa funzionalità e abbiamo scritto degli adattatori per loro, abbiamo un insieme di adattatori che implementano la stessa interfaccia

- possiamo quindi sostituire gli oggetti adattatori in fase di esecuzione

HOMEWORK

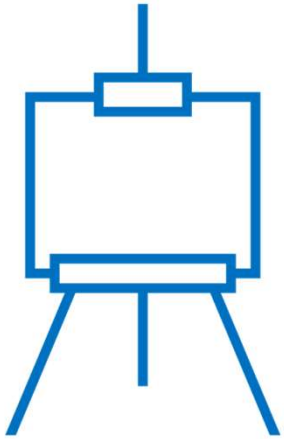
- Estendere l'esempio di `Duck` e `Swan` in modo da adattare il sistema anche ai `Turkey`

- Progettare una soluzione che combini i pattern Adapter e Strategy

IL PATTERN FAÇADE

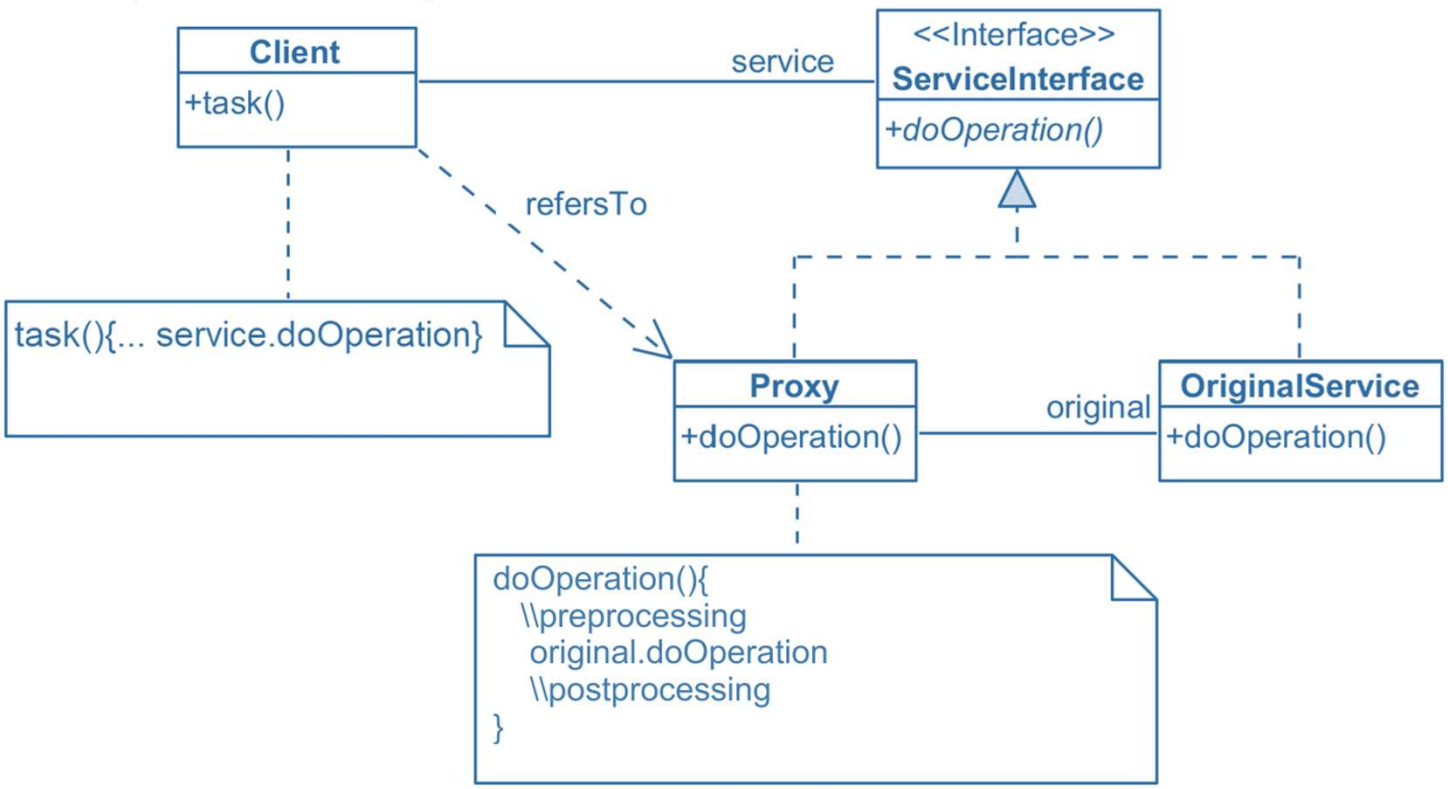
- Pattern GoF

- Realizza un adattatore che inoltra le richiesta a più adaptee, simultaneamente

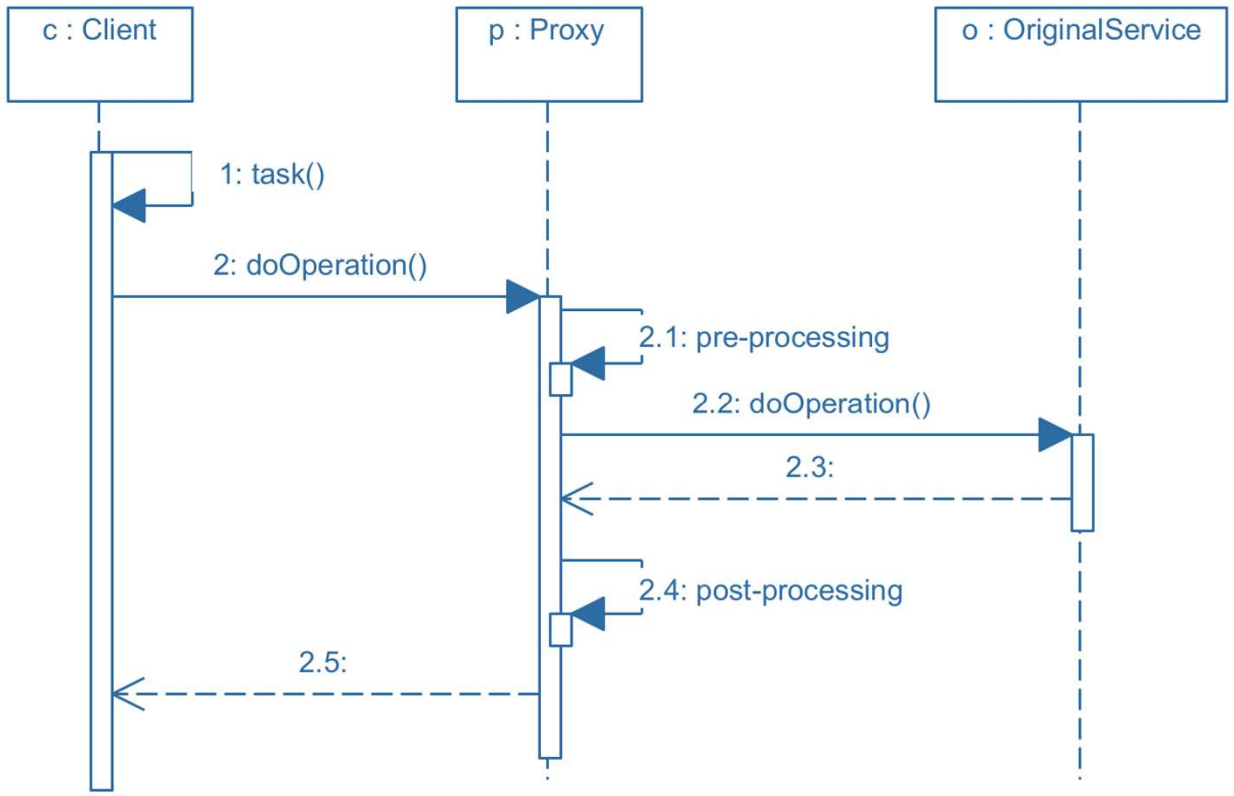


PROXY

IL PATTERN PROXY



PROXY: MODELLO DINAMICO



OSSERVAZIONI

Il **proxy** fornisce un **surrogato** (o segnaposto) per un altro oggetto, per controllarne l'accesso

Potrebbe sembrare simile al pattern Adapter (si introduce un intermediario) ma **sono diversi**

- Il proxy e l'oggetto originale hanno la **stessa interfaccia** (adapter e adaptee no)
- Il proxy può eseguire **pre-/post-processing** (adapter no)

Esistono **diversi tipi** di proxy:

