

# 21

## Progettazione OO e pattern

Mirko Viroli  
`mirko.viroli@unibo.it`

C.D.L. Ingegneria e Scienze Informatiche  
ALMA MATER STUDIORUM—Università di Bologna, Cesena

a.a. 2018/2019



## Goal della lezione

- Illustrare il ciclo di vita del software
- Illustrare il concetto di Design Pattern
- Discutere i dettagli di vari Pattern
- Ripassare vari concetti del corso

## Argomenti

- Progettazione architetturale
- La nozione di Pattern
- La lista di Pattern della GoF
- Dettagli su Pattern già visti
- Dettagli su nuovi Pattern

- 1 Ciclo di vita del software
- 2 Progettazione architetturale
- 3 Design Pattern e Progettazione di Dettaglio
- 4 Pattern già incontrati
- 5 Nuovi pattern: creazionali



# Macro-fasi del processo di sviluppo

## Analisi

Si definisce in modo preciso il **problema** da risolvere

- i requisiti del problema, ben “ingegnerizzati”
- il modello del dominio (terminologia, entità e relazioni)

## Design

Si definisce la struttura del sistema da sviluppare

- progetto architettuale + progetto di dettaglio
- si descrive la **soluzione**, ad uno specifico livello di dettaglio

## Implementazione/codifica

Si realizza il sistema sulla base del progetto, scegliendo le tecnologie adeguate (efficienti, efficaci) – p.e. il linguaggio di programmazione

## Post-codifica: Collaudo, Manutenzione, Deployment

Fasi necessarie, che spesso impiegano più del 70% delle risorse complessive

# Micro-fasi in analisi e progettazione

## Analisi dei requisiti – interazione col committente

I requisiti raccolti dalle interazioni col committente vengono analizzati e formalizzati, diventando una sorta di “contratto”

## Analisi del problema – valutazioni preventive

A partire dai requisiti, si analizza il problema identificandone gli aspetti più complessi da gestire, e dettagliando le funzionalità da realizzare

## Design architetturale – inizio della progettazione

Progettazione “ad altissimo livello”, in cui si definisce solo la struttura complessiva del sistema in termini dei principali moduli (classi, o meglio interfacce) di cui esso è composto e delle relazioni macroscopiche (“uses”, “has-a” o “is-a”) fra di essi.

## Design di dettaglio – progettazione

Rappresenta una descrizione del sistema molto vicina alla codifica, ovvero che la vincola in maniera sostanziale. Descrive interfacce, classi astratte e concrete che rappresentano la *soluzione* ai principali problemi identificati in analisi



# Sul modello di sviluppo

## Quale approccio?

- A cascata: le fasi rigorosamente in ordine temporale
- A spirale: le fasi rigorosamente in ordine ma svolte in più “cicli”
- A fontana: si può tornare temporaneamente nella fase precedente/successiva
- Agile: sviluppo iterativo e incrementale

⇒ non esiste il modello “perfetto”

## Note e consigli per piccoli progetti (come quello d'esame)

- Convieni darsi 2-3 obiettivi intermedi incrementali
- Ogni obiettivo usi un approccio simile a quello in cascata
- La documentazione finale campiona il sistema finale, simulando un modello a cascata: ciò consente una coerente analisi a posteriore del sistema costruito

# Outline

- 1 Ciclo di vita del software
- 2 Progettazione architetturale**
- 3 Design Pattern e Progettazione di Dettaglio
- 4 Pattern già incontrati
- 5 Nuovi pattern: creazionali

# Progettazione architeturale

## Idee chiave

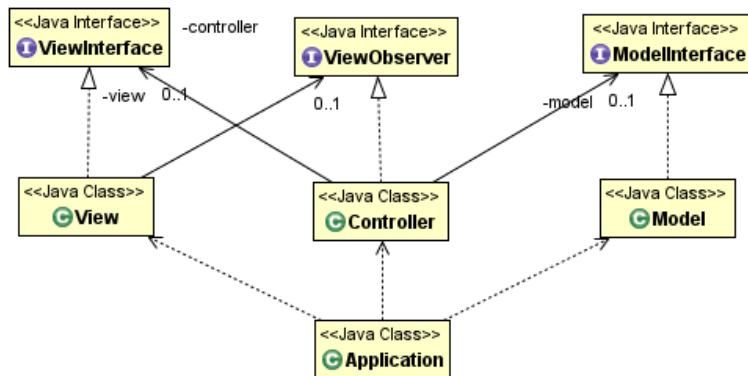
- Costruire un unico diagramma dei concetti principali (classi/interfacce)
- Idealmente, meglio se fatto di sole interfacce, indicativamente 5-12 in tutto
- Non ci siano classi/interfacce isolate dal resto
- Il sistema va diviso in componenti il più possibile isolati e non dipendenti fra loro
- Una descrizione in prosa indichi il principale ruolo di ogni entità e le relazioni esistenti
- Meglio se non estratto dal codice, ma scritto “a mano”

