



# SOFTWARE DESIGN PATTERNS (PARTE 2)

Ingegneria del Software a.a. 2023-2024

# SPAZIO DEI DESIGN PATTERN

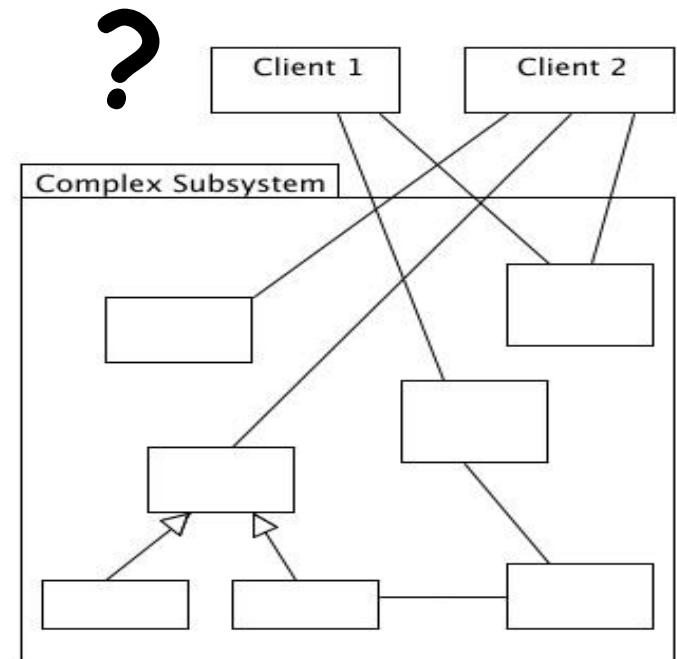
<i>Creazione</i>	<i>Struttura</i>	<i>Comportamento</i>
<b>Factory Method</b> <b>Abstract Factory</b> Builder Prototype Singleton	<b>Adapter</b> Bridge Composite Decorator <b>Facade</b> Flyweight Proxy	Interpreter <b>Template Method</b> Chain of responsibility Command Iterator Mediator Memento <b>Observer</b> <b>State</b> Strategy Visitor

Anche cenni di 'Model View Controller'

# FAÇADE (FACCIATA)

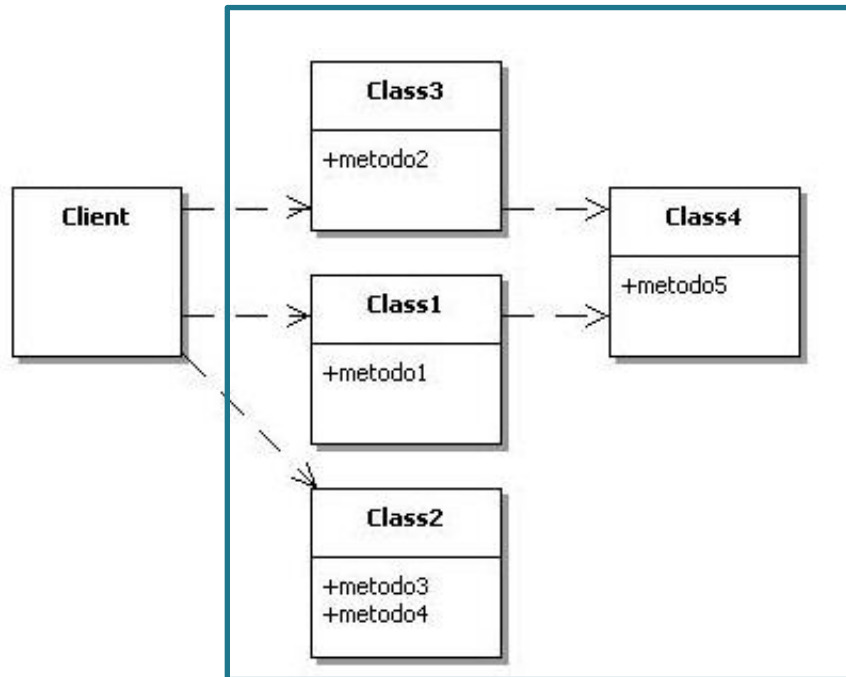
## ○ Problema

- Rendere **più semplice** l'accesso a sottosistemi che **espongono** interfacce complesse
- Fornire **un'unica interfaccia** per un insieme di funzionalità “sparse” su più interfacce/classi



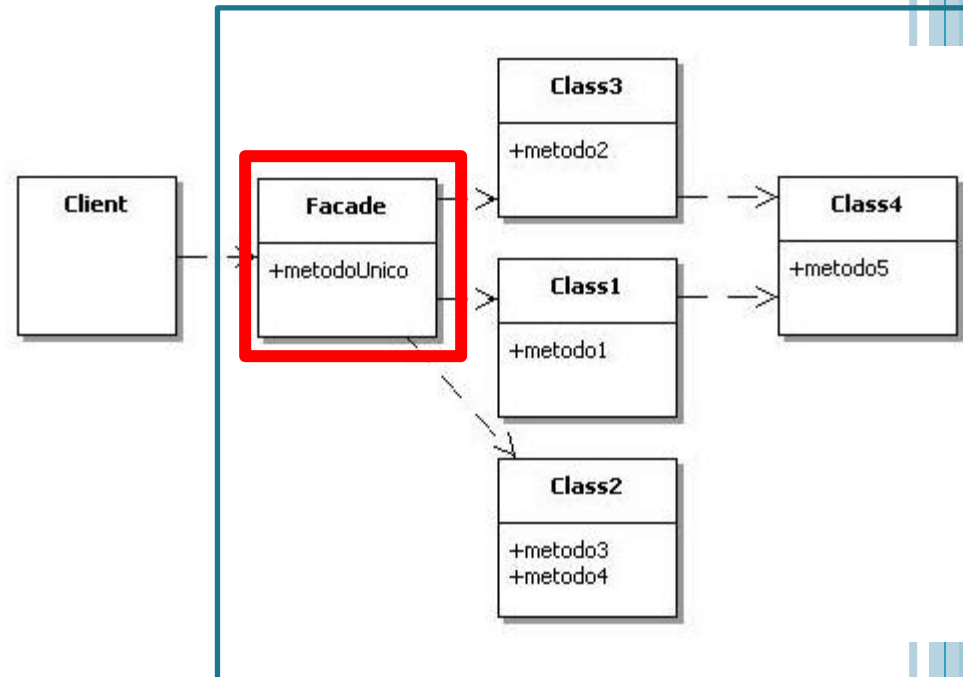
# ESEMPIO

Sottosistema



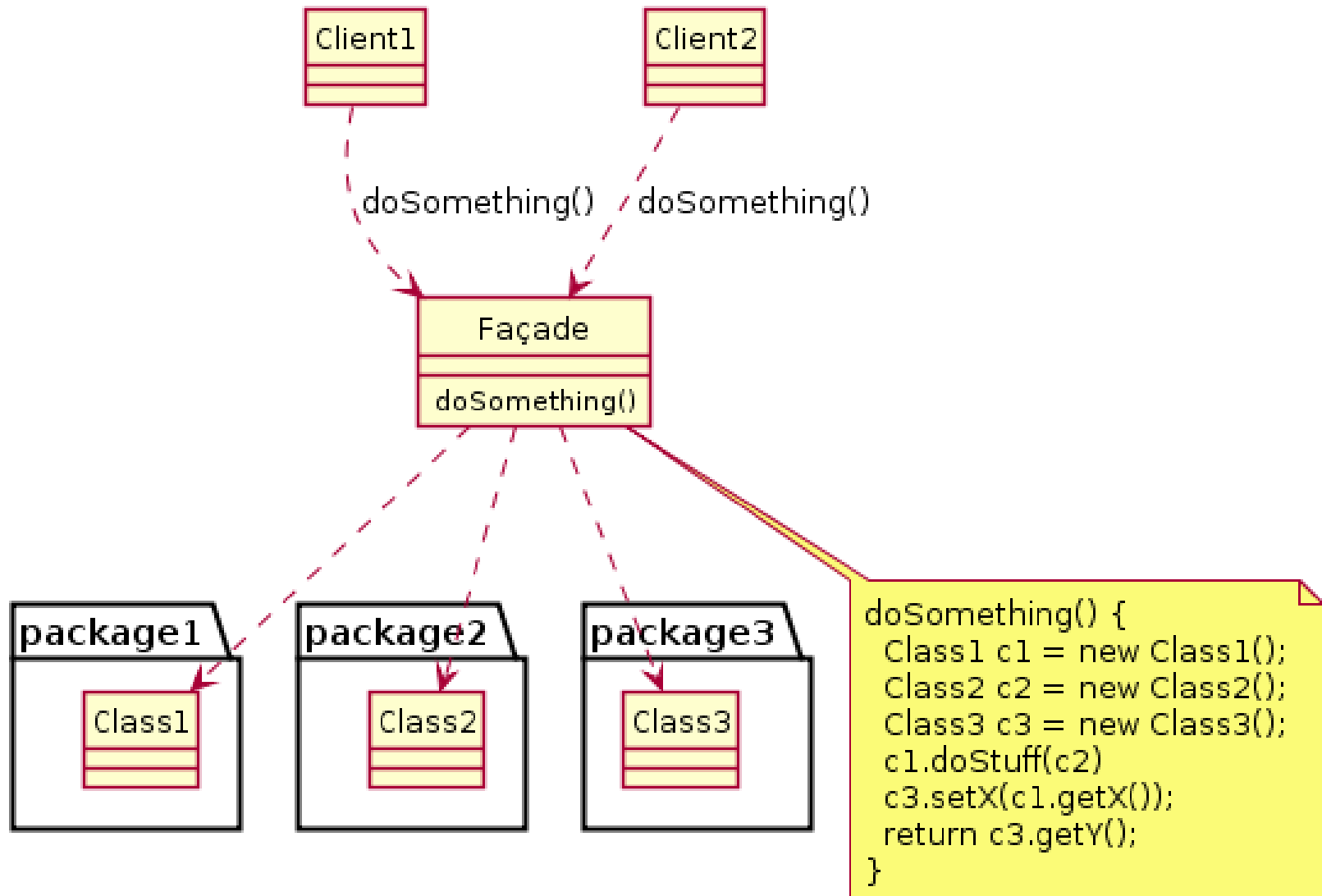
Un client per eseguire un task deve richiamare diversi metodi sparsi in classi diverse

Sottosistema



Facade nasconde la complessità poiché Client chiama solo *metodoUnico* per realizzare lo stesso task

# ALTRO ESEMPIO



# ESEMPI DI UTILIZZO

## ○ Compilatore

- Classi: LexicalAnalyzer, SyntaxAnalyzer, SemanticAnalyzer, CodeGenerator, ecc.

vs.

