14. Design pattern

IS 2024-2025

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

VI RICORDATE QUESTA SLIDE?

Principi generali
- Information hiding (tramite controllo delle interfacce)
- Astrazione di dati e controllo
- Coesione
- Disaccoppiamento

Collezioni di principi
- SOLID
- GRASP

Design pattern
- Strategy
- State
- Factories
- Decorator
- …

L'IMPORTANZA DEI PATTERN

Filippo Leonardi per Comune di Venezia (https://www.comune.venezia.it/it/archivio/19479)

https://www.ilgazzettino.it/nordest/venezia/cartello_ironico_ponte_calatrava_venezia-1798313.html

PATTERN PER PROGETTARE GRADINI

Formula di **Blondel**

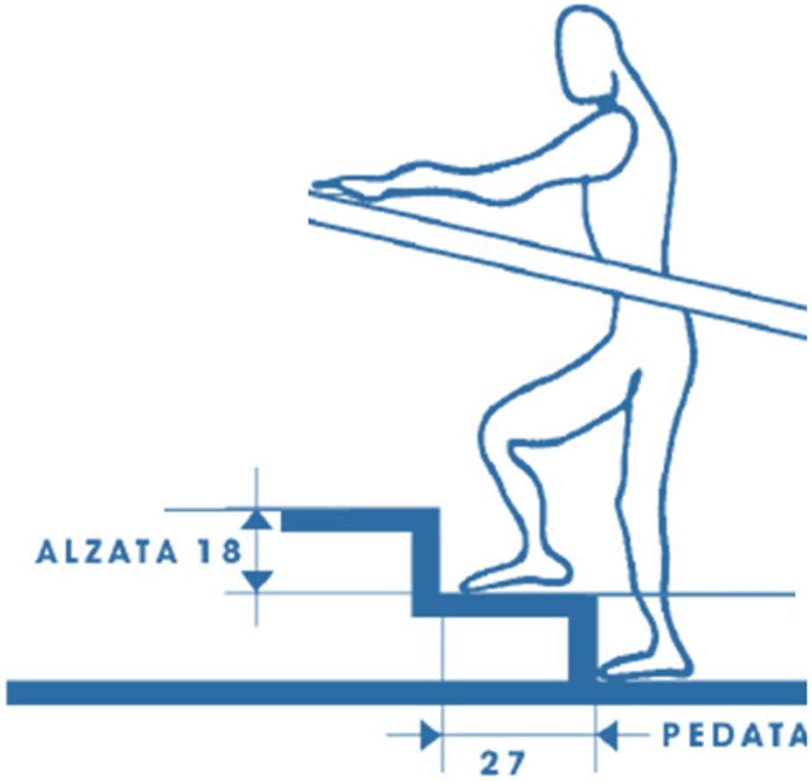
$$2 * alzata + pedata \, \epsilon \, [62cm, 65cm]$$

Limiti sulle piccole altezze di alzata:

- gradini bassi ⇒ risultati insoddisfacenti

Regola di **Hermant**

$$alzata * pedata = 600cm$$



PONTE DELLA COSTITUZIONE (CALATRAVA)

Pedata = 50 cm

Alzata = 8 cm

Formula di **Blondel**

$$2 * 8 + 50 = 66\ cm$$

(vicino, ma fuori da $[62cm, 65cm]$)

Regola di **Hermant**

$$50 * 8 = 400 (\neq 600)$$

Filippo Leonardi per Comune di Venezia
(https://www.comune.venezia.it/it/archivio/19479)

UN ALTRO PATTERN: DOV'È IL GRADINO?

Pietra d'Istria alternata alla trachite scura
(veneziani, XV secolo)

Ponte della costituzione
(Calatrava)

E QUINDI?

La progettazione **non è solo** un processo creativo!

«Leggi cento pagine di architettura al giorno» (Carlo Scarpa ad un giovane architetto)

Esistono una serie di regole pratiche che il progettista può seguire:

- Rapporto alzata/pedata (Blondel e Hermant)
- Materiali

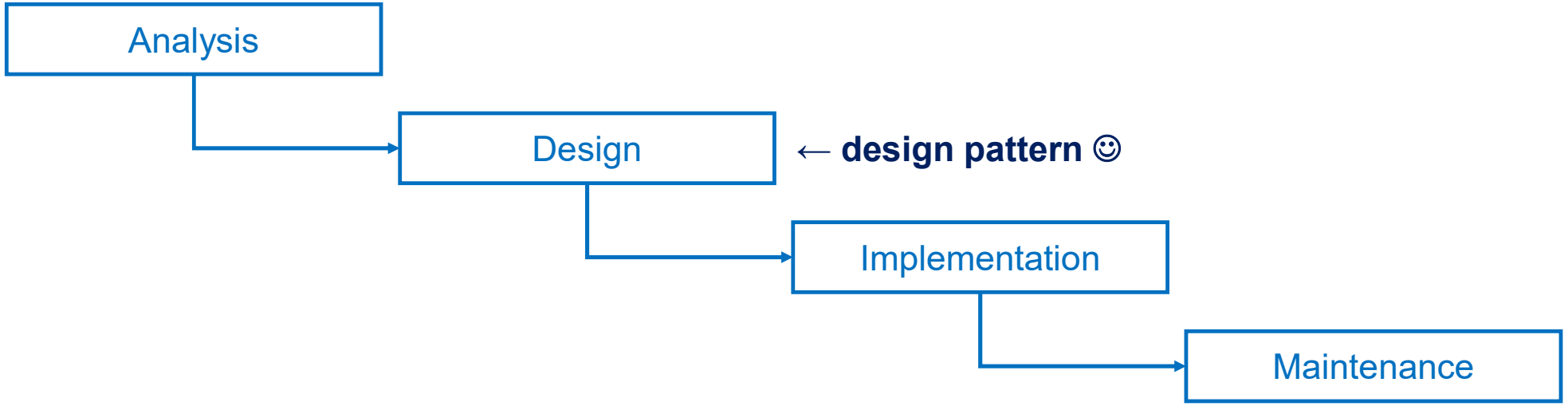
→ Queste regole pratiche sono i **design patterns**

(definiti grazie ad anni – a volte secoli – di studi ed esperienza)

COS'È UN PATTERN?

«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

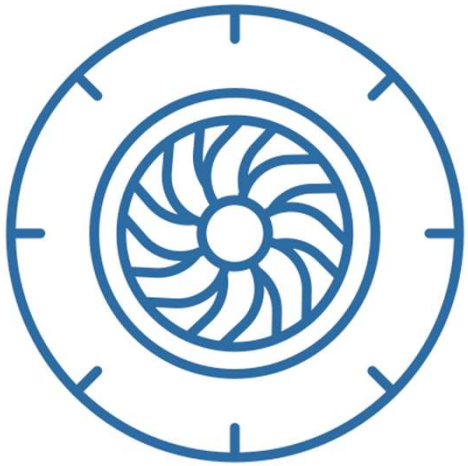


PATTERN NEL SOFTWARE: PERCHÉ?

«Progettare software OO è difficile e progettare software OO riutilizzabile è ancora più difficile» (E. Gamma)

I progettisti esperti **riusano le soluzioni** che hanno funzionato in passato

- Maggiore produttività
- Progetti risultanti più flessibili e riutilizzabili
- Don’t reinvent the wheel ☺



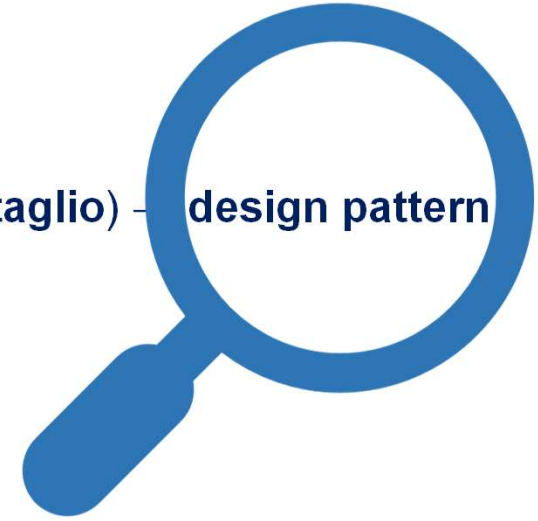
I sistemi **ben strutturati** hanno **modelli ricorrenti** di classi e oggetti

DALL'ARCHITETTURA AL CODICE

Architettura → pattern o **stili architecturali**

- pipes and filters
- publish-subsribe
- ...

Progettazione e raffinamento dei componenti (aka. **progettazione di dettaglio**) → **design pattern**

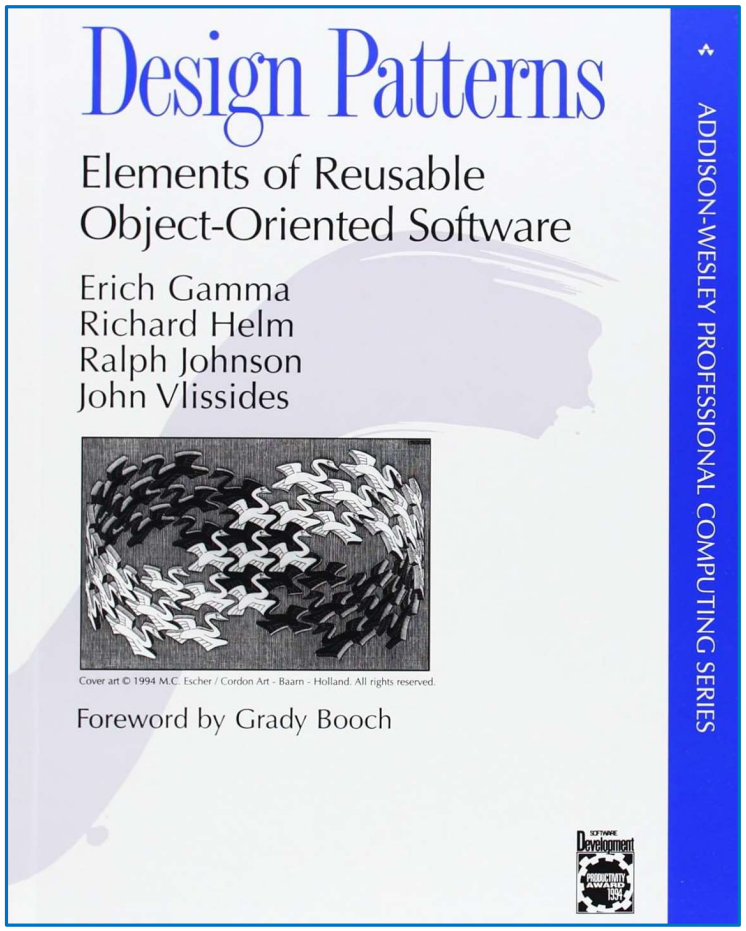


- abstract factory,
- decorator
- …

Codifica → **idiom** o **coding design patterns** (pattern specifici di linguaggi di programmazione)

- Idiom limitato a modello di progettazione (ma descrive comunque un problema ricorrente)
- In C, allocazione e deallocazione della memoria, convenzioni sui nomi delle variabili, ecc.