## Quale “pattern” architettura?

- MVC è un ottimo punto di partenza
- Non l'unico, ma non ne vedremo altri
- Attenzione a “copiare” architetture/progetti dalla rete!
  - ▶ per OOP: meglio partire dal foglio bianco



# Il caso di MVC – schema generale (nomi di comodo)



# Linee guida per un buon MVC pt.1

## Ruoli/responsabilità delle 3 parti

- View: gestire la parte di presentazione e di interazione con l'utente
- Model: gestire i dati e la logica di dominio dell'applicazione
- Control: gestire la “meccanica” dell'applicazione, coordinando View, Model ed eventuali interazioni con altri componenti (SISOP)

## Interazioni

- Le interazioni dell'utente in V diventano eventi catturati da C, o metodi `void` chiamati su C
- C osserva / modifica / o interroga M
- Quando necessario C comanda modifiche a V

⇒ non vi sono altre interazioni possibili!



# Linee guida per un buon MVC pt.2

## Linee guida

- Cambiare la view e la sua tecnologia (Swing, AWT, Console) non deve far toccare M e C
- Il modello gestisce in modo elegante e ben costruito la logica di dominio dell'applicazione
  - ▶ In M c'è margine per fare dell'ottima e pulita progettazione OO
  - ▶ Qualsiasi comportamento e/o fenomeno possibile è riproducibile da una opportuna sequenza di chiamate di metodo su M fatte da un `main`
  - ▶ Di fatto, C sostituisce questo `main`, creando l'applicazione indipendentemente dai dettagli della UI
- Modello e vista espongono interfacce ben progettate (senza dettagli implementativi) attraverso cui C agisce in modo ben disaccoppiato
- Se esistono Thread o accessi al sistema operativo, sono incapsulati in C
- M, V e C potrebbero essere fatti da “sotto-parti” diverse

# Outline

- 1 Ciclo di vita del software
- 2 Progettazione architetturale
- 3 Design Pattern e Progettazione di Dettaglio**
- 4 Pattern già incontrati
- 5 Nuovi pattern: creazionali

# Progettazione di dettaglio

## Elementi

- Non descrive ogni singola classe/interfaccia del sistema
- Solo quelle ritenute più importanti per capire come il sistema è organizzato
- Quelle che nascondono elementi non banali
- Documentata da più diagrammi UML sempre di 5-10 classi ognuno

## Come progettare una buona classe o gruppo di classi?

1. buona conoscenza della programmazione OO e delle linee guida di buona programmazione/progettazione note e discusse
2. utilizzo di cataloghi noti di pattern di progettazione (design pattern)



## I Pattern di progettazione

- Idea: trasmettere esperienze (positive) e ore di lavoro (di identificazione, rifattorizzazione) ad altri per essere usate *tout court*
- Sono elementi riusabili (semplici ed eleganti) di progettazione OO
- Sono stati ottenuti in passato (e tuttora) dall'analisi di soluzioni ricorrenti in progetti diversi
- Alcuni sono particolarmente famosi, come quelli della “Gang of Four” (detti anche Pattern della GoF, o Pattern di Gamma)
  - ▶ Testo famosissimo (in C++): “Design Patterns: Elements of Reusable Object-Oriented Software” di E.Gamma, R.Helm, R.Johnson, J.Vlissides
  - ▶ 23 in tutto. Esempi: Decorator, Singleton, Template Method, Observer
  - ▶ (Cit. “SW di grosse dimensioni li usano praticamente tutti”)
- Il loro uso migliora molto il codice
  - ▶ Ne favorisce la comprensione se li si indicano nella documentazione
  - ▶ Rende il codice più flessibile (nascono per questo)
  - ▶ Portano più direttamente ad una buona organizzazione

