Il materiale presentato corrisponde a (passi scelti di):
- Go4: *Design Patterns*. Addison Wesley, 2002
- S.J. Metsker: *Design Pattern in Java*. Addison Wesley, 2003

Dipartimento di Informatica
e Sistemistica Antonio Ruberti

Sapienza
Università di Roma

(Alcuni) Design Pattern e Tecniche di Riuso, Architettura Software e sue Qualità

Progetto di Applicazioni Software

Pattern

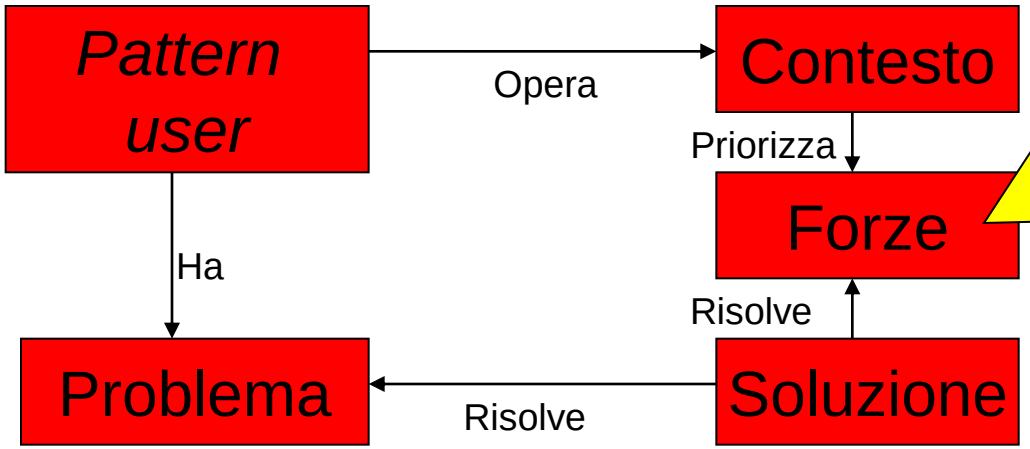
- Il concetto di pattern nasce dall'idea del **prof. Alexander** nel contesto dell'architettura ma può essere applicato in molte altre discipline compresa la progettazione del software
  - Alexander, nella sua lunga esperienza, aveva sviluppato una serie di *best practice*, cioè idee progettuali/realizzative che si erano dimostrate efficaci ed efficienti.
  - Il messaggio di Alexander ai futuri esperti era: "Se l'idea che ha portato ad una *soluzione* del *problema* ha una valida generale e se il *contesto* ne permette l'applicazione, perché non riutilizzarla?"
  - L'idea è che non è necessario "reinventare la ruota" ogni volta: i pattern sono un sistema eccellente per catturare ed esprimere l'esperienza maturata in un determinato mestiere
- I design pattern (letteralmente "pattern di progettazione") sono pattern (cioè nati per perseguire un intento: la soluzione di un problema) che utilizzano metodi e classi di un linguaggio OO.
  - I design pattern sono ad un livello più elevato rispetto ai linguaggi di programmazione Object Oriented perché indipendenti dagli stessi
    - I design pattern più importanti originariamente erano stati sviluppati per *SmallTalk*, un linguaggio Object Oriented che non si è mai diffuso

Gli Elementi in Gioco

- I pattern originali di Alexander erano formulati nel modo seguente:

```
SE ti trovi in CONTESTO
come nel caso ESEMPI
con questo PROBLEMA
che implica REQUISITI
ALLORA per queste MOTIVAZIONI
applica FORMATI_PROGETTO AND/OR REGOLE_PROGETTO
per realizzare SOLUZIONE
che guida a NUOVO_CONTESTO AND NUOVI_PATTERN
```

- Ad ogni design pattern è stato associato un nome compatto che ne chiarisse l'essenza



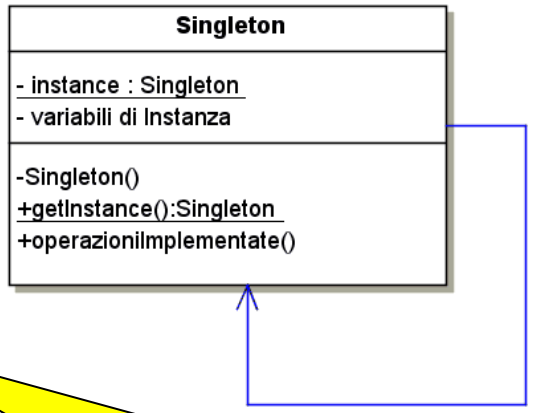
uno scenario che illustra il problema di progettazione e come le classi e gli oggetti del pattern lo risolvono. Devono essere descritte tutte quelle forze (problemi, vincoli e implicazioni varie) che influenzano la parte del progetto che il pattern deve affrontare, le interazioni e i conflitti

Singleton / 1

- Questo *design pattern* è usato per assicurare che una classe abbia una sola istanza ed un unico punto di accesso globale.
- In molte situazioni c'è la necessità di garantire l'esistenza di un unico oggetto di una classe
  - Ad esempio in un sistema ci possono essere tante stampanti ma solo una classe *PrintSpooler*
- Le classi *Singleton* vengono progettate con i costruttori privati per evitare la possibilità di istanziare un numero arbitrario di oggetti della stessa.
- Esiste un metodo statico con la responsabilità di assicurare che nessuna altra istanza venga creata oltre la prima, restituendo contemporaneamente un riferimento all'unica esistente

Singleton / 2

- La classe mantiene all'interno il riferimento all'unica istanza *Singleton* della classe
  - L'istanza alla prima esecuzione del metodo statico (**inizializzazione pigra**) oppure contemporaneamente alla definizione della variabile di istanza riferimento all'oggetto
- La classe contiene poi tutti i metodi, le proprietà e gli attributi tipici dell'astrazione per cui è stata concepita



L'inizializzazione pigra viene spesso usata quando non si hanno abbastanza informazioni per istanziare l'oggetto Singleton nel momento di dell'inizializzazione statica. Un Singleton potrebbe richiedere valori calcolati più avanti

Esempio Singleton

```
public class Singleton {
          // istanza allocata in modo pigro
          private static Singleton instance = null;
          // specifica delle proprieta' caratteristiche
          // dell'oggetto
          // ...
          // costruttore definito come privato in modo
          // da proibire ai client di creare oggetti
          private Singleton() {}
          public static Singleton getInstance() {
                    if ( instance == null )
                              instance = new Singleton();
                    return instance;
          }
}
public class Application {
          public static void main(String args[])  {
                    Singleton obj=Singleton.getInstance();
                    Singleton obj2=Singleton.getInstance();
                    System.out.println("Primo oggetto:"+obj);
                    System.out.println("Secondo oggetto:"+obj2);
                    if (obj==obj2)
                              System.out.println("Sono lo STESSO oggetto");
          }
}
```

Variazioni per il singleton non pigro:

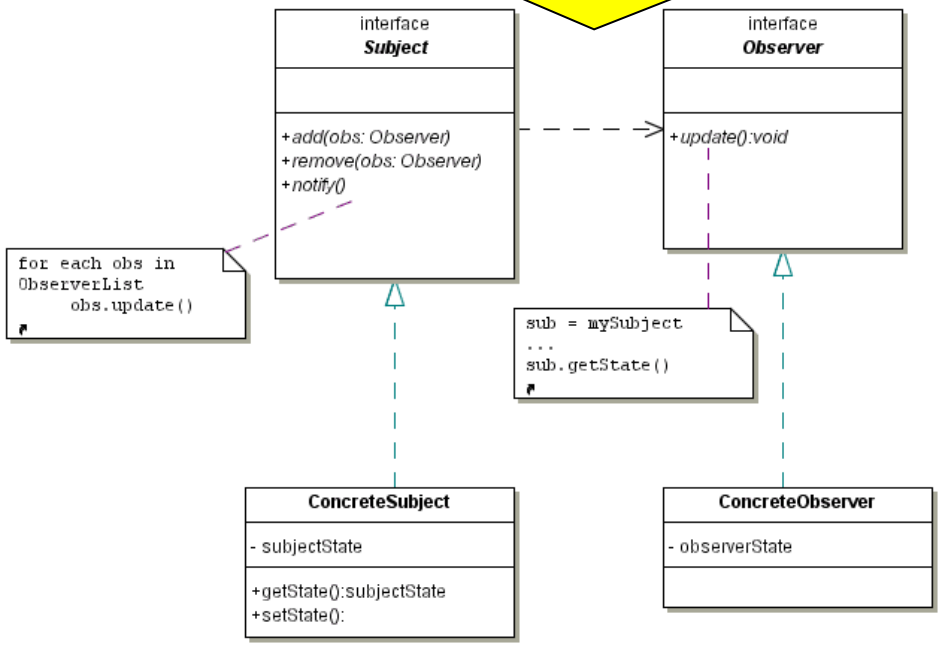
```
private static Singleton
instance = new Singleton();

private Singleton() {}

public … getInstance() {
   return (instance);
}
```

Observer / 1

- L'intento del pattern *Observer* è definire una dipendenza uno-a-molti tale che quando un oggetto cambia stato tutti quelli che ne dipendono vengono automaticamente notificati del fatto ed aggiornati di conseguenza
  - L'oggetto osservato è chiamato ***Subject*** (soggetto) mentre gli oggetti osservatori sono noti come ***Observer***
- In risposta alla notifica, ogni osservatore richiederà al soggetto le informazioni necessarie per sincronizzare il proprio stato con il nuovo stato del soggetto

Observer / 2

Gli ascoltatori delle Swing sono un esempio di pattern Observer. Infatti il componente che scatena l'evento è il subject mentre gli ascoltatori formano una lista di *Observer*



Noto anche come
Publish/Subscribe

- Il soggetto *pubblica* le modifiche al proprio stato
  - La pubblicazione agli osservatori è fatta chiamando il metodo *notify* che internamente chiama il metodo *update* di tutti gli Observer registrati

Gli osservatori *si sottoscrivono* per ottenere gli aggiornamenti

- La sottoscrizione di un Observer x è realizzata da x invocando sul subject il metodo *add(x)*

Implementaz. Observer/1

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

```
public interface Observer {
        void Update();
}

public class ConcreteObserver implements Observer {
        private ConcreteSubject subject;

        public ConcreteObserver(ConcreteSubject aSubject)
        {
                subject = aSubject;
                subject.add(this);
        }
        public void Update()
        {
                System.out.println("Io sono l'observer " + this +
                " ed il nuovo stato del mio subject e' " +
                subject.getState());
        }
}
```

Implementaz. Observer/2

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

```
public interface Subject {
        void add(Observer anObserver);
        void remove(Observer anObserver);
        void notify();
}

public class ConcreteSubject implements Subject {
        private int state;
        // la lista degli osservatori
        private LinkedList observers;
        public ConcreteSubject(int initialState){
                state = initialState;
        }
        public int getState() { return state; }
        public void setState(int aState) { state = aState; }
```

Implementaz. Observer/3

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

```
public void Notify() {
        Iterator iter=observers.iterator();
        Observer obs;
        while(iter.hasNext()) {
                obs=(Observer)iter.next();
                obs.update();
        }
}

public void add(Observer anObserver) {
        observers.add(anObserver);
}

public void remove(Observer anObserver) {
        observers.remove(anObserver);
}
```

Implementaz. Observer/4

DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI
SAPIENZA
UNIVERSITÀ DI ROMA

```
public static void main(String[] args) {
int concreteObservers[3]; int initialState = 10;
ConcreteSubject mySubject = new ConcreteSubject(initialState);
System.out.println("E' stato creato un subject con stato " +
mySubject.getState());
for (int i = 0; i < 3; i++) {
        concreteObservers[i] = new ConcreteObserver(mySubject);
}
System.out.println("..."); mySubject.setState(17);
mySubject.Notify(); System.out.println("...");
mySubject.setState(06); mySubject.Notify(); }
```

Un possibile output è il seguente:

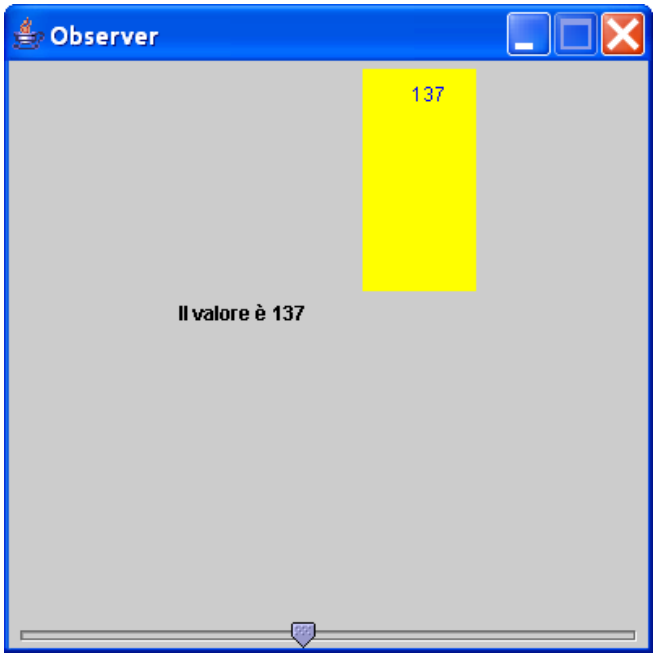
E' stato creato un subject con stato 10
...
Io sono l'observer ConcreteObserver@288051 ed il nuovo stato del mio subject e' 17
Io sono l'observer ConcreteObserver@45eb ed il nuovo stato del mio subject e' 17
Io sono l'observer ConcreteObserver@ee7a14 ed il nuovo stato del mio subject e' 17
...
Io sono l'observer ConcreteObserver@288051 ed il nuovo stato del mio subject e' 6
Io sono l'observer ConcreteObserver@45eb ed il nuovo stato del mio subject e' 6
Io sono l'observer ConcreteObserver@ee7a14 ed il nuovo stato del mio subject e' 6

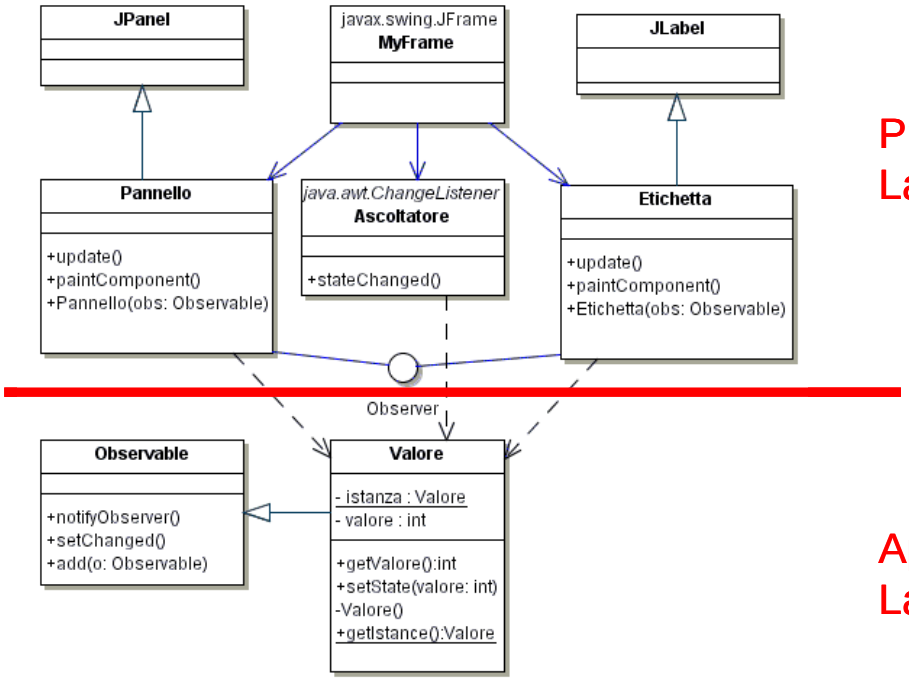
Implementazione Java

- Java fornisce già implementate le classi per realizzare il pattern *Observer*
  - Gli *osservatori* devono implementare l'interfaccia `java.util.Observer` la quale definisce il metodo
    - `public void update(Observable o,Object arg)`
  - Il *subject* per essere tale deve estendere la classe `java.util.Observable` che tra gli altri fornisce i seguenti metodi:
    - `public void addObserver(Observer o)`
    - `public void removeObserver(Observer o)`
    - `public void notifyObserver([Object arg])`
    - `protected void setChanged()`
  - Il subject notifica il cambiamento dello stato invocando *notifyObserver* il quale chiama i metodi *update* degli osservatori installati quando vengono chiamati in sequenza
    - Si intende che il *subject* cambia stato quando viene chiamato il metodo *setChanged*

Esempio / 1

- Si vuole realizzare una finestra grafica con una JSlider. La modifica del valore aggiorna una JLabel e un istogramma contenuto in un pannello





Presentation Layer

Application Layer

Esempio / 2

```
class Valore extends java.util.Observable
{
        private int stato;
        private static Valore val=new Valore();
        private Valore() {}

        public static Valore getIstance() {
                return(val);
        }

        public void setState(int unoStato) {
                stato=unoStato;
                this.setChanged();
                this.notifyObservers();
        }

        public int getState() {
                return(stato);
        }
}
```

Notifica agli ascoltatori la modifica dello stato del *subject*

Esempio / 3

```
class Etichetta extends JLabel implements java.util.Observer {
  public Etichetta(java.util.Observable obs) {
    obs.addObserver(this);
  }

  public void update(java.util.Observable obs,Object arg) {
    if (obs instanceof Valore) {
      int state = ( (Valore) obs).getState();
      this.setText("Il valore è "+state);
    }
  }
}

class Listener implements ChangeListener {
  public void stateChanged(ChangeEvent ce) {
    JSlider obj=(JSlider)ce.getSource();
    Valore.getIstance().setState(obj.getValue());
  }
}
```

Si registra come osservatore del subject passato come param

Aggiorna i contenuto dell'etichetta quando viene notificato l'aggiornamento dello stato del subject

ChangeListener si occupa di gestire gli eventi di modifica del valore scelto nella *JSlider*. Dichiara il metodo *stateChanged*, invocato allo scatenarsi dell'evento.

Esempio / 4

```
class Pannello extends JPanel implements java.util.Observer {
 int state;
 public void paintComponent(Graphics g)   {
   super.paintComponent(g);
   Color c;
   if (state<100) c=Color.green; else if (state<200)
   c=Color.yellow; else c=Color.red;
   Graphics2D g2=(Graphics2D)g;
   g2.setColor(c);
   g2.fillRect(30,0,70,state);
   g2.setColor(Color.blue);
   g2.drawString(String.valueOf(state),60,20); }
 public void update(java.util.Observable obs,Object arg) {
   if (obs instanceof Valore) {
     state=Valore.getIstance().getState();
     repaint(); }  }

 public Pannello(java.util.Observable obs) {
   obs.addObserver(this);
   this.setPreferredSize(new Dimension(100,300)); } }
```

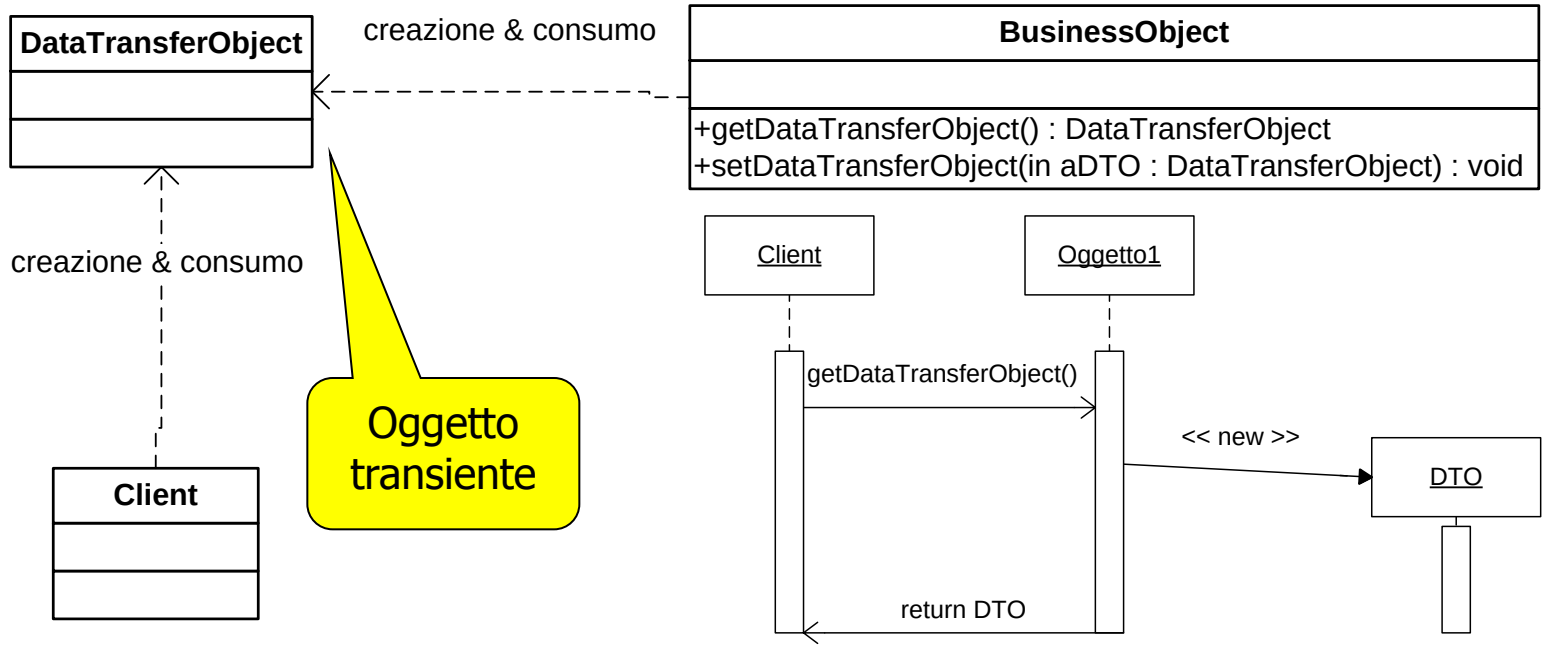
Esempio / 5

```
public class MyFrame extends JFrame {
  JSlider sudSlider = new JSlider(0,300);
  JPanel centroPnl = new JPanel();
  Etichetta et;
  Listener ascoltatore=new Listener();
  Pannello pannello;
  public MyFrame()  {
    super("Observer");
    et=new Etichetta(Valore.getIstance());
    pannello=new Pannello(Valore.getIstance());
    this.getContentPane().add(sudSlider, BorderLayout.SOUTH);
    this.getContentPane().add(centroPnl,  BorderLayout.CENTER);
    centroPnl.add(et);
    centroPnl.add(pannello);
    sudSlider.addChangeListener(ascoltatore);
    sudSlider.setValue(0);
    ...
  }
}
```

Data Transfer Object / 1

- Quando l'invocazione tra oggetti è remota, è più conveniente permettere al client di gestire molti valori in un'unica operazione piuttosto che avere invocazioni distinte
  - Infatti quello che "pesa" nell'invocazione remota non sono tanto i dati trasportati, ma l'invocazione stessa
- Analogamente permettere al client di gestire più oggetti con una singola invocazione piuttosto che singolarmente

Data Transfer Object / 2



- **`DataTransferObject`** (DTO): contiene un sottoinsieme dei dati disponibili nel `BusinessObject`, ma più di uno. Le istanze sono transienti, create all'occorrenza per il solo scopo di muovere I dati tra `Client` e `BusinessObject`. Non presenta metodi comportamentali e (se in Java) permette accesso diretto agli attributi, senza metodi `get`/`set`

Data Transfer Object / 3

- **`BusinessObject`**: i dati vengono acceduti in gruppi, attraverso metodi `get`/`set`. Ad ogni richiesta del client viene tipicamente creato una nuova istanza di `DataTransferObject`
- **`Client`**: è remoto rispetto al `BusinessObject`. Se fosse locale il pattern avrebbe poco senso
- In caso di client diversi con esigenze differenti, ci possono essere diversi DTO a fronte dello stesso `BusinessObject`