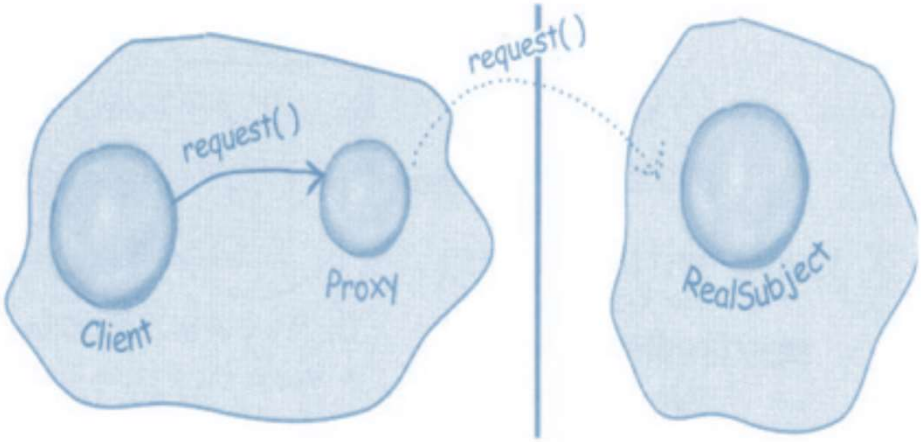
- **Remote proxy**: permette l'accesso a un oggetto remoto (usato, per esempio, RMI e in corba)
- **Protection proxy**: implementa un controllo sugli accessi
- **Cache proxy**: mantiene coppie richiesta-risposta (per alleggerire il carico sul server)
- **Synchronization proxy**: gestisce gli accessi concorrenti a un servizio
- **Virtual proxy**: si comporta come l'originale mentre l'originale viene costruito, poi passa le richieste a quest'ultimo

ESEMPIO

Supponiamo di voler **monitorare** un servizio **`GumballMachine`**

Possiamo farlo attraverso un **remote proxy** (ovvero mediante una rappresentazione «locale» di un oggetto «remoto»)

- Il client effettua una richiesta al proxy
- La richiesta viene inoltrata (attraverso la rete) all'oggetto remoto
- Il risultato è restituito al proxy
- Il proxy inoltra il risultato al client



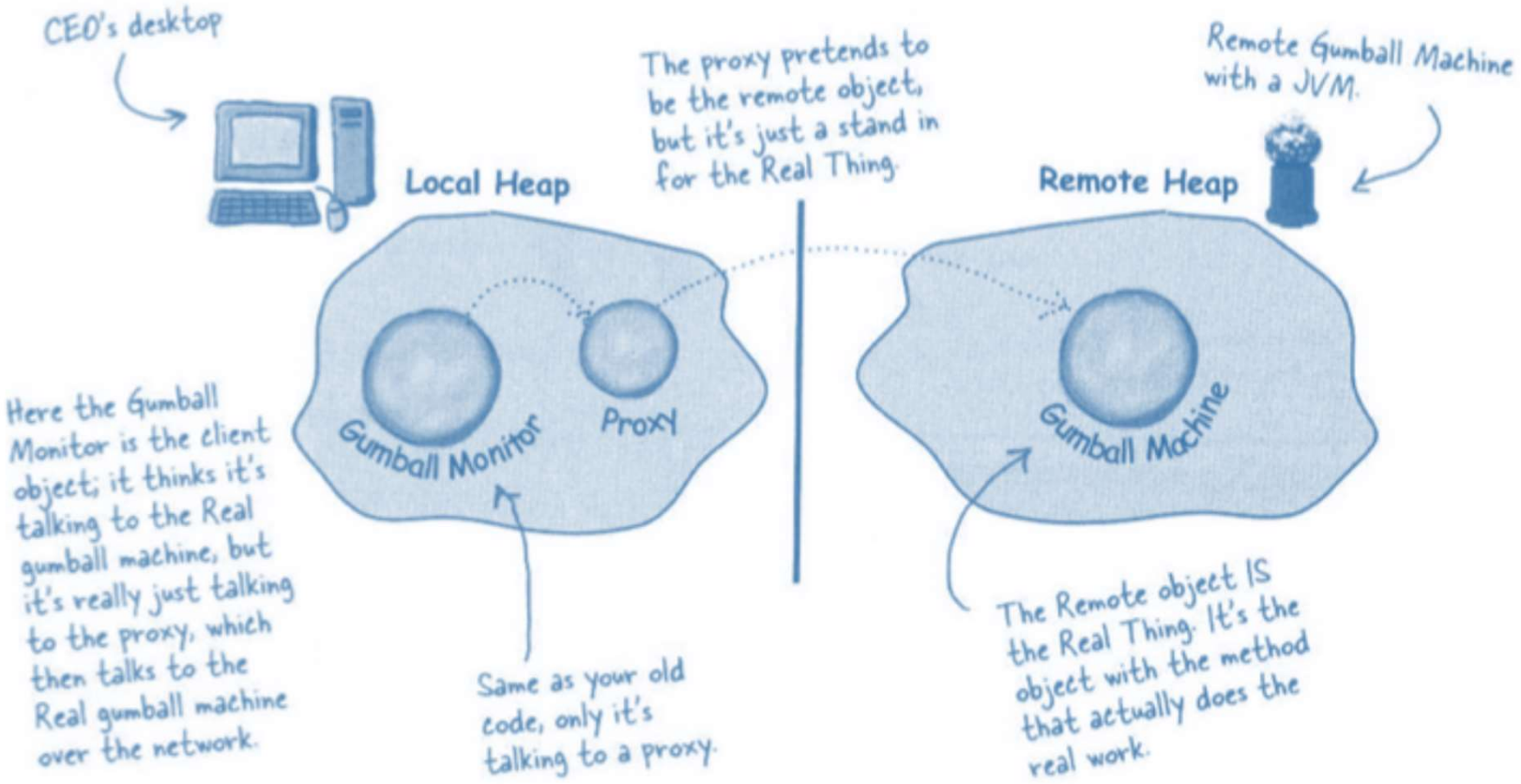
ESEMPIO (CONT.)

```
public class GumballMachine { // PROXY
  String location; // indirizzo dell'oggetto remoto (come stringa)
  // ...altri campi
  public GumballMachine(String location, int count) {
    this.location = location; // indirizzo salvato nel costruttore
    // ...resto del codice
  }
  // getter per l'indirizzo
  public String getLocation() {
    return location;
  }
  // ...resto del codice
}
```

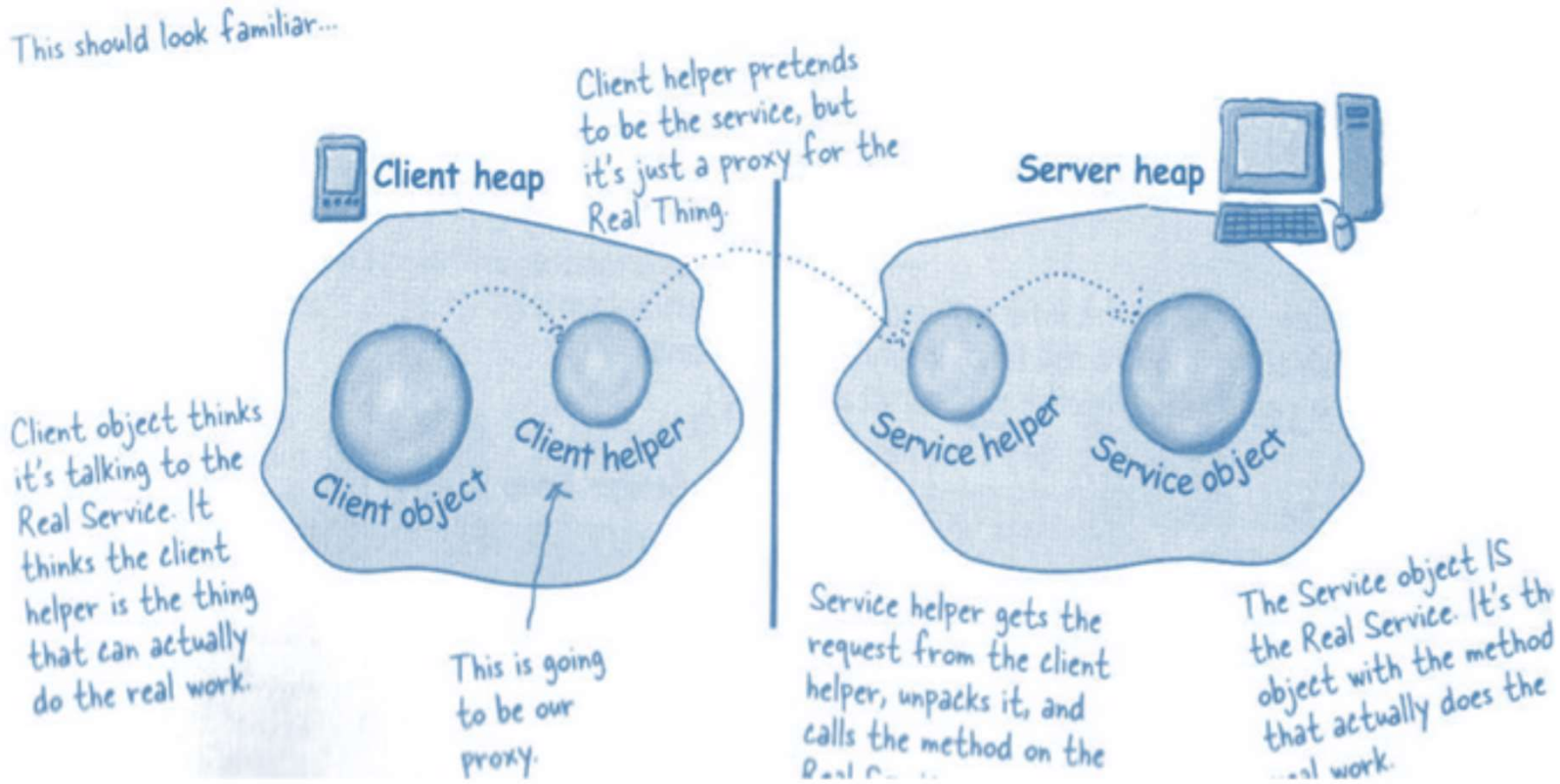
ESEMPIO (CONT.)

```
public class GumballMonitor { // CLIENT
  GumballMachine machine;

  // costruttore collegato al proxy della macchina
  public GumballMonitor(GumballMachine machine) {
    this.machine = machine
  }

  // report testuale con informazioni fornite dal proxy (in modo trasparente)
  public void report() {
    System.out.println("Gumball Machine: " + machine.getLocation())
    System.out.println("Inventory: " + machine.getCount() + " gumballs");
    System.out.println("State: " + machine.getState());
  }
}
```

ESEMPIO (CONT.)



