

DESIGN PATTERN E ANTI-PATTERN

Angelo Di Iorio

Università di Bologna

Design Pattern in Java

- I Design Pattern sono metodi di risoluzione a problemi ricorrenti
- Favoriscono il riuso di soluzioni che si sono dimostrate efficaci e/o efficienti
- Esistono in molti contesti e non sono stati inventati in ambito informatico ma in architettura
- Per lo sviluppo software, un testo fondamentale:
Gamma, Helm, Johnson, Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1994
- Questi progettisti saranno poi conosciuti come la “Gang of Four” – GoF



Perchè usare i pattern?

- I pattern risolvono **problemi reali e ricorrenti**.
- Catturano e trasmettono l'esperienza di progettisti esperti che li hanno utilizzati per realizzare soluzioni reali d'eccellenza, e promuovono **buone prassi progettuali**.
- Costituiscono un **vocabolario comune** per discutere sulle soluzioni dei problemi (ad esempio, ma non solo) durante lo sviluppo di un progetto
- Supportano la documentazione e il riuso, e permettono di capire più facilmente il funzionamento di un sistema software

Un passo indietro: Alexander

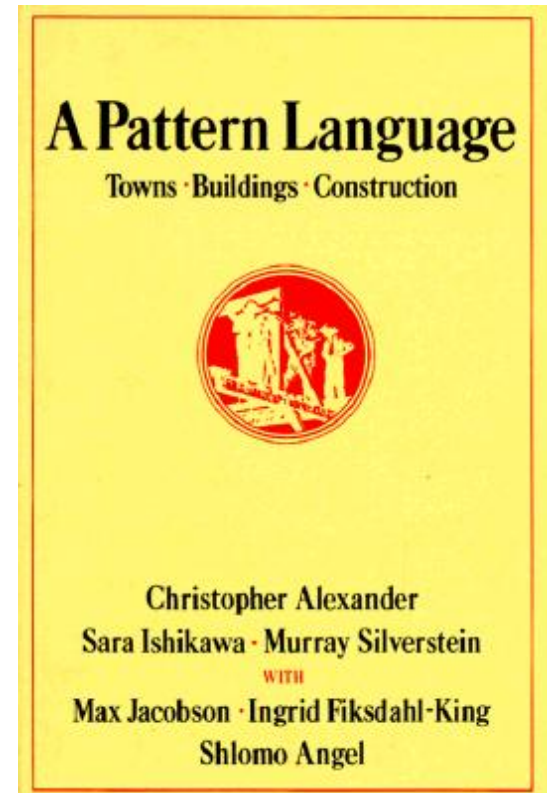
«Gli elementi di questo **linguaggio** sono entità chiamate pattern.

Ogni pattern riguarda un **problema** che si presenta in modo **ricorrente** nel nostro ambiente

e ne descrive il **nucleo di una soluzione**

in modo tale che sia possibile usare questa stessa soluzione un milione di volte senza **mai** realizzarla allo stesso modo.»

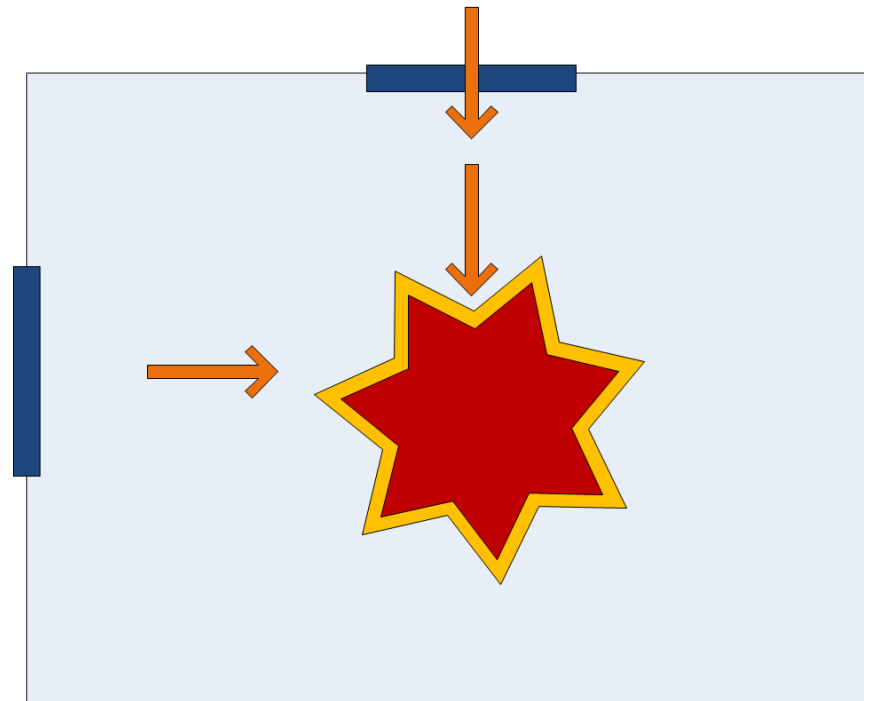
(C. Alexander e altri, “*A Pattern Language: Towns, Buildings, Construction*”, Oxford University Press, New York, 1977)



Esempio: dove posizionare porte in una stanza?

Conrner doors
[196 in Alexander]

“Se le porte creano un pattern di movimento che distrugge le zone di una stanza, la stanza non consentirà mai alle persone di stare comode”



Esempio: le porte

Soluzione:

“... nella maggior parte delle stanze, e in maniera particolare in quelle piccole, metti le porte il più vicino possibile agli angoli della stanza”

“Se una stanza ha due porte, e le persone si muovono attraversandole, mantieni entrambe le porte ad un lato della stanza”



Pattern – schema generale

- I pattern sono solitamente costituiti di quattro parti principali:
 - **Titolo:** il nome del pattern, facile da usare e da comunicare
 - **Problema:** una frase che descrive il problema / intento della soluzione, anche in termini di pro e contro dettagliando le *forze* coinvolte
 - **Forze:** trade-offs, goals + constraints, fattori/interessi in gioco
 - **Soluzione:** forma o regola da applicare per risolvere il problema rispettando le forze in gioco
 - descrive **come** generare la soluzione: struttura, partecipanti e collaborazioni

GoF Design Patterns

- Esistono diversi cataloghi di Design Pattern
- Il nucleo originale dei design pattern proposti dalla *Gang of Four* ne include ventitré, organizzati in tre gruppi in base al tipo di problema che si cerca di risolvere:
 - **Creazionali (Creational)**: si occupano del modo in cui le classi sono istanziate
 - **Strutturali (Structural)**: si occupano del modo in cui le classi e gli oggetti sono composti
 - **Comportamentali (Behavioral)**: si occupano di come gli oggetti comunicano e interagiscono, con l'obiettivo primario di aumentare la flessibilità delle applicazioni