Data Transfer Object / 4

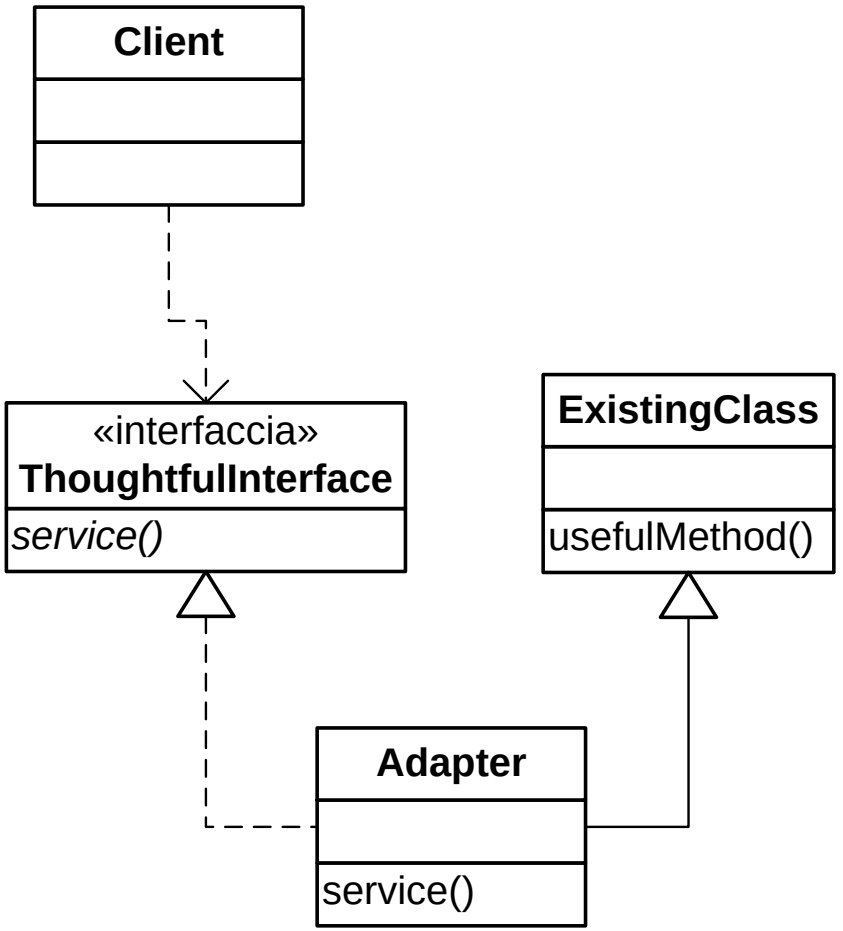
- Omogeneità nei dati del DTO
  - Eventualmente pensare più DTO in base a differenti esigenze di accesso
- Solo dati, nessun comportamento
- Spesso coincide con una transazione del `BusinessObject` (modifica coerente del livello di persistenza)
- In molti casi il DTO è realizzato in XML
  - definizione del DTO è un XSD
  - un'istanza passata come parametro è un documento XML

Adapter / 1

- Convertire l'interfaccia di una classe in un'altra richiesta dal client
  - Convertire l'interfaccia di (una porzione di software di) un legacy system in quella progettata
- Consente a classi diverse di operare insieme quando ciò non sarebbe altrimenti possibile a causa di interfacce incompatibili

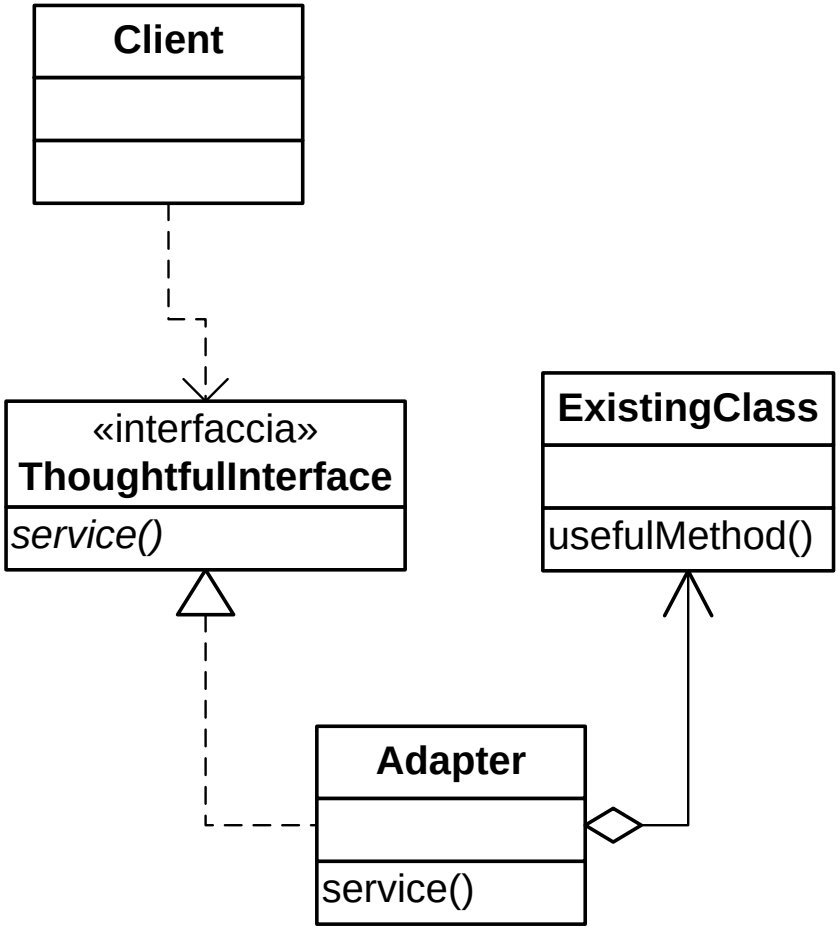
Adapter / 2

- L'Adapter implementa l'interfaccia, estende la classe esistente e ridefinisce il metodo chiamando quello esistente



Adapter / 3 (Adapter Object)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

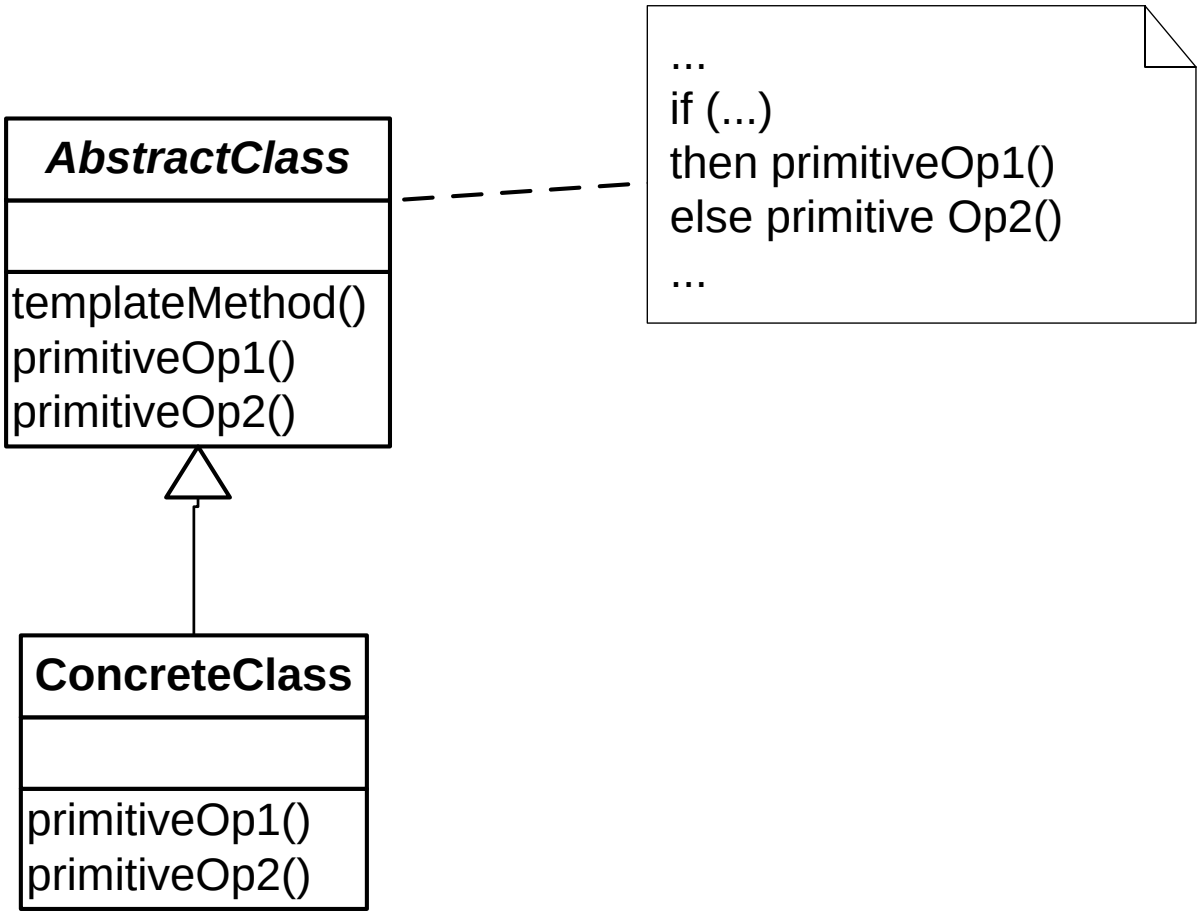


- Se la classe esistente non può essere specializzata (ovvero l'adapter è costretto a specializzare una classe astratta che definisce l'interfaccia), allora si realizza tramite un meccanismo di *class composition* e *forwarding*

Template Method / 1

- Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi dell'algoritmo alle sottoclassi
- Le sottoclassi ridefiniscono alcuni passi dell'algoritmo senza dover implementare di nuovo la struttura dell'algoritmo stesso
  - Operazioni *hook*
- Principio Hollywood: *don't call me, I'll call you*
  - La classe padre chiama le operazioni ridefinite nei figli e non viceversa

Template Method / 2



Dipartimento di Informatica
e Sistemistica Antonio Ruberti

Sapienza
Università di Roma

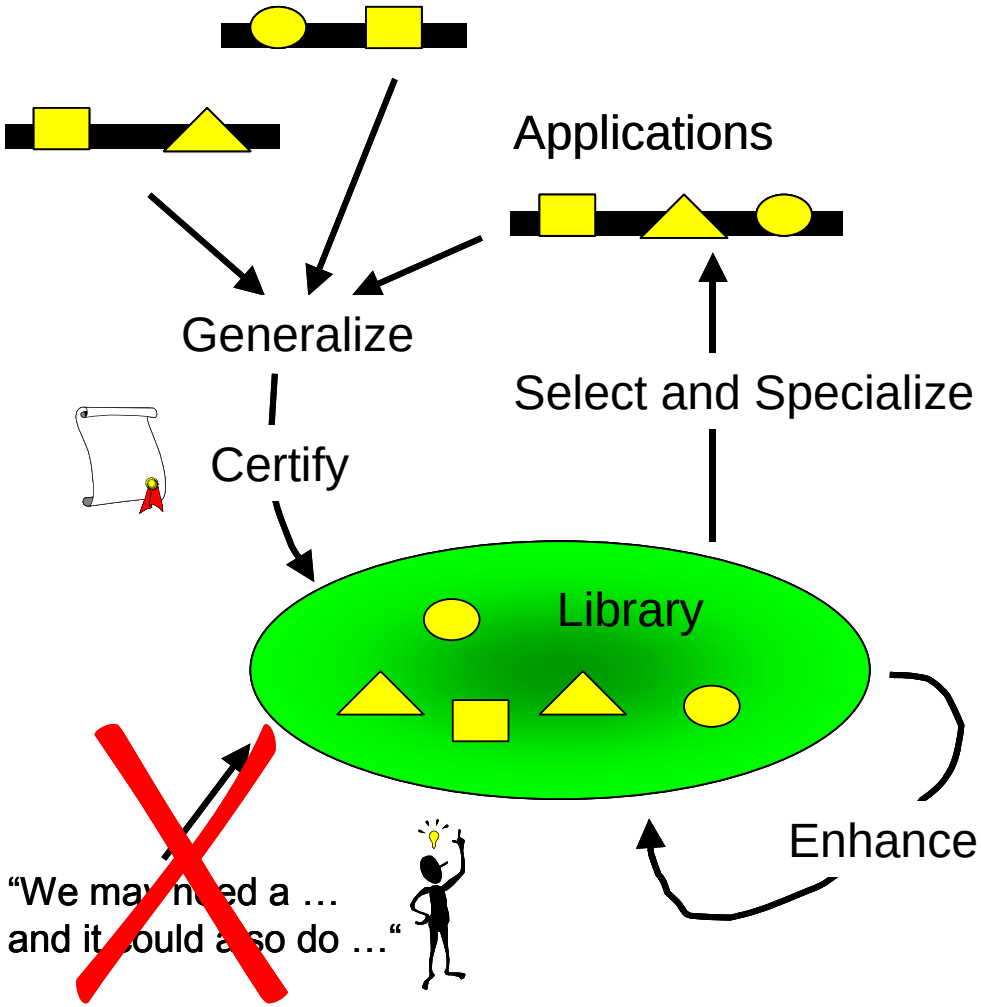
Riuso e le Relative Tecniche

Riuso (1)

- Il riuso *Cut&Paste* produce benefici solo sul 20% del lavoro
- Migliori risultati importando componenti generici (codice, sorgenti, design patterns, GUI *look&feel*, …)