- Classe **Compiler** con metodo compile()

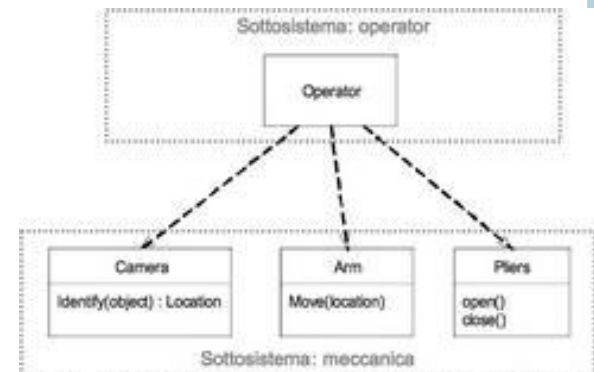
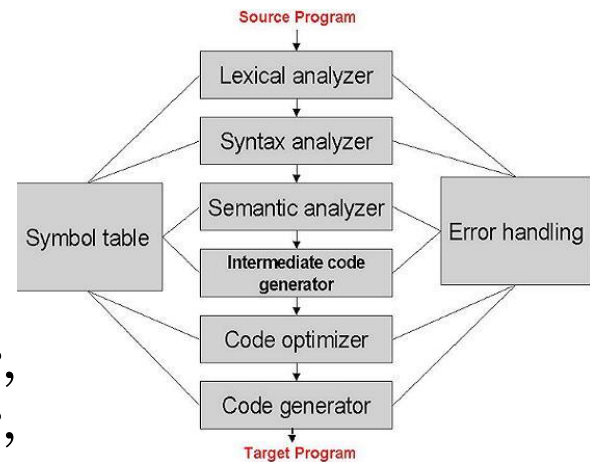


## ○ Robot

- Classi: Camera (per identificare gli oggetti), Arm (per muovere) e Pliers (per afferrare)

vs.

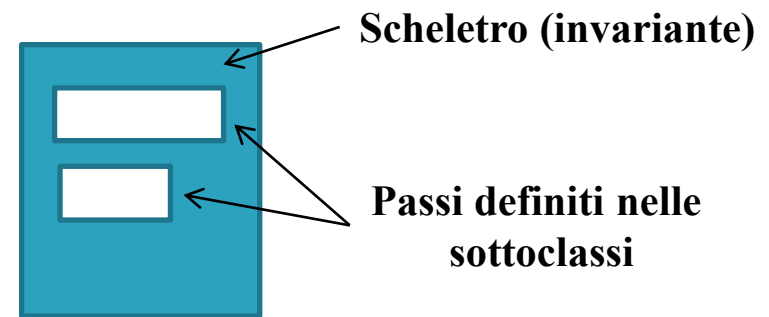
- Classe **Robot** con metodo moveObject()



# CONSEGUENZE: FAÇADE PATTERN

- Promuove un accoppiamento debole fra cliente e sottosistema 😊
  - Riduce la dipendenza tra client e sottosistema
- Nasconde al cliente le componenti (complesse) del sottosistema 😊
- Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema 😊
- **Façade vs. Adapter**
  - Entrambi sono “wrapper” (involucri)
  - Entrambi si basano su un'interfaccia, ma:
    - **Façade la semplifica**
    - **Adapter la converte**

# TEMPLATE METHOD



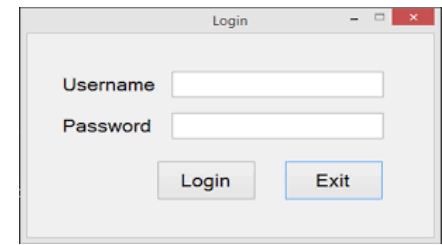
- Definisce lo **scheletro (template)** di un **algoritmo in un metodo** posponendo la definizione di alcuni passi a delle sottoclassi
  - permette di ridefinire alcuni passi di un algoritmo senza cambiare lo scheletro

## ○ Problema

- Implementare la parte **invariante** di un algoritmo una sola volta, e lasciare alle sottoclassi l'implementazione delle parti che possono variare
  - per evitare duplicazioni (migliora il riuso)!!



# TEMPLATE METHOD - ESEMPIO



- Vogliamo scrivere una classe **riusabile** per il **logging degli utenti** in una applicazione, che svolga i seguenti passi:

1. Chieda all'utente Username e Password
2. Autentichi l'utente **producendo un oggetto** che incapsuli eventuali informazioni che potranno essere richieste in seguito come prova dell'autenticazione
3. Mostri all'utente un display animato mentre l'autenticazione è in corso
4. Notifichi all'utente quando il login è completo, rendendo l'oggetto prodotto dall'autenticazione disponibile all'app

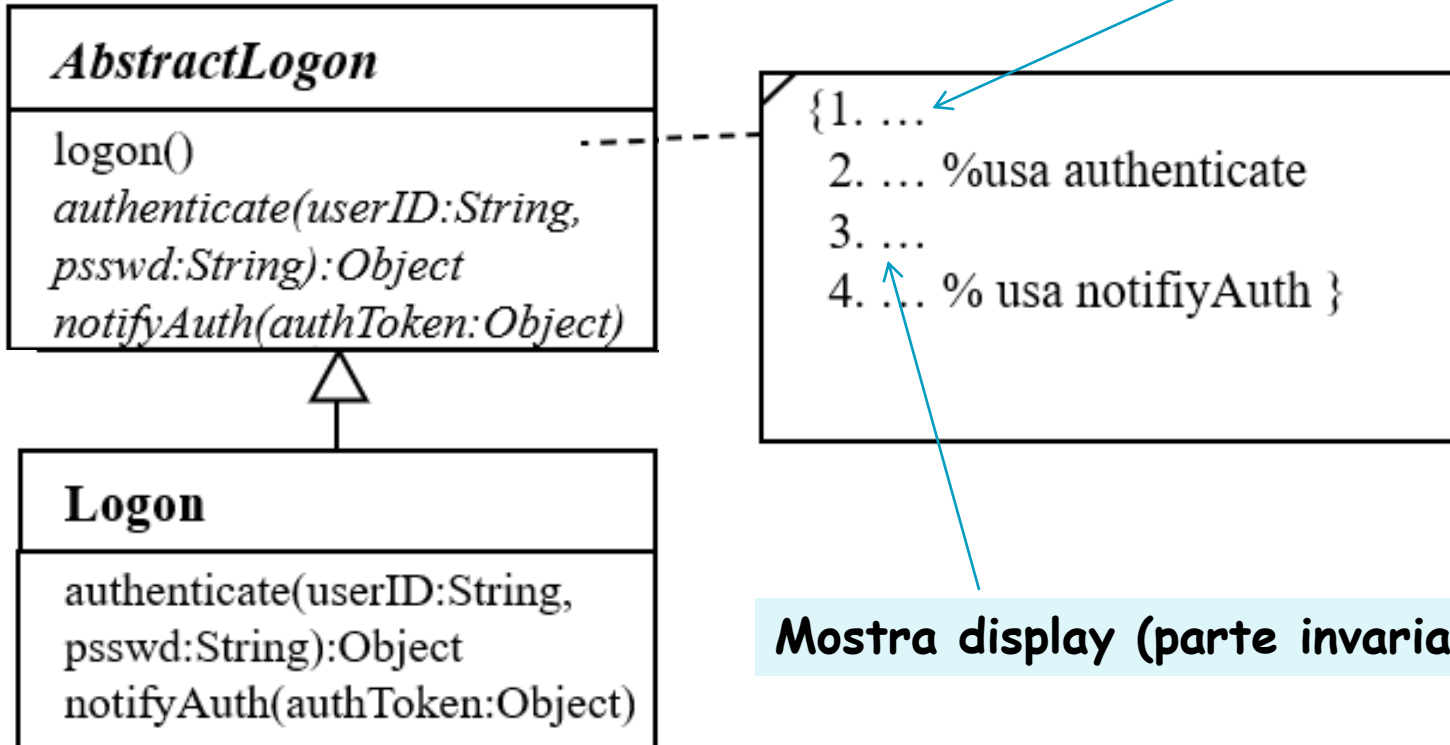
**Parte invariante/fissa** (anche se possono cambiare messaggi e immagini, la logica è la stessa)

**Parte variabile** (la logica dipende dall'applicazione)

# TEMPLATE METHOD

Classe astratta (riusabile)