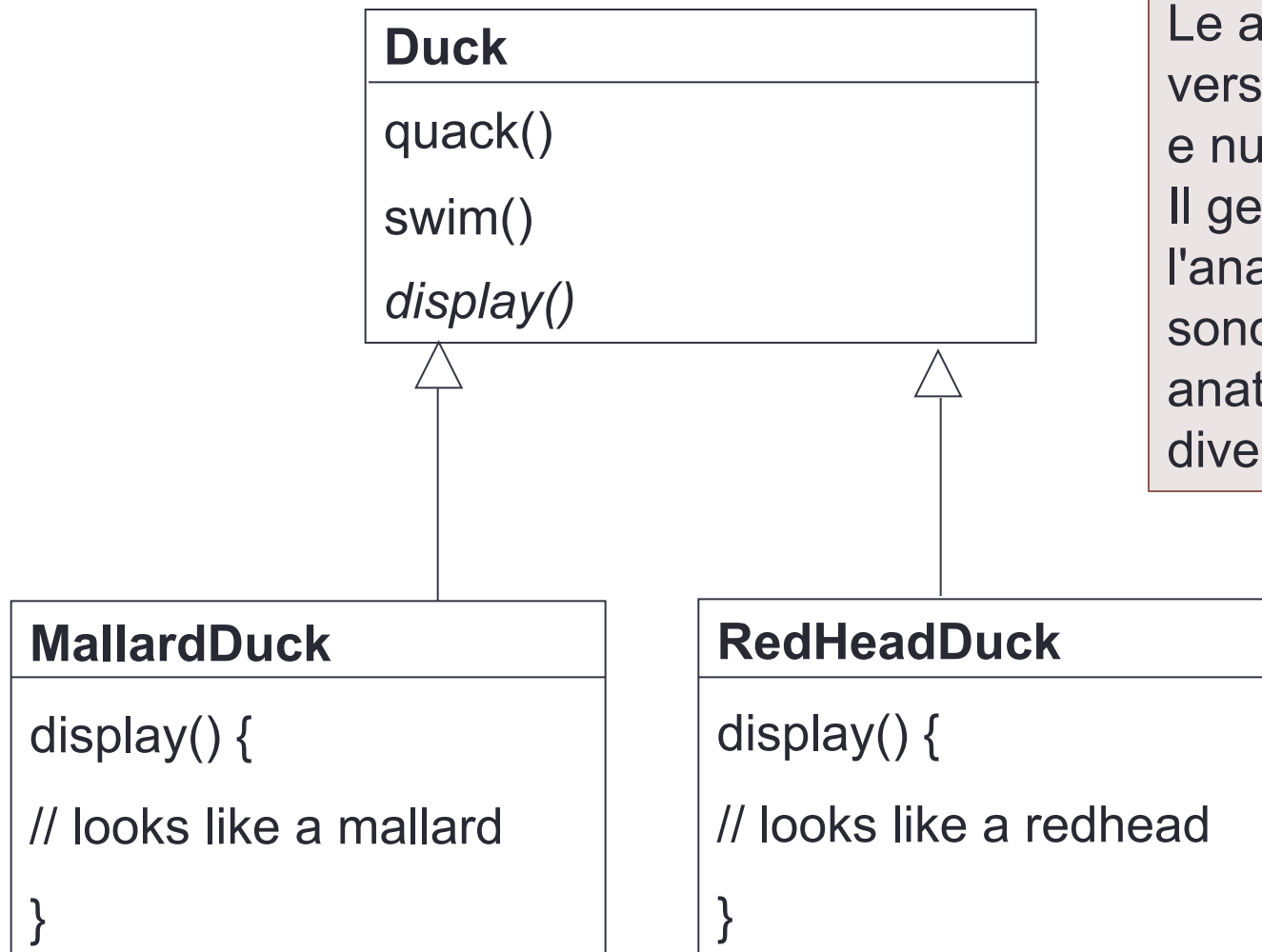
GoF Design Patterns

| Creational | Structural | Behavioral |
|------------------|------------|-------------------------|
| Factory | Adapter | Interpreter |
| Abstract Factory | Bridge | Template Method |
| Builder | Composite | Chain of Responsibility |
| Prototype | Decorator | Command |
| Singleton | Facade | Iterator |
| | Flyweight | Mediator |
| | Proxy | Memento |
| | | Observer |
| | | State |
| | | Strategy |
| | | Visitor |

- Non li guardiamo tutti ma alcuni tra i più utilizzati, utili per capire l'idea stessa di pattern e le sue applicazioni in Java per aumentare flessibilità e riusabilità del codice

BEHAVIORAL PATTERNS

Scenario: Duck



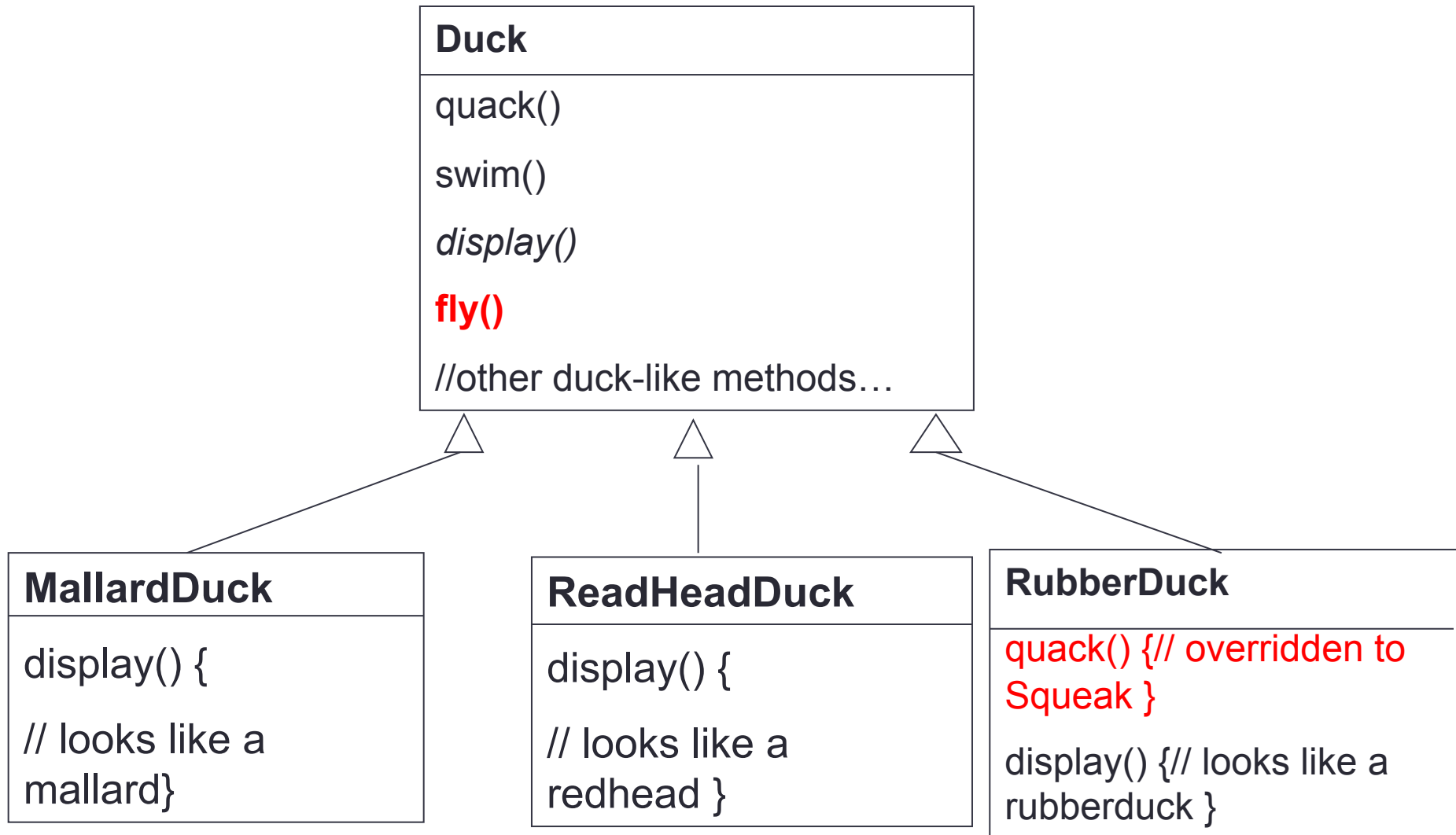
Le anatre fanno un verso (quack) e nuotano. Il germano reale e l'anatra testarossa sono due tipi di anatre con tratti diversi.

[da: 'Head First. Design Patterns.']

Altri tipi di anatre e comportamenti?

- Come aggiungere un nuovo tipo di anatra che fa un verso diverso dalle altre?
- Come aggiungere il comportamento *fly()*?