Principio Open-Closed: *ogni componente deve essere aperto all'estensione ma chiuso alle modifiche*

Riuso (2)



Due cicli:

- *Product development*
- *Asset develpment*

Importanza della documentazione

Commitment

Riuso del codice (1)

Un componente riusabile deve:

- essere generico per catturare le cose in comune tra differenti contesti
- offrire meccanismi per essere specializzato

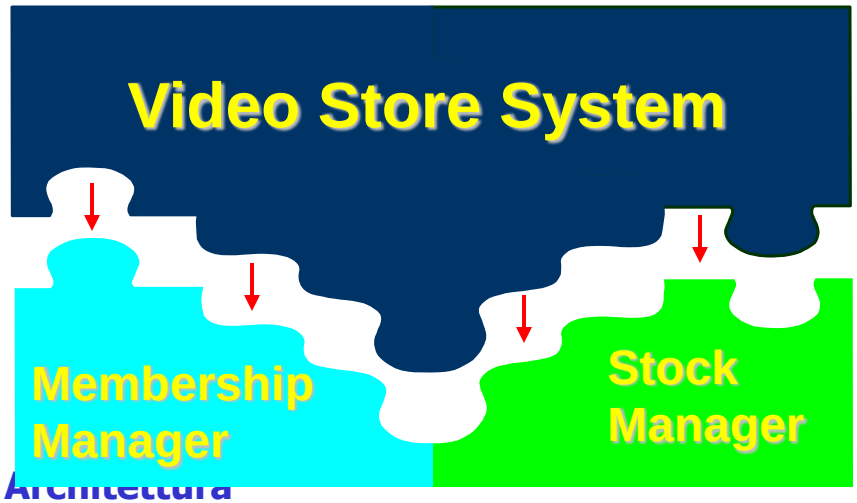
Plug-points

- per la composizione con altri componenti per costruire qualcosa di più grande
- per inserire parti che lo specializzano

Riuso del codice (2)

Upper Interfaces - "l'uso normale"

- connessione diretta e visibile di parti che forniscono servizi ben definiti
- Interfaccia pubblica di una classe, API di un DBMS, window manager, … e insieme dei servizi offerti da un componente



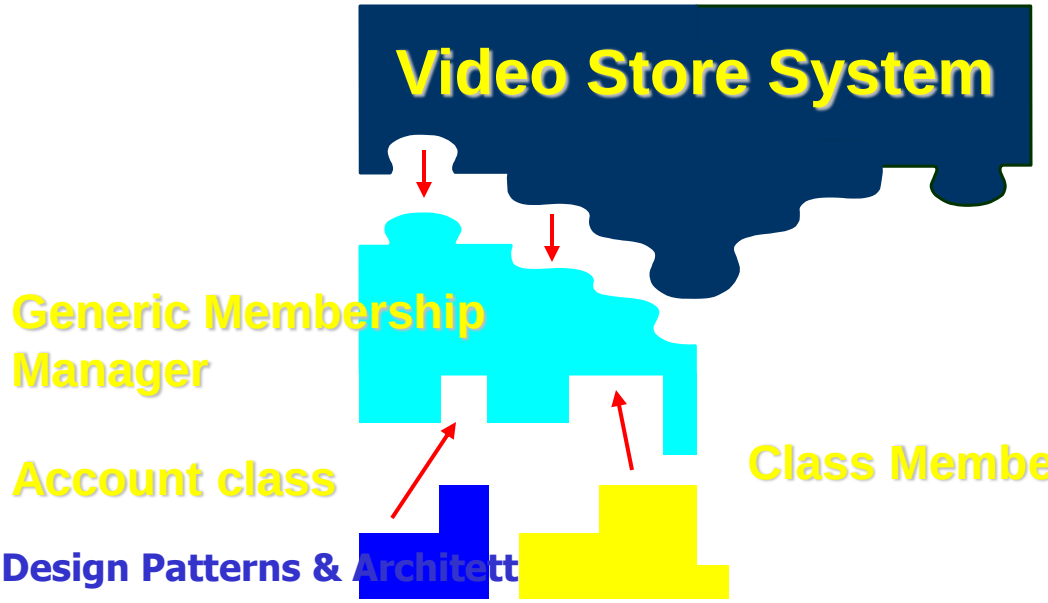
Per costruire questo componente … …

… … usiamo questi componenti

Riuso del codice (3)

Lower Interfaces - “customizzazione”

- plug-in per specializzare il comportamento di un componente
- Web browser, word processor e spreadsheet utilizzano tecnologie di linking dinamico per estendere le loro funzionalità



Per costruire questo componente … …

… … usiamo questo componente … …

… … provvedendo dei plug-in per customizzarlo

Riuso del codice (4)

Un OOP Framework è un insieme di classi astratte e concrete che collaborano per offrire lo scheletro dell'implementazione di un'applicazione

- E' adattabile, componendo ad-hoc sottoclassi selezionate, ovvero definendo nuove sottoclassi ed implementando metodi che fanno il *plug-in* o l'*override* delle superclassi
- Lo scheletro del comportamento comune è:
  - un metodo interno specificato su un interfaccia che deve essere implementata da una classe specializzata
  - un metodo template nella superclasse con parti varianti delegate alla sottoclasse

Riuso del codice (5)

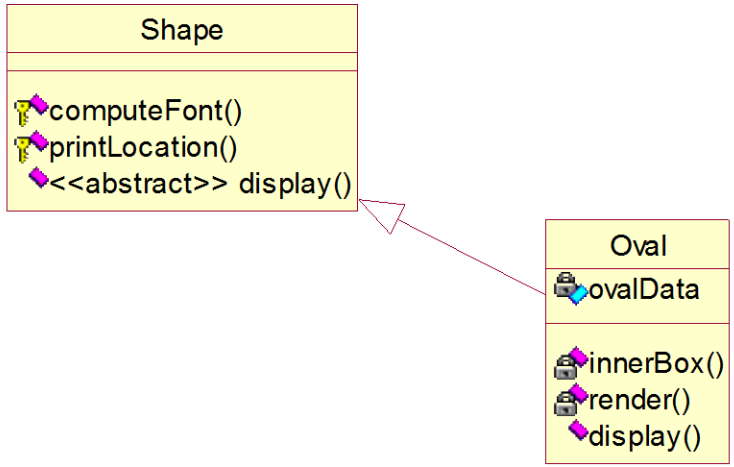
Esempio - *Progetto ed implementazione di un programma per manipolare forme. Differenti forme sono visualizzate differentemente. Quando una forma è visualizzata, essa mostra un rendering del suo bordo ed una stampa testuale della sua locazione nel font più grande che può essere contenuto nella forma*

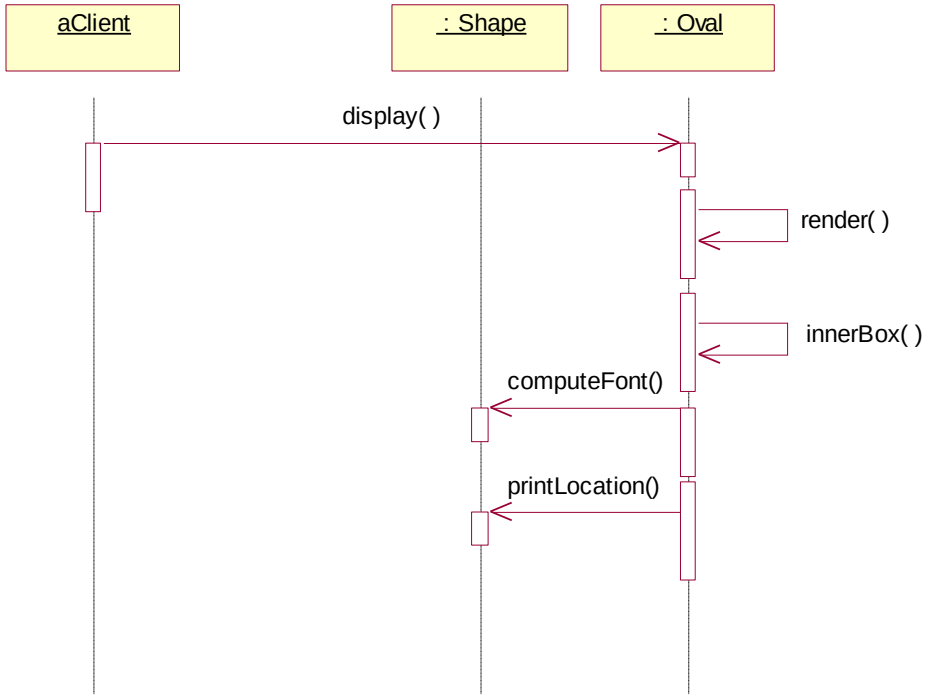
Class Library

(slide 36)

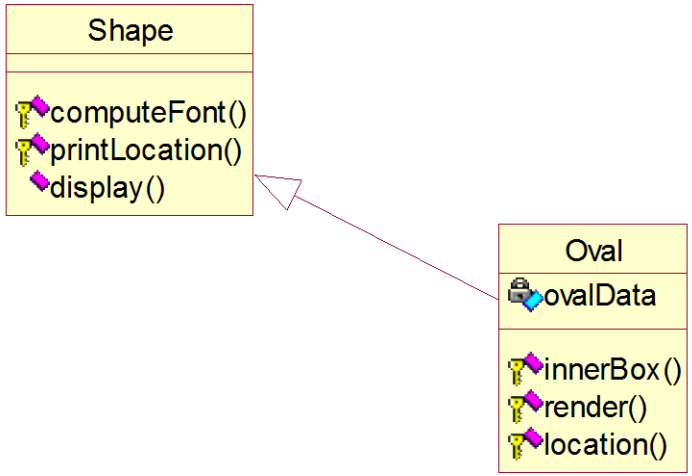
Framework style (template)

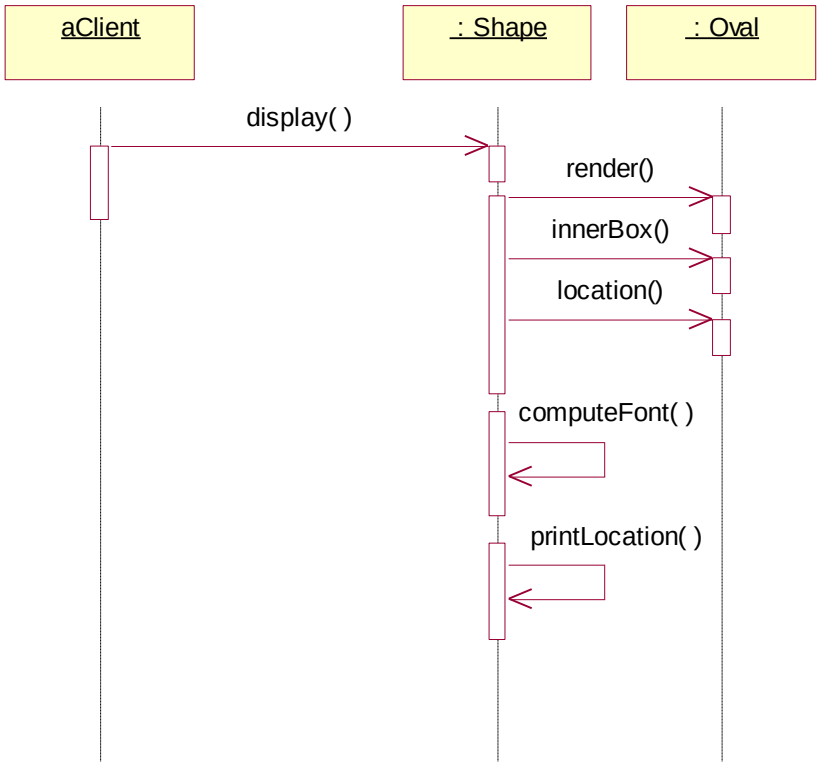
(slide 37)






```
Class Shape
{  // called from subclass
   protected Font computeFont
   (BoundingBox b, String s) {… …}
   protected void printLocation
   (GraphicsContext g, Font f,
   Point location) {… …}
   public abstract void display
   (GraphicsContext g);
}
Class Oval extends Shape
{  private LocationInfo ovalData;
   private BoundingBox innerBox() {… …}
   private void render (GraphicsContext g) {… …}
   public void display
   (GraphicsContext surface) {
      render();
      BoundingBox box = innerBox();
      Font font = super.computeFont(box,
         ovalData.location).asString());
      super.printLocation(surface, font,
         ovalData.location;
    }
}
```






```
Class Shape
{  public void display(GraphicsContext surface)
   {  // delegate to subclass to fill in the
      // pieces …
      render(surface);                  // plug-point
      BoundingBox box = innerBox(); // plug-point
      Point location = location();  // plug-point
      // … then do the rest based on those bits
      Font font = computeFont(box, location);
      surface.printLocation (location, font);
   }
}
Class Oval extends Shape
{  // implement 3 specific plug-ins for
   // the plug-points in Shape
   protected void render (GraphicsContext g) {… …}
   protected BoundingBox innerBox() {… …}
   protected Point location () {return center;}

   private Point center;
   private int majorAxis, minorAxis, angle;
}
```