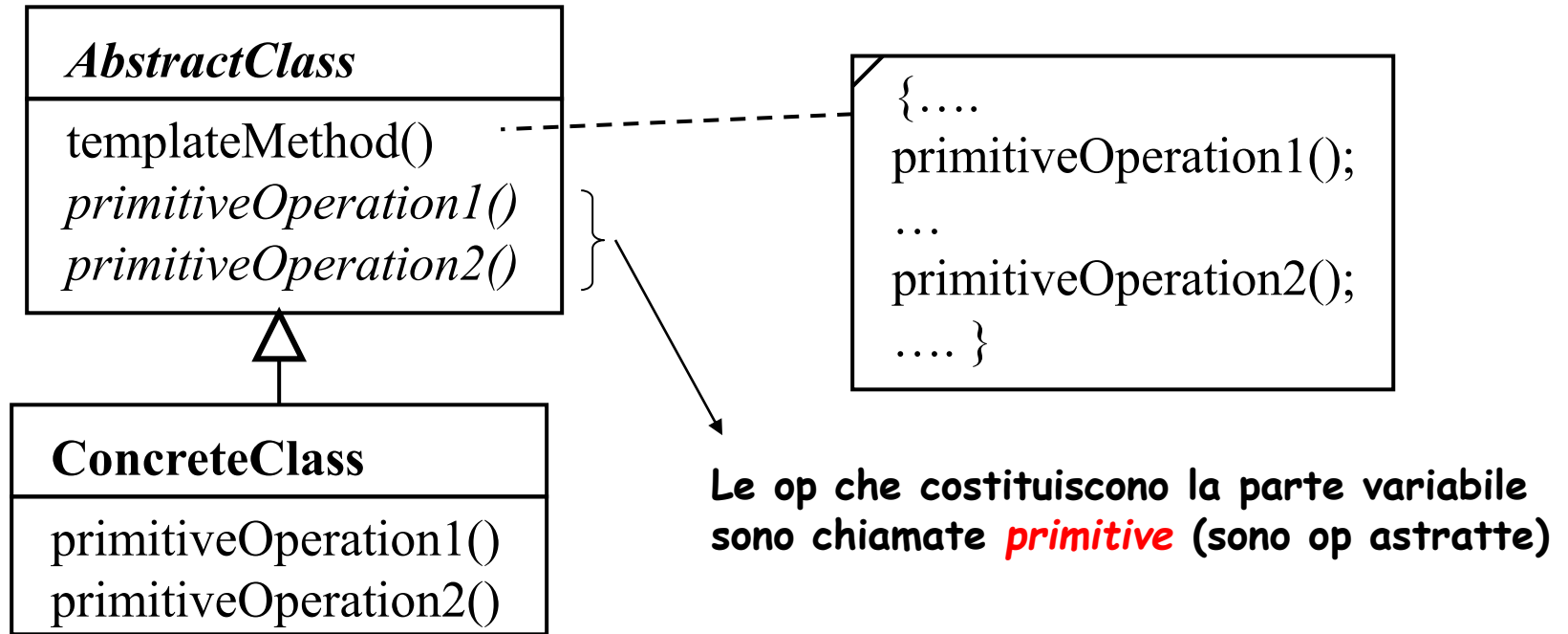
Chiede id e pwd (parte invariante)



Mostra display (parte invariante)

Classe concreta che fa override di `authenticate()` e `notifyAuth()` (parti variabili)

# IL PATTERN REALIZZA INVERSIONE DI CONTROLLO



Hollywood principle (Inversione di controllo)

*Non chiamate, richiameremo noi ...*

Normalmente sono le sottoclassi a chiamare i metodi delle superclassi. Con questo pattern è `templateMethod()` a chiamare i metodi specifici ridefiniti nelle sottoclassi

# CONSEGUENZE: TEMPLATE METHOD

- Tecnica fondamentale per il riuso del codice
  - molto usata nelle librerie e nei **framework**

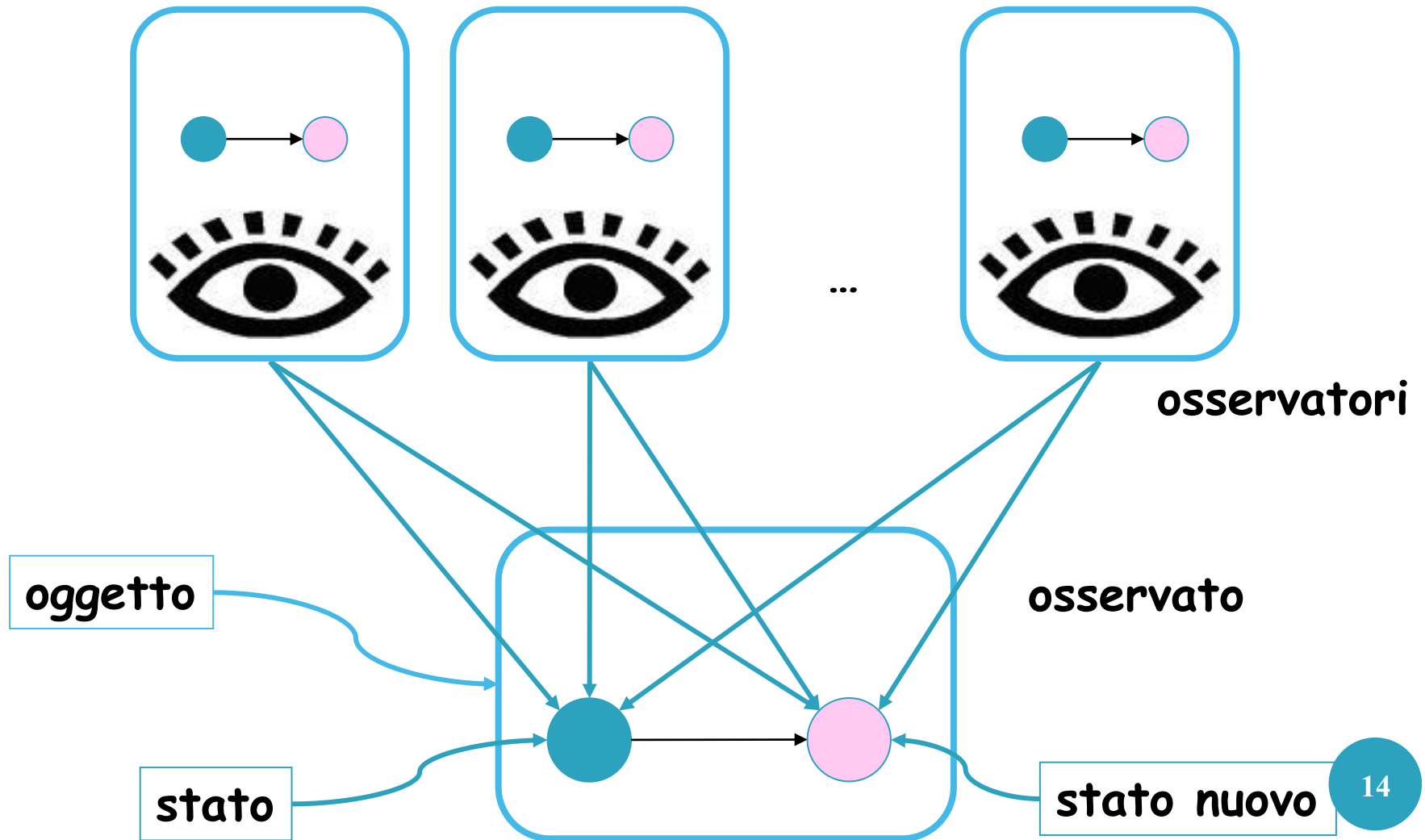


Reuse reduces the risks and costs of software development and maintenance. Figure 2


- Realizza inversione del flusso di controllo
- Permette di avere anche più sottoclassi concrete
  - Rappresentano differenti parti variabili
    - Ad esempio Logon1, Logon2, Logon3
- Importante chiarire bene quali operazioni devono essere ridefinite nelle sottoclassi
  - Parte fissa vs. parte variabile



# OBSERVER - COSA VORREMMO



# PERÒ COME LO IMPLEMENTIAMO?

- Attributi di stato pubblici?!?
  - Non scherziamo ... 
- **Con un operazione**: gli osservatori invocano “continuamente” un operazione **cambiato()** dell'oggetto osservato e poi **getStato()**

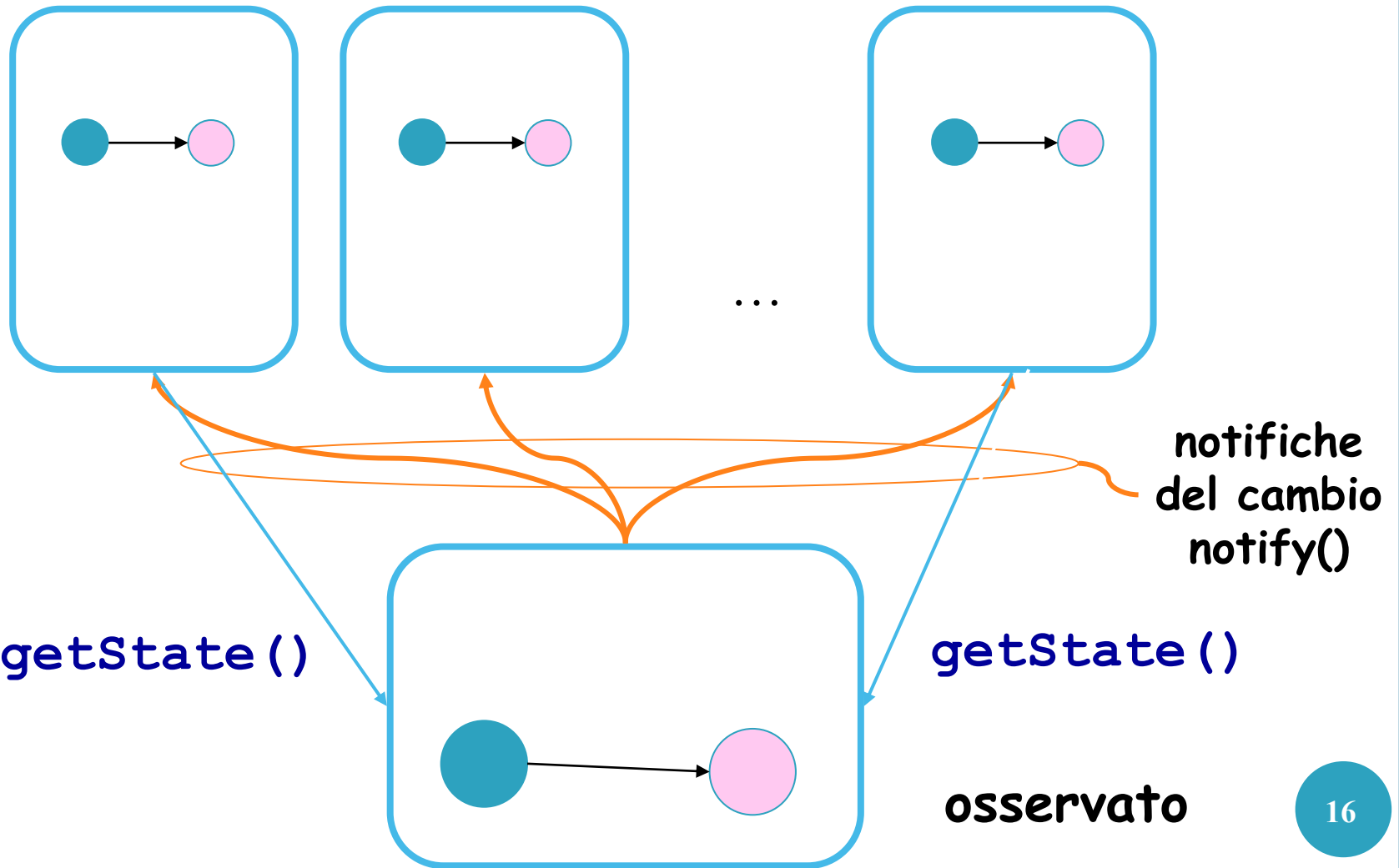
OggettoOsservato
stato: Stato
cambiato(): boolean

?