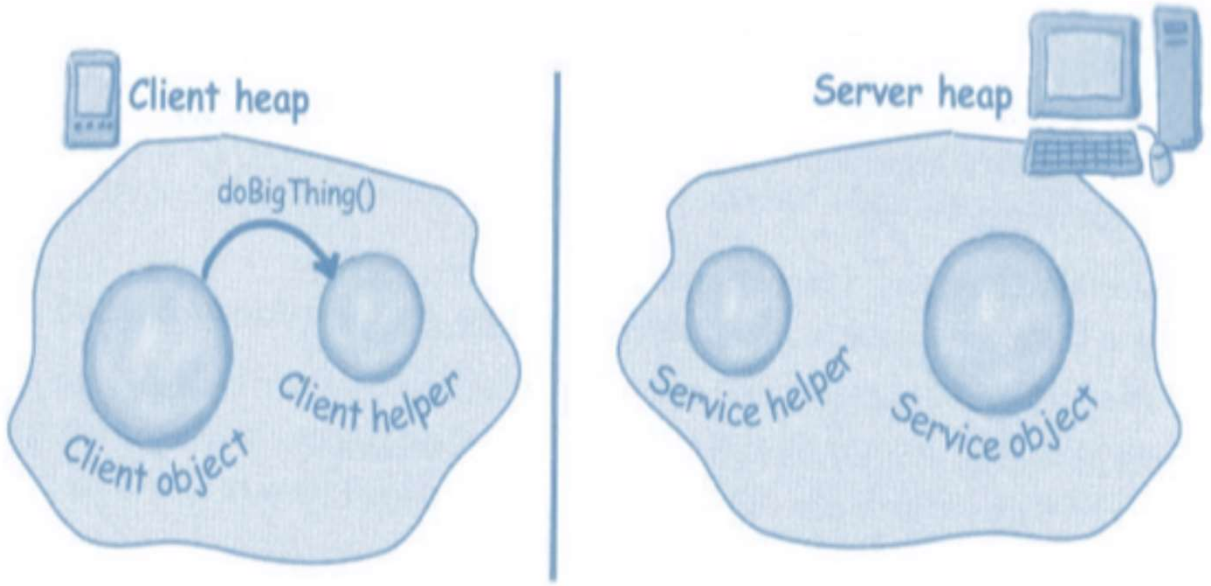
UN ALTRO ESEMPIO, CON DUE PROXY



UN ALTRO ESEMPIO (CONT.)

Come avviene l'interazione?

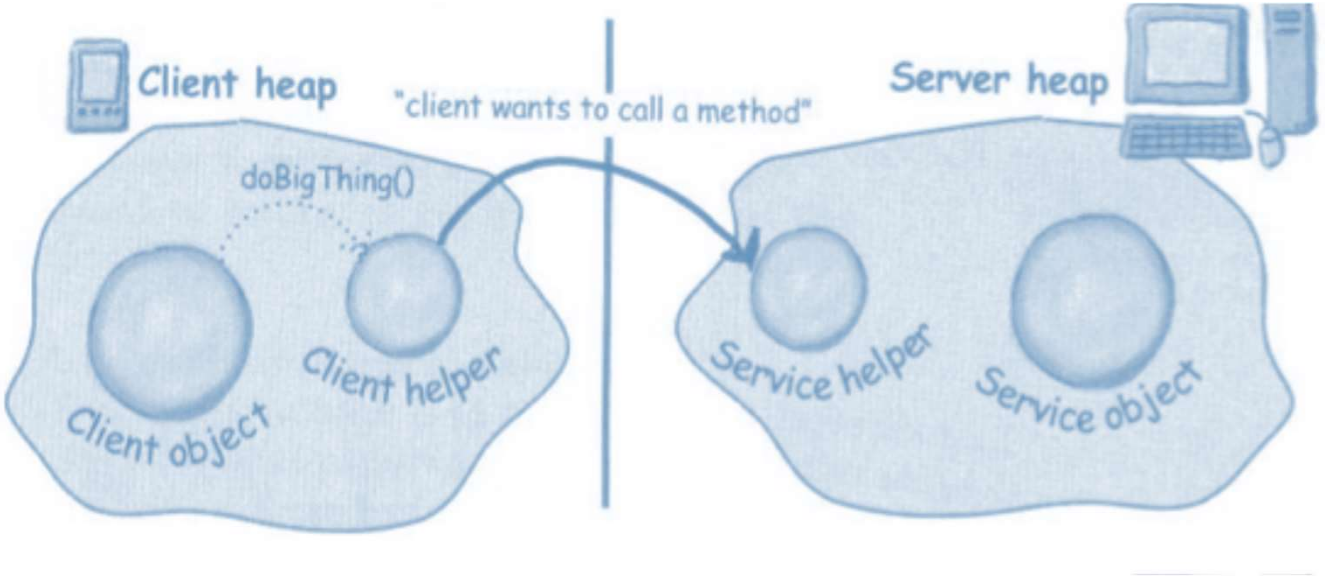
1) Il client invoca il metodo `doBigThing` offerto dal client helper



UN ALTRO ESEMPIO (CONT.)

Come avviene l'interazione?

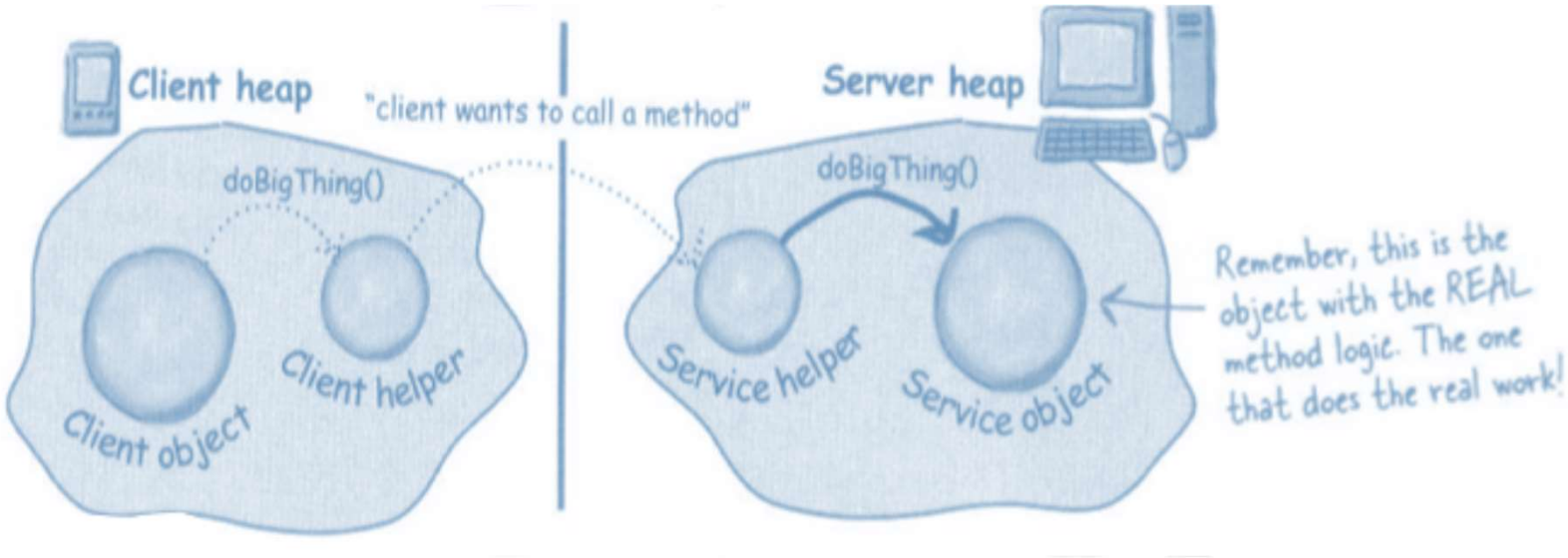
2) Il client helper inoltra la richiesta la service helper



UN ALTRO ESEMPIO (CONT.)

Come avviene l'interazione?