LETTURE CONSIGLIATE



GoF (Gang of Four)

CLASSIFICAZIONE GOF DEI DESIGN PATTERN

23 design pattern suddivisi in base al loro **scopo** (*purpose – what a pattern does*)

Pattern **creazionali** (creational pattern)

- Riguardano la creazione di oggetti
- *Abstract Factory, Builder, Factory Method, Prototype, Singleton*

Pattern **strutturali** (structural pattern)

- Riguardano la composizione di classi e oggetti
- *Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy*

Pattern **comportamentali** (behavioural pattern)

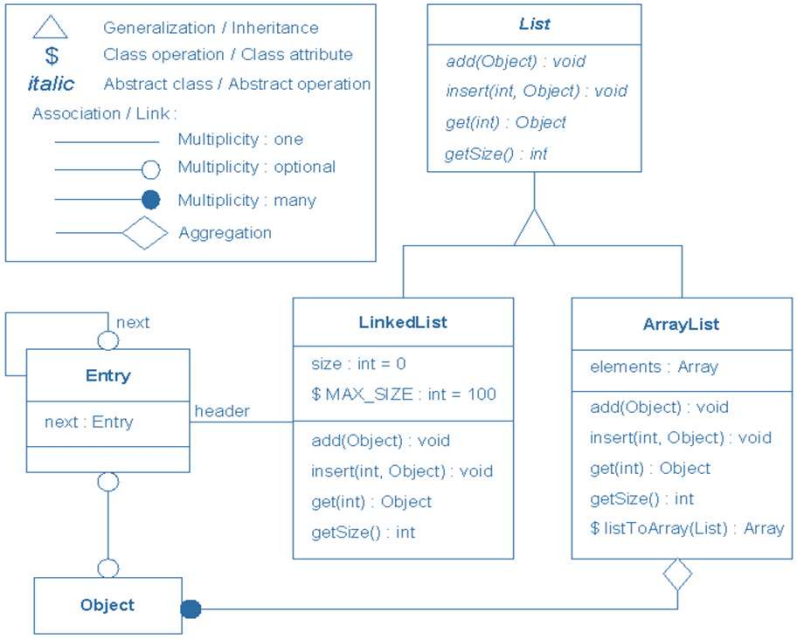
- Riguardano le interazioni tra classi e oggetti (e la suddivisione delle loro responsabilità)
- *Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template, Visitor*

GOF PATTERN TEMPLATE

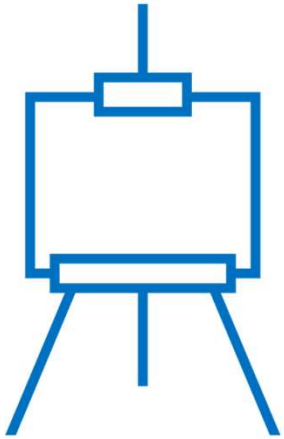
Pattern name and classification	A good, concise name for the pattern and its type
Intent	Short statement about what the pattern does
Also known as	Other names for the pattern
Motivation	A scenario that illustrates where the pattern would be useful
Applicability	Situations where the pattern can be used
Structure	A graphical representation of the pattern
Participants	The classes and objects participating in the pattern
Collaborations	How do the participants carry out their responsibilities
Consequences	What are the pros and cons of the pattern
Implementation	Hints and techniques for implementing the pattern
Sample code	Code fragments for a sample implementation
Known uses	Examples of the pattern in real systems
Related patterns	Other patterns that are closely related to the pattern

NOTAZIONE

GoF usa OMT (Object Modeling Technique)

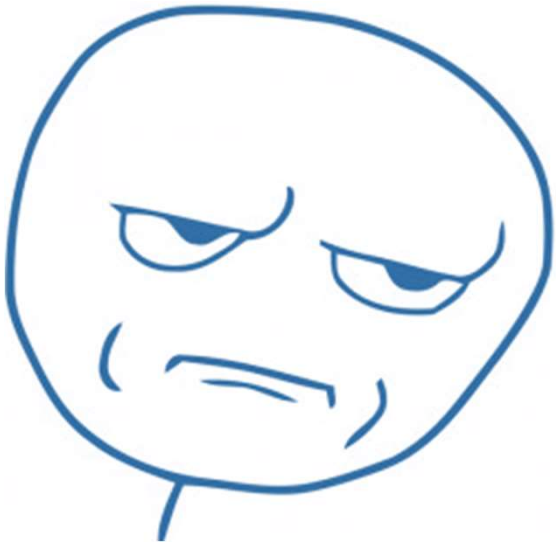


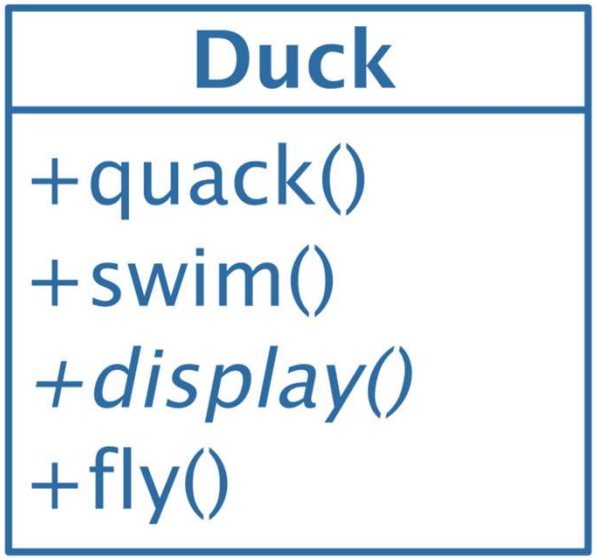
Altri libri (come Head First Design Patterns) usano UML