# Design Pattern in questa sede

## In questa lezione

- Illustreremo il catalogo della GoF
- Verranno approfonditi alcuni Pattern anche con esempi

## Nel corso

- Vari Pattern già stati utilizzati (es.: nelle librerie)
- Vanno usati dove opportuno nel progetto d'esame (e nella relazione)
- Possono essere tema dell'esame in laboratorio
- Quelli visti a lezione sono da conoscere tassativamente

## Per il vostro futuro

- Noi porremo le basi per uno loro studio in autonomia
- Un ottimo progettista li conosce e usa (ove opportuno) **tutti**
- Alla magistrale verranno approfonditi

# Motivazioni

## Rifattorizzazione (refactoring)

- Operazione di modifica del codice che non aggiunge funzionalità
- Ha lo scopo di migliorare programmazione e struttura del SW
- Ha lo scopo di attrezzare il codice a possibili cambiamenti futuri
- Può/deve quindi comportare una riprogettazione di alcune parti

## La necessità del refactoring

- Una buona progettazione non la si ottiene al primo “colpo”, ma richiede vari refactoring
- Brian Foote identifica tre fasi nello sviluppo di un sistem: prototyping, expansionary, consolidating; nel consolidamento si rifattorizza
- Nell'agile programming, ogni ciclo di sviluppo non parte se non si è rifattorizzato il codice del ciclo precedente

## I pattern

- L'esperienza pregressa risulta fondamentale per velocizzare il processo di rifattorizzazione
- I Pattern di progettazione forniscono direttamente “ricette” di buona costruzione del SW





## Un Pattern ha quattro elementi fondamentali

- Un nome. (È un aspetto fondamentale!)
- Un problema che risolve. (La causa che porta al suo uso)
- La soluzione che propone. (Gli elementi del progetto)
- La conseguenza che porta. (Riuso, variabilità, performance,..)

## Granularità

- Gruppo ristretto di (1-5) oggetti/classi generali dipendenti fra loro
- Sistemi più specifici o più complessi sono utili, ma non propriamente dei “Pattern”
  - ▶ Non singole classi riusabili (liste, hash-table)
  - ▶ Non “pattern architetturali” (come MVC)
  - ▶ Non framework complessi (gerarchia Swing, Reflection)

# Classificazione dei Pattern: categorie

## Livello “proposito del Pattern”

- Creazionali: Riguardano la creazione degli oggetti
- Strutturali: Riguardano la composizione di classi/oggetti
- Comportamentali: Riguardano la interazione e distribuzione di responsabilità fra classi/oggetti

## Livello “scope”

- Classi: Il Pattern riguarda primariamente le relazioni fra classi (e sottoclassi), e quindi tratta aspetti statici (compile-time)
- Oggetti: Il Pattern riguarda primariamente le relazioni fra oggetti (l'esistenza di riferimenti fra oggetti), e quindi tratta aspetti dinamici (run-time)



# I 23 Pattern GoF

## Creazionali

- A livello di classe: **Factory Method**
- A livello di oggetto: **Abstract Factory**, **Builder**, **Prototype**, **Singleton**

## Strutturali

- A livello di classe: **Adapter**
- A livello di oggetto: **Adapter**, **Bridge**, **Composite**, **Decorator**, **Facade**, **Proxy**

## Comportamentali

- A livello di classe: **Interpreter**, **Template Method**
- A livello di oggetto: **Chain of Responsibility**, **Command**, **Iterator**, **Mediator**, **Memento**, **Flyweight**, **Observer**, **State**, **Strategy**, **Visitor**

# “Design for change”

## Problemi nel cercare di realizzare modifiche ad un sistema..

- Dipendenza dal nome di classe concreta
  - ▶ Abstract Factory, Factory Method, Prototype
- Dipendenza da operazioni (metodi) specifici
  - ▶ Chain of Responsibility, Command
- Dipendenza dalla interfaccia/implementazione di un oggetto
  - ▶ Abstract Factory, Bridge, Memento, Proxy
- Dipendenza da un algoritmo specifico
  - ▶ Builder, Iterator, Strategy, Template Method, Visitor
- Dipendenza stretta fra due classi
  - ▶ Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer
- Estendere funzionalità via subclassing non è pratico
  - ▶ Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy

# Schema di descrizione per ogni pattern

Aderiremo al seguente schema, che è una semplificazione di quello proposto alla GoF

## Ingredienti

1. Descrizione in prosa (nome, motivazione, esempi, soluzione)
2. Rappresentazione grafica (diagramma delle classi generale)
3. Esempio (già visto/nuovo)

## Pattern già incontrati, alcuni da approfondire

Singleton, Template Method, Strategy, Observer, Iterator, Decorator

## Nuovi

Factory Method, Abstract Factory

# I pattern nel corso OOP

## Esame laboratorio

- Qualche esercizio potrebbe riguardare il loro uso

## Esami di progetto

- Identificarne/usarne “pochi” è considerato poco soddisfacente
- Scegliere di usarli non è arbitrario, ma è indice di buona progettazione e/o di buona rifattorizzazione
- Gli argomenti: “in questo progetto non servivano” e “non c'è stato tempo” sono pessimi
- Argomento di probabile discussione all'esame

## Di conseguenza

- I pattern qui presentati vanno conosciuti
- Gli altri pattern sono facoltativi, e importanti per il vostro futuro

# Outline

- 1 Ciclo di vita del software
- 2 Progettazione architetturale
- 3 Design Pattern e Progettazione di Dettaglio
- 4 Pattern già incontrati**
- 5 Nuovi pattern: creazionali



# Singleton: creazionale, su oggetti

## Intento/motivazione

Garantire che una classe abbia una unica istanza, accessibile globalmente e facilmente a molteplici classi, senza doversi preoccupare di fornirne il riferimento a chi lo richiede (ad esempio passandolo al costruttore)

## Esempi

- Un unico gestore di stampanti in un sistema
- Un unico gestore del “log”
- `java.lang.Runtime`

## Soluzione

- La classe sia responsabile di tenere traccia di tale unica istanza
- La classe impedisca la creazione di altri oggetti
- La classe fornisca l'accesso a tale oggetto staticamente
- Attenzione: singleton accoppia clienti e implementazione



# Singleton: UML



# Singleton: Il caso di java.lang.Runtime

```
1 public class UseRuntime {
2     public static void main(String[] s){
3
4         // r punta l'unico oggetto di Runtime
5         final Runtime r = Runtime.getRuntime();
6         System.out.println(r.availableProcessors());
7         System.out.println(r.freeMemory());
8
9         // Accessibile anche senza depositare in r
10        System.out.println(Runtime.getRuntime().maxMemory());
11
12        /*
13         * Questa soluzione è migliore rispetto a utility class con
14         * solo metodi statici, specialmente quando l'implementazione
15         * potrebbe fornire istanze di tipo diverso a seconda
16         * del contesto di esecuzione specifico.
17         * Quindi è ok che java.lang.Math sia una utility class
18         */
19    }
20 }
```



# Singleton: Il caso di una classe Log

```
1 public class Log {
2     // si potrebbe cambiare assegnando una specializzazione
3     private static final Log SINGLETON = new Log();
4     // rendo invisibile da fuori il costruttore (o protected)
5     private Log(){};
6     // torno il singleton
7     public static Log getLog(){
8         return SINGLETON;
9     }
10    // da qui seguono i metodi istanza
11    // unico metodo di questo singleton
12    public void add(String s){
13        System.err.println(s);
14    }
15 }
```

```
1 public class UseLog {
2     public static void main(String[] s){
3         Log.getLog().add("Prova 1");
4         Log.getLog().add("Prova 2");
5     }
6 }
```

# Singleton: conseguenze

## Effetti benefici

- C'è un controllo “incapsulato” di chi vi accede
- Evita di dover portare i riferimenti all'oggetto nei campi di tutti le classi che lo usano
- È facile raffinare l'implementazione del singleton (via subclassing)
- Può gestire la creazione by-need (detta anche *lazy*) dell'oggetto
- Più flessibile dei metodi statici (che non hanno overriding)

## Critiche

- Il Singleton può essere problematico col multi-threading
- Crea dipendenze nascoste, gli user dipendono dal nome della classe
- Difficile tornare indietro dalla scelta di usare il singleton
- Incapsula due responsabilità distinte (creazione + aspetti interni)