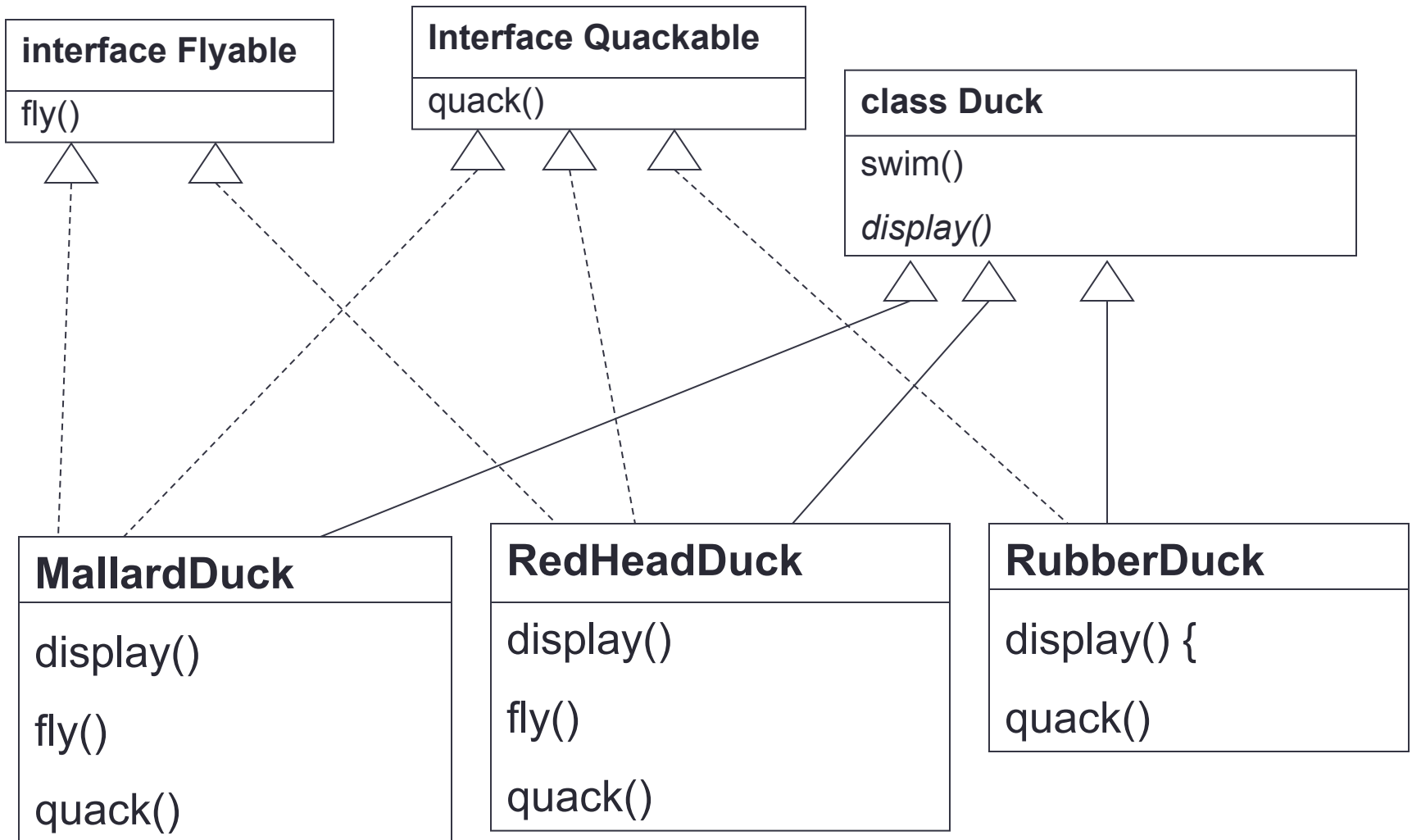
Ereditarietà



Estendibile?

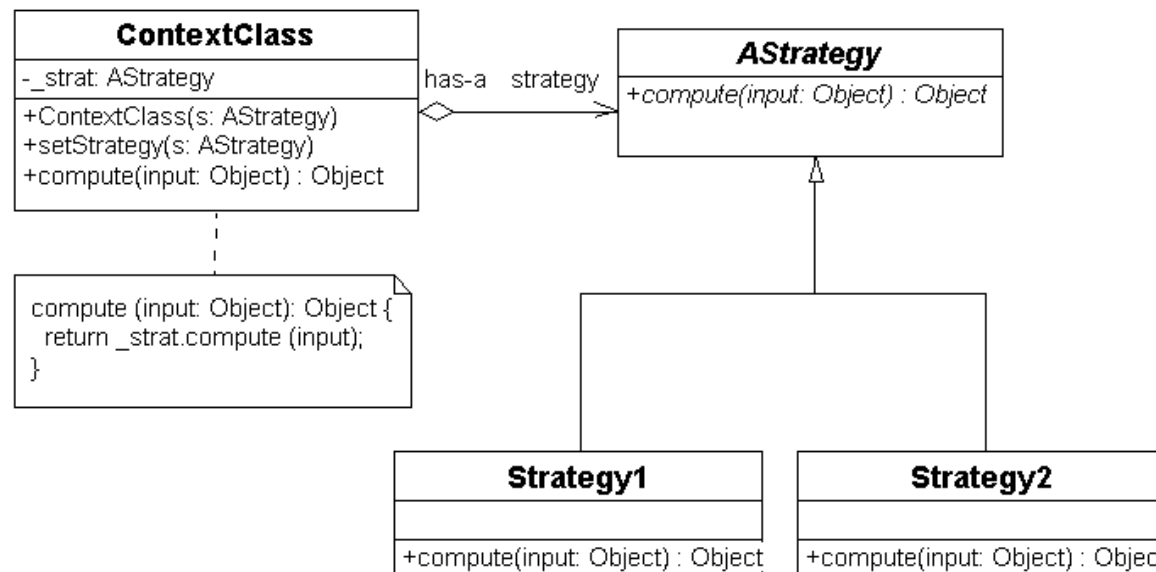
- Come aggiungere un nuovo tipo di anatra muta e che non vola?
- Basta fare *override* dei metodi *quack()* e *fly()*?
- E per nuovi tipi di anatra che hanno comportamenti – *quack()* e *fly()* – parzialmente sovrapposti alle altre?
- Come gestire le diverse combinazioni?

Interfacce. Si può fare meglio?

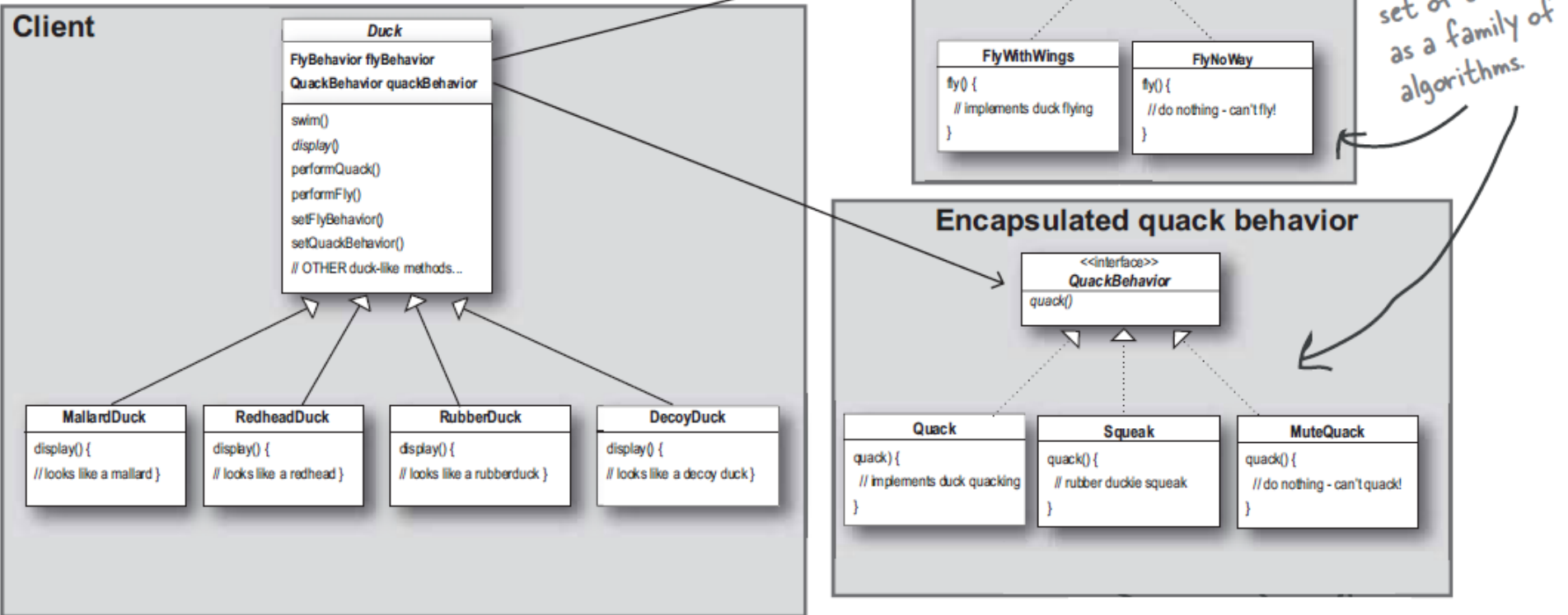


Pattern Strategy

- *Problema*: definire una famiglia di algoritmi e renderli interscambiabili
 - modificare il comportamento di una classe a run-time e disaccoppiare il comportamento (Algoritmo) dalla classe (Client) che lo usa



Client makes use of an encapsulated family of algorithms for both flying and quacking.



[da: 'Head First. Design Patterns.']

Pattern strategy in Java

- Un esempio di applicazione di questo pattern in Java è l'interfaccia `Comparator<T>`
- `Comparator<T>` espone diversi metodi tra cui `compare(T o1, T o2)`
- che confronta i due argomenti e restituisce un valore intero con lo stesso schema di `compareTo()` in `Comparable`
- Implementata da classi usate in algoritmi di ordinamento per cambiare la **strategia di confronto** ma mantenere l'algoritmo invariato
- Nella prossima slide l'algoritmo *SelectionSort*

Java Comparator

```
public static void sort(Object[] a,  
                        Comparator comparator) {  
    int n = a.length;  
    for (int i = 0; i < n; i++) {  
        int min = i;  
        for (int j = i+1; j < n; j++) {  
            if ( comparator.compare(a[j],a[min]) < 0 )  
                min = j;  
        }  
        swap(a, i, min); // scambia a[i] e a[min]  
    }  
}
```

Scenario: Clock

- Come modellare questa situazione?

Un orologio ha due pulsanti: MODE e CHANGE

MODE permette di scegliere la modalità: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”.