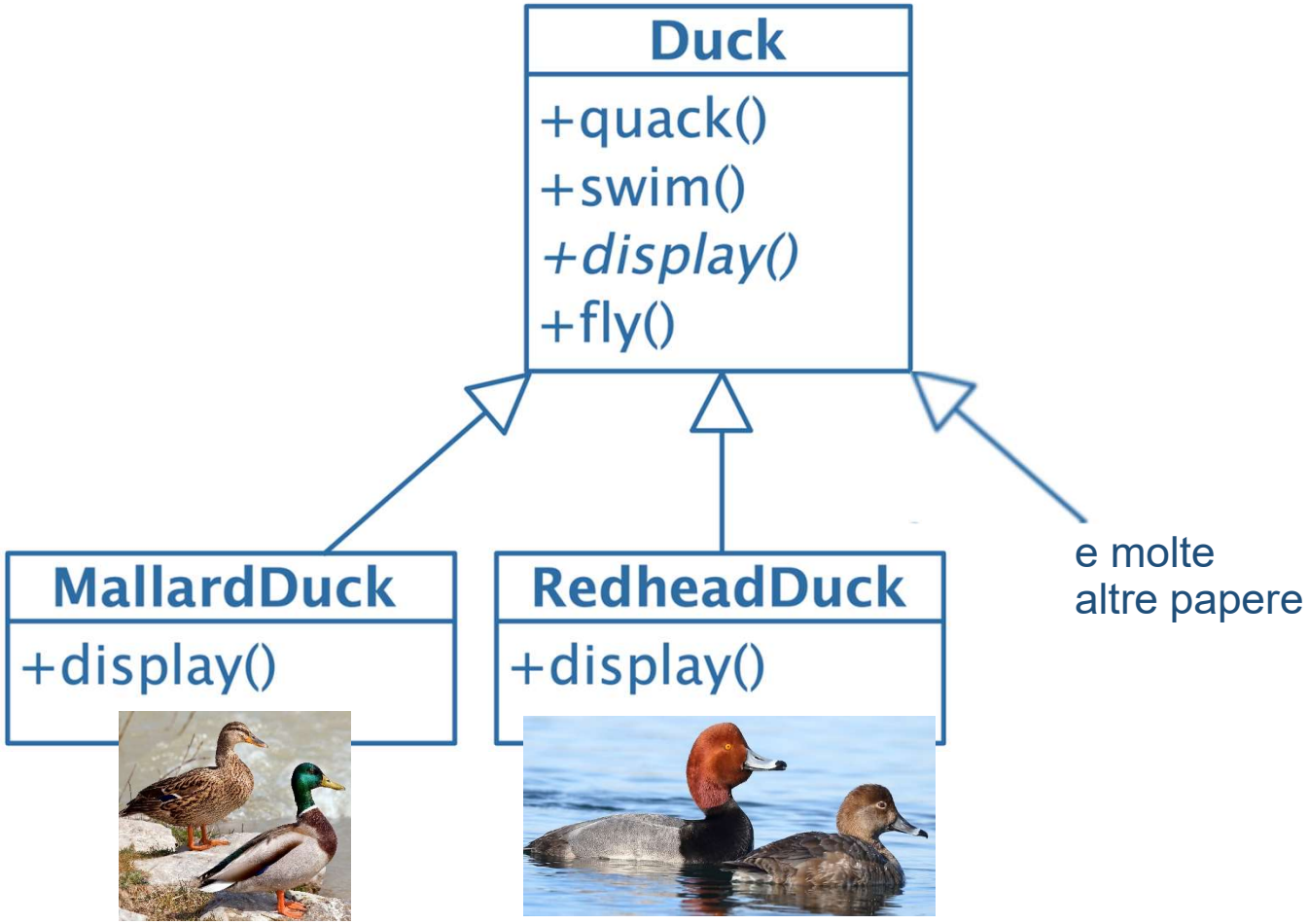
STRATEGY

UNA PAPERA

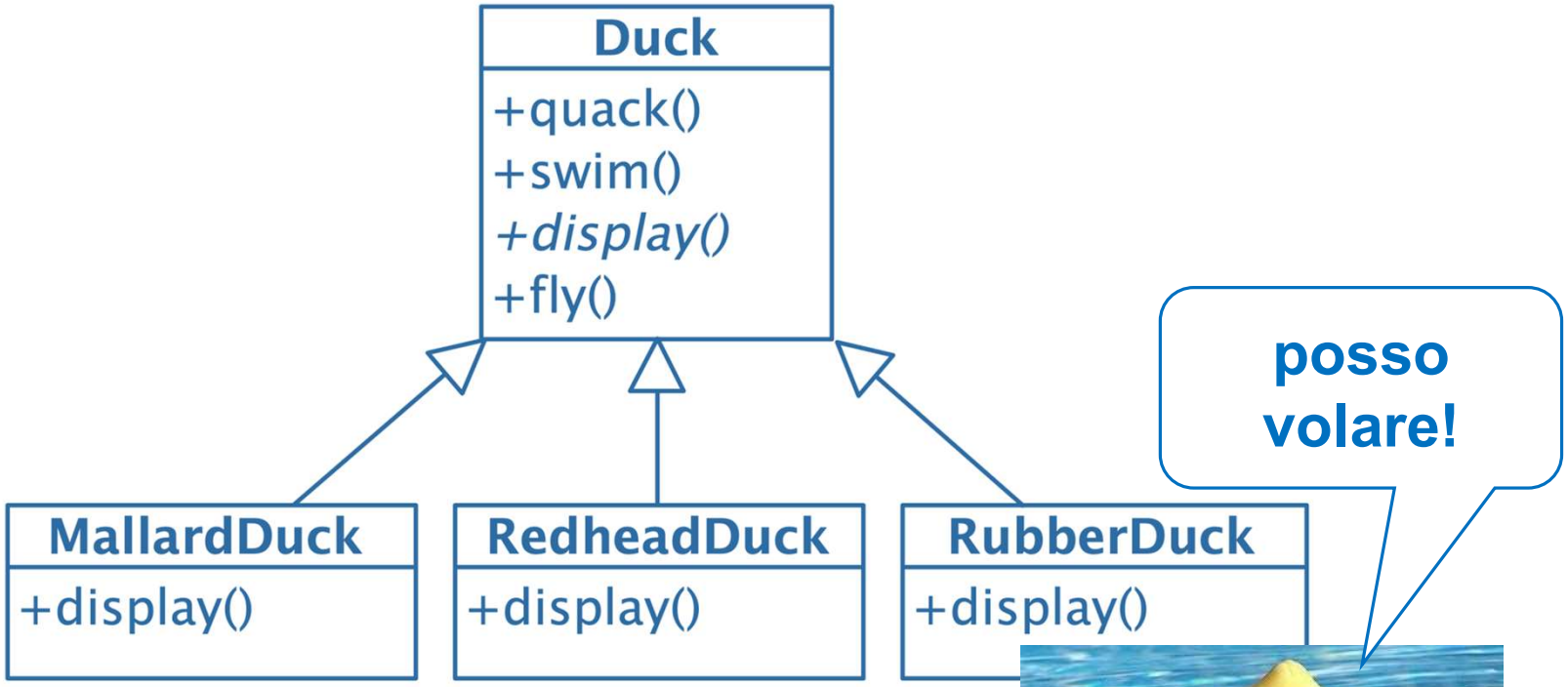


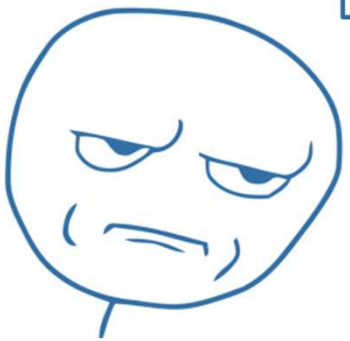


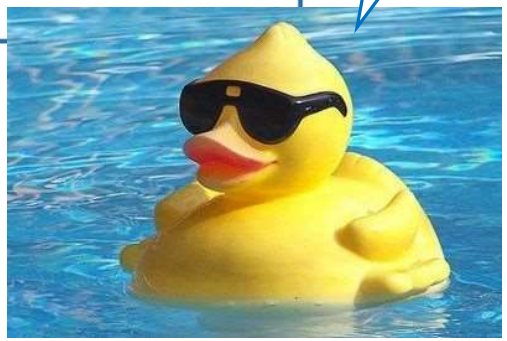
ANCORA PAPERE



ANCHE QUELLA DI GOMMA







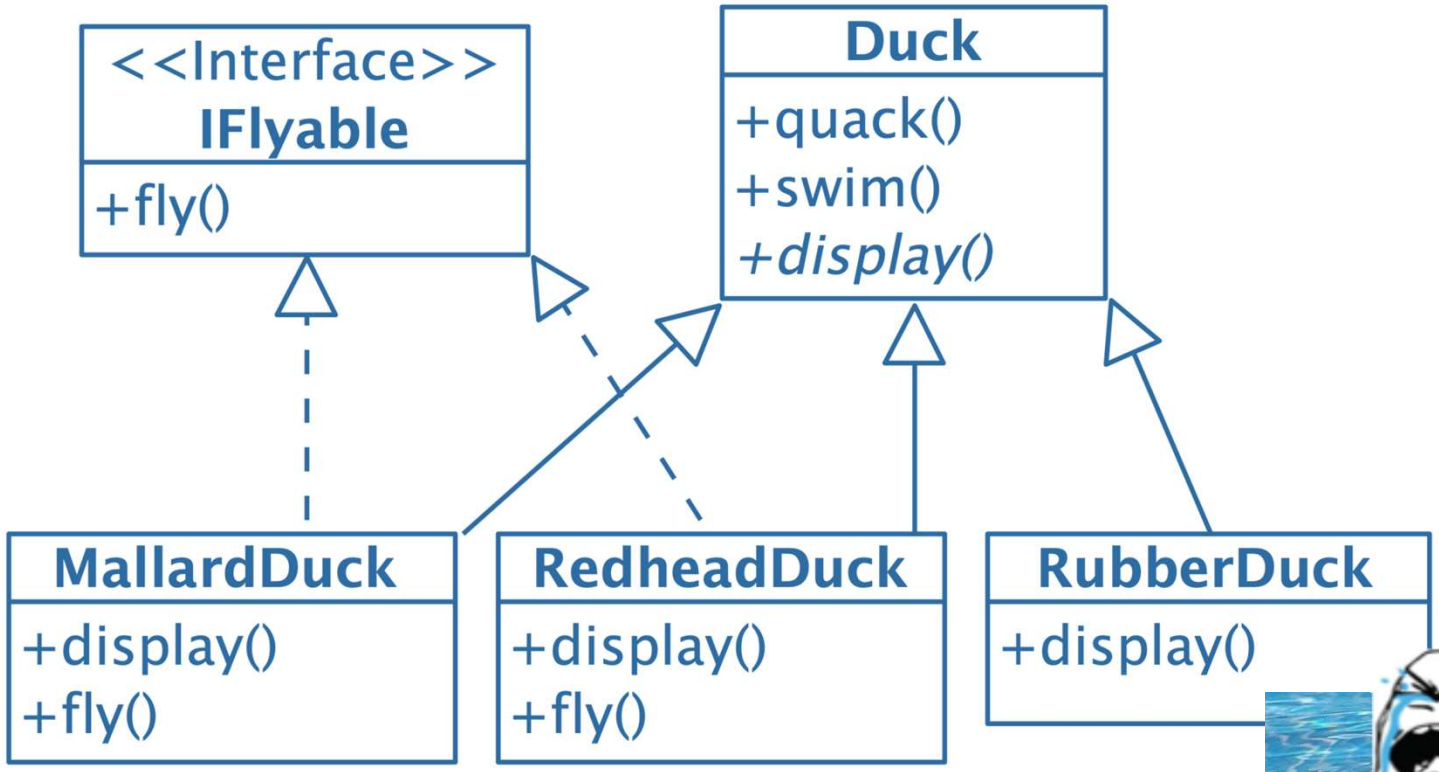
UNA PRIMA SOLUZIONE: OVERRIDE

```
class Rubberduck{

  fly() {
    \\ do nothing
  }

  quack(){
    \\ override to squeak
  }

}
```

Problemi

- Estensione di classi di cui si usa **solo una parte** del comportamento
- **Ogni volta** che aggiungiamo una nuova papera
    - Verificare se `fly` va sovrascritto
    - Verificare se `quack` va sovrascritto

UN'ALTRA SOLUZIONE: INTERFACCE



MA È LA SOLUZIONE MIGLIORE?

Devo simulare 50 papere

⇒ Vanno scritte 50 istanze del metodo `fly`

Cambia lo stile di volo di 20 papere

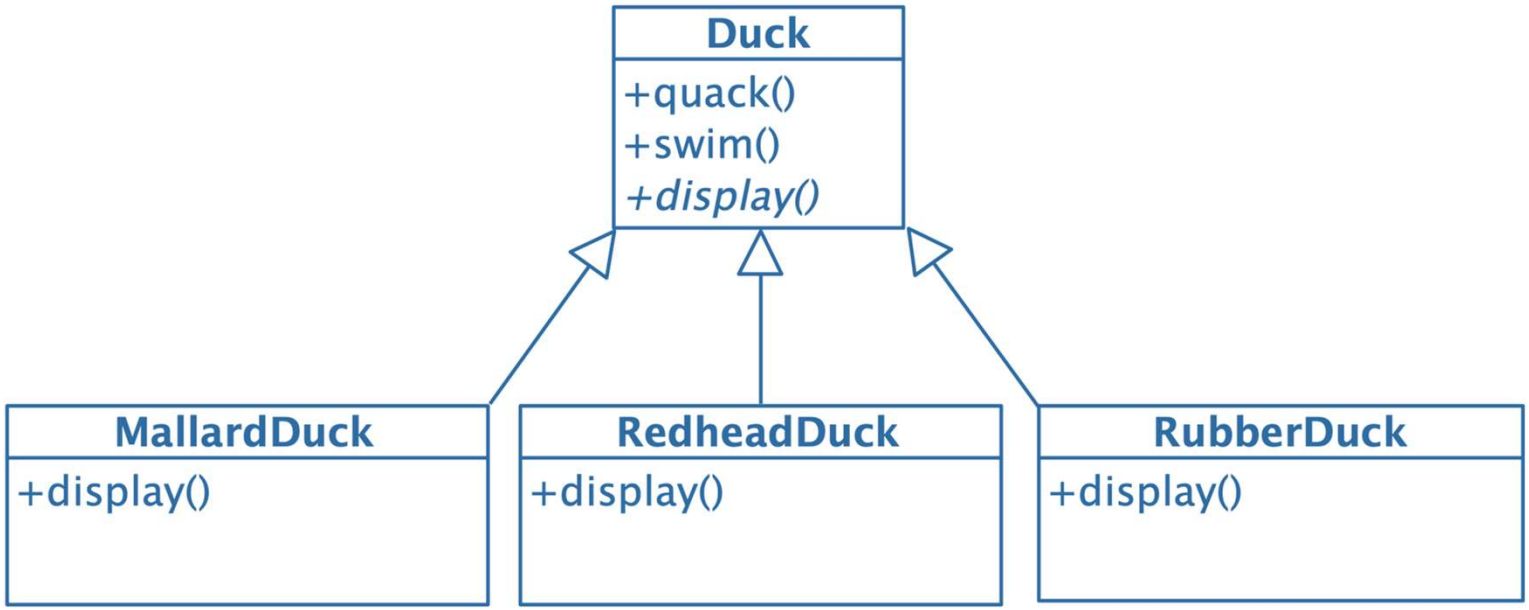
⇒ Vanno riscritte 20 istanze del metodo `fly`

Oggi (ri)scrittura costa! 💸 💸 💸 💸

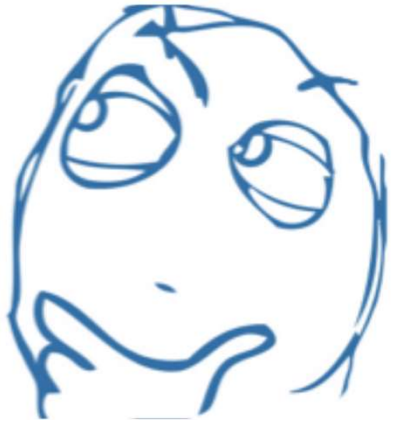
ANALIZZIAMO LA SITUAZIONE

- Non possiamo mettere il metodo `fly` in una classe base
- Se usiamo delle interfacce, non possiamo riusare il codice
- Serve un «dependency split»
    - Volare è un comportamento e dovrebbe essere separato da `Duck`
    - Diversi comportamenti di volo potrebbero essere riusati da oggetti diversi
    - Papere diverse potrebbero volare in modi diversi

SERVE UNA «STRATEGIA»



come separare la responsabilità di `fly`?