- Così l'osservatore potrebbe:
  - “rompere” troppo all'osservato
  - scoprire la variazione troppo tardi
  - “perdersi” la prima di due variazioni “vicine”
- E in più se ho tanti osservatori? **poco scalabile**
  - l'osservato passa il tempo a rispondere (magari senza nessuna variazione)

# L'IDEA ...

**osservatori**



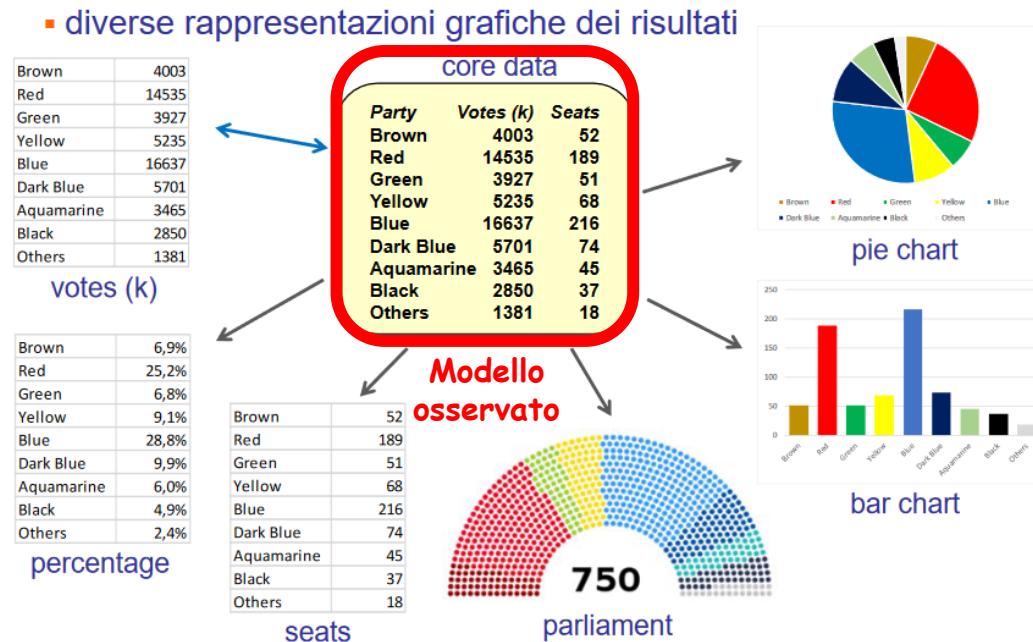
# QUINDI ...

- Gli osservatori si **registrano** presso l'oggetto osservato
  - Gli osservatori si possono aggiungere e togliere dinamicamente (**runtime**)
- Quando l'oggetto osservato cambia stato, **notifica** tutti gli osservatori
  - invoca l'operazione **notify()**
- Quando notificato, ogni osservatore decide cosa fare:
  - Niente
  - **Richiedere** all'osservato informazioni sullo stato
    - **getState()**
  - ...



## VISTE SULLO STESSO OGGETTO

- Gli osservatori potrebbero essere varie “viste” sullo stesso oggetto
  - Esempio: dei Dati di un foglio elettronico e loro rappresentazioni tabellari o grafiche



**Modello e viste ad esso collegate sono consistenti!**

# INTRODUZIONE MVC

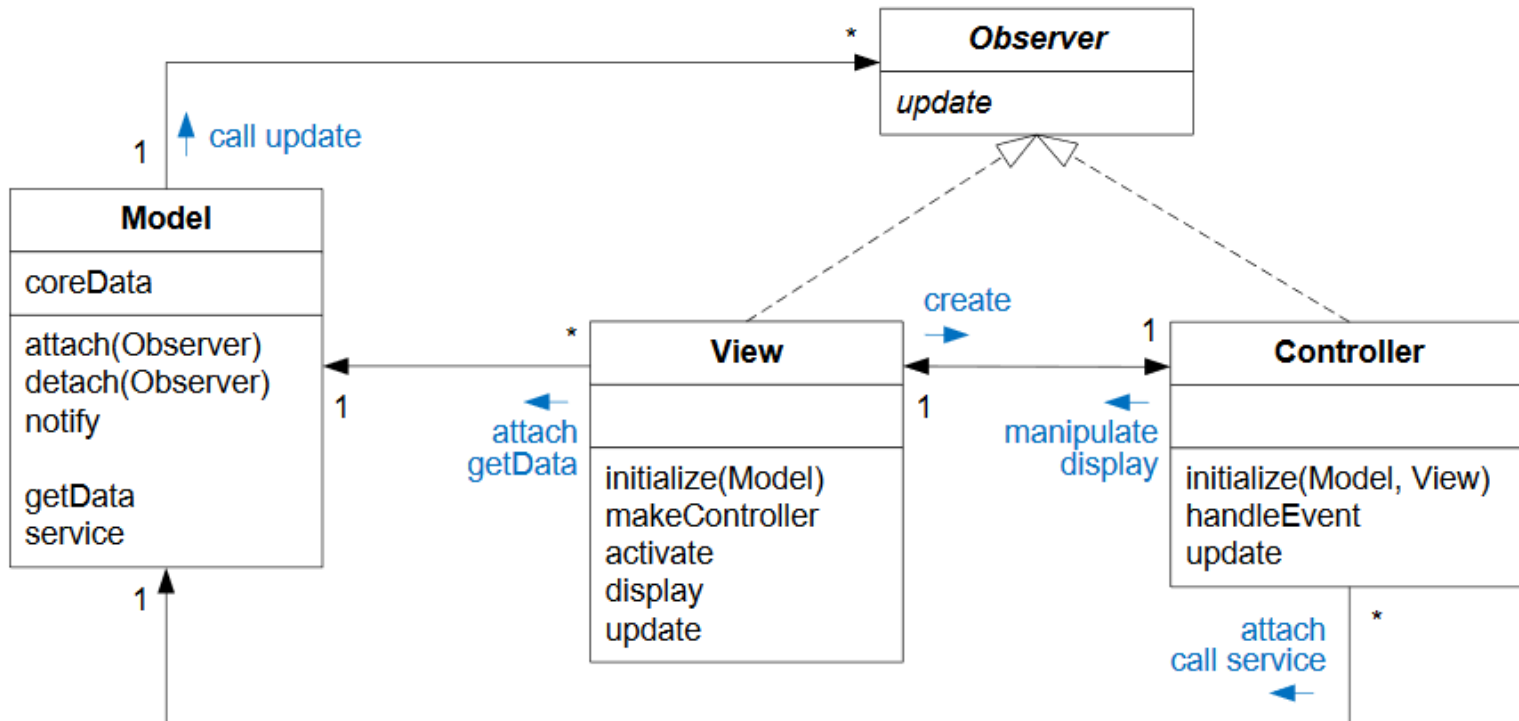
- **E' un pattern (stile) architetturale!**

- Granularità maggiore rispetto ai design pattern
- Design pattern risolvono 'piccoli' problemi
- MVC definisce uno stile architetturale di un applicazione