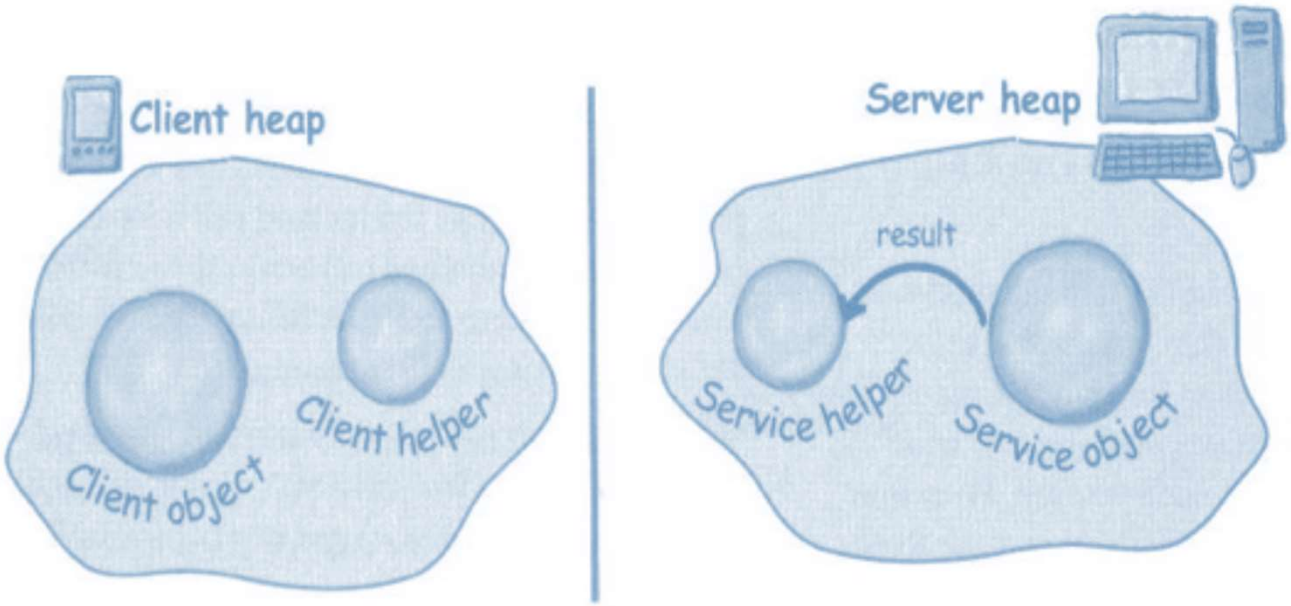
3) Il service helper invoca l'oggetto «vero»



UN ALTRO ESEMPIO (CONT.)

Come avviene l'interazione?

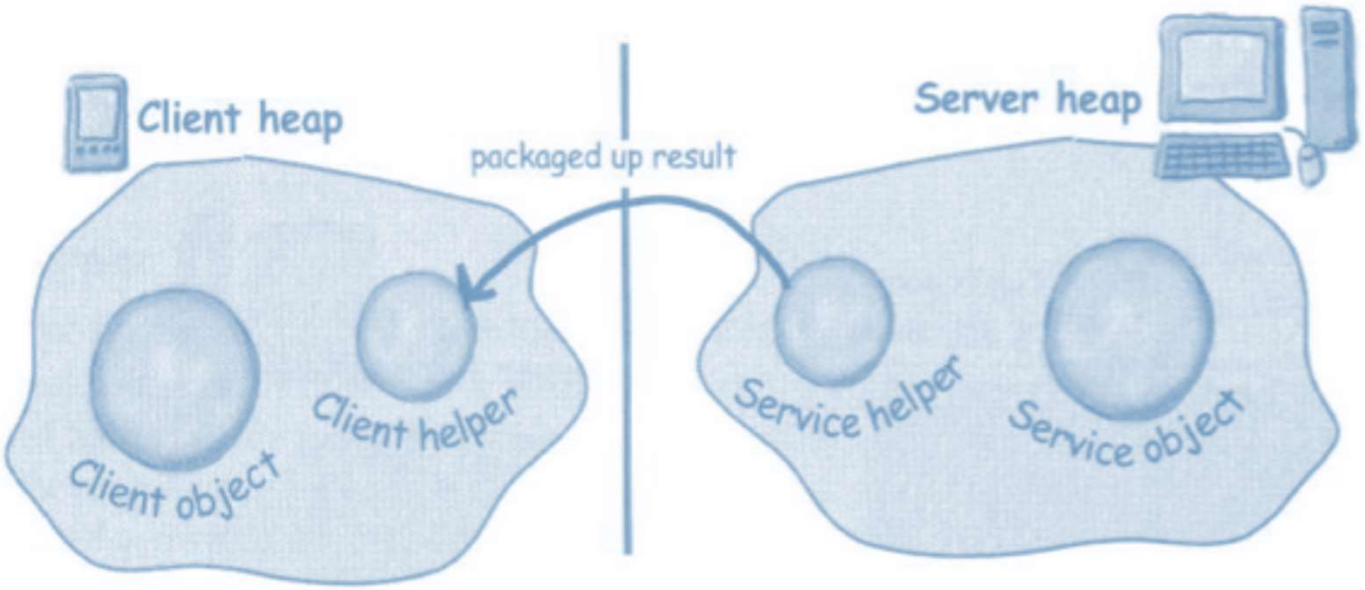
4) L'oggetto «vero» restituisce il risultato



UN ALTRO ESEMPIO (CONT.)

Come avviene l'interazione?

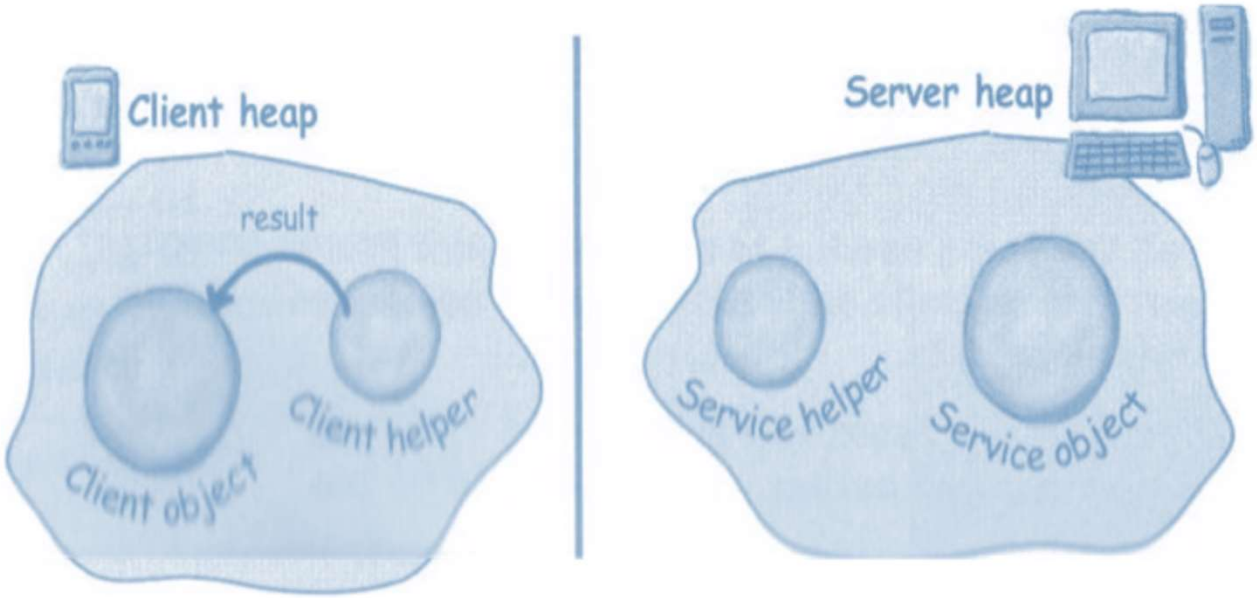
5) Il service helper inoltra il risultato al client helper



UN ALTRO ESEMPIO (CONT.)

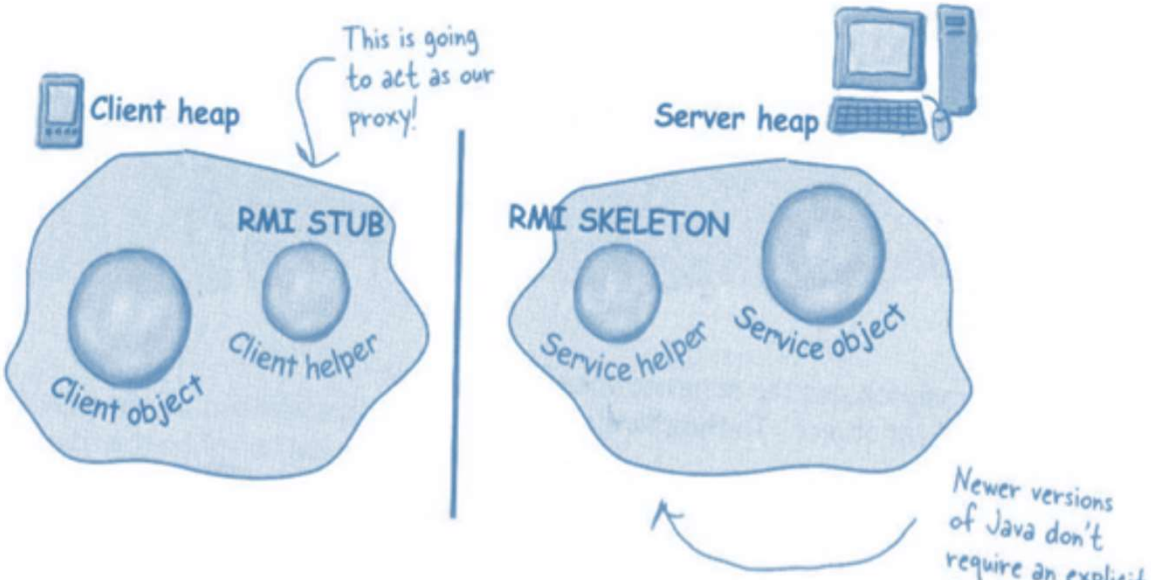
Come avviene l'interazione?