# Singleton con “lazy initialization” (non thread-safe)

```
1 public class LogLazy {
2
3     private static LogLazy SINGLETON = null;
4
5     private LogLazy(){};
6
7     // Creo il SINGLETON alla prima chiamata
8     public static LogLazy getLog(){
9         if (SINGLETON == null){
10             SINGLETON = new LogLazy();
11         }
12         return SINGLETON;
13     }
14     // metodo del singleton
15     public void add(String s){
16         System.err.println(s);
17     }
18 }
```



# Singleton con “lazy initialization” e “thread-safe”

```
1 public class LogLazyTS {
2
3     // inner static classes are initialised at first use
4     private static class LazyHolder {
5         private static final LogLazyTS SINGLETON = new LogLazyTS();
6     }
7
8     private LogLazyTS(){};
9
10    // Creo il SINGLETON alla prima chiamata
11    public static LogLazyTS getLog(){
12        return LazyHolder.SINGLETON;
13    }
14    // metodo del singleton
15    public void add(String s){
16        System.err.println(s);
17    }
18 }
```



# Template Method: comportamentale, su classi

## Intento/motivazione

Definisce lo scheletro (template) di un algoritmo (o comportamento), lasciando l'indicazione di alcuni suoi aspetti alle sottoclassi.

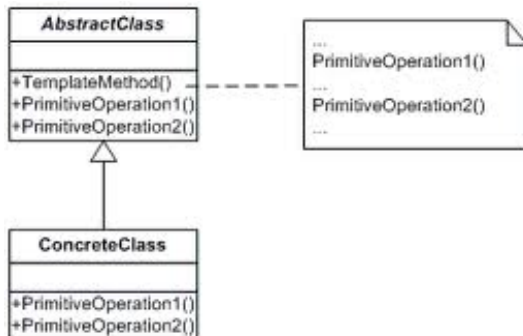
## Esempi

- In un input stream (`InputStream`), i vari metodi di lettura sono dei Template Method: dipendono dall'implementazione del solo concetto di lettura di un `int`
- Le interfacce funzionali con metodi di default che chiamano l'astratto

## Soluzione

- L'algoritmo è realizzato attraverso un metodo non astratto (il template method) di una classe astratta
- Questo realizza l'algoritmo, chiamando metodi astratti quando servono gli aspetti non noti a priori
- Una sottoclasse fornisce l'implementazione dei metodi astratti

# Template Method: UML





# Template Method: Una estensione di InputStream

```
1 import java.io.*;
2 import java.util.*;
3
4 public class UseRandomInputStream {
5
6     public static void main(String[] args) throws IOException {
7
8         // Il metodo InputStream.read(byte[]) è un Template Method
9         // Dipende dall'implementazione (non nota) di InputStream.read()
10
11         // Creo una sottoclasse in cui concretizzo read()
12         InputStream input = new InputStream(){ // Uno stream ad-hoc
13             int ct = 0;
14             public int read() throws IOException {
15                 return ct++;
16             }
17         };
18
19         // Ora provo InputStream.read(byte[]) e lo trovo concretizzato
20         final byte[] b = new byte[100];
21         input.read(b);
22         System.out.println(Arrays.toString(b));
23         // [0,1,2,3,4,5,6,...]
24     }
25 }
```

# Template Method: esempio BankAccount

```
1 public abstract class BankAccount {
2     private int amount;
3
4     public BankAccount(int amount){
5         this.amount = amount;
6     }
7
8     public abstract int operationFee(); // costo bancario operazione
9
10    public int getAmount(){
11        return this.amount;
12    }
13
14    public void withdraw(int n){ // template method
15        this.amount = this.amount - n - this.operationFee();
16    }
17    // main di prova
18    public static void main(String[] args){
19        final BankAccount b = new BankAccount(100){
20            public int operationFee(){ return 1; }
21        };
22        b.withdraw(20);
23        System.out.println(b.getAmount()); // 79
24    }
25 }
```

# Template Method: esempio con i metodi default

```
1 public interface SimpleIterator<X> {  
2  
3     X getNext();  
4  
5     default List<X> getListOfNext(int n){  
6         final List<X> l = new LinkedList<>();  
7         for (int i = 0; i < n; i++){  
8             l.add(getNext());  
9         }  
10        return l;  
11    }  
12  
13 }
```

```
1 public class UseSimpleIterator {  
2  
3     public static void main(String[] args){  
4         final Counter c = new Counter();  
5         final SimpleIterator<Integer> si =  
6             () -> {c.inc(); return c.getValue();};  
7  
8         System.out.println(si.getNext());  
9         System.out.println(si.getListOfNext(20));  
10    }  
11  
12 }
```