- Divide **un'applicazione interattiva** in tre tipologie di componenti

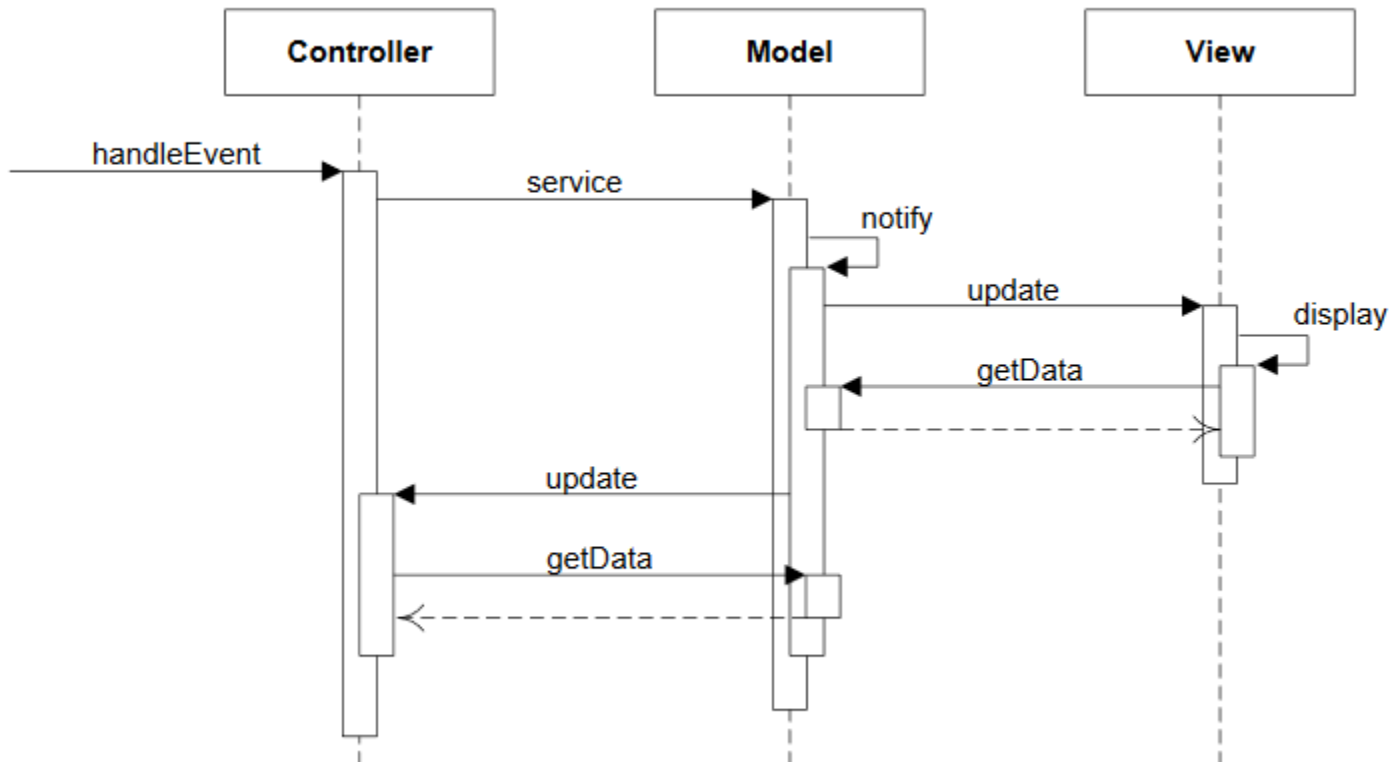
- **un modello** – **contiene i dati e le funzionalità di base**
  - il modello si occupa dell'elaborazione dei dati
- **una o più viste** – mostrano informazioni agli utenti
  - una vista si occupa della gestione **dell'output**
- **uno o più controller** – gestiscono le richieste degli utenti
  - un controller si occupa della gestione **dell'input**
- **Un'interfaccia utente** è formata da una vista e un controller

# (POSSIBILE) STRUTTURA



**Viste e controllori ‘sono degli observer’ del modello ....**

# SCENARIO PROPAGAZIONE INPUT



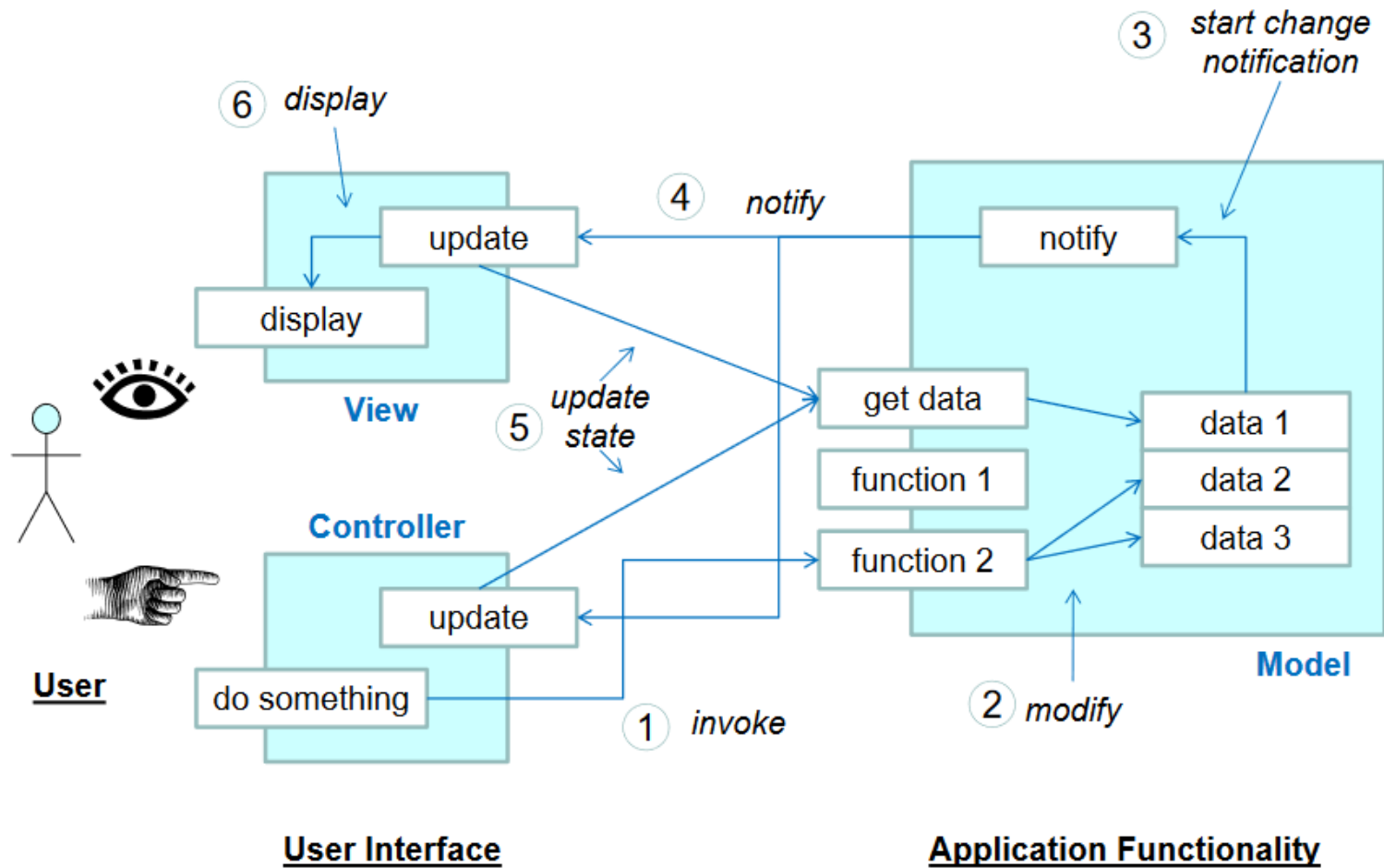
{for all o in observers  
o.update() }

**notify() chiama  
update su tutti gli  
observer**

# SCENARIO PROPAGAZIONE INPUT

- uno dei controller accetta una richiesta di input tramite la sua procedura di gestione degli eventi – interpreta l'evento e chiede al modello l'esecuzione di un servizio
- il modello esegue il servizio richiesto – questo può portare a un cambiamento del suo stato interno
- il modello notifica tutte le viste e i relativi controller dei cambiamenti di stato significativi – tramite un meccanismo di propagazione dei cambiamenti – ad es., mediante Observer
  - ogni vista chiede al modello i dati di interesse che sono cambiati – e aggiorna la sua visualizzazione
  - anche i controller interrogano il modello – ad es., per capire se devono abilitare o disabilitare certe funzionalità
- il controllo torna al controller originale, considerando conclusa la gestione dell'evento di input

# SCENARIO PROPAGAZIONE INPUT



# UTILIZZO DEL PATTERN MVC

## Utilizzo [ [modifica](#) | [modifica wikitest](#) ]

Storicamente il pattern MVC è stato implementato lato server. Recentemente, con lo sviluppo e la parziale standardizzazione di [JavaScript](#) sono nate le prime implementazioni lato client.<sup>[3]</sup>

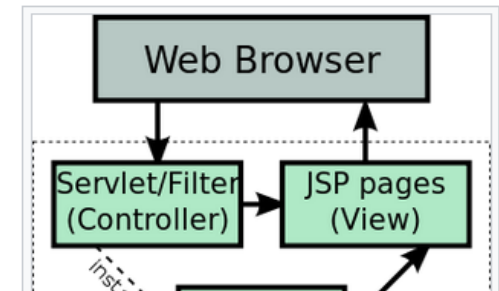
### Lato server [ [modifica](#) | [modifica wikitest](#) ]

Originariamente impiegato dal linguaggio [Smalltalk](#), il pattern è stato esplicitamente o implicitamente sposato da numerose tecnologie moderne, come [framework](#) basati su [PHP](#) ([Symfony](#), [Laravel](#), [Zend Framework](#), [CakePHP](#), [Yii framework](#), [CodeIgniter](#)), su [Ruby](#) ([Ruby on Rails](#)), su [Python](#) ([Django](#), [TurboGears](#), [Pylons](#), [Web2py](#), [Zope](#)), su [Java](#) ([Spring](#), [JSF](#) e [Struts](#)), su [Objective C](#) o su [.NET](#).

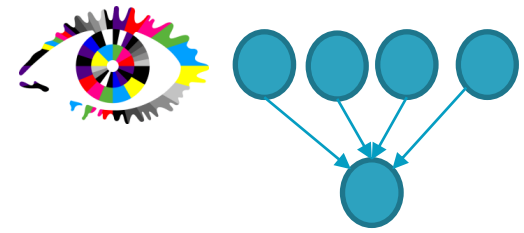
A causa della crescente diffusione di tecnologie basate su MVC nel contesto di [framework](#) o piattaforma [middleware](#) per [applicazioni web](#), l'espressione **framework MVC** o **sistema MVC** sta entrando nell'uso anche per indicare specificamente questa categoria di sistemi (che comprende per esempio [Ruby on Rails](#), [Struts](#), [Spring](#), [Tapestry](#) e [Catalyst](#)).

### Lato client [ [modifica](#) | [modifica wikitest](#) ]

Negli ultimi anni è aumentata la richiesta di [Rich Internet application](#) che facciano chiamate asincrone al server ([AJAX](#)), senza fare redirect per visualizzare i risultati delle elaborazioni. Col crescere della quantità di codice JavaScript eseguito sul client, si è sentita l'esigenza di creare i primi framework che implementassero MVC in puro JavaScript. Uno dei primi è stato [Backbone.js](#), seguito da una serie interminabile di altri framework, tra cui [JavaScriptMVC](#), [Ember](#) ed [AngularJS](#).