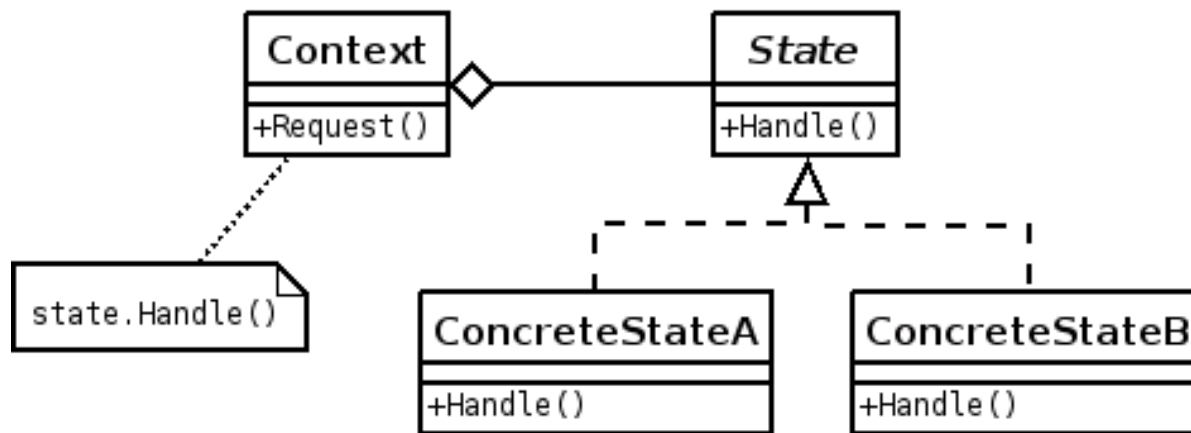
CHANGE esegue operazioni diverse in base alla modalità:

- *accendere la luce del display, se è in modalità di visualizzazione normale,*
- *incrementare in una unità le ore o i minuti, se è in modalità di modifica di ore o di minuti*

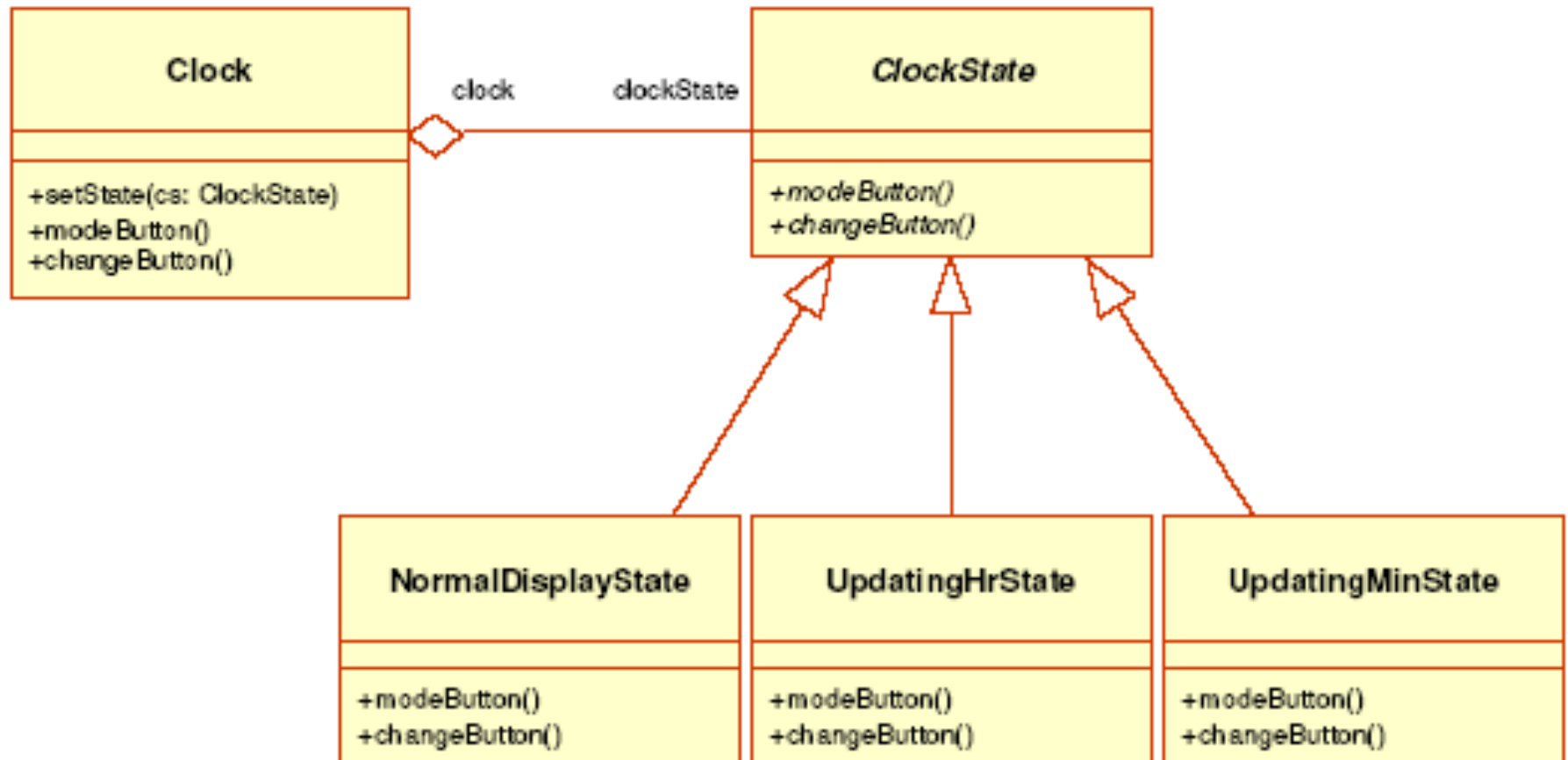
- E se i possibili stati sono molti?

Pattern State

- Il pattern **State** permette ad un oggetto di cambiare il proprio comportamento in base allo stato in cui si trova