6) Il client helper restituisce il risultato al client



MOLTO SIMILE A JAVA RMI…

…con il client helper che fa da **stub**, mentre il service helper fa da **skeleton**



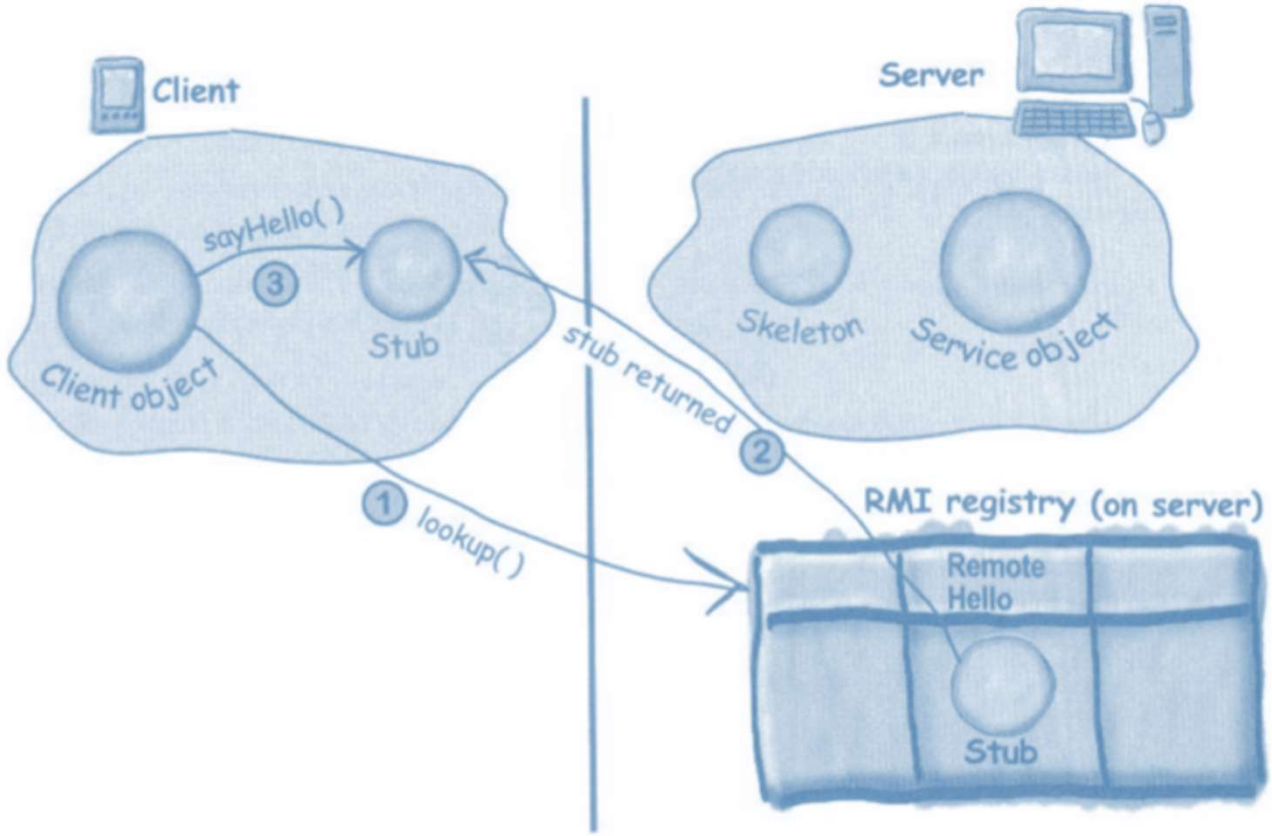
Ma come funziona?

JAVA RMI: COME FUNZIONA?

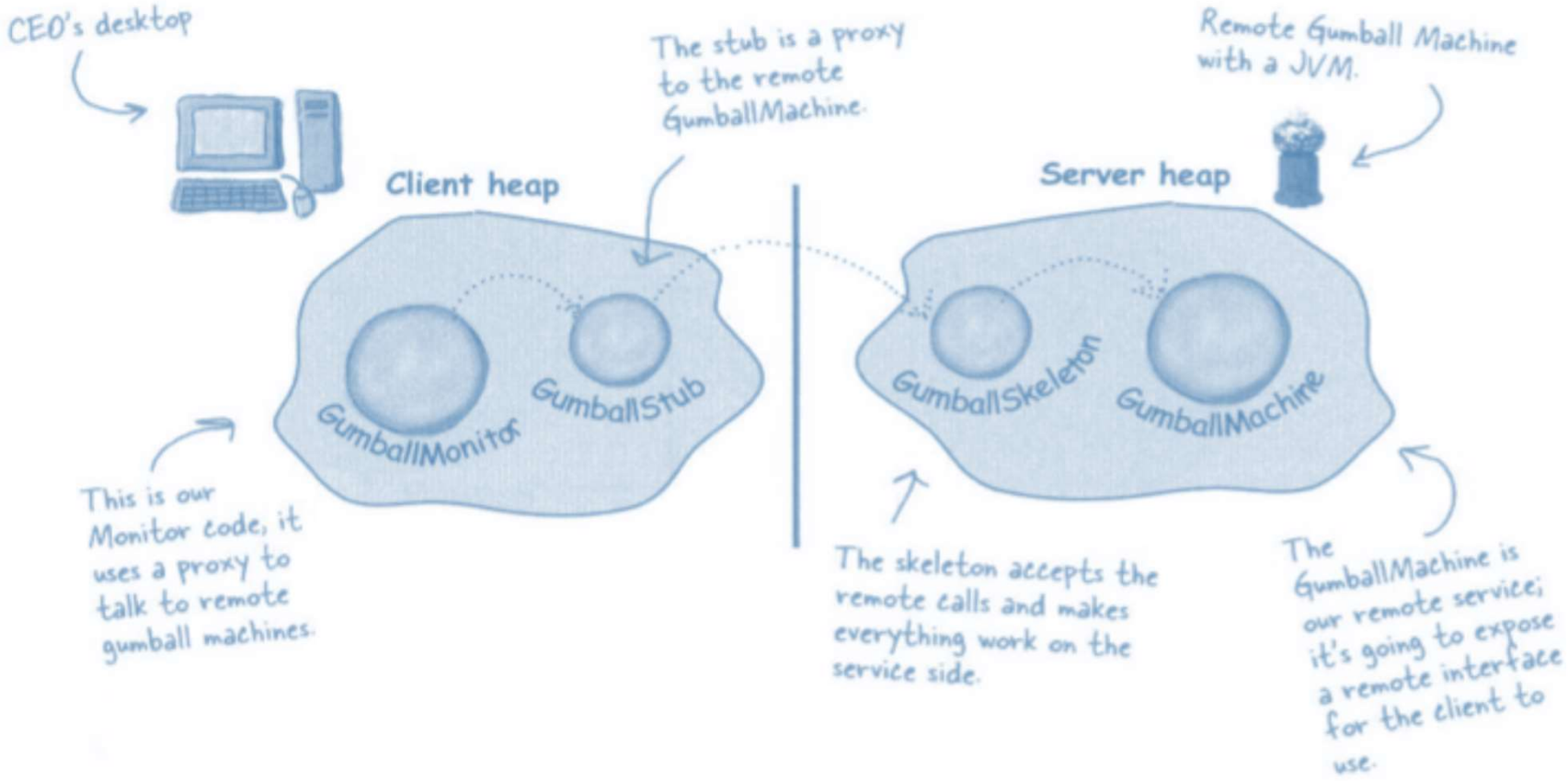
(1) Il client fa un ricerca nel registro RMI

(2) Il registro RMI restituisce un oggetto **stub** (remote proxy del servizio)