Riuso del codice (6)

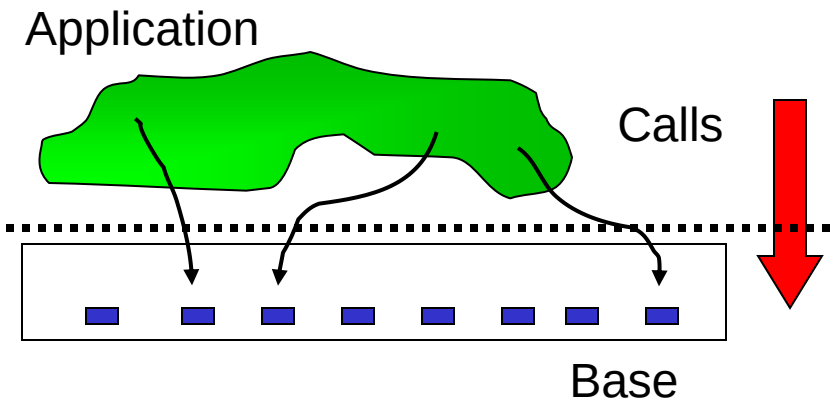
- I vari comportamenti sono differenti; si cerca di fattorizzare le parti comuni
- Condivisione delle operazioni *low-level*. Logica *high-level* duplicata
- Le chiamate vanno dall'applicazione alla base condivisa
- Definire un'interfaccia che l'applicazione può usare per chiamare le parti riusabili

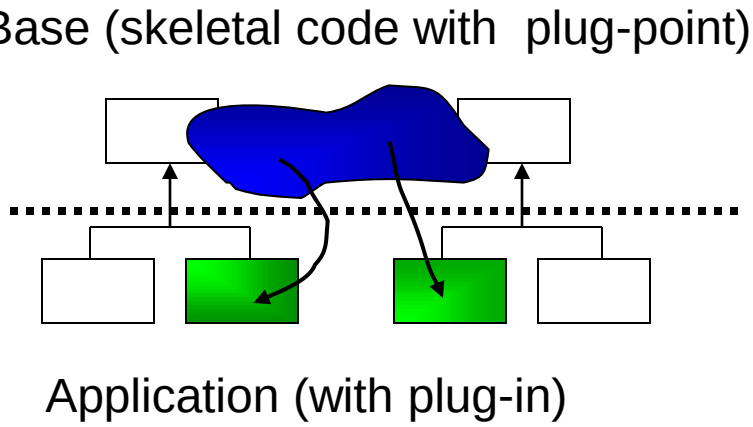
- I comportamenti sono essenzialmente gli stessi, identificare le differenze da delegare alle sottoclassi
- Condivisione dello skeleton applicativo; ogni applicazione *plug-in* le parti richieste per completarlo
- Le chiamate vanno dal framework alle applicazioni; *"Don't call me - I'll call you"*
- Definire un'interfaccia per i *plug-point* (chiamate dello skeleton riusabile alle applicazioni)

Riuso del codice (7)

- L'applicazione contiene il codice più recente; la base contiene codice più vecchio; il codice più recente chiama quello esistente
- L'applicazione implementa regole e policy (l'architettura)

- L'applicazioni contiene il codice più recente; la base (codice più vecchio) chiama il codice più recente
- Il framework implementa l'architettura ed impone regole e policy sull'applicazione





Dipartimento di Informatica
e Sistemistica Antonio Ruberti

Sapienza
Università di Roma

Concetto di Architettura e sue Qualità

Architettura / Architetture

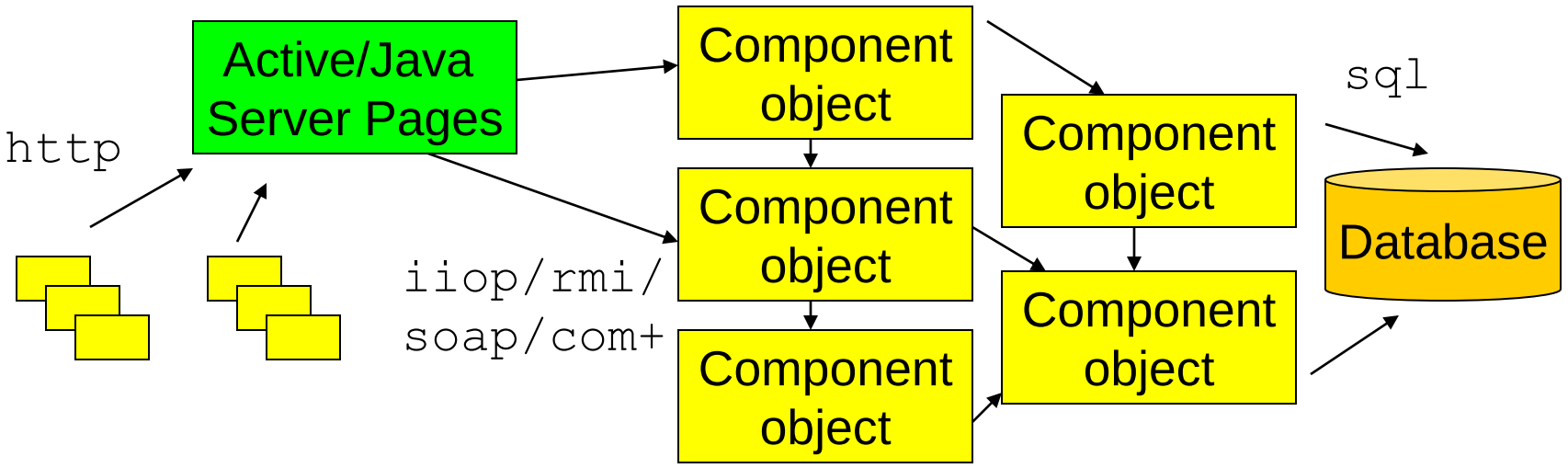
Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma

- Architettura è oramai una *"buzzword"*
  - **system architecture**: *"the structure of the pieces that make up a complete sw installation, including the responsibilities of these pieces, their interconnection, and possibly the appropriate technology"*
  - **component architecture**: *"a set of application-level components*, their structural relationships, and their behavioral dependecies"*

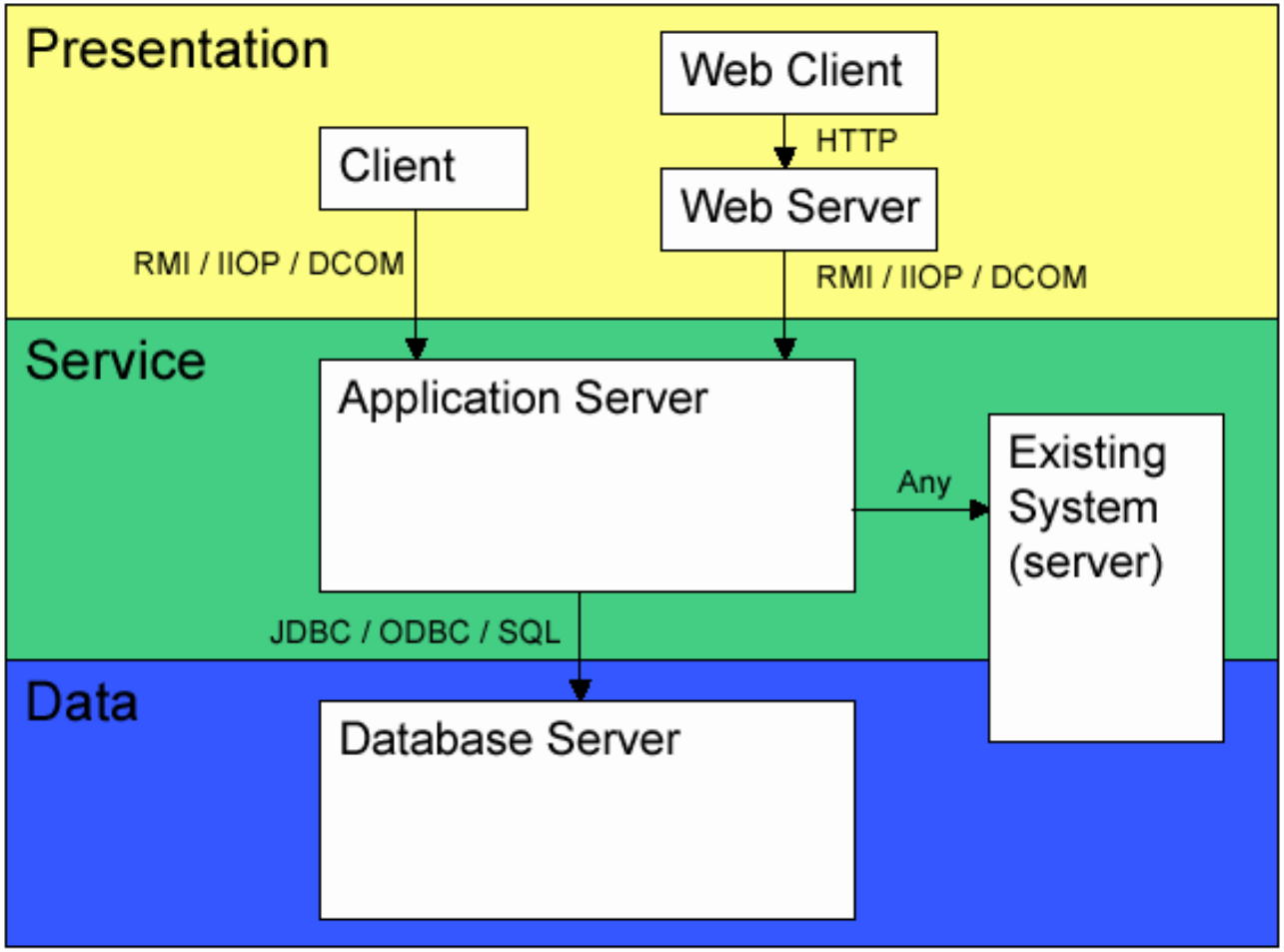
(*) **component/componente** non indica qui una particolare tecnologia, ma un modulo software con obiettivi ben definiti ed omogenei, relazioni strutturali con altri moduli/componenti, offre un insieme ben definito di servizi agli altri componenti (*server*) e può utilizzare servizi di altri componenti (*client*)

System Architecture (Architettura di Sistema)

- Dà la forma finale del sistema installato, e come varie tecnologie vengono assemblate per formarlo
  - Si considereranno prevalentemente N-tier distributed architecture



System Architecture
(Architettura di Sistema)



System Architecture (Architettura di Sistema)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Identificare differenti livelli nei quali i componenti vengono utilizzati
- User Interface
  - la presentazione dell'informazione agli utenti (anche altri sistemi) a la cattura dei loro input
  - crea cosa l'utente vede; tratta la UI logic
- User Dialog
  - management dello user's dialog in una sessione
  - stato transiente corrisponde al dialogo