SERVE UNA «STRATEGIA» (CONT.)

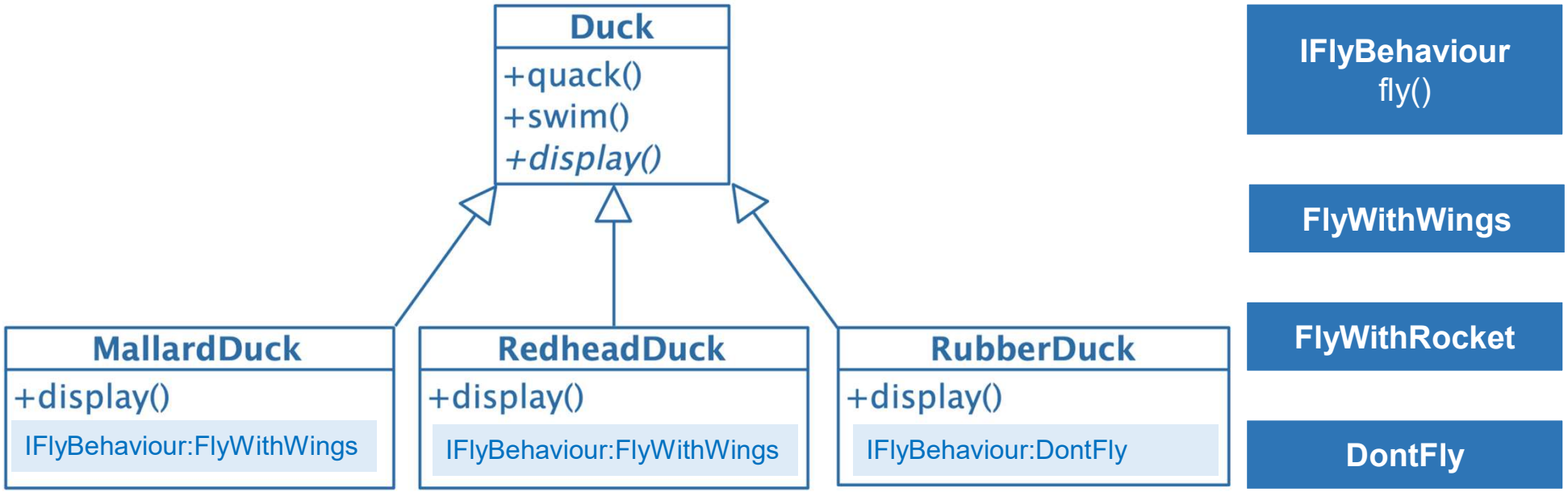
Richiamiamo alcuni principi di progettazione

Identify what vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don’t

Program to an interface, not an implementation

Favor composition over inheritance

SERVE UNA «STRATEGIA» (CONT.)



LO STRATEGY DESIGN PATTERN

Definire una **famiglia di algoritmi**, incapsularli separatamente e renderli intercambiabili

Lo **strategy** consente agli algoritmi di variare indipendentemente dai clienti che li usano

Un programma potrebbe dover fornire **più varianti di un algoritmo/comportamento**

- Variazioni sono incapsulate in **classi separate**
- Metodo **uniforme** per l'accesso alle classi

LO STRATEGY DESIGN PATTERN (CONT.)

Strategy (partecipante)

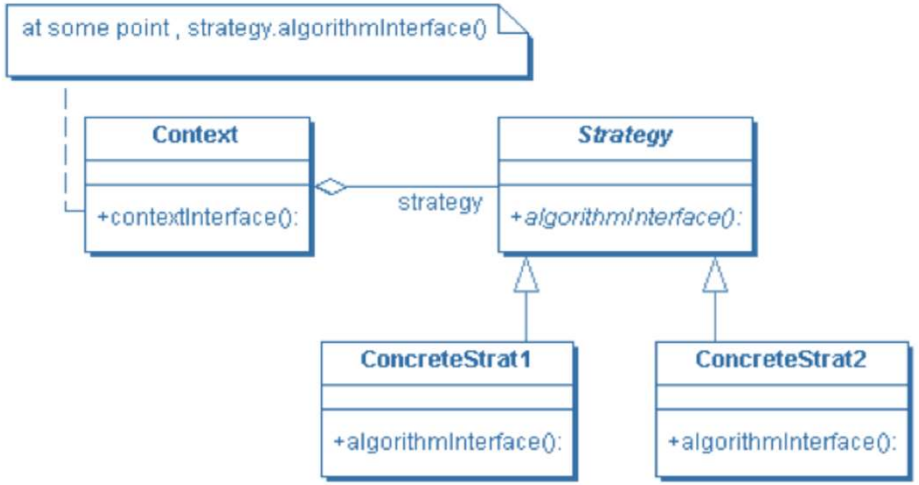
- Definisce un'interfaccia comune per gli algoritmi

ConcreteStrategy (partecipante)

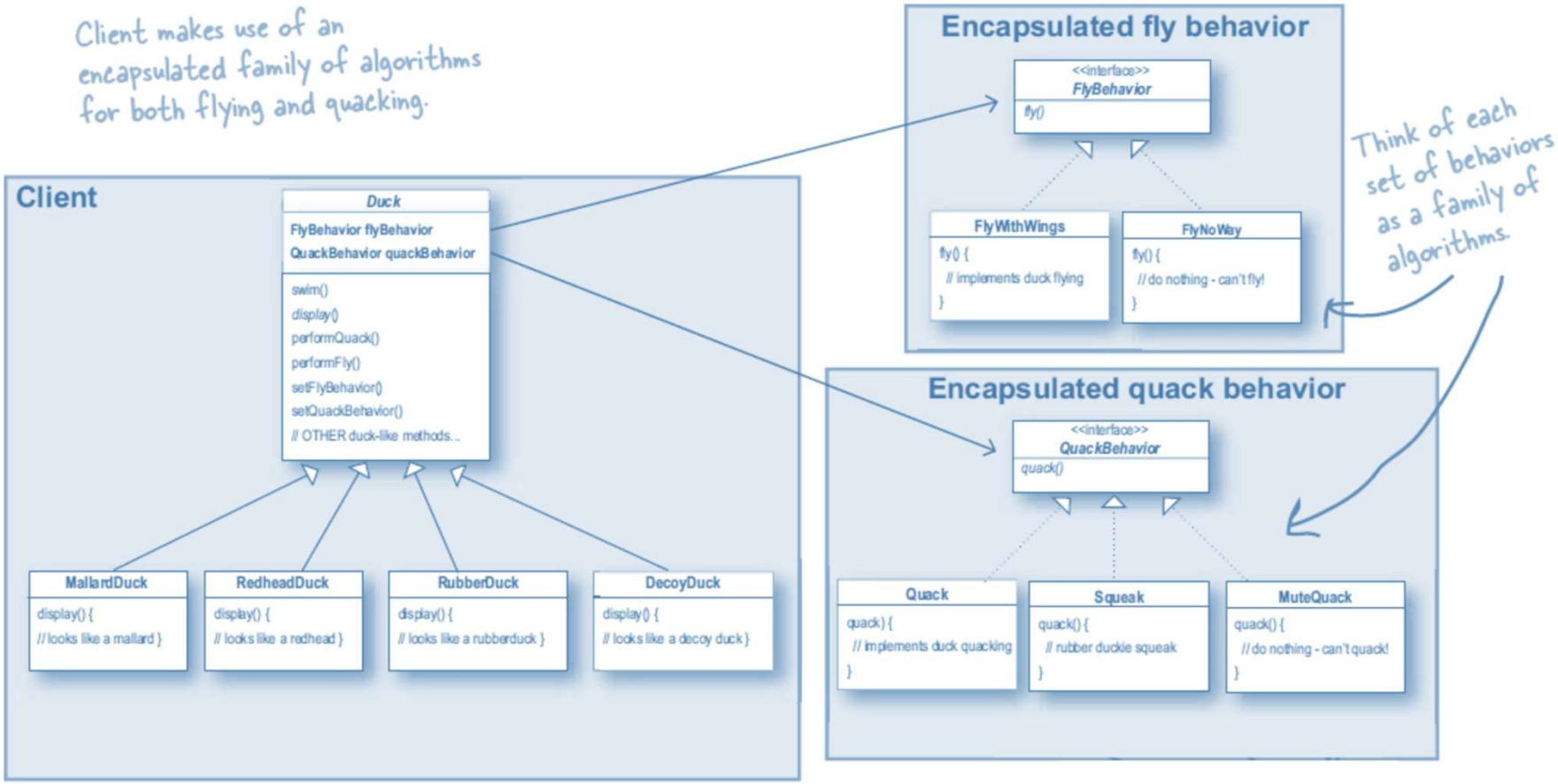
- Ogni strategia concreta implementa un algoritmo

Context (partecipante)

- Contiene un riferimento ad un oggetto **Strategy**
- Può definire un'interfaccia che consente alla strategia di accedere ai dati di interesse (invece di passarli come argomenti quando viene invocata la strategia stessa)



DI NUOVO LE PAPERE



UN ALTRO ESEMPIO

La classe `MyArray` rappresenta un vettore di numeri

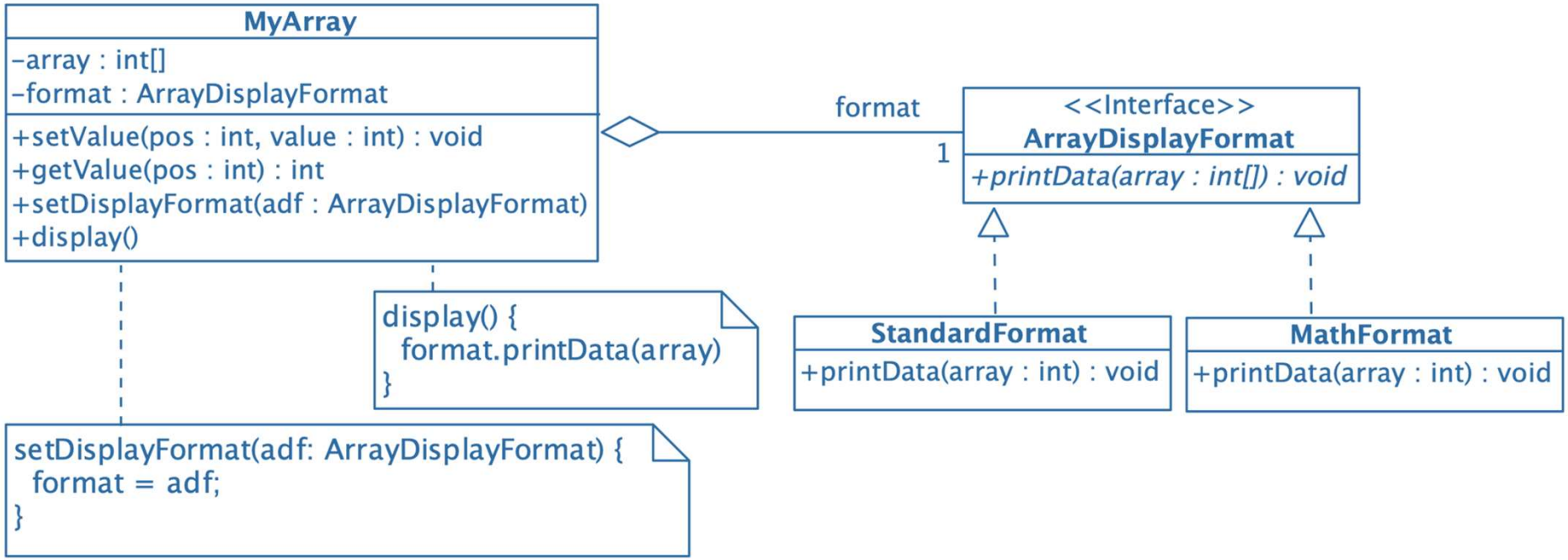
- Uno dei suoi metodi stampa l'array

- Due formati di stampa attualmente supportati
  - MathFormat (es. {12, -7, 3, ...})
  - StandardFormat (es. ar[0] = 12, ar[1] = -7, ar[2] = 3, …)

- I formati potrebbero cambiare in futuro (estesi o sostituiti da altri)

Come isolare l'algoritmo di formattazione, così che possa variare in modo indipendente dagli altri metodi offerti dalla classe?