# Strategy: comportamentale, su oggetti

## Intento/motivazione

Definisce una famiglia di algoritmi, e li rende interscambiabili, ossia usabili in modo trasparente dai loro clienti

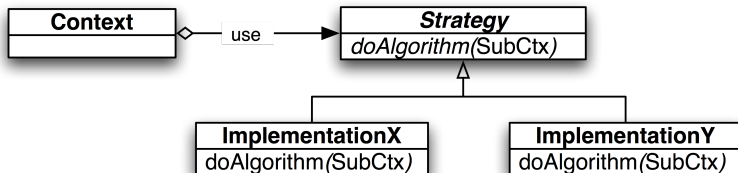
## Esempi

- Strategia di disposizione di componenti in una GUI (LayoutManager)
- Strategie di confronto fra due elementi per sorting (Comparable)
- Strategie di map, filter, etc.. negli Stream

## Soluzione

- Gli algoritmi sono realizzati tramite specializzazioni di una classe/interfaccia base
- Ai clienti passo un oggetto (di una specializzazione) della classe base
- Se la strategia è funzionale si usano facilmente le lambda (e viceversa)

# Strategy: UML



# Strategy: Sorting con comparatori

```
1 public class UseComparator {
2
3     public static void main(String[] args){
4         final List<Integer> list = Arrays.asList(200,31,142,65,35);
5         System.out.println(list);
6         // Creo tre strategie di comparazione diverse
7         final Comparator<Integer> c1 = new Comparator<Integer>(){
8             public int compare(Integer o1, Integer o2) {
9                 return o1 - o2;
10            }
11        };
12        final Comparator<Integer> c2 = new Comparator<Integer>(){
13            public int compare(Integer o1, Integer o2) {
14                return o2 - o1;
15            }
16        };
17        final Comparator<Integer> c3 = (o1,o2) -> o1%10 - o2%10;
18        // Uso le tre strategie
19        Collections.sort(list,c1);
20        System.out.println("cres: "+list);
21        Collections.sort(list,c2);
22        System.out.println("decr: "+list);
23        Collections.sort(list,c3);
24        System.out.println("unità: "+list);
25    }
26 }
```

# Strategy: Caso del BankAccount

```
1 @FunctionalInterface
2 public interface BankOperationFees {
3
4     int fee(int operationAmount);
5 }
```

```
1 public class StandardBankOperationFees implements BankOperationFees {
2
3     public int fee(int operationAmount){
4         return operationAmount < 1000 ? 1 : 2;
5     }
6 }
```



# Strategy: Caso del BankAccount

```
1 public class BankAccount {
2
3     private final BankOperationFees fees;
4     private int amount;
5
6     public BankAccount(BankOperationFees fees, int amount) {
7         this.fees = fees;
8         this.amount = amount;
9     }
10
11     public int getAmount() {
12         return this.amount;
13     }
14
15     public void withdraw(int n) {
16         this.amount = this.amount - n - this.fees.fee(n);
17     }
18
19     // main di prova
20     public static void main(String[] args) {
21         final BankOperationFees f = new StandardBankOperationFees();
22         final BankAccount b = new BankAccount(f, 100);
23         //final BankAccount b = new BankAccount(f, a -> a < 1000 ? 1 : 2);
24         b.withdraw(20);
25         System.out.println(b.getAmount()); // 79
26     }
27 }
```



# Strategy vs Template Method

## In comune

- Entrambi li si ottengono dall'esigenza di scorporare da una classe la gestione di una strategia o specializzazione
- Entrambi richiedono un behaviour aggiuntivo da realizzare

## Differenze

- Strategy è più flessibile, perché gli oggetti che rappresentano la specializzazione sono liberi dal dover estendere una certa classe, e quindi sono più facilmente riusabili (p.e. un `Comparator` è usabile con collection diverse)
- Template Method si integra con il subtyping, e quindi va usato quando a strategie specializzate devono corrispondere classi specializzate

## Altre note

- Negli `InputStream` le limitazioni del Template Method sono mitigate dal Decorator
- Con le lambda, l'approccio a Strategy diventa più naturale
- Valutare di usare il Template Method insieme a Strategy, ossia per definire gerarchie di strategie



# Decorator: strutturale, su oggetti

## Intento/motivazione

Aggiunge ad un oggetto ulteriori responsabilità, dinamicamente, e in modo più flessibile (e componibile) rispetto all'ereditarietà.