Scenario: Clock

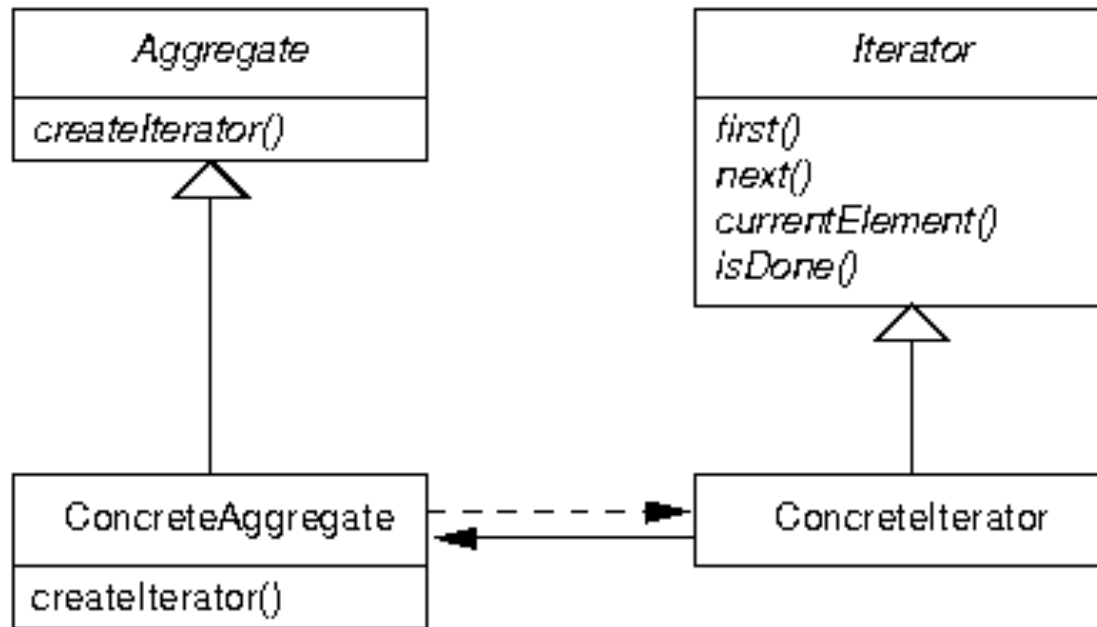


Scenario: collezione di oggetti

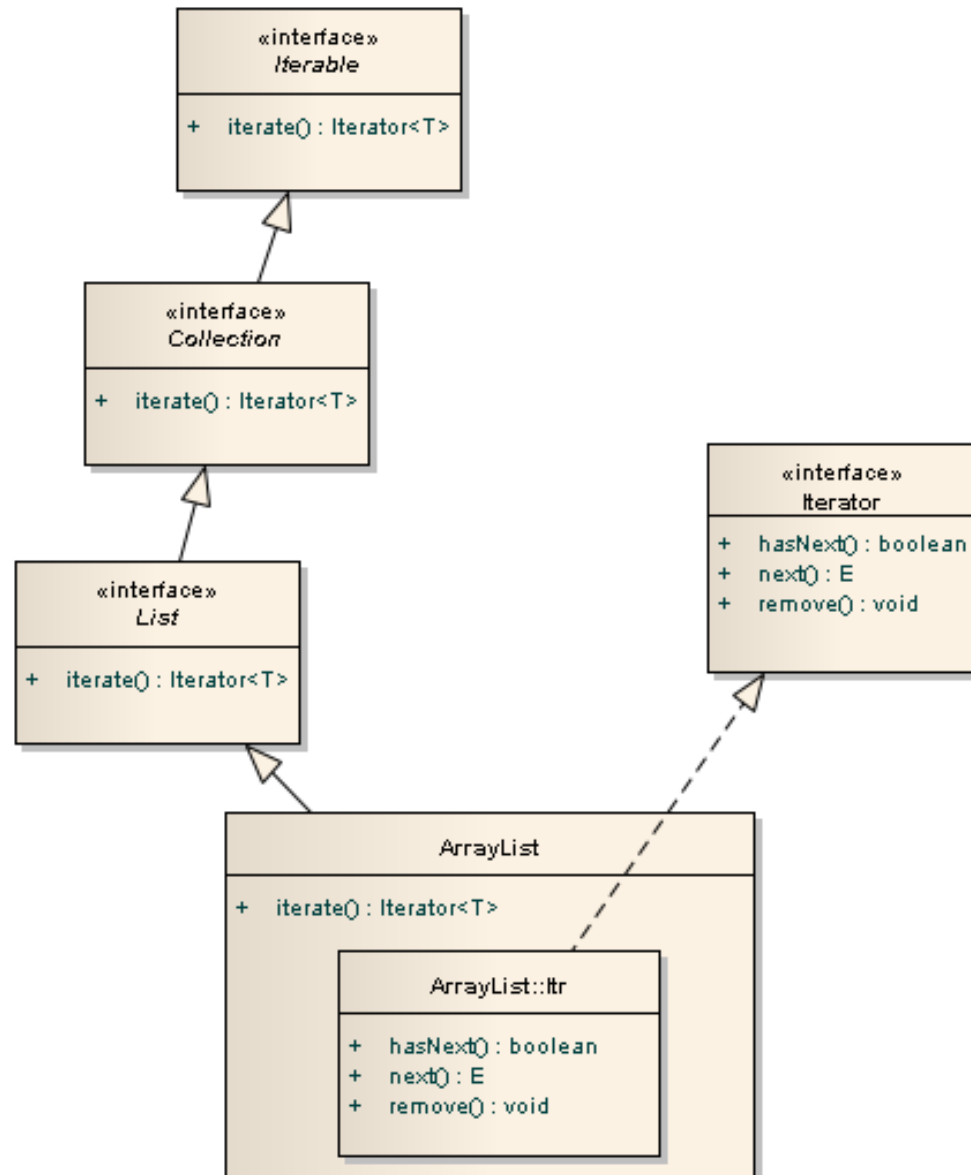
- Una collezione di oggetti può essere memorizzata in un vettore, un insieme, una lista
- E' utile poter scrivere algoritmi che scandiscono tutti gli elementi di una collezione **senza conoscere** i dettagli di **come la collezione è realizzata e il tipo degli oggetti che contiene**
- E' utile poter usare lo stesso algoritmo su collezioni di oggetti di tipo diverso, senza imporre che questi oggetti siano istanze di classi collegate tra loro (es. classi concrete di una stessa classe astratta)
- Si vuole quindi disaccoppiare l'iterazione sulla collezione dalla rappresentazione degli oggetti
- Come fare?

Iterator

- Il pattern **Iterator** permette di accedere agli elementi di un aggregato in modo sequenziale e indipendente dagli elementi dell'aggregato



Iterator Pattern in Java



STRUCTURAL PATTERNS

Scenario: Beverage

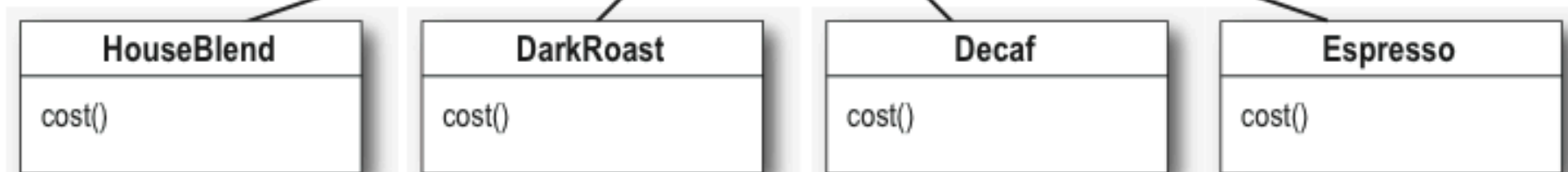
Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.

The `cost()` method is abstract; subclasses need to define their own implementation.



In un caffetteria si vendono quattro tipi di caffè: HouseBlend, DarkRoad, Decaf, Espresso.

Il costo dipende dal tipo di caffè.



[da: 'Head First. Design Patterns.']

Aggiungere extra al caffè

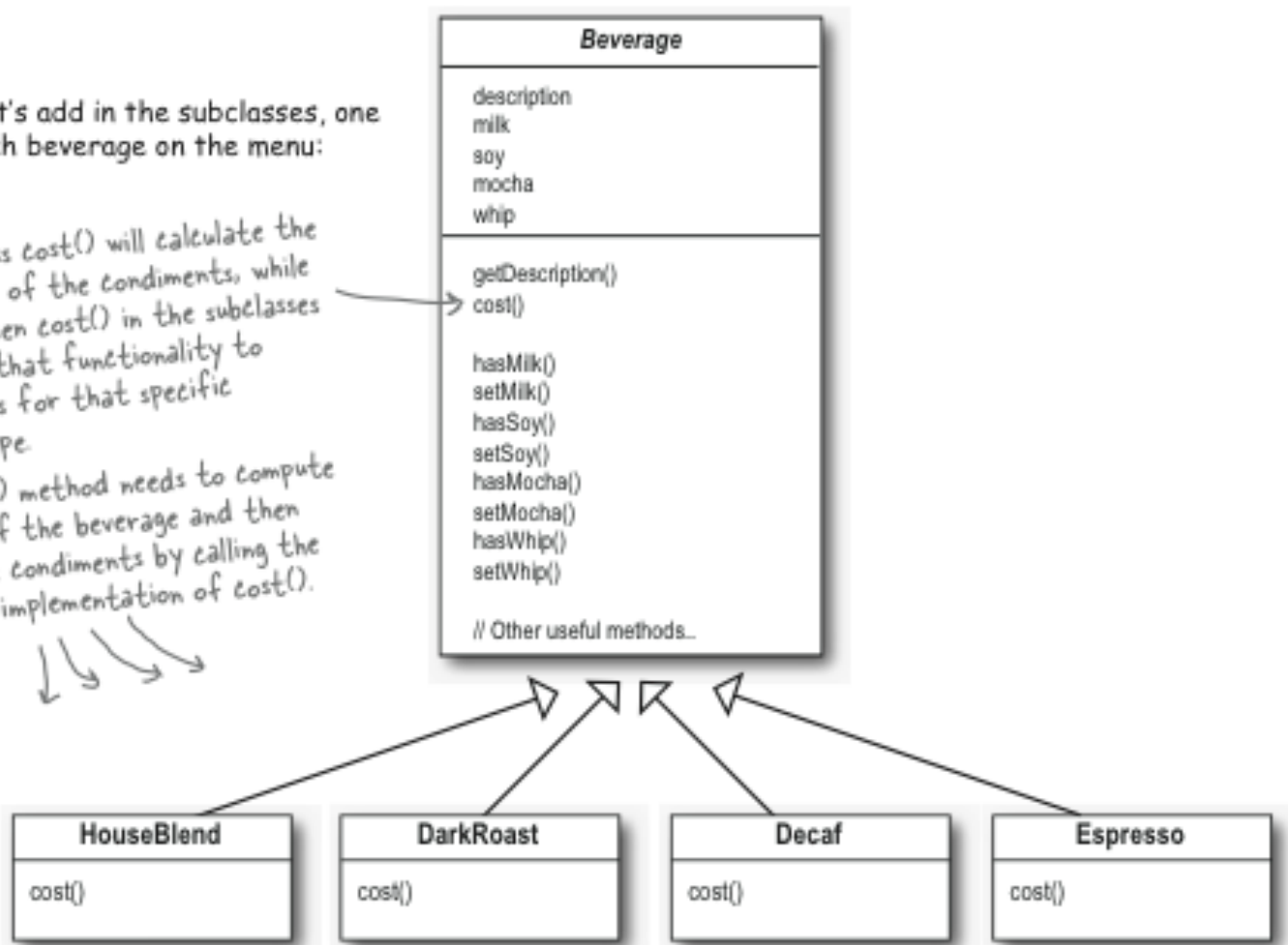
- Si vuole permettere al cliente di scegliere:
 - Caffè base
 - Extra da aggiungere al caffè: latte, soia e mocha (cioccolato)
- Il costo dipenderà quindi dal tipo di caffè base e dagli ingredienti aggiunti
- Idee?