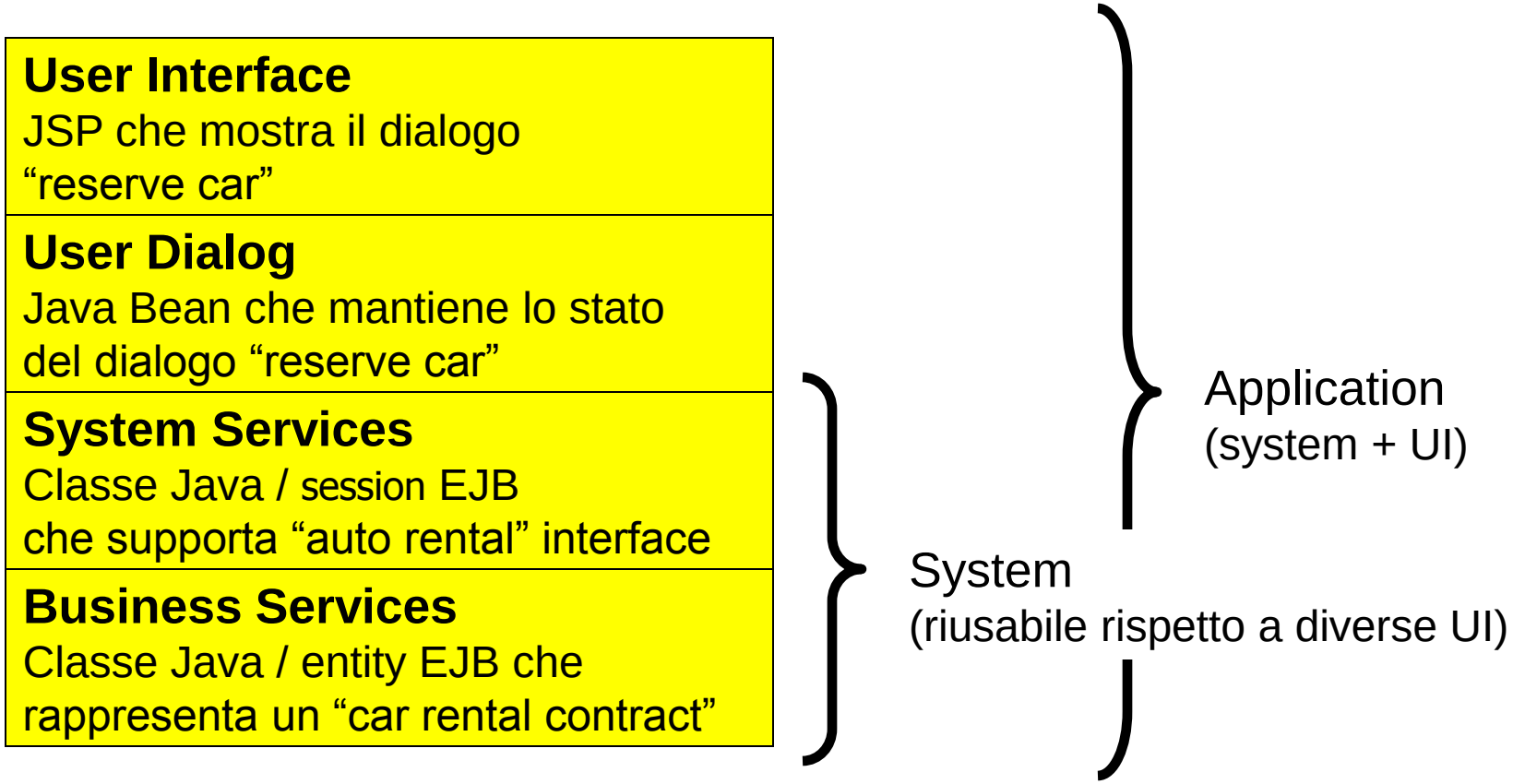
System Architecture (Architettura di Sistema)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- System Services (Servizi di Sistema)
  - la rappresentazione esterna del sistema, che fornisce accesso ai suoi servizi
  - é una *facade* per il layer sottostante, fornendo un contesto nel quale i business service più generali sono utilizzati per un particolare sistema
  - nessun stato dialog- o client-related
  - operazioni = nuove transazioni
- Business Services (Servizi di Business)
  - l'implementazione delle core business information, regole e trasformazioni
  - operazioni possono essere combinate con altre in transazioni
  - tipicamente database associato

System Architecture
(Architettura di Sistema)

Dipartimento di Informatica
e Sistemistica Antonio Ruberti
Sapienza
Università di Roma



Component Architecture (Architettura a Componenti)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

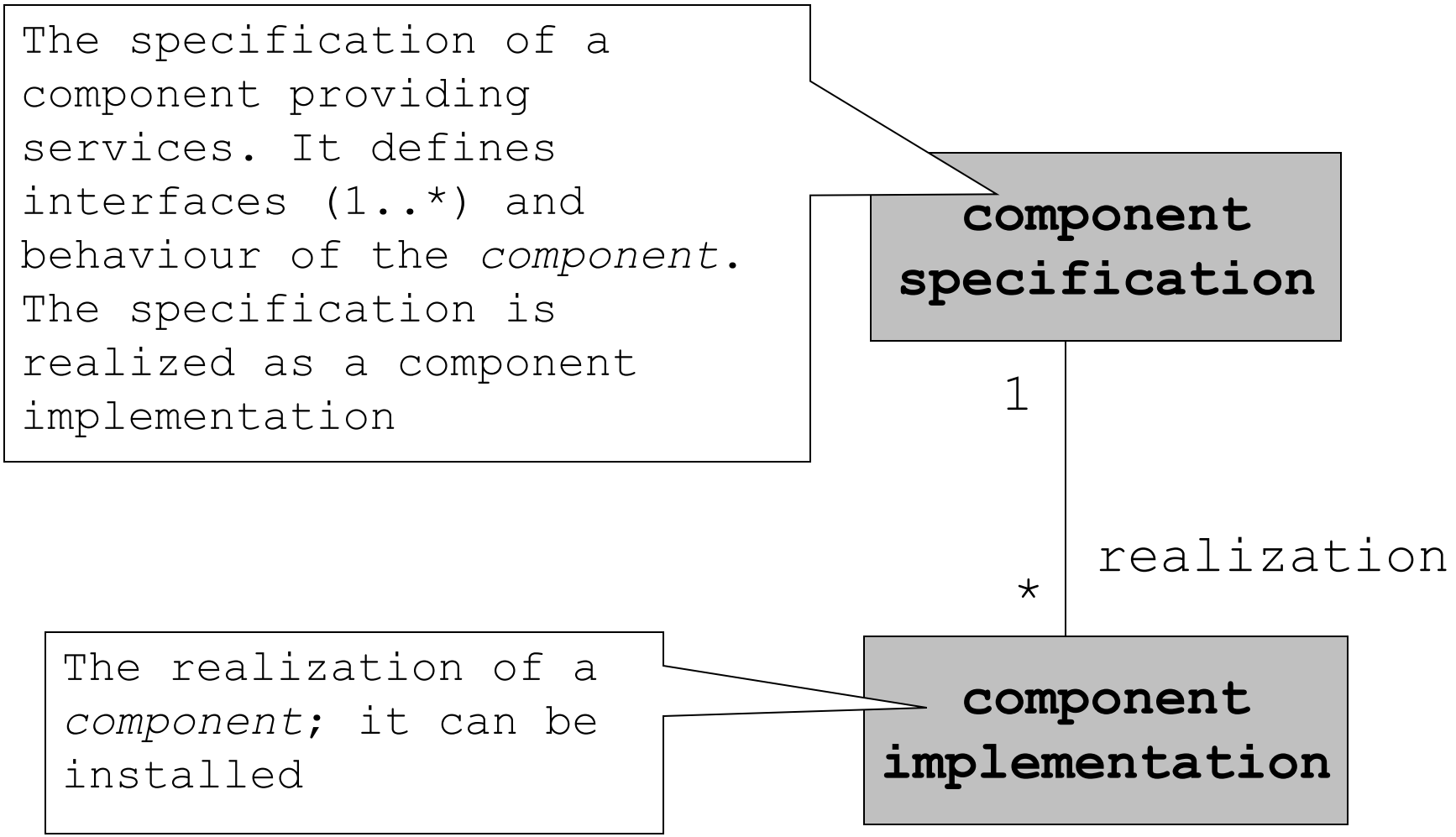
- Nella component architecture ci si concentra sui component, le loro relazioni strutturali e dipendenze comportamentali
  - Relazioni strutturali: associazioni e inheritance tra specifications e interfaces, e composizione tra components
  - Dipendenze comportamentali: dipendenze tra components, tra components e interfacce, tra interfacce stesse

Component Architecture (Architettura a Componenti)

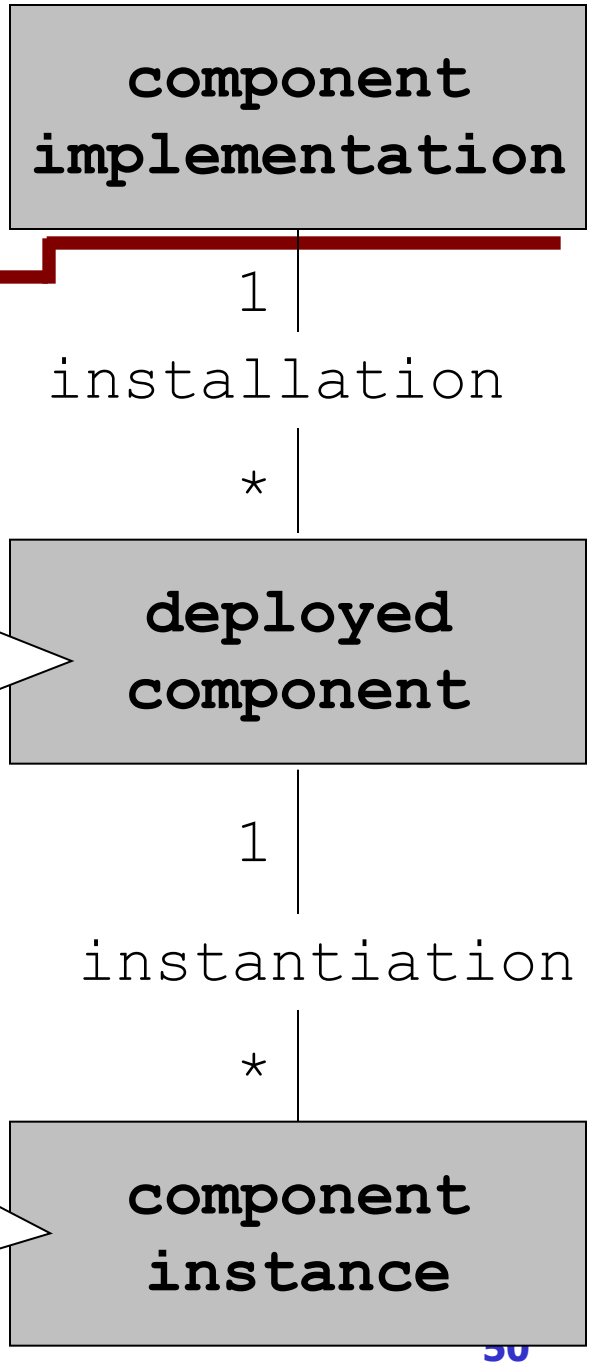
Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Component Specification Architecture
  - *"any dependency emanating from a CSA is part of the definition of that component specification and must be adhered to by all implementations"*
- Component Implementation Architecture
  - Dipendenze che esistono tra particolari implementazioni; in genere l'implementazione aggiunge vincoli ulteriori a quelli della specification

Forme


The specification of a component providing services. It defines interfaces (1..*) and behaviour of the *component*. The specification is realized as a component implementation
component specification
1
realization
*
The realization of a *component*; it can be installed
component implementation

Forme



An installed copy of a *component* implementation; it is deployed by registering it with the environment, thus enabling the environment to identify it to use when creating an instance

A run-time concept: an "object" with its own state and a unique identity, the "thing" that performs the implemented bahavior. A deployed *component* may have multiple instances

Incapsulamento

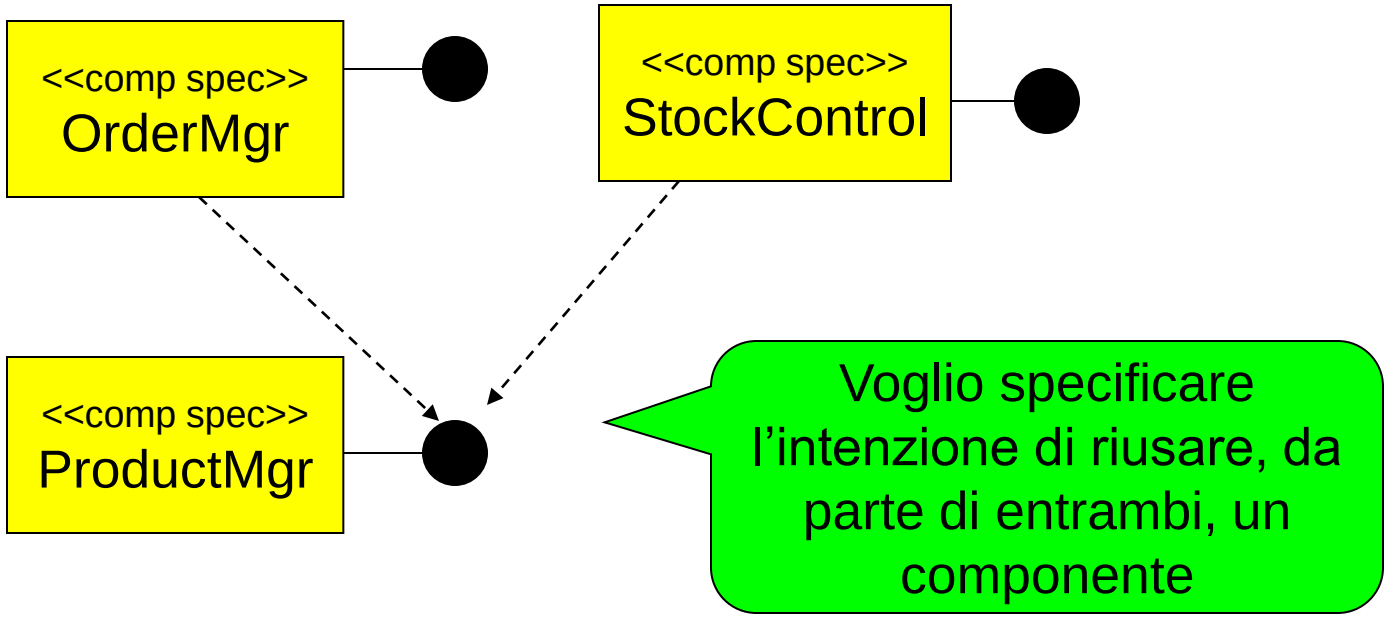
- La specifica di un componente deve descrivere le interfacce che un utilizzatore può sfruttare per accedere al componente
- Le interfacce dovrebbero essere il più possibile separate dall'implementazione
  - Autocontenute
  - Indipendenti da un particolare linguaggio di programmazione
- Le interfacce dovrebbero essere bidirezionali
  - Descrivere anche i servizi *attesi* dal componente