(3) Il client invoca lo **stub** come se fosse il servizio

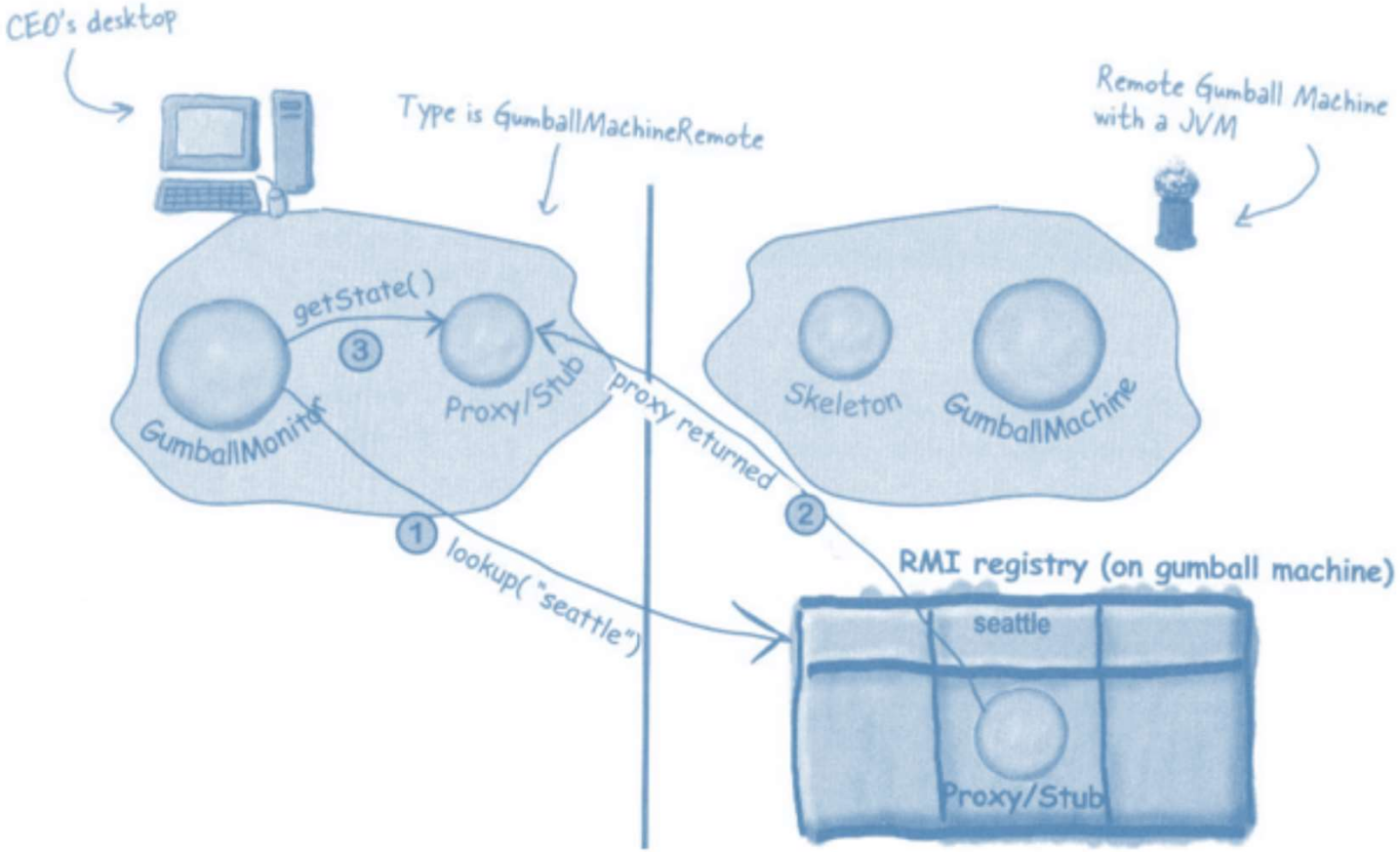


RMI E GUMBALL MACHINE



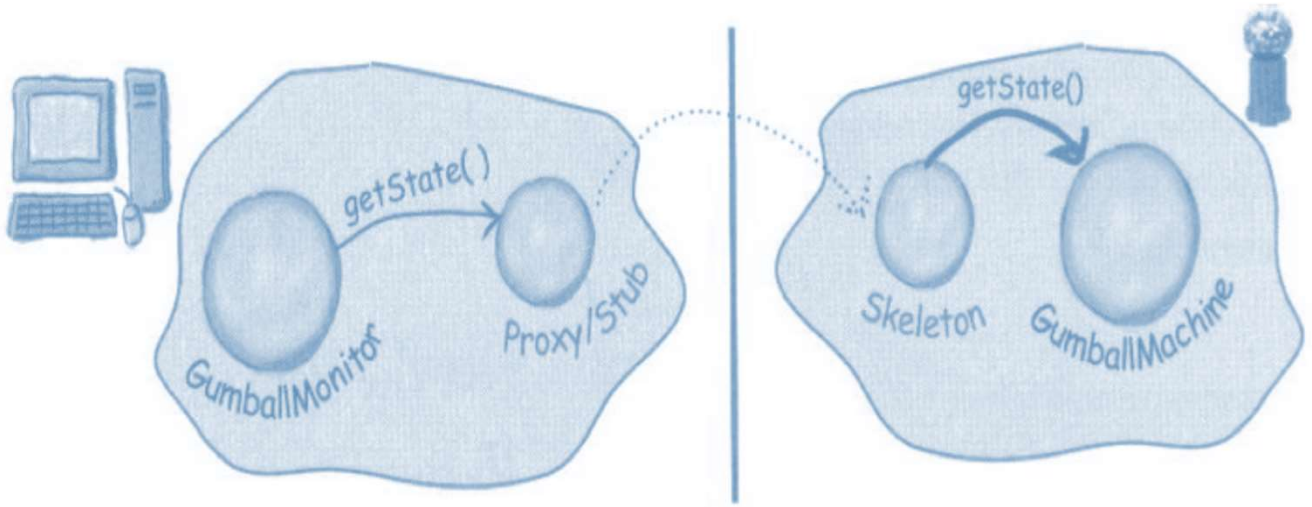
RMI E GUMBALL MACHINE (CONT.)

Il CEO avvia il monitor, che recupera il proxy dal registro RMI



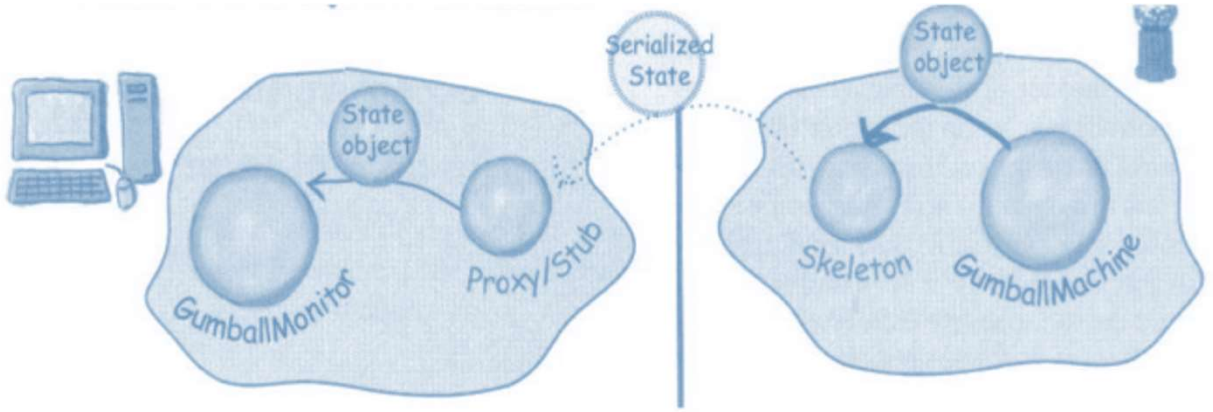
RMI E GUMBALL MACHINE (CONT.)

`GumballMonitor` invoca il metodo `getState` sul `Proxy`, che inoltra al servizio `GumballMachine` (attraverso lo `Skeleton`)



RMI E GUMBALL MACHINE (CONT.)

Il servizio `GumballMachine` restituisce lo stato allo `Skeleton`, che lo serializza e invia come risposta al `Proxy`



The monitor hasn't changed at all, except it knows it may encounter remote exceptions. It also uses the GumballMachineRemote interface rather than a concrete implementation.

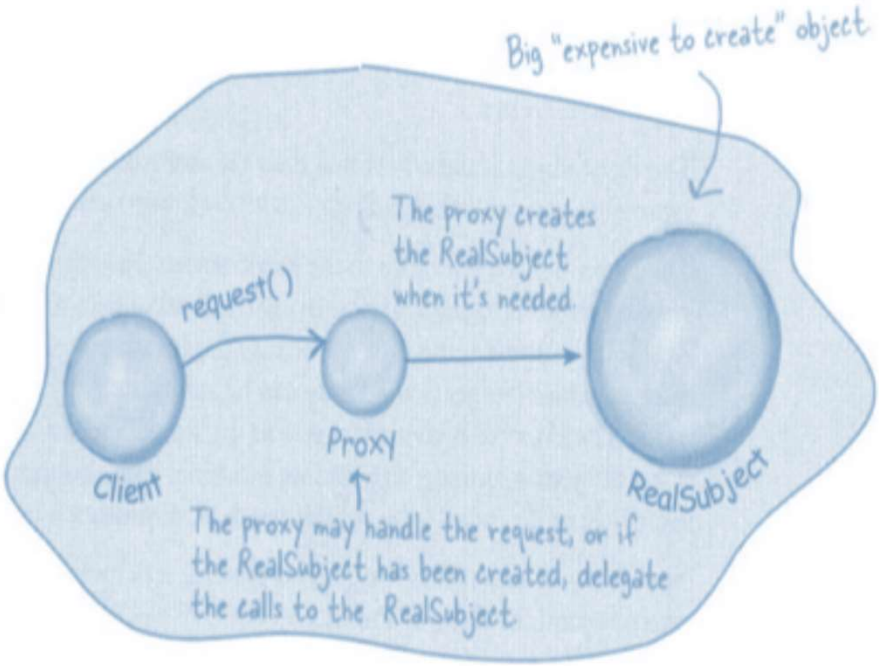
Likewise, the GumballMachine implements another interface and may throw a remote exception in its constructor, but other than that, the code hasn't changed.