# RITORNIAMO SULL'OBSERVER



- Definisce una **dipendenza** “**lasca**” uno a molti tra oggetti, in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da lui sono avvertiti ...

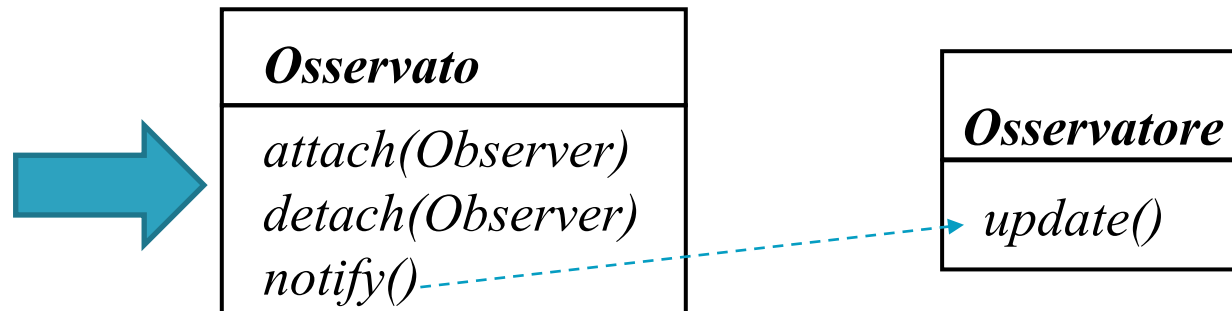
## Problema

- Associare più “viste” differenti ad un modello (dati)
- Implementare il **broadcast**
- Il cambiamento di un oggetto richiede il cambiamento di altri oggetti
  - Non si conosce quanti oggetti devono cambiare
- Notificare oggetti senza fare assunzioni a priori su quali siano questi oggetti
  - Evita l'accoppiamento “forte”



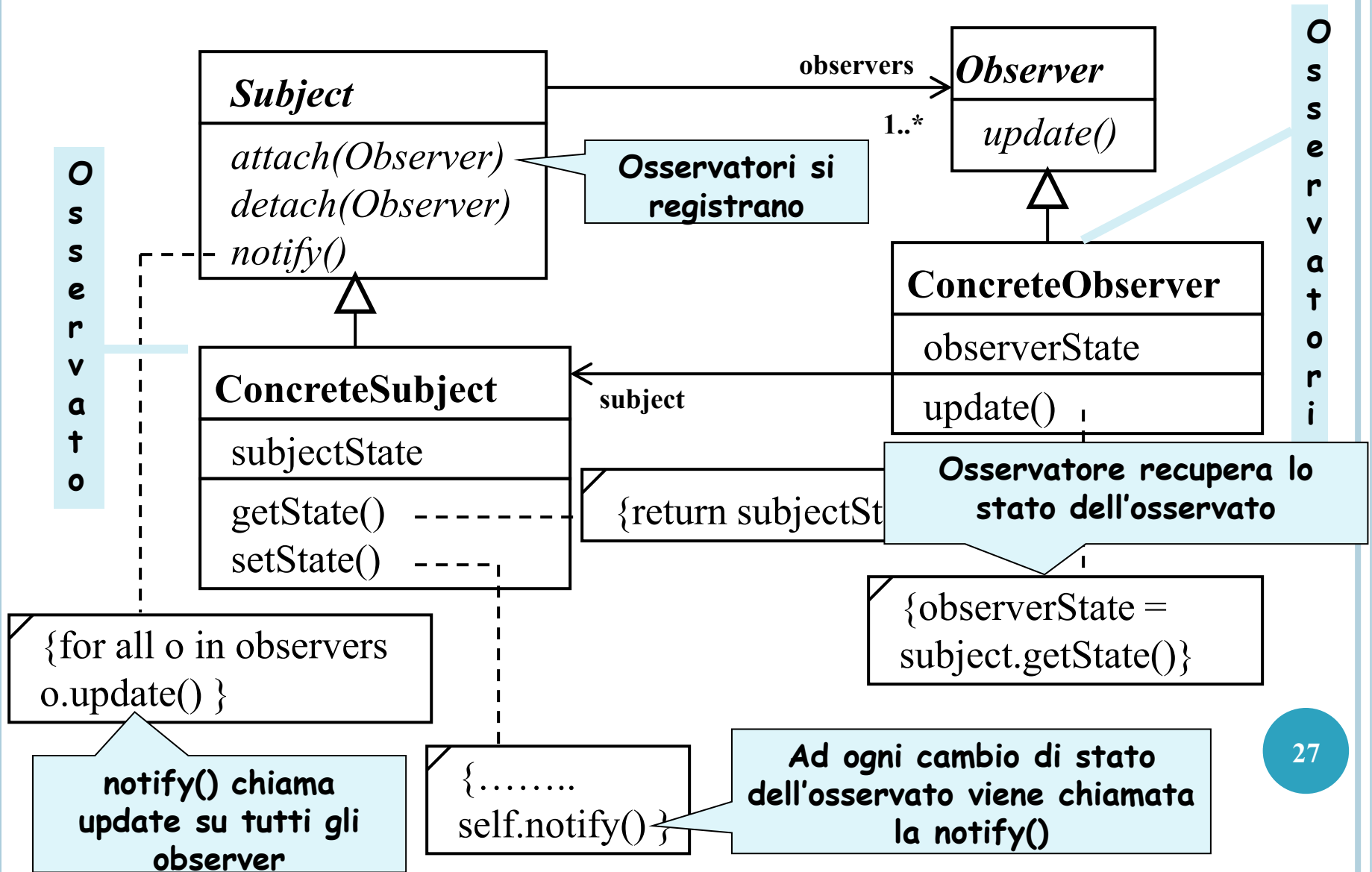
# INTERFACCE

- L'oggetto osservato deve fornire un'interfaccia standard per la registrazione `attach(Osservatore)` e `detach(Osservatore)`



- L'oggetto osservato deve poter notificare:
  - l'oggetto osservato ha `notify()` che chiama tutti gli `update()` dei registrati
  - gli osservatori devono fornire un'interfaccia standard per la notifica `update()`

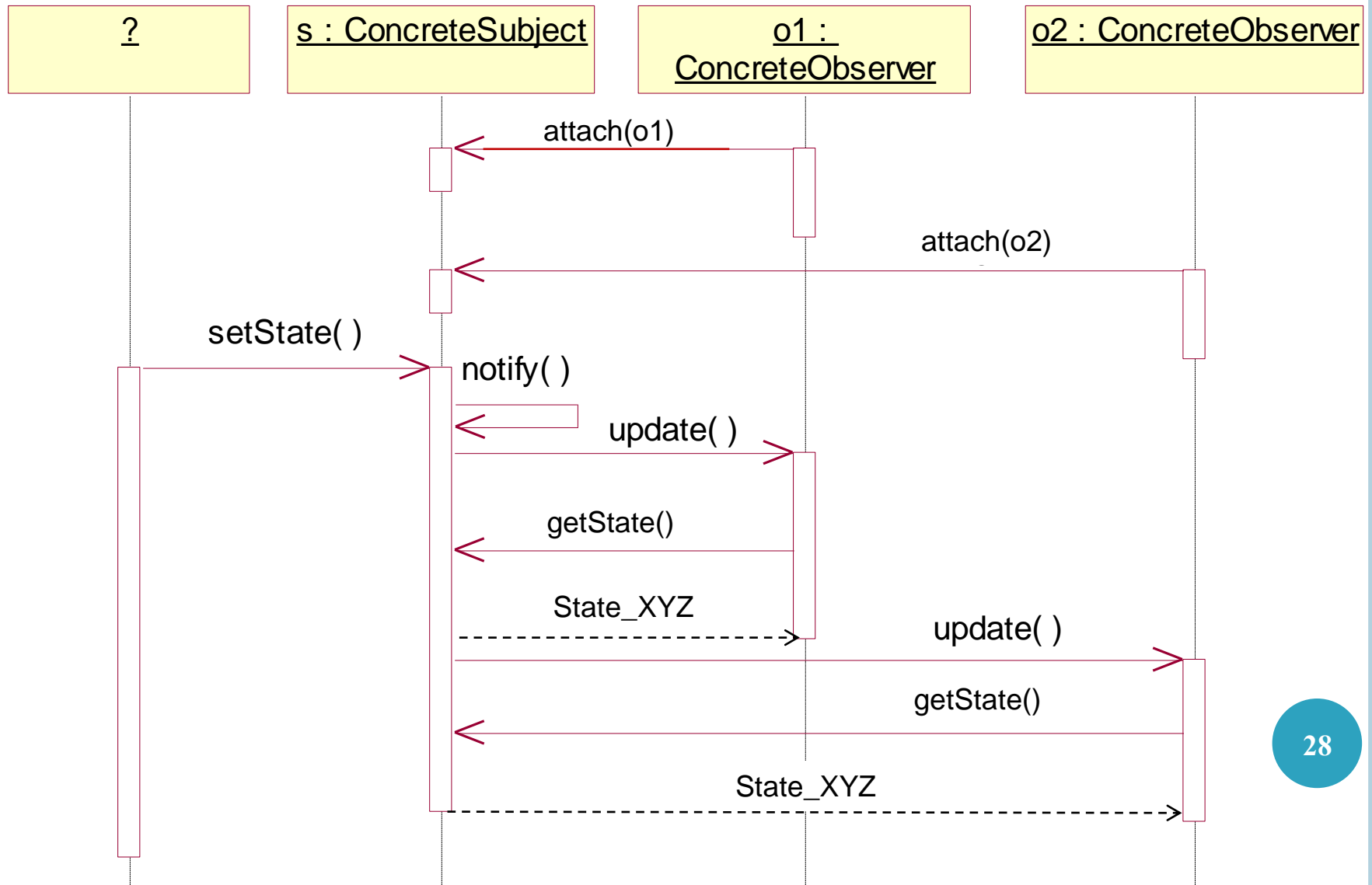
# OBSERVER DESIGN PATTERN



# SEQUENCE DIAGRAM OBSERVER

**osservato**

**osservatori**



# CONSEGUENZE: OBSERVER PATTERN

- Collegamento lasco tra Osservato e Osservatori 😊
  - L'Osservato sa che vi è una lista di Osservatori, ma non conosce la loro struttura (è **dinamica**)
  - Osservato e Osservatori possono appartenere a diversi strati di astrazione in un sistema
- Supporto per la **comunicazione broadcast** 😊
  - publish/subscribe
- Modifiche inaspettate 😞
  - una modifica innocente nell'osservato può scatenare una cascata di modifiche negli Osservatori e nei loro dipendenti