## Esempi

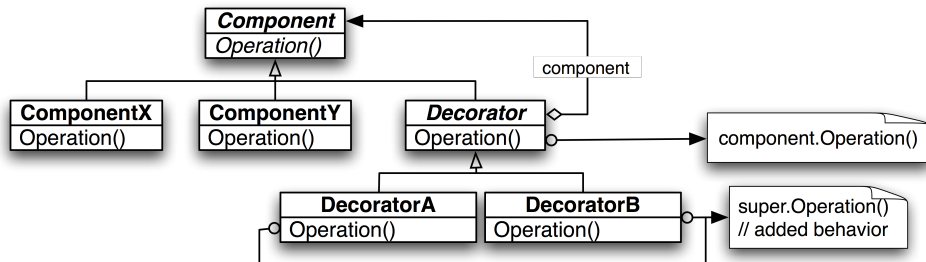
- Aggiungere (in modo componibile) la gestione “buffered” ad uno stream
- Aggiungere una barra di scorrimento ad un pannello
- Ottenere uno stream ordered da uno unordered

## Soluzione

- La classe base viene estesa con una nuova classe che è anche wrapper di un oggetto della classe base
  - Uno o più metodi potrebbero delegare semplicemente all'oggetto wrappato, altri modificare opportunamente, altri essere aggiuntivi
- ⇒ può essere visto come variante dello strategy (in cui la strategia è la realizzazione base del comportamento), e potrebbe includere dei template method



# Decorator: UML



# Esempio di problema

Data la seguente interfaccia Pizza..

```
1 public interface Pizza {  
2  
3     int getCost(); // prezzo in centesimi di euro  
4  
5     String getIngredients(); // lista di ingredienti a menù  
6  
7 }
```

Realizzare le seguenti astrazioni

- Margherita (6.50 euro, ingredienti: pomodoro + mozzarella)
- Aggiunta Salsiccia (1.50 euro), anche doppia o tripla eccetera
- Aggiunta Funghi (1 euro), anche doppia o tripla eccetera
- Pizza senza glutine (+10% costo)

Forniamo la soluzione col decoratore (tipica del concetto di “ingrediente”)

- Class concreta Margherita
- Decoratore astratto IngredientDecorator
- Specializzazioni Salsiccia, Funghi e GlutenFree



# Funzionalità di testing

```
1 public class Test {
2
3     // Completare l'assegnamento di p1,p2,p3,p4 opportunamente
4     @org.junit.Test
5     public void test(){
6         Pizza p1 = null; // funghi + salsiccia su base margherita
7         Pizza p2 = null; // margherita
8         Pizza p3 = null; // doppia salsiccia su base margherita
9         Pizza p4 = null; // come p3 ma stesa
10        // stampe di comodo
11        System.out.println(p1.getCost()+" "+p1.getIngredients());
12        System.out.println(p2.getCost()+" "+p2.getIngredients());
13        System.out.println(p3.getCost()+" "+p3.getIngredients());
14        System.out.println(p4.getCost()+" "+p4.getIngredients());
15        // test correttezza di costi e stringhe
16        assertEquals(p1.getCost(),900); // 6.50 + 1.00 + 1.50
17        assertEquals(p2.getCost(),650);
18        assertEquals(p3.getCost(),950); // 6.50 + 1.50 + 1.50
19        assertEquals(p4.getCost(),1045); // (6.50 + 1.50 + 1.50) + 10%
20        assertEquals(p1.getIngredients(),"Pomodoro, Mozzarella, Salsiccia,
21        Funghi");
22        assertEquals(p2.getIngredients(),"Pomodoro, Mozzarella");
23        assertEquals(p3.getIngredients(),"Pomodoro, Mozzarella, Salsiccia,
24        Salsiccia");
25        assertEquals(p4.getIngredients(),"[GF] Pomodoro, Mozzarella, Salsiccia,
26        Salsiccia");
27    }
28 }
```