We also have a small bit of code to register and locate stubs using the RMI registry. But no matter what, if we were writing something to work over the Internet, we'd need some kind of locator service.

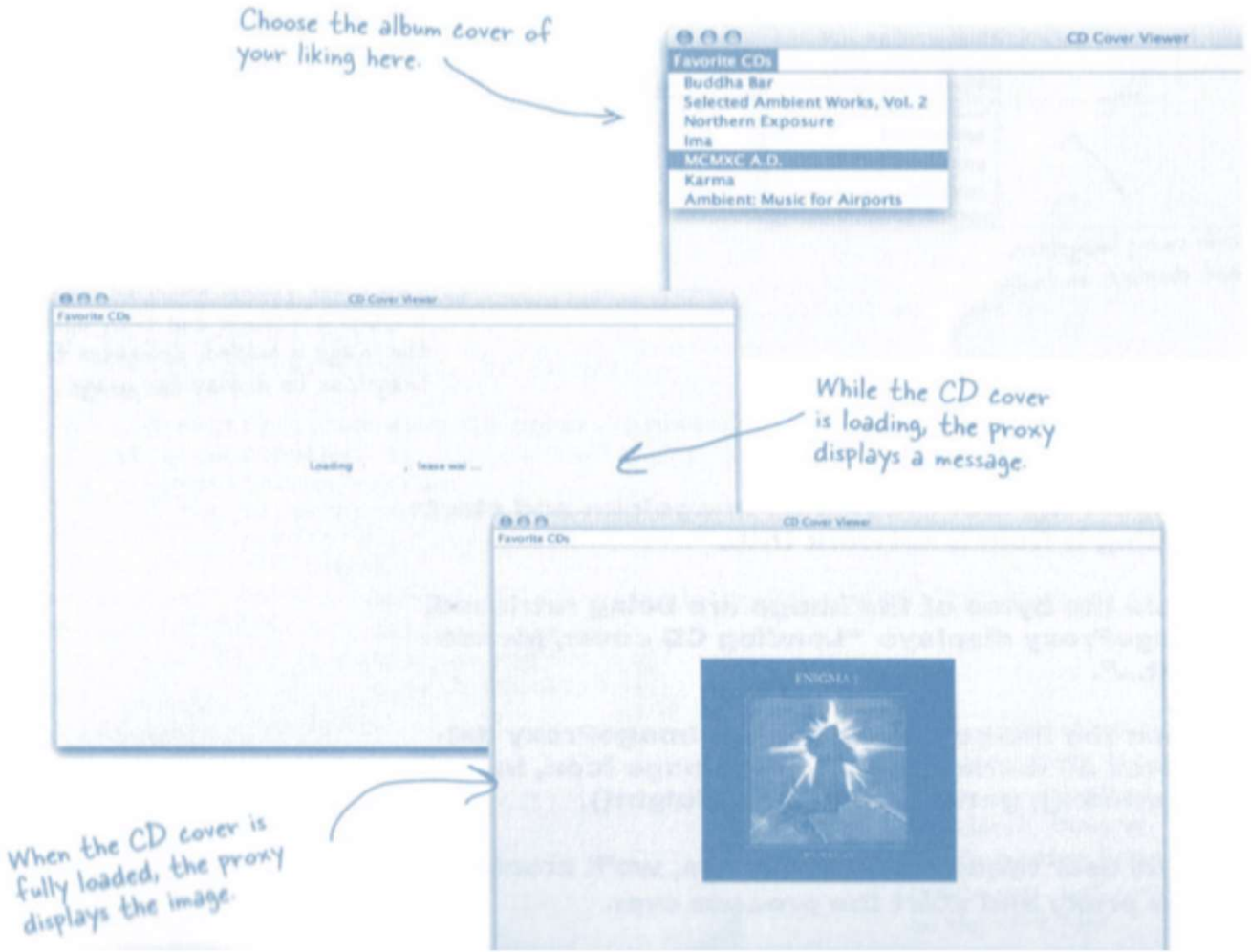
ESEMPIO DI VIRTUAL PROXY

Il **virtual proxy** risulta utile quando si vuole ritardare la creazione di un oggetto «costoso»

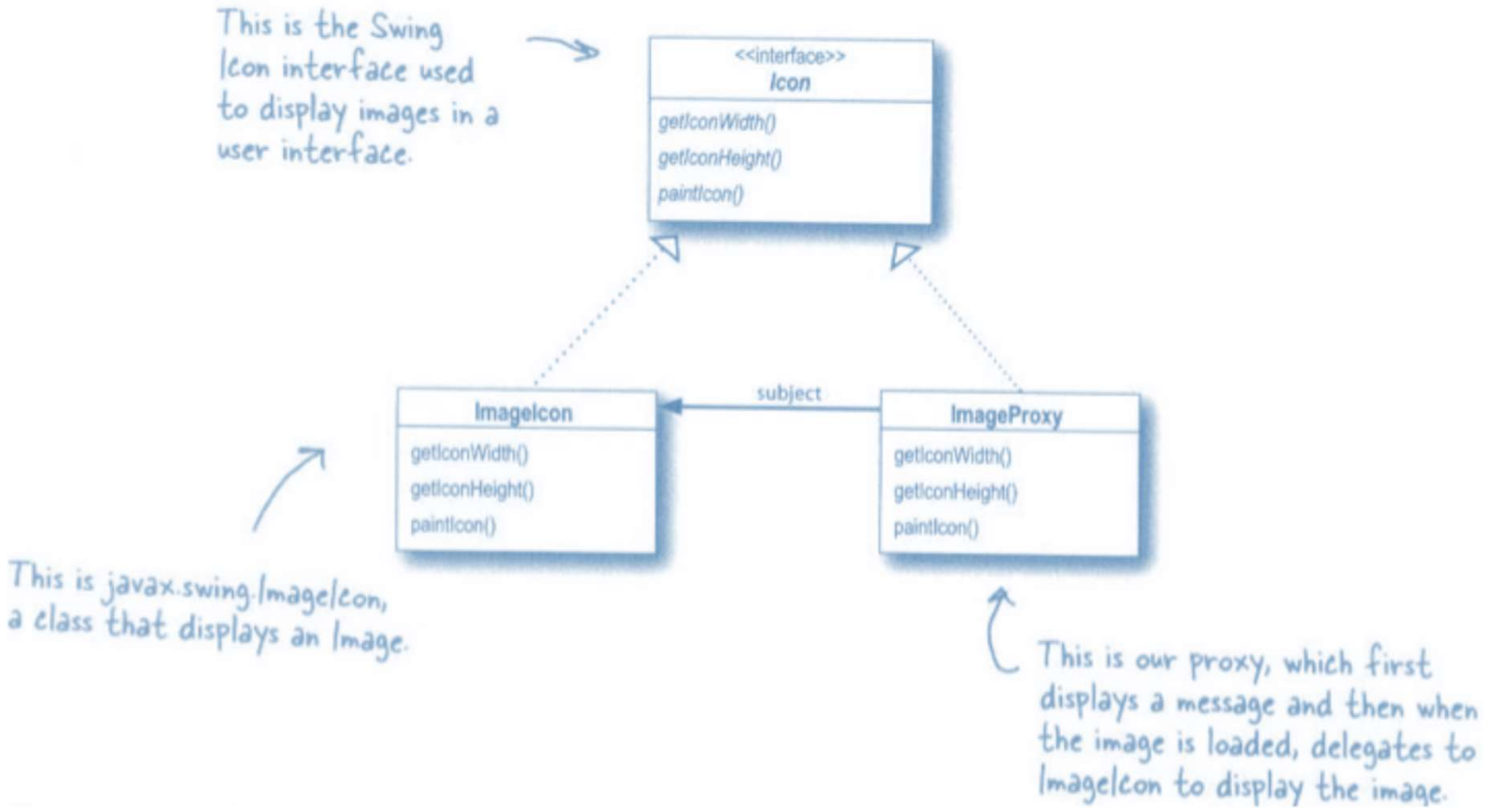
- Oggetto creato solo quando necessario
- Il virtual proxy funge da «surrogato» durante la creazione
- A creazione ultimata, il virtual proxy si limita ad inoltrare le richieste all'oggetto creato



ESEMPIO DI VIRTUAL PROXY (CONT.)



ESEMPIO DI VIRTUAL PROXY (CONT.)



ESEMPIO DI VIRTUAL PROXY (CONT.)

1. Viene creato un `ImageProxy`

2. Lo `ImageProxy` prima crea una `ImageIcon` e comincia il download da una URL

3. Durante il download, lo `ImageProxy` mostra la scritta "`Loading CD cover, please wait…`"

4. Quando il download è completo, lo `ImageProxy` delega tutte le richieste alla ImageIcon

5. Se l'utente richiede una nuova immagine, si riparte da 1

ESEMPIO DI VIRTUAL PROXY (CONT.)