# STATE PATTERN



PushButton(eroga caffè) → eroga (stato=soldi inseriti OK)

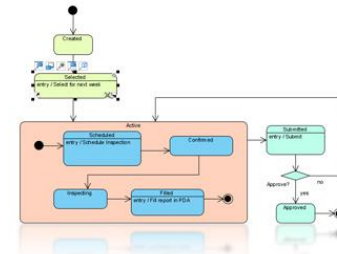
PushButton(eroga caffè) → msg (stato=soldi inseriti insuff)

## ◉ Scopo

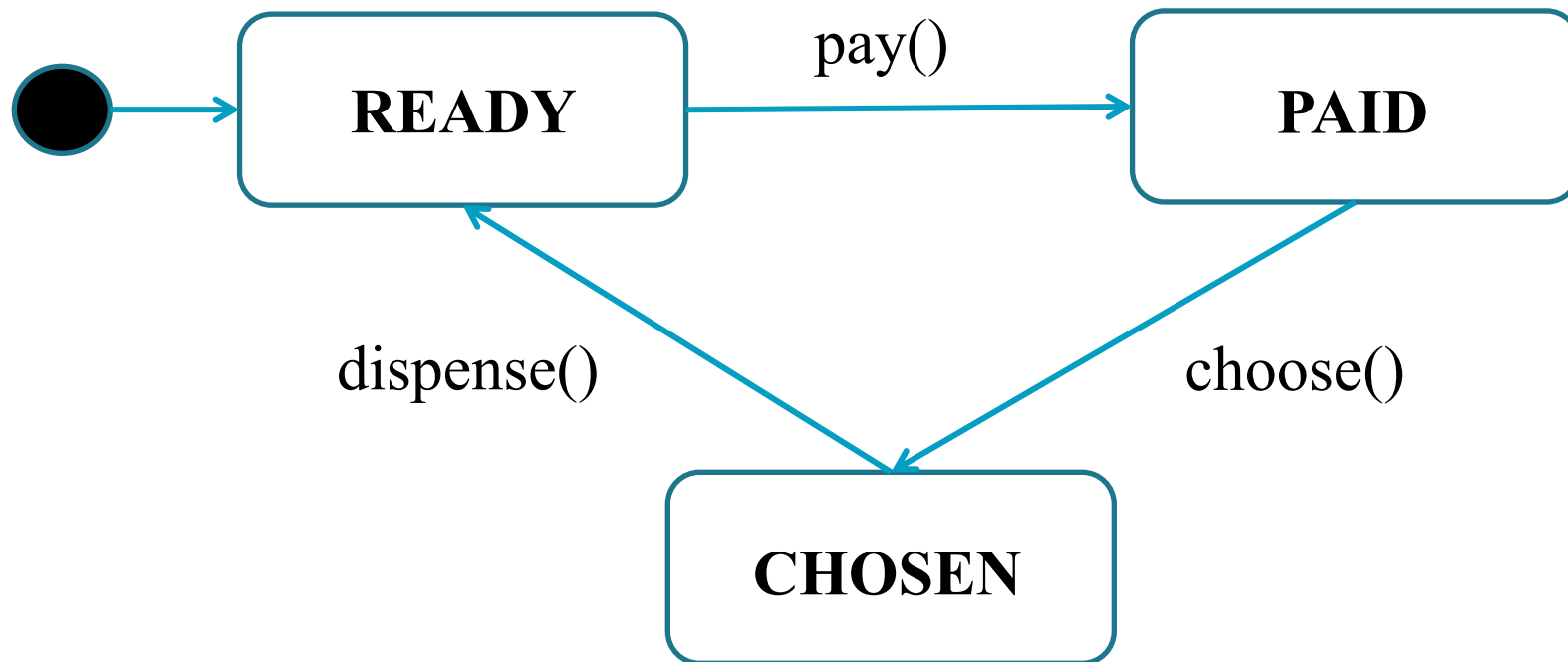
- Permettere a un oggetto di cambiare il suo comportamento al variare del suo stato interno
- **Permette di implementare le state machine** (bene!) in un linguaggio che non le supporta nativamente

## ◉ Come funziona?

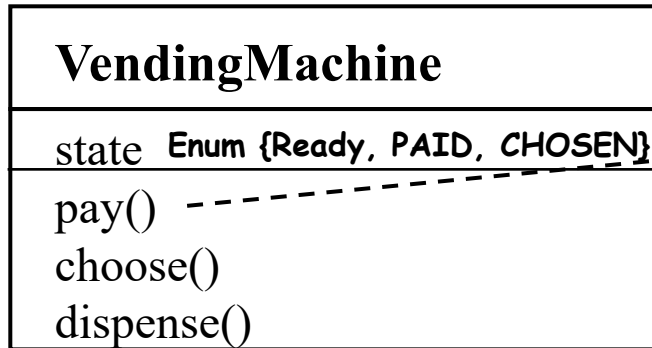
- Si estrae la **rappresentazione dello stato** in classi esterne
  - ◉ organizzate in una gerarchia!
- Si sfrutta il **polimorfismo** per variare il comportamento



# ESEMPIO: VENDING MACHINE



# POSSIBILE IMPLEMENTAZIONE (IF/SWITCH)



```
{...  
if (state==READY )  
...  
else if (state == PAID)  
...  
else if (state == CHOSEN)}
```

**ESEMPIO:**

```
public void pay(money) {  
    switch(state) {  
        case READY:  
            if (money are sufficient) {  
                ... cambia il totale ... msg("money ok") ...  
                state=PAID;  
            } else { // money not sufficient  
                ... cambia il totale ... msg("insert more money") ...  
            }  
        PAID  
        CHOSEN  
        default: msg("Op not possibile!!");  
    }
```

# PERÒ ...

Tanti if... (campanello d'allarme)



- Difficile da comprendere e mantenere

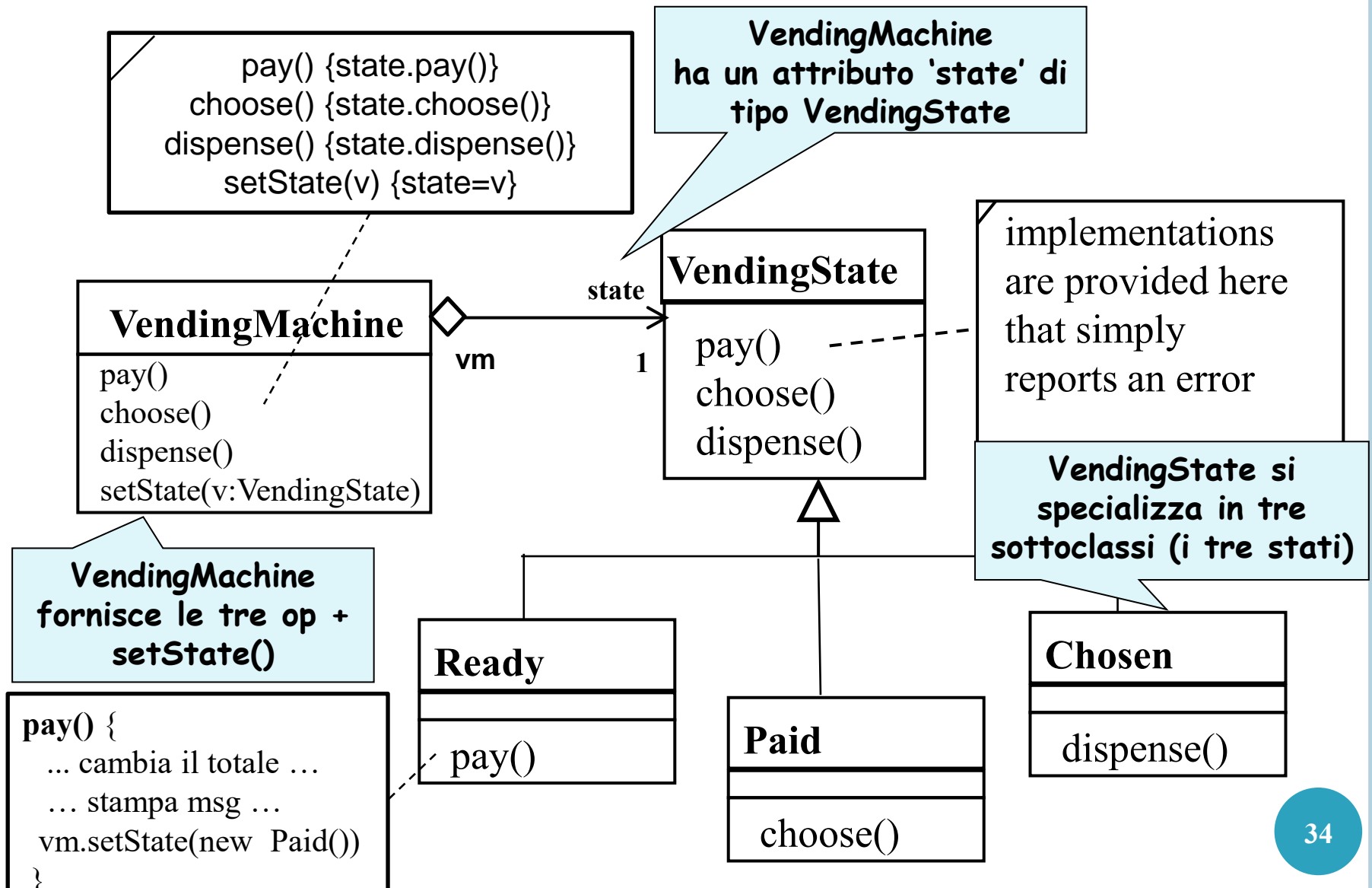
- Aggiunta di uno stato implica molte modifiche
  - Un 'If' per ogni metodo di evento