SOLUZIONE, CON STRATEGY DP



SOLUZIONE: CONTEXT

```
public class MyArray {
  private int[] array;
  private int size;
  ArrayDisplayFormat format; // placeholder for strategy

  public MyArray(int size) { array = new int[size]; }
  public void setValue(int pos, int value) { array[pos] = value; }
  public int getValue(int pos) { return array[pos]; }
  public int getLength() { return array.length; }

  // set and use strategy
  public void setDisplayFormat(ArrayDisplayFormat adf) { format = adf; }
  public void display() { format.printData(array); }
}
```

SOLUZIONE: STRATEGY

```
// interface
public interface ArrayDisplayFormay {
  public void printData(int[] arr);
}

// first concrete strategy
public class StandardFormat implements ArrayDisplayFormat {
  public void printData(int[] arr) {
    System.out.println("{");
    for(int i=0; i<arr.length-1; i++)
      System.out.println(arr[i] + ",");
    System.out.println(arr[arr.length-1] + "}");
  }
}
```

SOLUZIONE: STRATEGY (CONT.)

```
// second concrete strategy
public class MathFormat implements ArrayDisplayFormat {
  public void printData(int[] arr) {
    for(int i=0; i<arr.length-1; i++)
      System.out.println("Arr[" + i + "] = " + arr[i]);
  }
}
```

COME FUNZIONA?

Il **cliente** crea e passa una **strategia concreta** al **contesto** (e poi interagisce solo con il contesto)

```
public class StrategyExample {
  public static void main(String[] args) {
    MyArray m = new MyArray(10);
    m.setValue(1,6); m.setValue(0,8); m.setValue(4,1); m.setValue(9,7);
    // standard format
    System.out.println("This is the array in 'standard' format");
    m.setDisplayFormat(new StandardFormat());
    m.display();
    // math format
    System.out.println("This is the array in 'math' format");
    m.setDisplayFormat(new MathFormat());
    m.display();
  }
}
```

IL RISULTATO

```
This is the array in 'standard' format
{ 8, 6, 0, 0, 1, 0, 0, 0, 0, 7 }

This is the array in 'math' format
Arr[0] = 8
Arr[1] = 6
Arr[2] = 0
Arr[3] = 0
Arr[4] = 1
Arr[5] = 0
Arr[6] = 0
Arr[7] = 0
Arr[8] = 0
Arr[9] = 7
```

QUANDO SI USA?

Lo **strategy** pattern è **utile**

- quando più classi (correlate) differiscono **solo nel comportamento**
- quando servono **più varianti** diverse di un algoritmo
- quando un algoritmo usa dei dati che i clienti non conoscono
- per **evitare** di esporre **strutture dati complesse** e **algorithm-specific**

NB: Quando una classe definisce molti comportamenti, e questi comportamenti appaiono come alternative in costrutti condizionali, può essere utile trasformare i branch dei costrutti condizionali in classi ottenute con lo strategy pattern!

OSSERVAZIONI

Lo **strategy** pattern **consente** di

- **definire comportamenti diversi** fornendo un'alternativa all'estensione della classe Context
- **eliminare costrutti condizionali** estesi (per definire le alternative)
- scegliere **implementazioni diverse** per uno stesso task

Questo viene al **prezzo** di

- **incrementare il numero di oggetti** presenti in un'implementazione
- garantire che le implementazioni alternative rispettino la **stessa interfaccia**

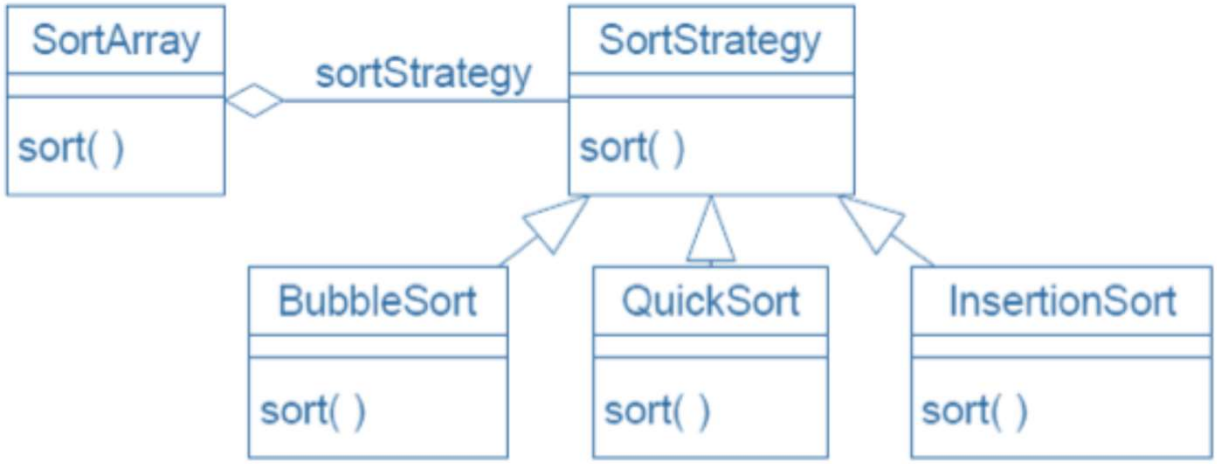
UN ESEMPIO

Problema:

- Decidere a runtime quale algoritmo utilizzare per l'ordinamento di un array
- Sono disponibili più algoritmi di ordinamento

Soluzione:

- Incapsulare gli algoritmi di ordinamento disponibili mediante lo strategy pattern



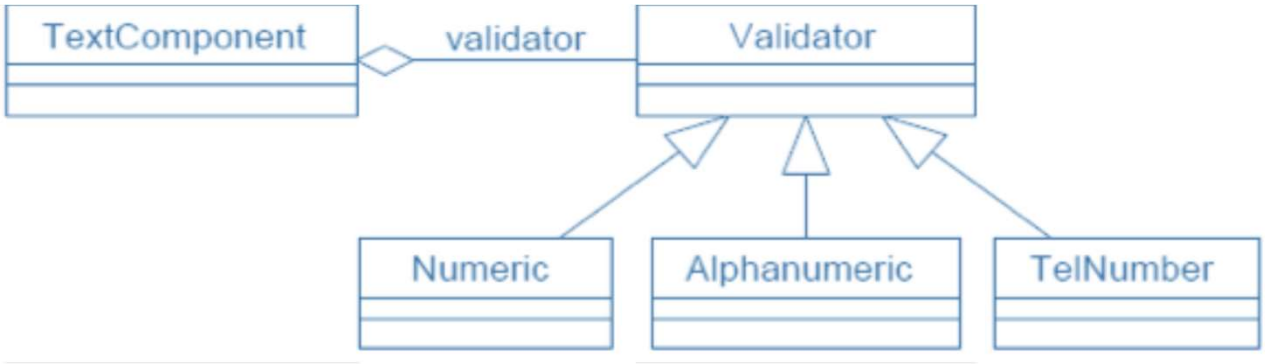
UN ALTRO ESEMPIO

Problema:

- Decidere a runtime come validare i dati inseriti una textbox di una GUI
- Diversi metodi di validazione disponibili (per campi numerici, alfanumerici, o numeri telefonici)

Soluzione:

- Incapsulare i metodi di validazione disponibili in strategie diverse



NB: Questa è la tecnica utilizzata da Java Swing per i componenti testuali (ogni componente ha un riferimento ad un document model che fornisce la strategia di validazione richiesta)

STRATEGIE DIVERSE, DATI DIVERSI (?)

Strategie diverse potrebbero operare su dati diversi!

Può accadere che alcune strategie concrete non utilizzino tutti i dati passati tramite l'interfaccia

- Il contesto crea e inizializza parametri che non saranno utilizzati

- Workaround:
  - Coupling maggiore tra strategie concrete e contesto
  - Strategie concrete che chiedono al contesto solo i dati di cui hanno bisogno

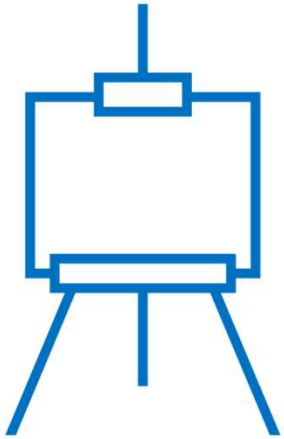
HOMEWORK

Pizzami vende pizze e offre sconti ai propri clienti. Ci sono diversi tipi di sconti, quali

- 10% di sconto
- sconto di €5 ogni €20 spesi
- 3 pizze al prezzo di 2
- 20% di sconto sulla pizza con l'ananas

Pizzami ci chiede di sviluppare un sistema di gestione delle vendite. In particolare, richiedono di progettare uno schema per calcolare lo sconto automaticamente quando vengono vendute le pizze. Il sistema deve consentire la selezione del tipo di sconto da applicare (tra quelli elencati sopra). Inoltre, quando **Pizzami** vuole aggiungere un nuovo tipo di sconto o modificarne uno esistente, il cambiamento deve essere facile da implementare e non deve «intaccare» il sistema.

Discutere inoltre le problematiche riguardanti i dati forniti. Ad esempio, cosa succede con sconti del tipo «riduci €5 ad ogni acquisto» o «riduci €5 al primo acquisto»?