1. Viene creato un `ImageProxy`

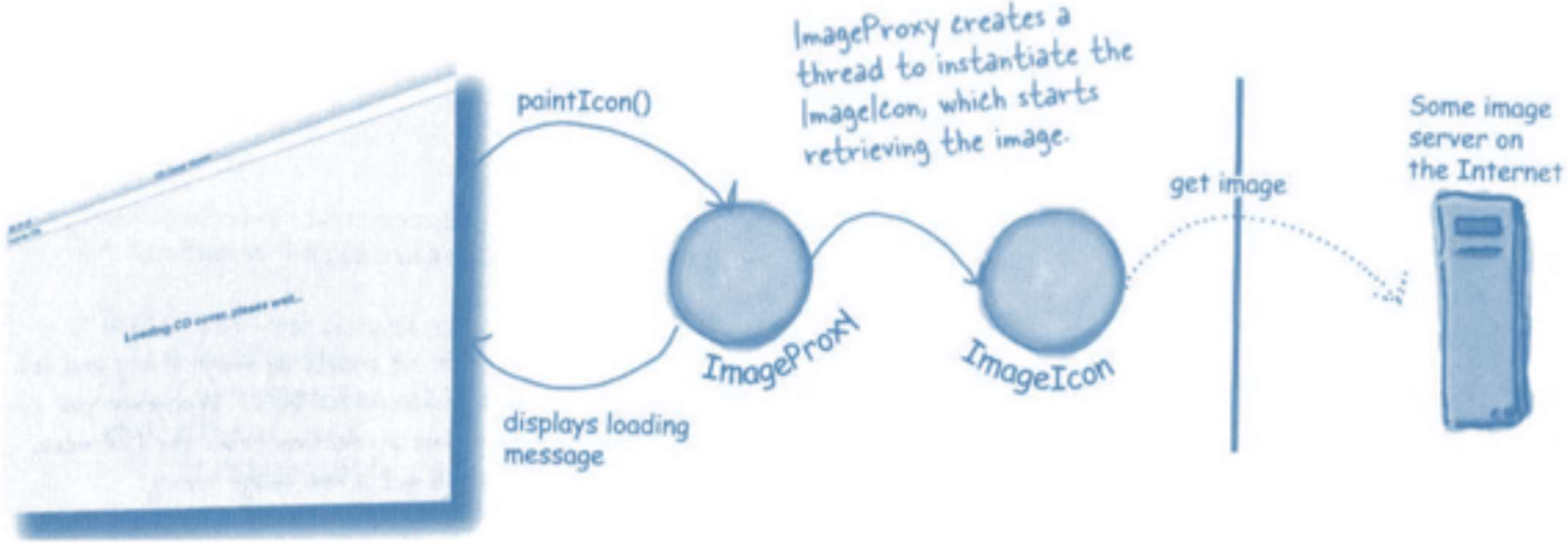
2. Lo `ImageProxy` prima crea una `ImageIcon` e comincia il download da una URL

3. Durante il download, lo `ImageProxy` mostra la scritta "`Loading CD cover, please wait…`"

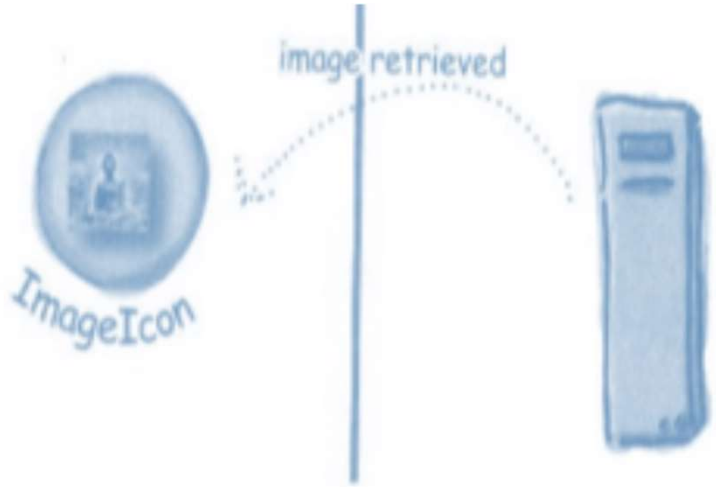
4. Quando il download è completo, lo `ImageProxy` delega tutte le richieste alla ImageIcon

5. Se l'utente richiede una nuova immagine, si riparte da 1

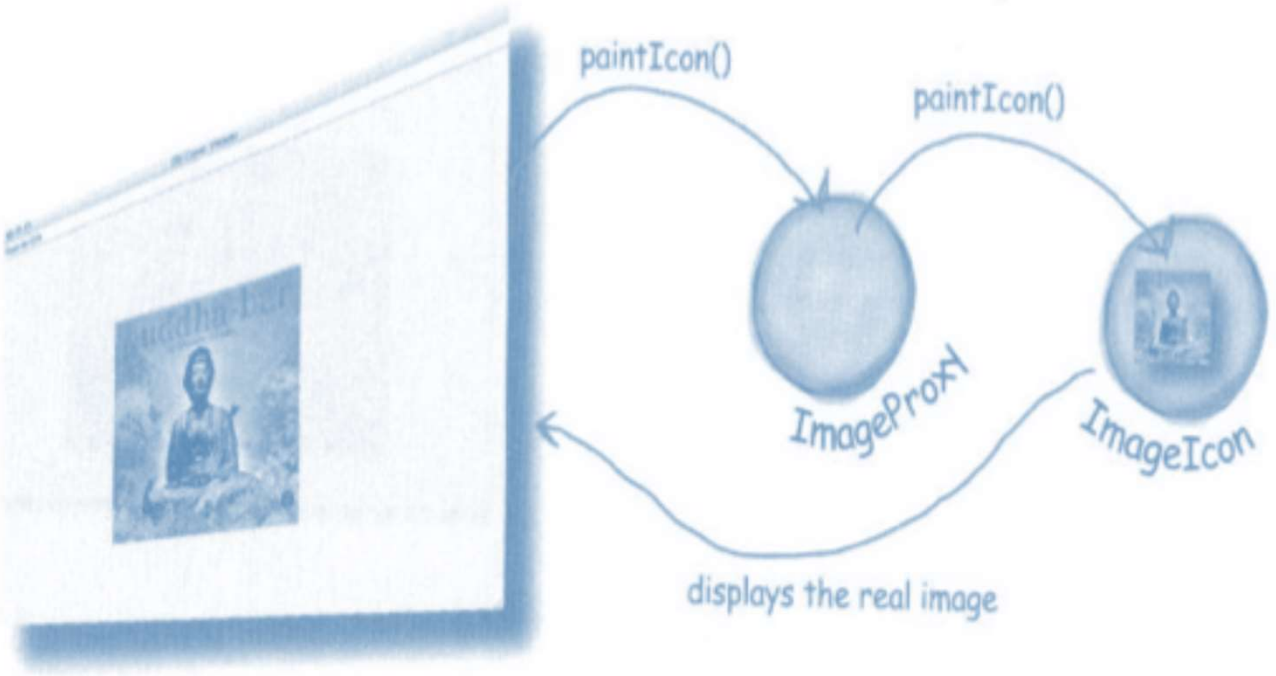
ESEMPIO DI VIRTUAL PROXY (CONT.)



ESEMPIO DI VIRTUAL PROXY (CONT.)



ESEMPIO DI VIRTUAL PROXY (CONT.)



ESEMPIO DI VIRTUAL PROXY (CONT.)

```
public class ImageProxy implements Icon {
  ImageIcon imageIcon;
  URL imageURL;
  Thread retrievalThread;
  boolean retrieving = false;

  public ImageProxy(URL url) { imageURL = url; }

  public int getIconWidth() {
    // se c'è l'oggetto, delega all'oggetto stesso
    if (imageIcon != null) return imageIcon.getIconWidth();
    // altrimenti fornisce una risposta di default
    else return 800;
  } //...
```

ESEMPIO DI VIRTUAL PROXY (CONT.)

```
public int getIconHeight() {
  if (imageIcon != null) return imageIcon.getIconHeight();
  else return 600;
}
public void paintIcon(final Component c, Graphics  g, int x,  int y) {
  if (imageIcon != null)  imageIcon.paintIcon(c, g, x, y);
  else {
    g.drawString("Loading CD cover, please wait...", x+300, y+190);
    if (!retrieving) {
      retrieving = true;
      retrievalThread = new Thread(new Runnable() {
        // codice del thread nella slide successiva
      });
      retrievalThread.start(); }}}}
```

ESEMPIO DI VIRTUAL PROXY (CONT.)

```
// codice del thread
public void run() {
  try {
    imageIcon = new ImageIcon(imageURL, "CD Cover");
    c.repaint();
  } catch (Exception e) { e.printStackTrace(); }
}
```

HOMEWORK

Progettare un firewall proxy per smartphone.

- Il proxy deve filtrare sms e chiamate provenienti da «stalker».
- Gli stalker sono elencati in una blacklist.
- La blacklist deve essere aggiornabile.


Q
A

RIFERIMENTI

Contenuti

- **Capitoli 16 e 17** di "Software Engineering" (G. C. Kung, 2023)

Approfondimenti