- **Alternativa: State pattern!**

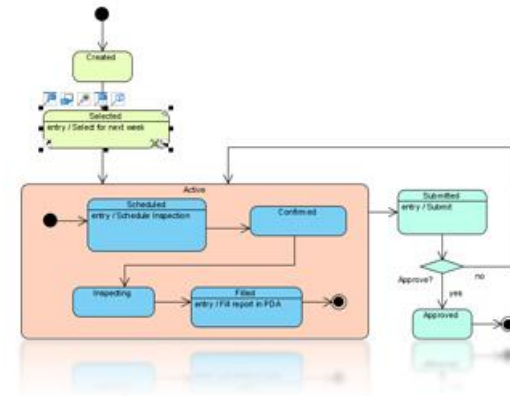
- Usiamo una gerarchia di classi per rappresentare gli stati della macchina
- Ogni (sotto)classe uno stato
- Ogni (sotto)classe ha la sua implementazione delle operazioni



# IMPLEMENTAZIONE CON STATE PATTERN



# STATE PATTERN



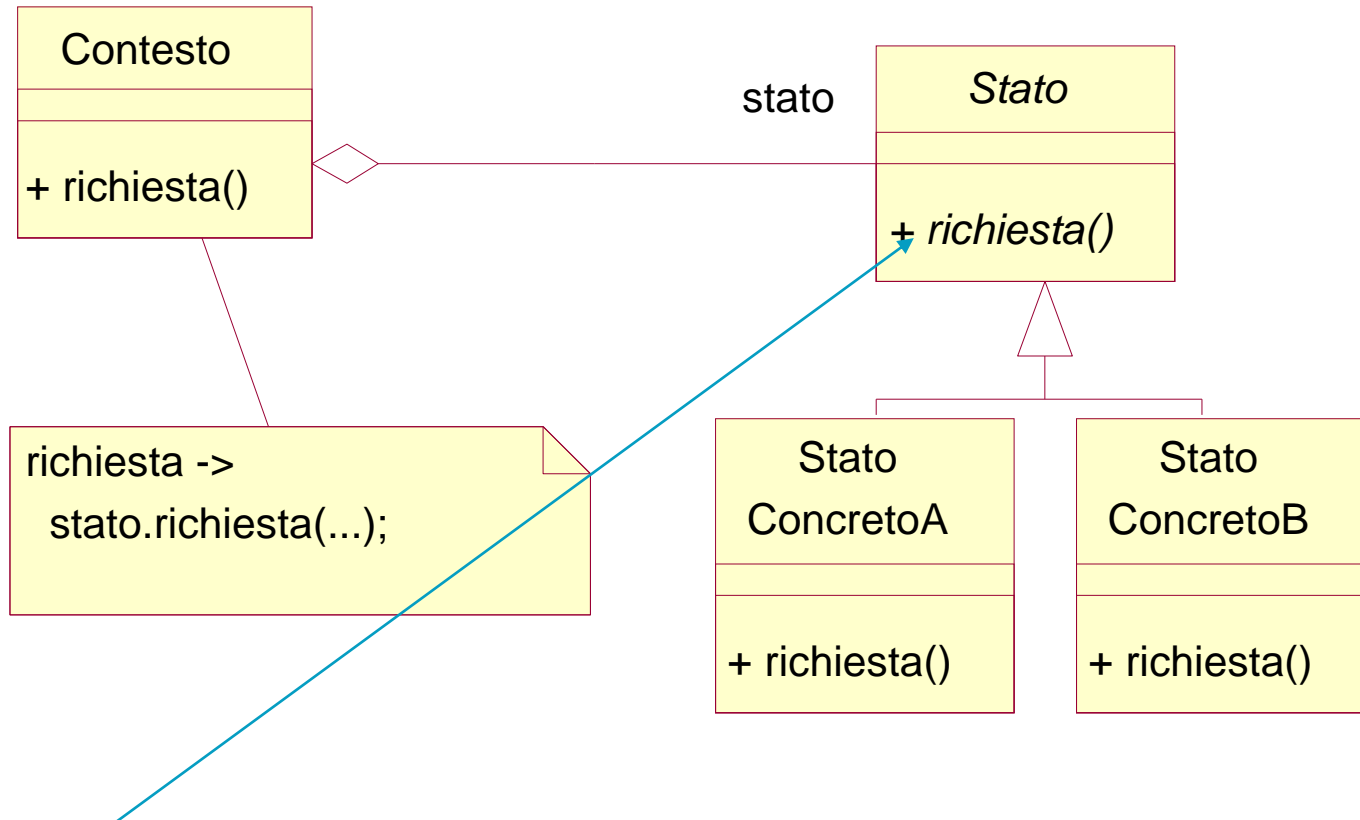
Consente ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello stato in cui si trova

- Es. Vending Machine

## ◦ Problema

- Un oggetto deve modificare dinamicamente il suo comportamento al variare del suo stato interno
  - State machine

# DIAGRAMMA STATE PATTERN



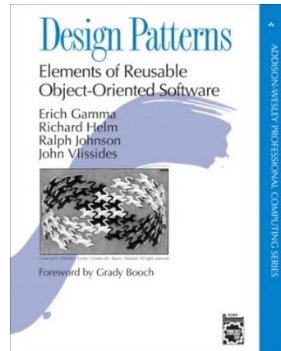
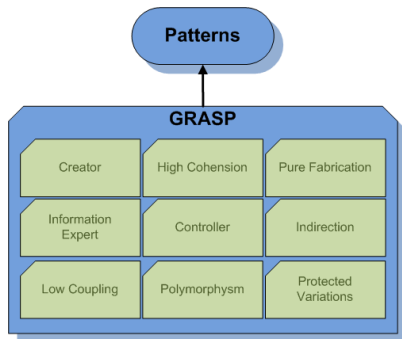
**Se richiesta è astratto nella superclasse (astratta) Stato, le sottoclassi concrete devono implementare anche i metodi che corrispondono a transizioni non ammesse in quello stato (che solleveranno errori)**

# CONSEGUENZE: STATE PATTERN

- Incapsula i comportamenti associati ad uno stato in un oggetto e la logica di transizione tra stati in uno oggetto piuttosto che in una grosso switch tra istruzioni
  - La soluzione “switch” è difficile da mantenere/estendere
- Il comportamento associato ad uno stato dipende solo da una classe
  - *StatoConcretoA, vedi diagramma*
- E' semplice cambiare il comportamento di uno specifico stato e aggiungere nuovi stati
- Incrementa il numero delle classi 😞



# RIASSUMENDO



## DESCRIZIONE DI UN PATTERN

- Template piuttosto preciso!
  - Versione semplificata (solo parti fondamentali)

### Nome (una/due parole)

- Significativo/esplicativo, che aiuta a capire a cosa serve il pattern

### Problema

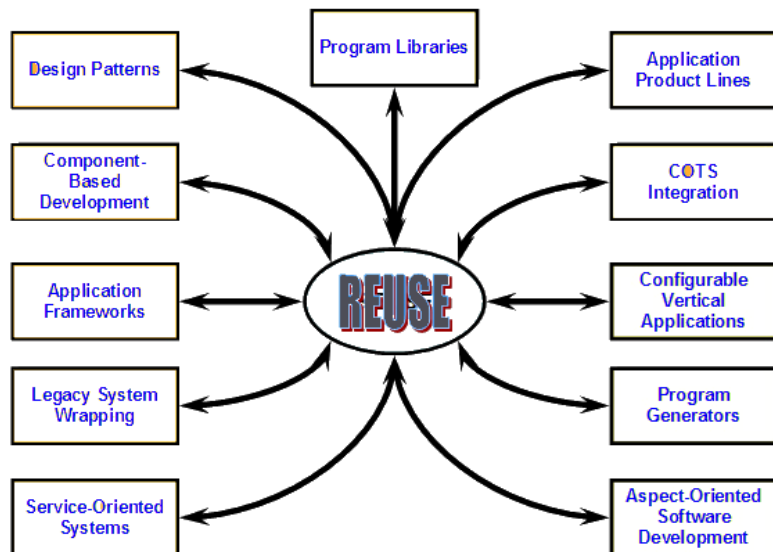
- descrive **quando** applicare il pattern
- può contenere anche delle precondizioni per poterlo applicare

### Soluzione **Non sempre precisa e completa, è uno sketch**

- descrive **astrattamente** gli elementi del design proposto, le loro relazioni, responsabilità e collaborazioni

### Conseguenze

- Pro/contro della sua applicazione



Creazione	Struttura	Comportamento
Factory Method <b>Abstract Factory</b> Builder Prototype Singleton	<b>Adapter</b> Bridge Composite Decorator <b>Facade</b> Flyweight Proxy	Interpreter <b>Template Method</b> Chain of responsibility Command <b>Iterator</b> Mediator Memento <b>Observer</b> <b>State</b> Strategy <b>Visitor</b>

# RIFERIMENTI

- (GoF) *Design Patterns: Elementi per il riuso di software ad oggetti*, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, Pearson, 1 gennaio 2002
- *Patterns in Java*, Mark Grand, Wiley (1998)
- GoF's Design Patterns in Java, Franco GUIDI POLANCO, Politecnico di Torino
- *Applicare UML e i pattern*, a cura di Luca Cabibbo Pearson; 3° edizione, 1 gennaio 2005, (Pattern GRASP)
- Slide del corso di Progettazione e analisi orientata agli oggetti, Stefano Mizzaro, Università di Udine