Composizione

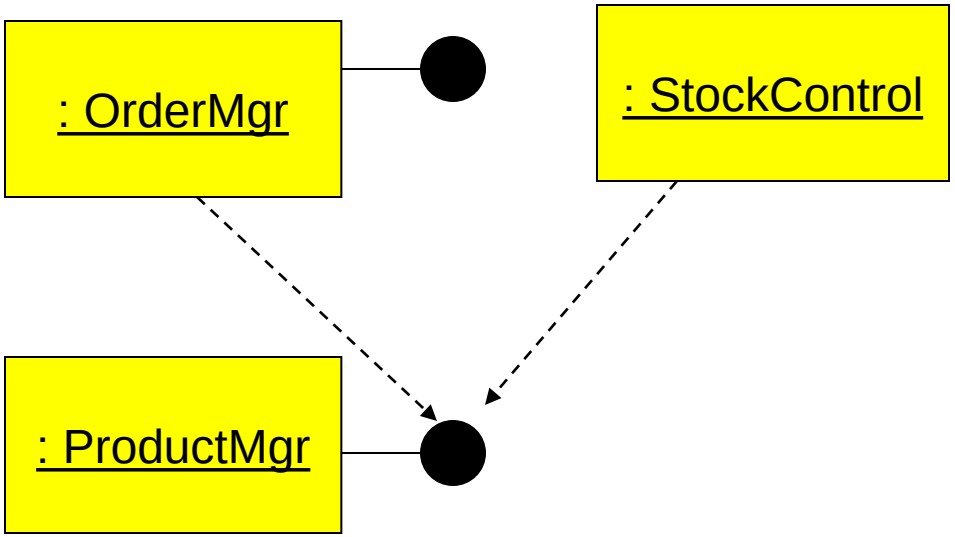
- Per realizzare il collegamento tra componenti è necessario fornire dei meccanismi per:
  - selezionare i componenti da utilizzare
  - definire la tipologia di interazione
- Questi meccanismi sono generalmente forniti dagli standard infrastrutturali

Component Architecture (Architettura a Componenti)

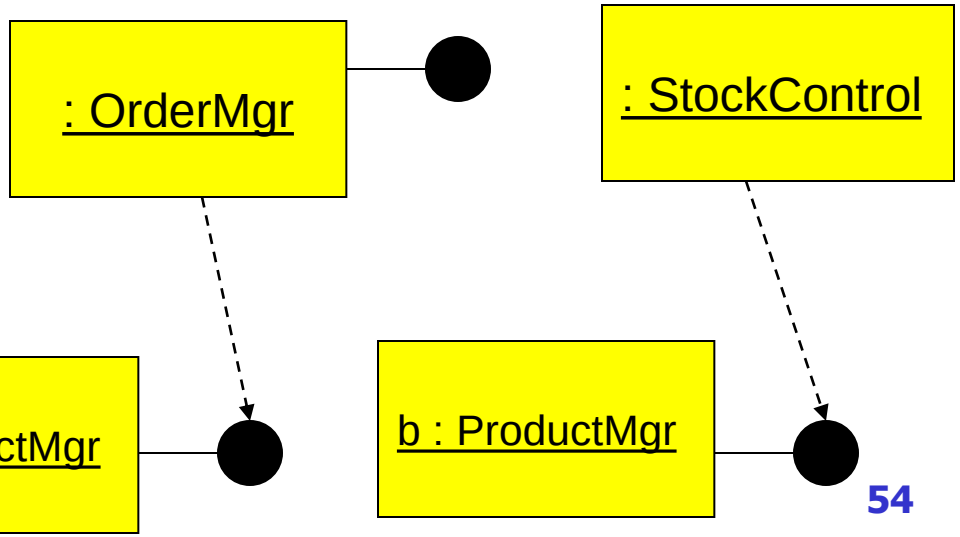
- Component Object Architecture
  - Utile soprattutto quando faccio riuso di components



Component Architecture
(Architettura a Componenti)



Quale delle due
è quella
desiderata ?



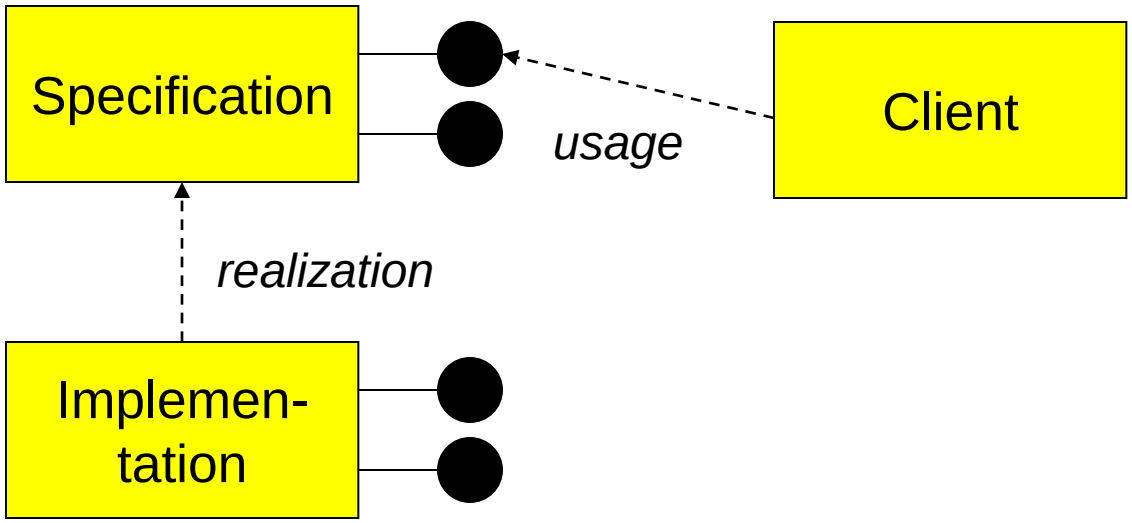
Design by Contract (Progettazione per Contratto)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

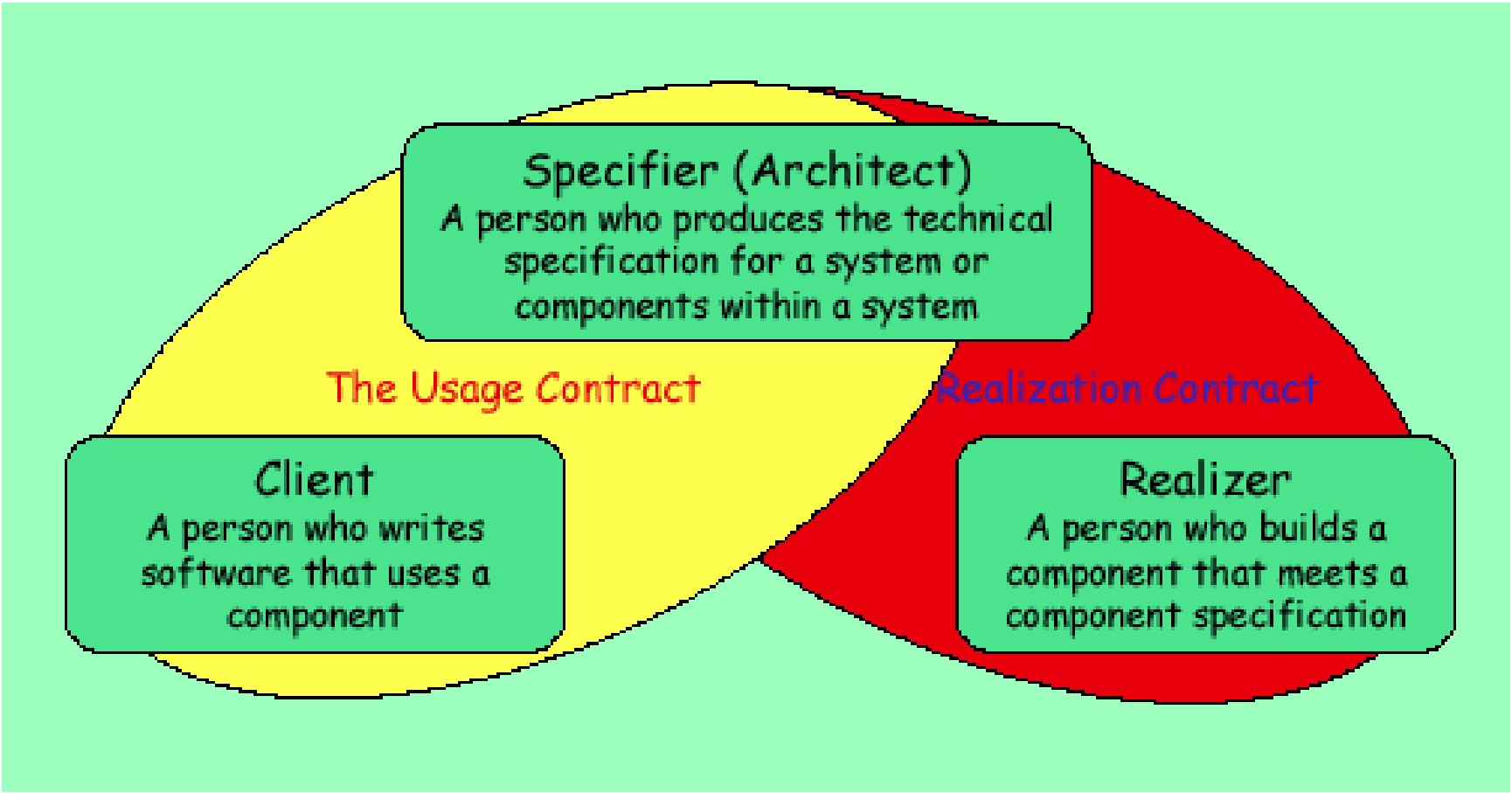
- (nel mondo reale) un contratto descrive (specifica) i dettagli di un accordo (per la fornitura di un servizio) in modo non ambiguo
  - Responsabilità delle parti: “cosa ogni parte fa ammesso che le altri parti facciano cosa hanno detto che faranno”
  - Asserisce anche cosa avviene se le parti non fanno quello che si sono impegnate a fare (falliscono nel fornire il servizio)

Design by Contract (Progettazione per Contratto)

- Nel software design
  - ***Usage contract***: contratto tra l'interfaccia di un component ed i suoi client
  - ***Realization contract***: contratto tra un component specification e le sue implementation



Design by Contract
(Progettazione per Contratto)



Usage Contract (Contratto d'Utilizzo)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Descrive la relazione tra un component object ed i suoi client
  - *run-time* contract
- E' specificato come un *interfaccia*
  - *Operazioni*: lista delle operazioni fornite, complete di segnatura e definizioni
  - *Information model*: definizione astratta di qualsiasi informazione/stato che viene mantenuto tra le richieste del client un oggetto che supporta l'interfaccia, e vincoli su quell'informazione

Usage Contract (Contratto d'Utilizzo)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Un'operazione è un mini-contratto a sua volta
  - (parametri) input / output
  - precondizione (situazione in cui è possibile richiedere l'operazione), postcondizione (effetto dell'operazione sui parametri e l'information model)

In linea teorica il client assicura il soddisfacimento della precondizione ... Se la precondizione non è vera, il risultato dell'operazione è indefinito