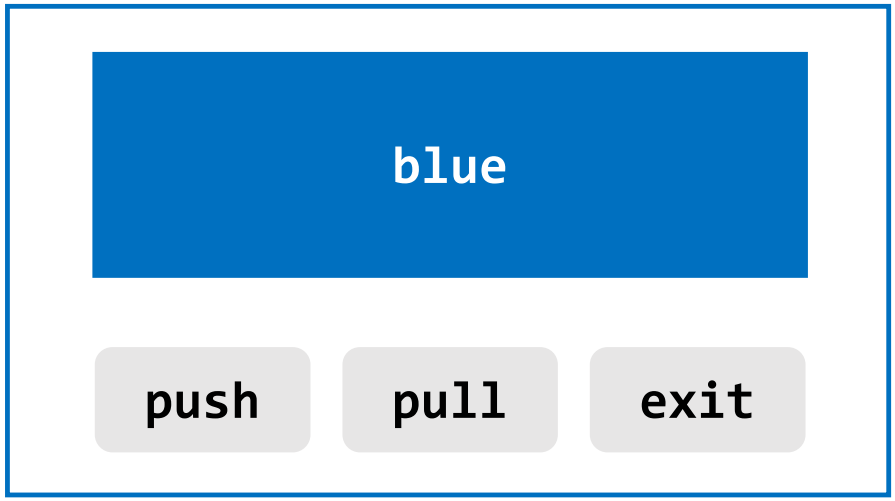
STATE

ESEMPIO «A COLORI»

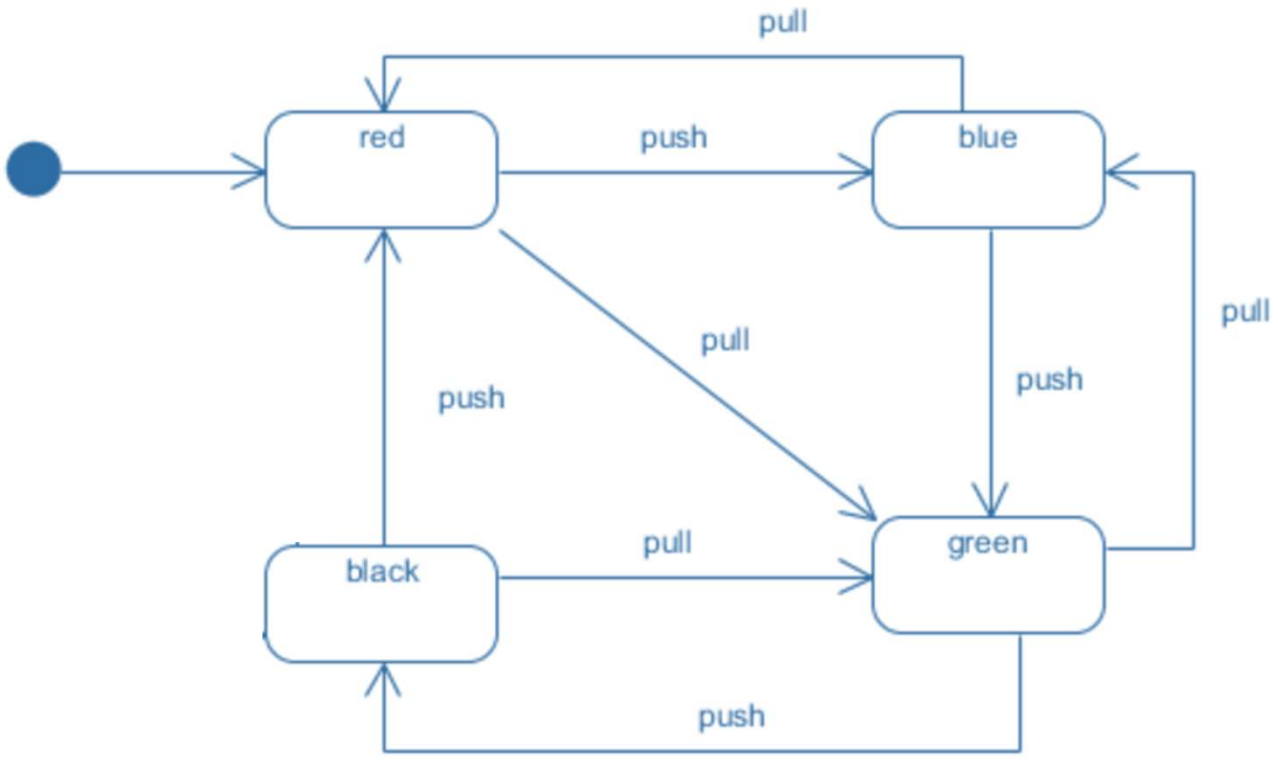
Consideriamo una classe che offre due metodi (**`push`** e **`pull`**), il cui comportamento varia in base all'effettivo stato di un oggetto

Supponiamo di avere a disposizione una GUI per

- mandare le richieste di **`push`**/**`pull`** (attraverso bottoni omonimi) e
- visualizzare lo stato dell'oggetto (nero, rosso, blu e verde)



ESEMPIO «A COLORI» (CONT.)

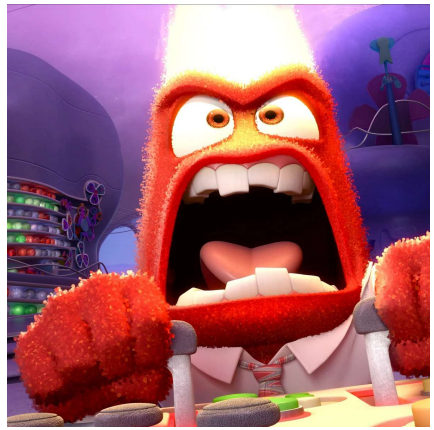


COME REALIZZARLO, SENZA STATE PATTERN

```
/**
 * Class ContextNoSP has behavior dependent on its state. The push() and pull()
 * methods do different things depending on the state of the object.
 * This class does NOT use the State pattern
 */
public class ContextNoSP {
  // The state!
  private Color state = null;

  // Creates a new ContextNoSP with the specified state (color)
  public ContextNoSP(Color color) { state = color; }

  // Creates a new ContextNoSP with the default state
  public ContextNoSP() { this(Color.red); }

 // Returns the state.
  public Color getState() { return state; }

  // Sets the state
  public void setState(Color state) { this.state = state; }
```

COME REALIZZARLO, SENZA STATE PATTERN (CONT.)

```
  /* The push() method performs different actions depending on the state of
     the object (right now the only action is to make a state transition) */
  public void push() {
    if (state == Color.red) state = Color.blue;
    else if (state == Color.green) state = Color.black;
    else if (state == Color.black) state = Color.red;
    else if (state == Color.blue) state = Color.green;
  }

  /* The pull() method performs different actions depending on the state of
     the object (right now the only action is to make a state transition) */
  public void pull() {
    if (state == Color.red) state = Color.green;
    else if (state == Color.green) state = Color.blue;
    else if (state == Color.black) state = Color.green;
    else if (state == Color.blue) state = Color.red;
  }

}
```



COME REALIZZARLO, SENZA STATE PATTERN (CONT.)

```
/* Test program for the ContextNoSP class (NOT using the State pattern) */
public class TestNoSP extends Frame implements ActionListener {
  // GUI attributes
  private Button pushButton = new Button("Push Operation");
  private Button pullButton = new Button("Pull Operation");
  private Button exitButton = new Button("Exit");
  private Canvas canvas = new Canvas();

  // The Context
  private ContextNoSP context = null;

  // Setup
  public TestNoSP() {
    super("No State Pattern");
    context = new ContextNoSP();
    setupWindow();
  }
  private void setupWindow() { // Setup GUI }
```

COME REALIZZARLO, SENZA STATE PATTERN (CONT.)

```
// Handle GUI actions
public void actionPerformed(ActionEvent event) {
  Object src = event.getSource();
  if (src == pushButton) {
    context.push();
    canvas.setBackground(context.getState());
  }
  else if (src == pullButton) {
    context.pull();
    canvas.setBackground(context.getState());
  }
  else if (src == exitButton) {
    System.exit(0);
  }
}

// Main method
public static void main(String[] argv) {
   TestNoSP gui = new TestNoSP();
   gui.setVisible(true);
}
```

LO STATE DESIGN PATTERN

Si tratta di un pattern comportamentale (**behavioural pattern**)

Consente ad un oggetto di **alterare** il suo **comportamento** quando **cambia lo stato** (interno)

- Quasi come se l'oggetto avesse cambiato «classe»

Un esempio concreto viene dalle connessioni TCP

- Risposte diverse fornite ai clienti in base allo stato di una connessione
- Stati possibili: established, listening, closed

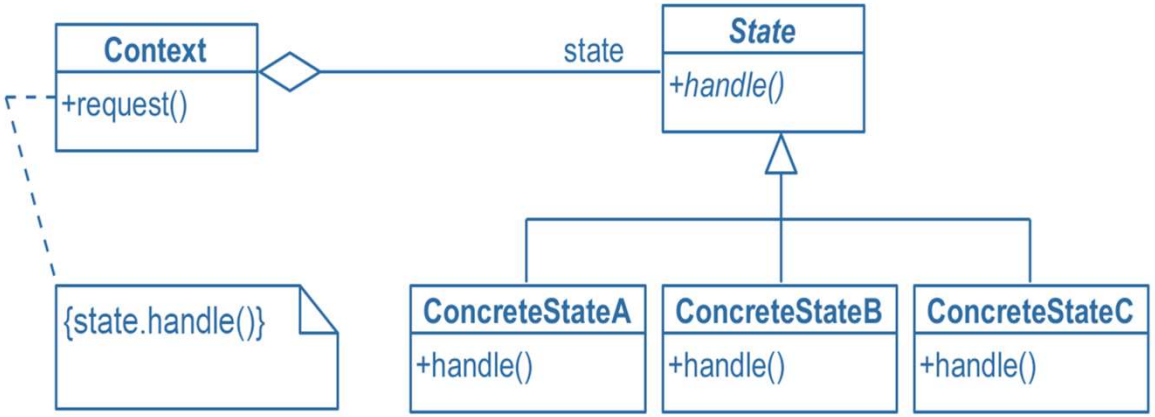
LO STATE DESIGN PATTERN (CONT.)