Ereditarietà e override

Now let's add in the subclasses, one for each beverage on the menu:

The superclass `cost()` will calculate the costs for all of the condiments, while the overridden `cost()` in the subclasses will extend that functionality to include costs for that specific beverage type.

Each `cost()` method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of `cost()`.

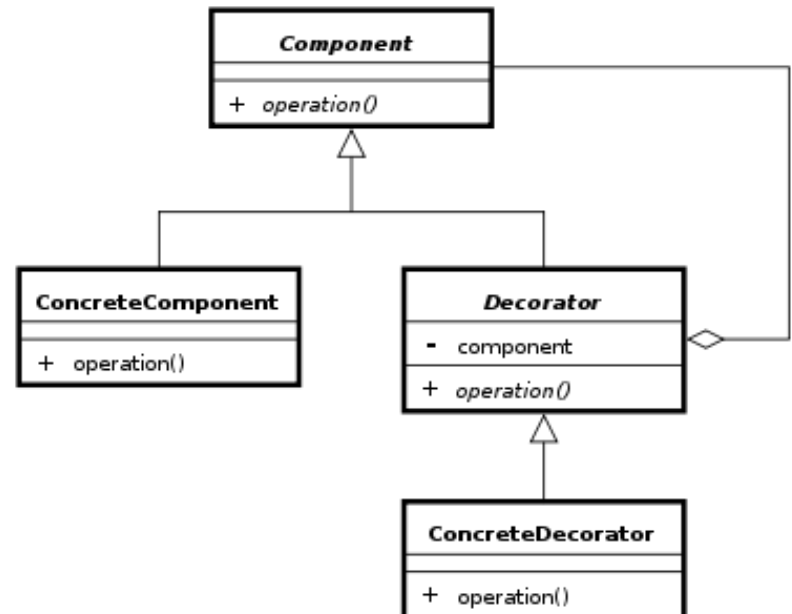


Si può fare meglio?

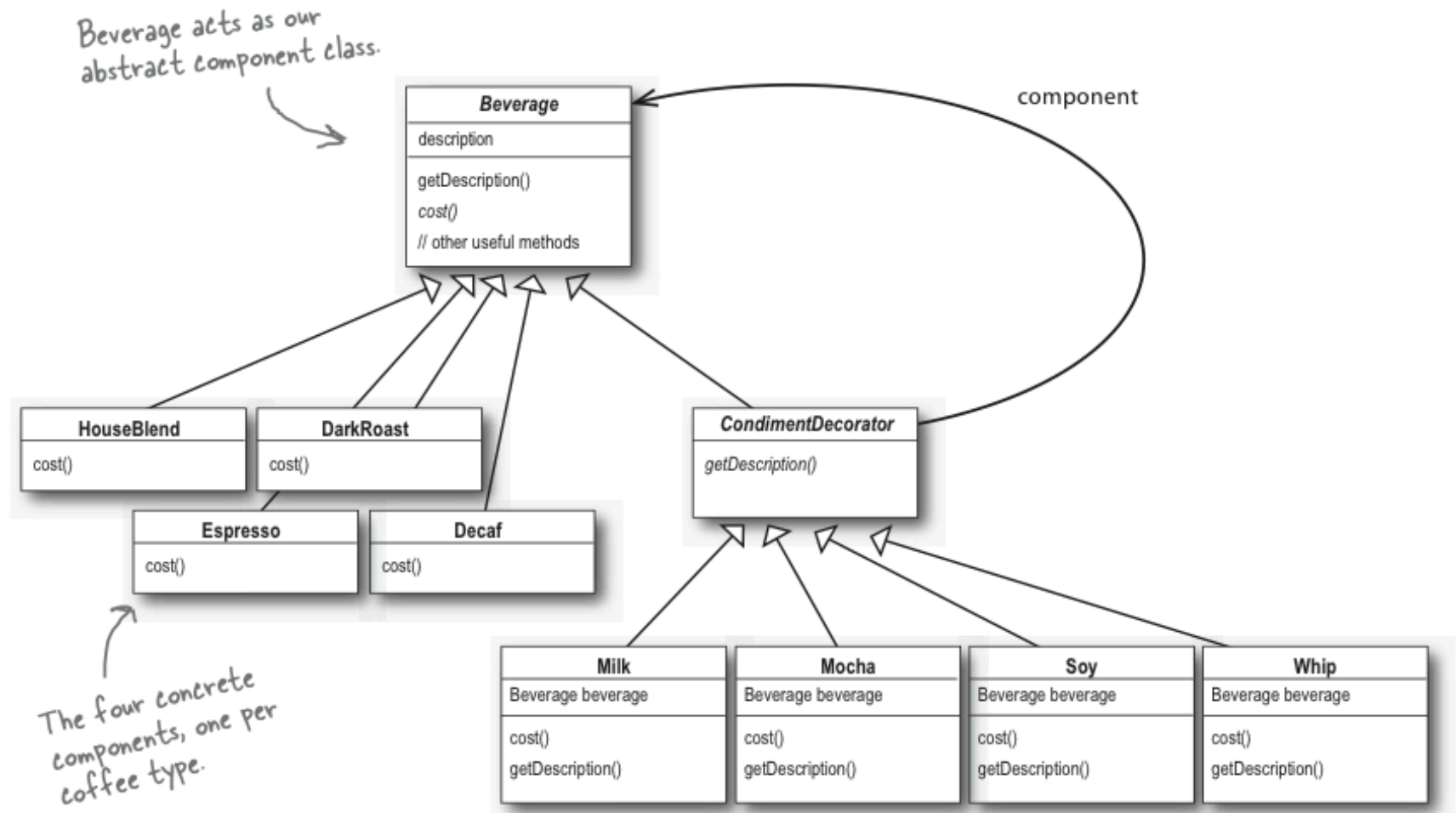
- Cosa succede se cambia il prezzo di un'extra?
- E se vogliamo aggiungere un nuovo extra?
- Tutti gli extra vanno bene per tutte le bevande?
- E se un cliente vuole più volte lo stesso extra?

Pattern Decorator

- Il pattern Decorator permette di aggiungere a runtime un comportamento ad un oggetto senza modificare il comportamento dell'oggetto di partenza
- L'oggetto "decorato" può essere a sua volta "decorato" ri-applicando lo stesso pattern
- Comportamento "stratificato"



Decorator



[da: 'Head First. Design Patterns.']

Esempio codice decoratore

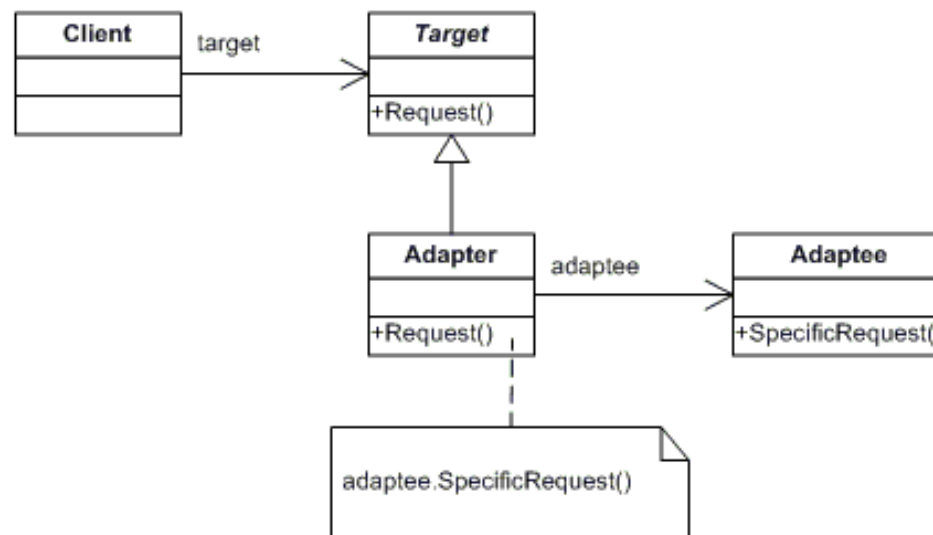
```
public class Mocha extends CondimentDecorator {  
    Beverage beverage;  
  
    public Mocha(Beverage beverage) {  
        this.beverage = beverage;  
    }  
  
    public double cost() {  
        return .20 + beverage.cost();  
    }  
}
```