Operazioni aggregate in interfacce per motivi pragmatici: in genere il loro uso è omogeneo e correlato

Realization Contract (Contratto di Realizzazione)

Dipartimento di Informatica e Sistemistica Antonio Ruberti
Sapienza Università di Roma

- Descrive come un implementatore deve realizzare una component implementation
  - *design-time* contract
- E' specificato attraverso
  - L'insieme di interfacce (comportamento complessivo di ogni oggetto)
  - Come un'implementazione interagisce con altri componenti
    - Collaborazioni e vincoli sugli information model
  - Come interfacce si corrispondono

Confronto

Interface

- Lista di operazioni
- Sottostante information model
- Contratto con il client
- Specifica come le operazioni agiscono sull'information model
- Effetti locali

Specification

- Lista di interfacce supportate
- Relazioni tra gli information model di differenti interfacce
- Contratto con l'implementatore
- Definisce l'unità di implementazione/run-time
- Specifica come le operazioni vengono realizzate in termini di uso di altre interfacce

Qualità Architetturali

- Per contribuire in maniera adeguata alla soluzione del problema applicativo, l'architettura deve possedere delle qualità
- In base alla situazione potrebbe essere necessario privilegiare delle caratteristiche di qualità a scapito di altre
- E' fondamentale identificare quali caratteristiche sono prioritarie, per guidare le decisioni di progetto sin dalle fasi iniziali

Qualità architetturali

- Performance
- Scalabilità
- Sicurezza
- Disponibilità
- Affidabilità

efficienza

robustezza

Qualità architetturali

- **Performance**: il sistema deve reagire adeguatamente rispetto al carico di lavoro atteso
- **Scalabilità**: capacità del sistema di adattarsi a situazioni di carico superiori a quelle previste inizialmente
  - quantità dati, numero utenti, frequenza richieste

Qualità architetturali

- **Sicurezza**: il sistema deve prevenire accessi non autorizzati o utilizzi non previsti
- Meccanismi:
  - autenticazione: assicurare che l'identità della sorgente di una richiesta sia effettivamente quella dichiarata
  - autorizzazione: assicurare che la sorgente di una richiesta, una volta autenticata, sia autorizzata ad eseguire l'operazione
- La sicurezza può essere dichiarata a diversi livelli di criticità all'interno del sistema, corrispondenti a diversi meccanismi di implementazione

Qualità architetturali

- **Disponibilità**: percentuale del tempo di esecuzione in cui il sistema appare come funzionante rispetto agli utenti
- **Affidabilità**: capacità del sistema di reagire autonomamente a situazioni di guasto
  - Misurano la tolleranza del sistema a situazioni di guasto

Qualità architetturali

- Ogni qualità architetturale si esprime in una serie di requisiti che investono il sistema a diversi livelli
  - Hardware
  - Progetto della rete
  - Progetto del software
  - Infrastruttura middleware
- Nel progetto dell'architettura sarà necessario identificare se un requisito dovrà richiedere il progetto di un componente opportuno oppure se è possibile sfruttare l'infrastruttura middleware

Disponibilità

- Fondamentale in applicazioni mission-critical, che devono fornire un servizio continuativo
  - es. applicazioni web di e-commerce
- In questi casi i guasti del sistema potrebbero portare problemi e perdite i cui effetti si possono protrarre anche a lungo termine
  - es. perdita clienti

Disponibilità: linee guida

- Progettare per aumentare la disponibilità significa introdurre tecniche e linee guida metodologiche per anticipare, rilevare e risolvere automaticamente i guasti hardware e software
- L'obiettivo è ridurre il downtime
  - non solo ridurre le possibilità di guasto, ma anche i tempi di riparazione
- Tecniche:
  - replicazione hardware e software
- Metodologie:
  - utilizzo di processi di progetto e sviluppo collaudati e con forte accento sul testing

Disponibilità: linee guida

Utilizzare tecniche di clustering

- Cluster: insieme di sistemi che offrono lo stesso servizio ma sono distribuiti su nodi fisici diversi, pur essendo logicamente e fisicamente connessi tra loro
- Diversi sistemi indipendenti in uno stesso cluster appaiono all'esterno come un sistema singolo
- In presenza di guasti, il carico di lavoro viene automaticamente spostato da un sistema all'altro
  - i client vengono spostati automaticamente (*failover*)
  - il guasto è trasparente: viene percepito semplicemente un ritardo
- Il clustering permette anche di aumentare la scalabilità dell'applicazione

Disponibilità: linee guida

Clustering (cont.)

- Il clustering permette di
  - ridurre il downtime dovuto a guasti
  - ridurre il downtime dovuto a manutenzione
  - scalare l'applicazione in maniera economica

Disponibilità: linee guida

Clustering (cont.)

- Implementare soluzioni di clustering non è semplice, a causa dei meccanismi necessari
  - redirezione automatica delle richieste
  - mantenimento della consistenza
- Esistono soluzioni già pronte
  - Molti server J2EE
  - Microsoft Cluster Server

Disponibilità: linee guida

Prevedere meccanismi di monitoring

- Per tollerare i guasti è prima di tutto necessario rilevare i guasti
  - Il rilevamento di guasti di componenti si basa generalmente su *timeout*
- E' possibile utilizzare due modalità di rilevamento
  - On-line: rilevamento durante le esecuzioni di richieste
  - Off-line: monitoring periodico a prescindere dalle richieste
- Monitoring significa inoltre utilizzare dei log per controllare l'andamento delle risorse e individuare i punti soggetti a guasti

Disponibilità: linee guida

Isolare applicazioni mission-critical

- E' possibile che più applicazioni condividano risorse hardware e software
- Nel caso di applicazioni mission-critical, è opportuno ridurre il più possibile il contatto con altre applicazioni, sia con tecniche hardware che software
  - Reti e DBMS dedicati
  - Middleware ad alte prestazioni

Disponibilità: linee guida

Prevedere un piano di test e di recupero adeguati

- Il test per la disponibilità deve essere il più possibile rigoroso e completo
- Lo scopo è provare le procedure di recupero in tutte le situazioni di guasto più estreme …
  - Staccare la spina o il cavo di rete
  - Togliere l'alimentazione generale

Disponibilità: linee guida

Piano di recupero (cont.)

- Il piano di recupero deve fornire istruzioni dettagliate sulle operazioni da eseguire in circostanze critiche
  - Operazioni richieste su ogni server per ripristinare l'operatività
  - Inventario delle parti
  - Materiale cartaceo

Affidabilità

- L'architettura deve essere progettata in modo da limitare le possibilità di malfunzionamento dell'applicazione
- Non confondere con disponibilità
  - capacità di recuperare dai guasti
- Anche in questo caso investe diversi aspetti del progetto:
  - tecnologia: hardware e software "robusti"
  - metodologia di progetto: test accurati e completi

Affidabilità: linee guida

Gestire gli errori all'interno dell'applicazione

- Una architettura affidabile è in grado di prevenire e adeguarsi alle situazioni di errore automaticamente
- E' necessario in fase di test prevedere tutte le possibili situazioni di errore a tutti i livelli e scrivere del codice di gestione adeguato
  - errori nel middleware
  - errori nel DBMS

Affidabilità: linee guida

Gestire con cura i cambiamenti

- Aggiungere o modificare un componente può essere causa di guasti nell'architettura
  - errori del nuovo componente
  - errori di integrazione
- I cambiamenti devono essere coordinati e controllati

Performance

- Dal punto di vista dell'utente, le performance vengono percepite come i tempi di risposta alle richieste
- Questo viene determinato da diversi aspetti dell'architettura
  - hardware
  - piattaforme middleware
  - basi di dati
  - progetto software
- Non confondere con la scalabilità
  - performance a carichi crescenti

Performance: linee guida

- Si possono identificare all'interno dell'architettura delle operazioni o dei pattern inevitabilmente costosi
- E' necessario progettare l'architettura in modo da identificare i possibili colli di bottiglia e limitarne l'impatto
- Esempi
  - mancanza di concorrenza
  - sincronizzazione tra componenti remoti
  - accesso frequente al DB
  - apertura connessioni
  - creazione thread

Performance: linee guida

Utilizzare tecniche di pooling

- Pooling significa attivare un insieme di risorse dello stesso tipo (pool) prima delle effettive richieste di utilizzo
- Quando un utilizzatore richiede una risorsa, viene reperita una istanza già attiva dal pool
- Tecniche di pooling permettono di accelerare l'accesso a risorse il cui tempo di attivazione diventa non trascurabile su applicazioni di larga scala
  - thread
  - componenti
  - connessioni a DB

Performance: linee guida

Pooling (cont.)

- Pooling di thread
  - spesso utilizzato per accelerare l'esecuzione di pagine web dinamiche
- Pooling di connessioni
  - riduce notevolmente i tempi di accesso al DB
- Pooling di componenti
  - dipende dalla tipologia di stato del componente
  - Se un componente contiene dello stato questo deve essere inizializzato al momento dell'attivazione e rilasciato

Performance: linee guida

Prevedere meccanismi di caching

- Caching: replicazione di dati o servizi in prossimità del loro utilizzo
- Il caching può essere applicato in diversi casi nell'architettura
  - pagine web dinamiche
  - dati
- Nel realizzare una soluzione di caching è necessario gestire le problematiche di consistenza tipiche di questo genere di problemi

Performance: linee guida

Caching (cont.)

- Nel caso in cui le operazioni sui dati sono prevalentemente letture, il caching dei dati può ridurre notevolmente il numero di accessi al DBMS
- Soluzioni per il caching dati
  - locale: risparmia la latenza di rete memorizzando i dati su una replica del DBMS in locale
  - in RAM: massima velocità di accesso ma necessaria una tecnologia specifica per la memorizzazione e l'interrogazione

Performance: linee guida

Ottimizzare gli accessi remoti

- Le invocazioni remote via middleware vanno utilizzate quando strettamente necessario
  - introducono un overhead sulle prestazioni dovuto alla sola connessione
- Inoltre, è necessario progettare i componenti di business in modo da limitare/ottimizzare gli accessi
  - più parametri nella stessa invocazione
  - …

Performance: linee guida

Ottimizzare le transazioni

- Le transazioni distribuite introducono un overhead notevole e vanno usate solo quando necessario

Ottimizzare i meccanismi di sicurezza

- Una connessione sicura (e.g. SSH) è notevolmente più costosa di una “in chiaro”
- Gli accessi sicuri non vanno necessariamente considerati nell’intera applicazione
- E’ possibile strutturare l’architettura in modo da inserire protezioni hardware, più efficienti

Scalabilità

- In una architettura scalabile, l'incremento di risorse comporta un aumento idealmente lineare nella capacità del servizio
  - Carico addizionale viene gestito aggiungendo risorse senza modifiche sostanziali al progetto
- Anche se la performance è parte della scalabilità, non è tutto
- La scalabilità è una parte integrante del progetto che va considerata a partire dalle prime fasi
- Ottenuta attraverso un bilanciamento accurato di elementi hardware e software

Scalabilità: linee guida

Utilizzare tecniche di clustering

- Il bilanciamento del carico su più macchine permette di ottenere risultati migliori che aumentando le risorse di una singola macchina e spesso è più economico
- L'applicazione deve essere progettata in maniera indipendente dalla locazione
- Inoltre sono necessari meccanismi per la gestione del cluster

Scalabilità: linee guida

Limitare le attese

- Operazioni molto costose rischiano di creare lunghe code di attesa quando ci sono molti processi richiedenti
- E' opportuno in questi casi sfruttare il più possibile modalità asincrone di interazione
  - Es. validazione carta di credito
- Questo principio, in casi estremi, potrebbe portare alla revisione dell'intero processo di business

Scalabilità: linee guida

Limitare i conflitti per le risorse

- L'utilizzo concorrente di risorse limitate è una delle principali cause di mancanza di scalabilità
  - Applicato a dischi, memoria, CPU, rete…
- *Principio*: acquisire le risorse il più tardi possibile e rilasciarle il prima possibile
- *Principio*: utilizzare "dopo" le risorse critiche
  - Se la transazione viene abortita prima ne ho limitato l'utilizzo

Scalabilità: linee guida

Curare il progetto dei componenti

- Limitare componenti stateful
  - Rispetto agli stateless, consumano più risorse e non sono condivisibili
- Separare i metodi transazionali da quelli non transazionali

Sicurezza

- Si ottiene attraverso il controllo degli accessi applicato a vari tipi di risorse dell'applicazione
  - Dati, componenti, hardware
- La sicurezza ruota intorno a quattro concetti principali
  - Autenticazione
  - Autorizzazione
  - Protezione dati
  - Auditing
- Viene affrontata in dettaglio in altri corsi