Il **contesto** (context)

- definisce l'**interfaccia** di interesse per i client
- mantiene un'istanza di **ConcreteState** che definisce lo stato corrente

Lo **stato** (state)

- **incapsula il il comportamento** associato a un particolare stato del contesto
- può anche essere una classe concreta con un'implementazione predefinita



Gli **stati concreti** (concrete state) sono sottoclassi dello stato che implementano il comportamento associato allo stato che rappresentano

L'ESEMPIO «A COLORI», CON STATE

```
/*
 * Abstract class which defines the interface for the behavior of a particular
 * state of the Context.
*/

public abstract class State {

	public abstract void handlePush(Context c);

	public abstract void handlePull(Context c);

	public abstract Color getColor();

}
```

L'ESEMPIO «A COLORI», CON STATE (CONT.)

```
/*
 * Concrete State classes for all the different states
 */
public class BlackState extends State {
  public void handlePush(Context c) {
    c.setState(new RedState()); // On a push(), go to "red"
  }
  public void handlePull(Context c) {
    c.setState(new GreenState()); // On a pull(), go to "green"
  }
  public Color getColor() { return (Color.black); }
}

public class RedState extends State { // Similar to above }

public class BlueState extends State { // Similar to above }

public class GreenState extends State { // Similar to above }
```

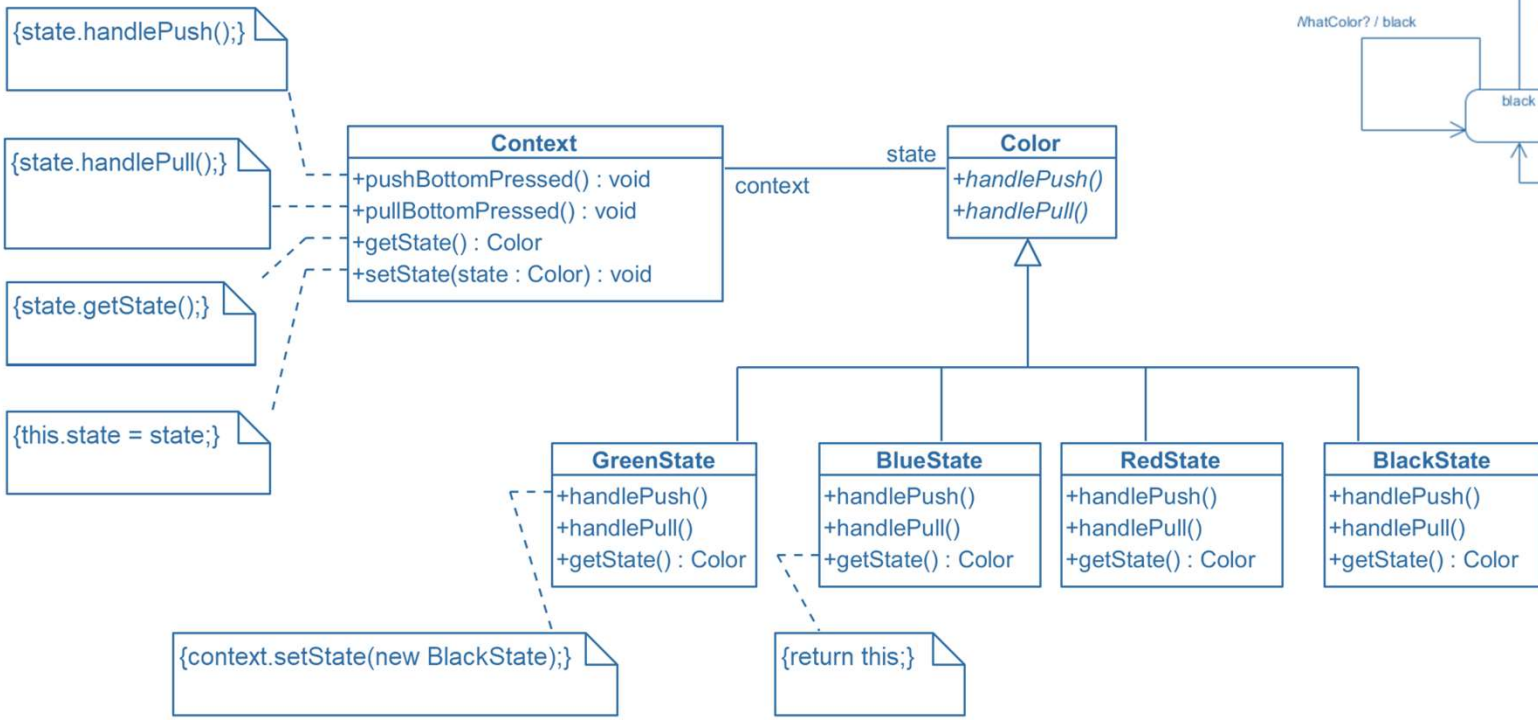
L'ESEMPIO «A COLORI», CON STATE (CONT.)

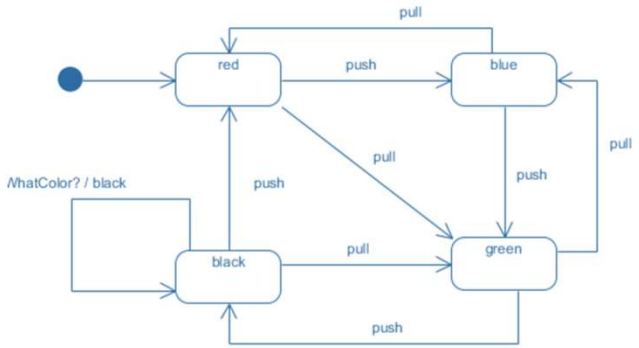
```
/**
 * Class Context has behavior dependent on its state. This class uses the State
 * pattern (delegating pull/push to the contained state object)
 */
public class Context {
  // The contained state
  private State state = null;  // State attribute

  // Creates a new Context with the specified state
  public Context(State state) { this.state = state; }

  // Creates a new Context with the default state
  public Context() { this(new RedState()); }

  // Returns the state
  public State getState() { return state; }

  // Sets the state
  public void setState(State state) { this.state = state; }
```

L'ESEMPIO «A COLORI», CON STATE (CONT.)

```
  /**
   * The push() method performs different actions depending on the state of
   * the object. Using the State pattern, we delegate this behavior to our
   * contained state object.
   */
  public void push() { state.handlePush(this); }

  /**
   * The pull() method performs different actions depending on the state of
   * the object. Using the State pattern, we delegate this behavior to our
   * contained state object.
   */
  public void pull() { state.handlePull(this); }

}
```

L'ESEMPIO «A COLORI», CON STATE (CONT.)





OSSERVAZIONI

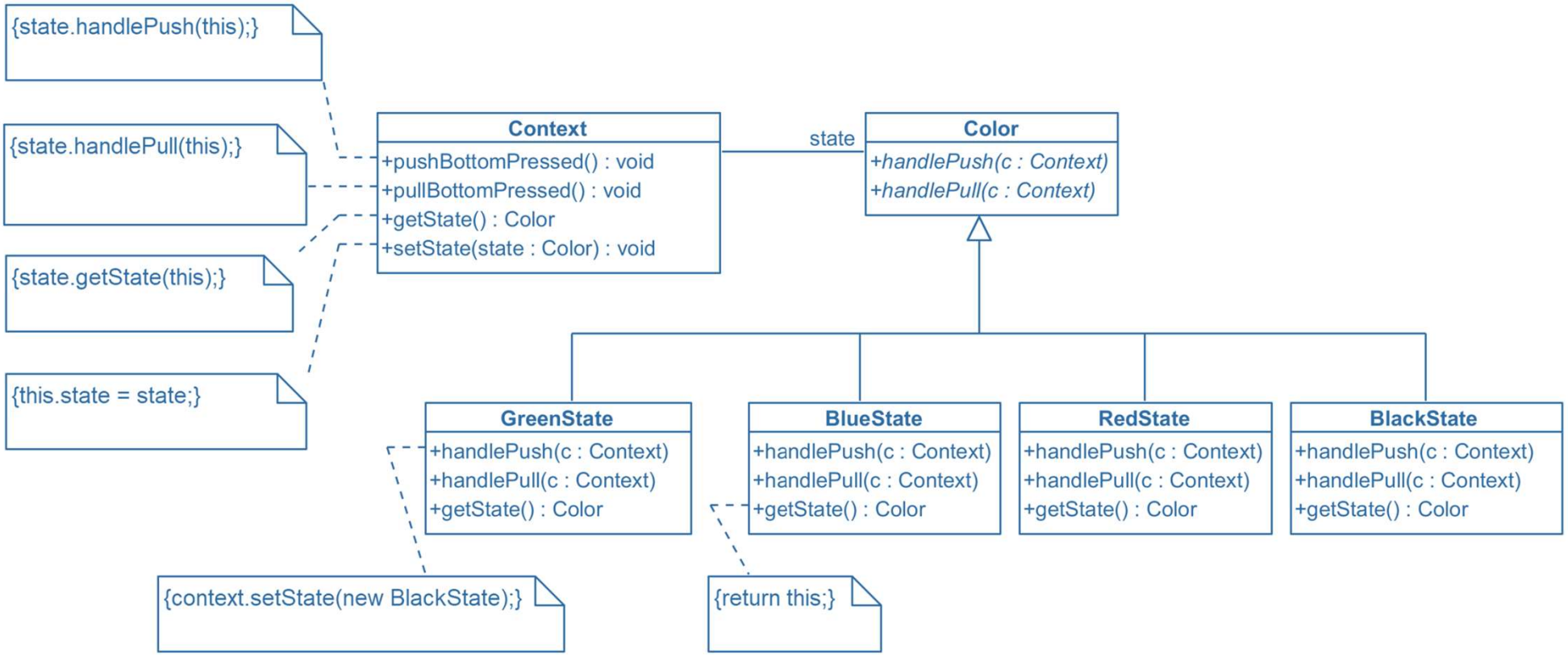
- L'invocazione dei metodi push() e pull() porta **solo a una transizione di stato**
  - nessun'altra azione viene eseguita
  - questo dipende dall'esempio scelto e potrebbe non valere in altri casi

- Gli **stati concreti** definiscono la **transizione di stato**
  - Conoscono il loro «next state» (soluzione più flessibile, ma causa dipendenze tra le classi «concrete state»)
  - Potrebbe essere il context a cambiare stato (nelle situazioni più semplici)

- Ogni volta che si **cambia stato** si crea un **nuovo oggetto** per rappresentare lo stato raggiunto
  - Questa è una scelta: creare oggetti «state» solo quando sono necessari e distruggerli in seguito
    (ok, se i cambiamenti sono poco frequenti)
  - In caso di cambiamenti frequenti, meglio creare gli oggetti «state» in anticipo e riusarli (senza distruggerli)

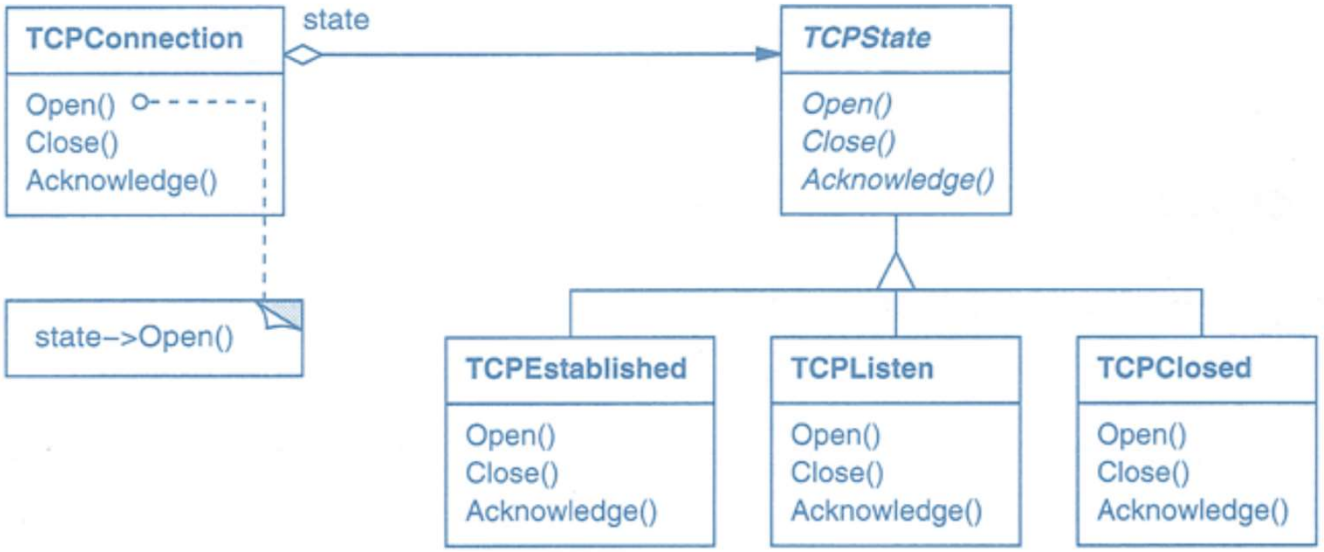
- Gli oggetti di tipo Color hanno una variabile (final) context
  - Anche questa è una scelta
  - Context può essere passato per riferimento

VARIANTE: CONTEXT PASSATO COME ARGOMENTO



UN ALTRO ESEMPIO: TCP

- Il **`TCPState`** è astratto e definisce l'interfaccia



- Le sottoclassi di **`TCPState`** rappresentano gli stati concreti

UN ALTRO ESEMPIO: TCP (CONT.)