Scenario: scene

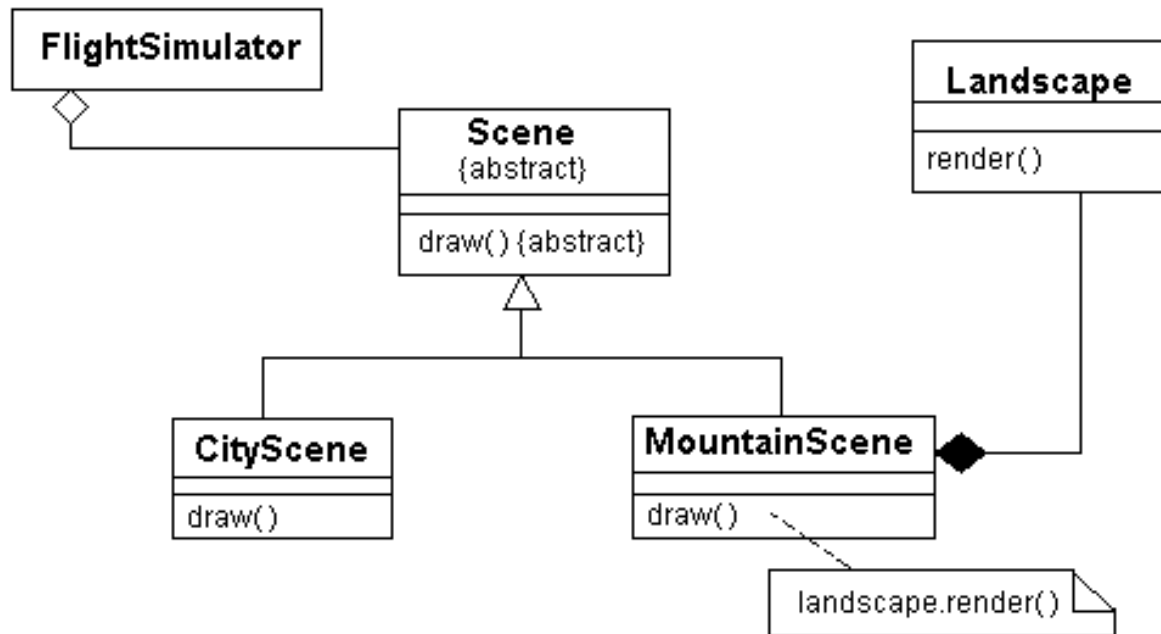
- Si vuole disegnare uno sfondo 2D in un'applicazione. Per permettere di cambiare sfondo si definisce una classe `Scene` con un metodo `draw()`
- Si implementano quindi le classi `CityScene`, `LunarScene`, `SportScene`, **ecc.**
- Si vorrebbe include una classe *legacy* che disegna un panorama alpino ma che non è una sottoclasse di `Scene`
- Lo stesso problema si presenterebbe volendo riusare classi che non implementano interfacce definite nell'applicazione
- Utile **adattare** classi che **non sono state progettate** per essere **interoperabili**

Pattern Adapter

- Il pattern **Adapter** converte l'interfaccia di una classe in un'altra interfaccia attesa dal *Client*
- Permette quindi di collegare classi non interoperabili nel caso in cui non sia possibile controllare la classe *Adaptee*

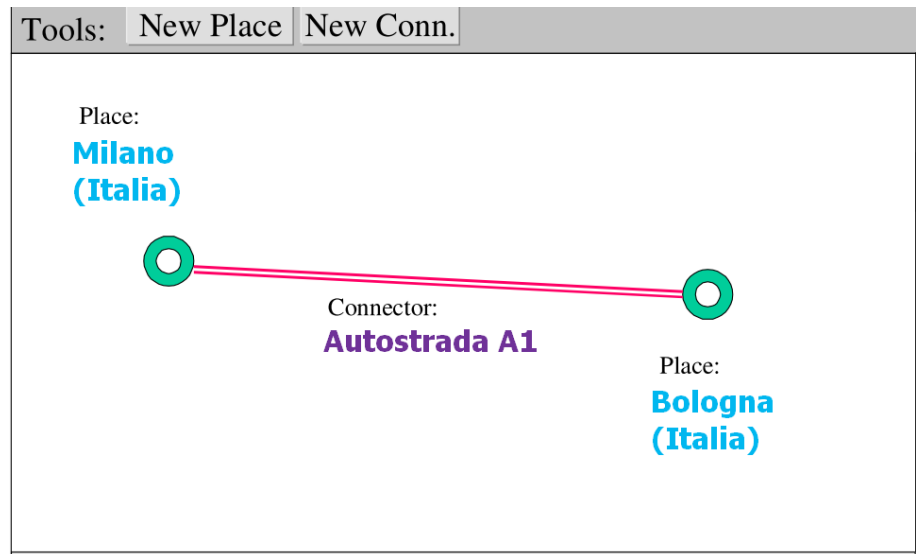


Scenario: scene



CREATIONAL PATTERNS

Scenario: framework grafico



Un **framework** per manipolare elementi grafici ha due componenti principali:

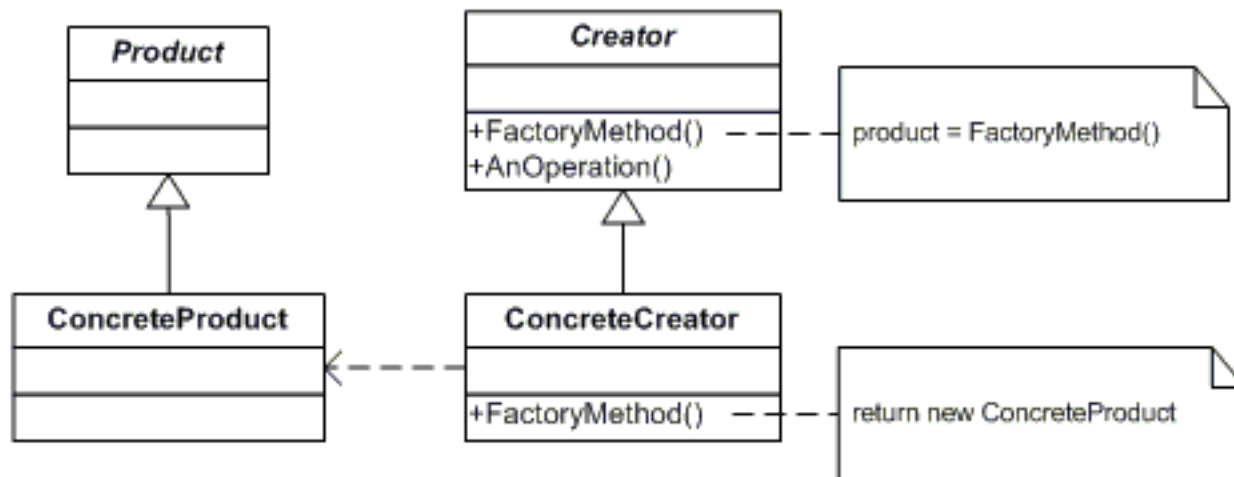
- **elementi**: oggetti da posizionare (es. luoghi e connettori)
- **strumenti**: oggetti che forniscono operazioni comuni a tutti gli elementi

Scenario: framework grafico

- Un framework usa classi astratte per definire e gestire gli oggetti e le loro relazioni
- Il framework **deve** creare oggetti (istanziare le classi) ma conosce solo le classi astratte che quindi non può istanziare
- Il comportamento del framework è indipendente dalle classi concrete che saranno istanziate
- Inoltre il framework potrebbe non essere in grado di anticipare quali classi saranno istanziate
- Si vuole portare via dal *framework* la creazione di particolari tipi di *elementi* (Place e Connector). Per fare ciò viene delegato alle sottoclassi dello *strumento* (PlaceHandler e ConnectorHandler), che specializzano le funzioni di gestione di ogni tipo di elemento, il compito di creare le particolari istanze di classi che sono necessarie per quello strumento

Factory Method

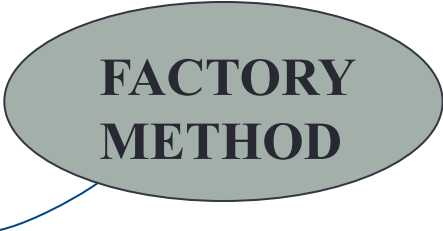
- Factory method permette ad una classe di differire l'istanziamento alle sue sottoclassi.
- La conoscenza di quali classi creare viene incapsulata nel Factory method, spostando tale conoscenza al di fuori del framework
- Definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione del tipo di classe da istanziare.