Stato come interfaccia o classe?

- Possiamo rappresentarlo come una **classe concreta** che fornisce un'implementazione **default**

```
class TCPState {
  TCPState() { }
  void open(TCPConnection t) { }
  void close (TCPConnection t) { }
  void acknowledge(TCPConnection t, TCPOctetStream os) { }
}
```

- Le **sottoclassi** poi implementano il **comportamento specifico** di ciascuno stato

```
class TCPClosed extends TCPState {
  ...
  void open(TCPConnection t) {
    // send SYN, receive SYN, ACK, etc.
    ChangeState(t, new TCPEstablished());
  }
}
```

QUANDO SI USA?

Lo **state** pattern è **utile** nei seguenti casi:

- Il **comportamento** di un oggetto **dipende** dal suo **stato**
  - Diversi comportamenti possibili a runtime
  - Ciascun comportamento determinato dallo stato a runtime

- Le operazioni hanno **dichiarazioni condizionali complesse**
  - Comportamenti alternativi definiti dalle condizioni
  - La scelta del comportamento dipende dallo stato dell'oggetto

LO STATE PATTERN IN 6 PASSI

1. Identificare/creare una **classe** che funga da «**macchina a stati**» dal punto di vista del cliente
   - Questa classe è la classe di contesto (**context**)
2. Creare una classe **state** che replichi i metodi dell'interfaccia della macchina a stati
   - Ogni metodo richiede un parametro aggiuntivo: un'istanza della classe di contesto
   - La classe state specifica qualsiasi comportamento utile «predefinito»
3. Creare una **sottoclasse** della classe **state** per ogni stato del dominio
   - Ogni sottoclasse sovrascrive solo i metodi che cambiano con lo stato
4. La classe di **contesto** mantiene lo **stato corrente**
   - Lo stato corrente è un oggetto della classe **state**
5. Le richieste dei client sono **delegate** dalla classe di contesto allo **stato corrente**
   - Passate all'oggetto della classe state
   - Si passa anche il puntatore `this` all'oggetto di contesto
6. I metodi della classe state modificano lo stato corrente
   - Modifica dell'oggetto di contesto passato

OSSERVAZIONI

- State «localizza» il comportamento che dipende dallo stato
  - Il comportamento è suddiviso tra i possibili stati
  - Le transizioni di stato sono esplicite
- Tutti i comportamenti specifici di uno stato sono memorizzati in una classe
  - L'alternativa è costituita da «valanga» di casi (condizionali) alternativi
  - Può produrre un gran numero di classi, ma è meglio dell'alternativa
- Lo stato corrente è memorizzato in un'unica posizione
- Gli oggetti della classe state possono essere condivisi
  - Quando non hanno variabili di istanza
- La classe **state** può implementare parte del comportamento
  - Se c'è del comportamento comune o di default

FAQ

Non possiamo usare una **tabella di transizione** degli stati per tutto questo?

Più **difficile** da capire e da estendere con altre azioni o comportamenti

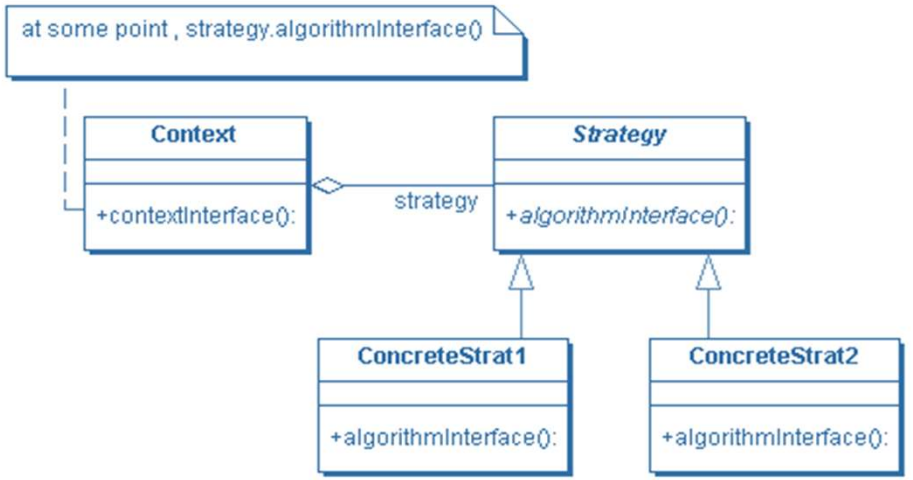
STATE VS STRATEGY

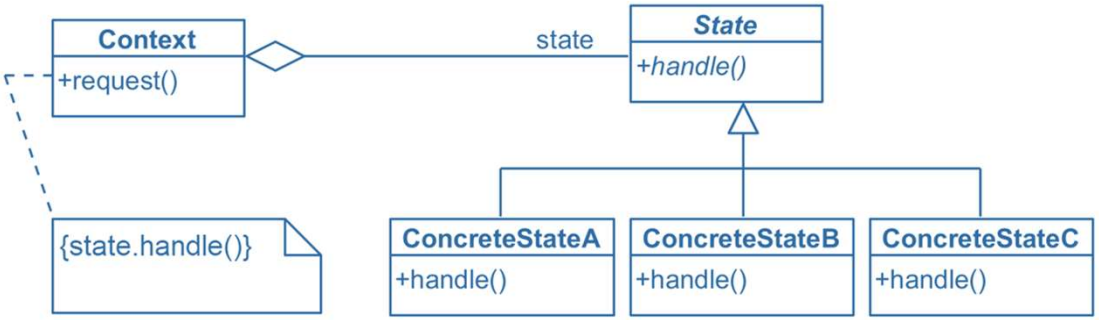
Ci sono similitudini tra i pattern **state** e **strategy**!

- Ad esempio, sono entrambi basati su **composizione con delega**

La differenza sta nell'**intento**

- Un oggetto **strategy** incapsula (una versione) di un **algoritmo**
- Un oggetto **state** incapsula un **comportamento** dipendente dallo stato (ed eventualmente transizioni di stato)
- Inoltre, in state un oggetto **context** cambia stato in base a transizioni di stato predefinite





HOMEWORK

Completare l'esempio «a colori» **implementando le differenti varianti** con i seguenti **vincoli**:

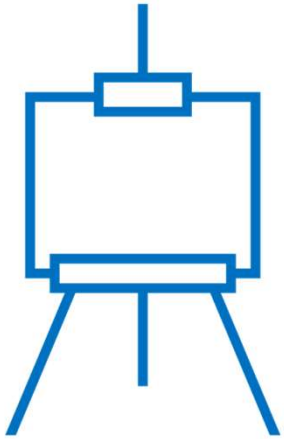
1. Creare gli oggetti **state** in **anticipo** e senza distruggerli ogni volta

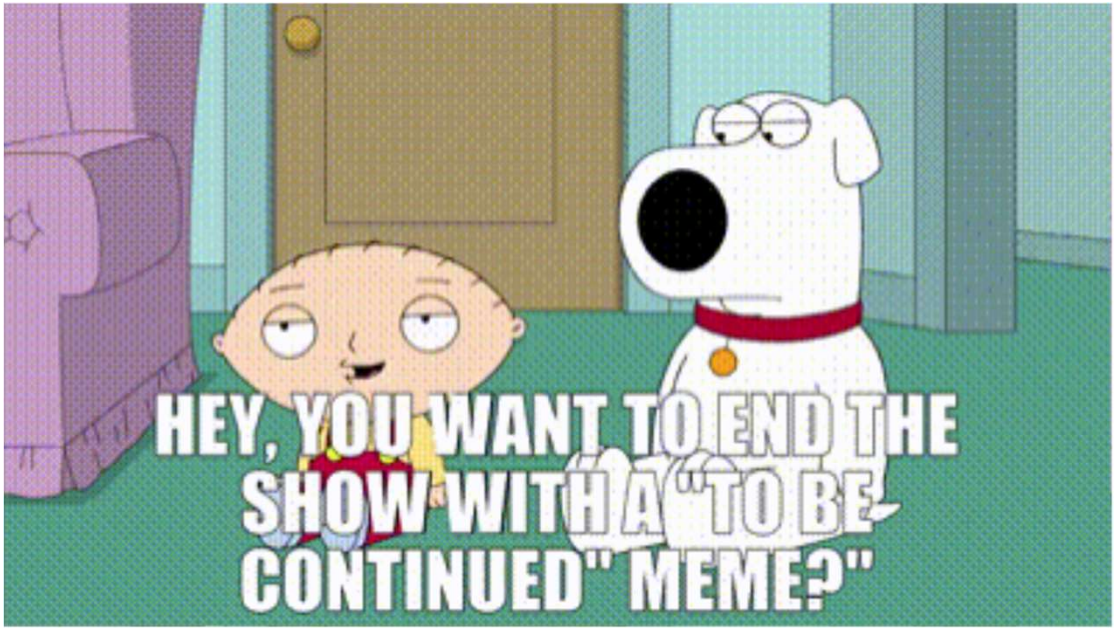
2. Lasciare al **contesto** la decisione sul flusso delle transizioni di stato

Attenzione: ci sono due differenti tipi di funzionalità che devono essere gestiti

- ```
  // handle state change: you can have different strategies here
  handlePush(Context c) {…}
  handlePull(Context c) {…}
  ```

- ```
  // handle a request that depends on the state: no variability in
  implementation, states deal with it
  public Color getColor() {…}
  ```




HEY, YOU WANT TO END THE SHOW WITH A "TO BE CONTINUED" MEME?"


Q
A

RIFERIMENTI

Contenuti

- **Capitolo 10** di "Software Engineering" (G. C. Kung, 2023) // design pattern
- **Sezione 13.5** di "Software Engineering" (G. C. Kung, 2023) // state pattern
- **Sezione 16.3** di "Software Engineering" (G. C. Kung, 2023) // strategy pattern

Approfondimenti