Elementi e strumenti: interfacce

```
interface MapElement {  
    abstract void setLabel(String id);  
    abstract String getPaintingData();  
}
```

```
abstract class ElementHandler {  
  
    abstract MapElement newElement();  
  
    MapElement createElement(String label) {  
        MapElement element = newElement();  
        element.setLabel(label);  
        return element;  
    }  
  
    public void paintElement(MapElement element) {  
        System.out.println(element.getPaintingData() );  
    }  
}
```



FACTORY METHOD

Elementi: implementazione

- `class Place implements MapElement {`
- `private String placeLabel;`
- `public void setLabel(String label) { placeLabel = label; }`
- `public String getPaintingData() {return "city:" + placeLabel;}`
- `}`
- `class Connector implements MapElement {`
- `private String connectorLabel;`
- `Place place1, place2;`
- `void setLabel(String label) { connectorLabel = label; }`
- `String getPaintingData() {`
- `return connectorLabel + "[from " + place1.getPaintingData()`
- `+ " to " + place2.getPaintingData() + "];"`
- `}`
- `void setPlacesConnected(Place origin, Place destination) {`
- `place1 = origin;`
- `place2 = destination;`
- `}`
- `}`

Strumenti: implementazione

```
class PlaceHandler extends ElementHandler {  
    MapElement newElement() {  
        return new Place();  
    }  
}
```

```
class ConnectorHandler extends ElementHandler {  
    MapElement newElement() {  
        return new Connector();  
    }  
}
```

```
void connect(Connector conn, Place origin,  
             Place destination){  
    conn.setPlacesConnected( origin, destination );  
}
```

ANTI-PATTERN E SUGGERIMENTI DI BUONA PROGRAMMAZIONE

Anti-pattern

- Termine coniato nel 1995 Andrew Koenig.
- *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Brown W., Malveau R. e altri, 1998.
- <http://www.antipatterns.com/> (mantenuto dagli autori)
- Controparte naturale dei design pattern, definiscono un vocabolario per identificare **errori implementativi o di processo comuni**
- Sono **soluzioni negative** che producono più problemi di quelli che risolvono
- Utile comprendere gli anti-pattern per prevenirli, o comunque individuare e risolvere gli errori

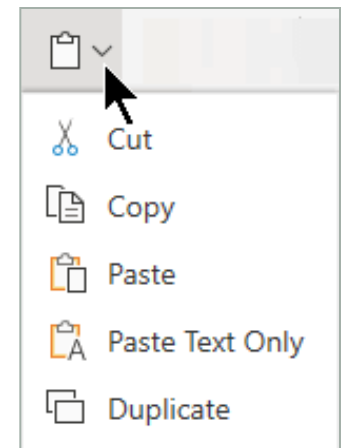
Cargo cult

- Il termine Cargo Cult si usa per indicare il comportamento di un programmatore che non riuscendo a risolvere un bug o ad implementare una nuova funzionalità, copia una soluzione esistente senza capirne il funzionamento
- Il più famoso esempio di cargo cult (in ambito non informatico) riguarda alcune popolazioni della Nuova Guinea che durante la seconda guerra mondiale notarono che nella loro isola arrivavano aerei (e scorte!). Per propiziare il ritorno di questi aerei alla fine della guerra iniziarono a imitare il comportamento militare, ad esempio i segnali (con fiamme e bastoni) per l'atterraggio o l'abbigliamento
- **Non vuole dire non riusare soluzioni esistenti ma riusarle con cognizione**



Copy-and-paste programming

- Consiste nel ripetere codice attraverso operazioni di copia&incolla
 - Ripetizioni di codice usato nella stessa applicazione e in situazioni simili o che sembrano simili ma hanno importanti differenze che rendono la soluzione non corretta
 - Ripetizione di codice usato in altre applicazioni
- Diversi rischi collegati:
 - Usare il codice in un posto in cui non è adatto
 - Duplicare errori
 - Più linee di codice da gestire, eventuali bug in più punti
- Attenzione anche al rischio inverso: sovraccaricare lo stesso codice per usarlo in più punti
- **E' corretto sfruttare esempi nelle prime fasi di sviluppo ma vanno interiorizzati**



Spaghetti code

- Probabilmente l'anti-pattern più famoso è lo *spaghetti code*
“an undocumented piece of software code that cannot be extended or modified without extreme difficulty due to its convoluted structure”
- Programmi il cui flusso di esecuzione è contorto e difficile da analizzare
- La progettazione OO mitiga il problema (rischio molto più alto con istruzioni GOTO) ma può verificarsi lo stesso, ad esempio per metodi molto lunghi e che fanno operazioni semplici in modo inutilmente complesso
- **Scrivere codice con un flusso di esecuzione facile da seguire**



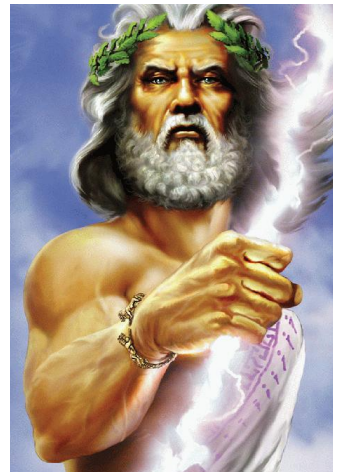
Kitchen Sink

- Il termine Kitchen Sink ("lavello della cucina") indica le situazioni in cui si creano gruppi di operazioni (interfacce) molto eterogenee e scollegate tra loro
- Diventa più difficile capire quale comportamento si sta individuando con quell'interfaccia
- **Dividere le interfacce complesse in più piccole ma più mirate**
- Principio di coesione



God Object

- In modo simile un altro anti-pattern descrive la situazione in cui si creano oggetti "super-potenti" che mantengono molte informazioni e svolgono molte operazioni, tra loro disomogenee e scollegate
- La manutenzione di questi oggetti diventa molto più difficile
- Aumentano le dipendenze
- Modificare una parte di questi oggetti potrebbe produrre errori in altre parti del codice, (che solitamente non vengono scoperti immediatamente ma in seguito)
- **Distribuire compiti su più classi**



Reinventing the wheel

- Questo antipattern descrive le situazioni in cui si prova a risolvere un problema prima di verificare se esiste una libreria, un'API, un servizio esterno che risolve esattamente lo stesso problema
- Sforzo ridondante e non richiesto
- Rischio di introdurre errori che altri hanno già risolto
- **Riusare librerie esistenti dove utili**
- Attenzione al rischio opposto: usare correttamente le librerie, rischio *cargo cult* e *copy&paste programming*



Boat Anchor

- Questo antipattern indica le situazioni in cui una parte del programma rimane nel codice sorgente anche se non è più necessaria
- Succede perché si pensa di poterla riutilizzare in seguito ma nella maggior parte dei casi non è così
- Più codice da mantenere, meno chiara la lettura
- **Cancellare il codice obsoleto e non più usato**
- Molto utile usare sistemi di versionamento (es. Git, SVN) per poter recuperare il codice, ma non mantenerlo nei sorgenti
- Lo stesso vale per i messaggi di debug ☺



Magic numbers

- Un errore comune consiste nell'uso di numeri direttamente nel codice a cui è associato un significato che però non è esplicitato

```
function potentialEnergy(mass, height) {  
    return mass * 9.81 * height;  
}
```

- Diversi problemi:
 - meno comprensibile per altri sviluppatori
 - valore probabilmente ripetuto all'interno del codice
 - aumenta la possibilità di errore
- **Definire costanti al posto di magic numbers e usarle coerentemente nel codice**



Corretto uso dei nomi

- Cosa fa questo metodo?

```
function doSomething($i) {  
    for ($i2 = 0; $i2 <= 52; $i2++) {  
        $j := $i2 + randomInt(53 - $i2) - 1;  
        swap($i, $i2, $j);  
    }  
    return $a;  
}
```

- Come renderlo più chiaro?
- **Usare nomi auto-esplicativi**: già il nome ci fa capire cosa ci aspettiamo da un metodo; se la scelta del nome è difficile, potrebbe esserci qualcosa che non va nel metodo stesso (es. Fa troppe cose?)
- Usare nomi comuni, chiari per gli altri sviluppatori
- **Adottare una convenzione** ed usarla sempre (es. maiuscole/underscore, italiano/inglese)



Conclusioni

- Oggi abbiamo visto un po' di *pattern* e *anti-pattern* nella programmazione
- Utile conoscerli per prevenire errori e aumentare estensibilità e manutenibilità del proprio software
- Ne esistono molti altri e altre classificazioni, ad esempio focalizzati su aspetti architetturali o anche management dei progetti software
- Altri che abbiamo visto durante il corso?
- Non è un pattern a se' stante ma concludo con una nota sulla documentazione: molto spesso sottovalutata, è fondamentale per comunicare con gli altri sviluppatori ma anche per lavorare bene sul proprio progetto (soprattutto dopo un po' di tempo)