# Classe Margherita

```
1  /*
2   * This is the basic Pizza of all
3   */
4
5
6  public class Margherita implements Pizza {
7
8      public int getCost(){
9          return 650;
10     }
11
12     public String getIngredients(){
13         return "Pomodoro, Mozzarella";
14     }
15
16 }
```

# Classe IngredientDecorator

```
1 public abstract class IngredientDecorator implements Pizza {
2
3     protected final Pizza decorated;
4
5     protected IngredientDecorator(Pizza decorated){
6         this.decorated = decorated;
7     }
8
9     public int getCost(){
10         return this.decorated.getCost();
11     }
12
13     public String getIngredients(){
14         return this.decorated.getIngredients();
15     }
16
17 }
```



# Classi per gli ingredienti..

```
1 public class Salsiccia extends BasicIngredient {
2
3     public Salsiccia(Pizza p){
4         super("Salsiccia",150,p);
5     }
6 }
```

```
1 public class Funghi extends BasicIngredient {
2
3     public Funghi(Pizza p){
4         super("Funghi",100,p);
5     }
6 }
```

```
1 public class GlutFree extends IngredientDecorator {
2
3     public GlutFree(Pizza p){
4         super(p);
5     }
6
7     public int getCost(){
8         return (int)(super.getCost() * 1.1);
9     }
10
11     public String getIngredients(){
12         return "[GF] "+super.getIngredients();
13     }
14 }
```



# Iterator: strutturale, su oggetti

## Intento/motivazione

Fornisce un modo per accedere agli elementi di un aggregato, sequenzialmente, senza esporne la rappresentazione interna.

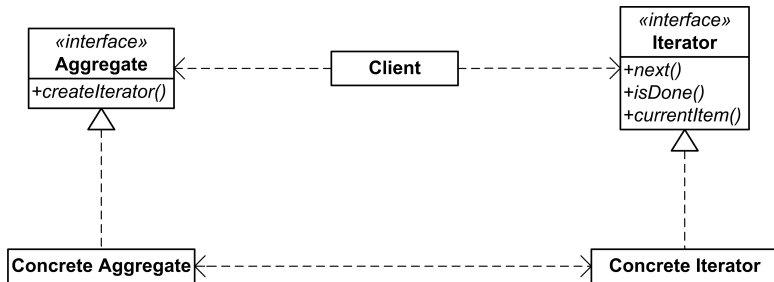
## Esempi

- Gli iteratori delle collezioni di Java
- Un iteratore sui componenti di una GUI
- Variante Splititerator negli stream, e ListIterator

## Soluzione

- La classe iterabile (collezione, stream) di partenza fornisce un metodo per creare un iteratore
- L'iteratore ha metodi per accedere sequenzialmente agli elementi dell'iterabile
- (Con multiple specializzazioni, se ne nascondono i dettagli interni)

# Iterator: UML



# Observer: comportamentale, su oggetti

## Intento/motivazione

Definisce una dipendenza dinamica uno-molti: quando uno cambia, molti vengono notificati/aggiornati

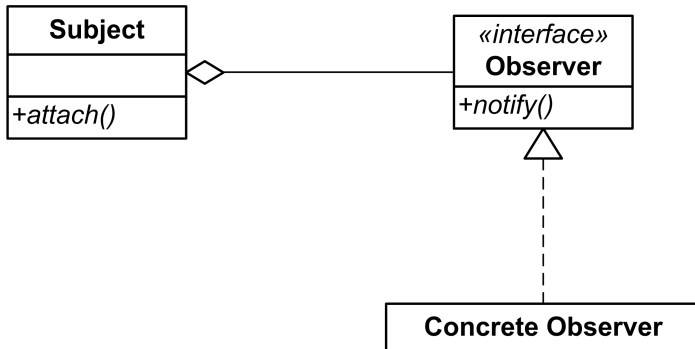
## Esempi

- Un componente grafico potrebbe avere agganciati vari osservatori
- Qualunque dispositivo del S.O. potrebbe essere agganciato a vari osservatori

## Soluzione

- Un *subject* ha un metodo per collegargli un nuovo ascoltatore
- Quando una determinata condizione vale nel *subject*, in tutti gli ascoltatori viene chiamato un certo metodo (*notify* o *update*)
- Molto usato nei sistemi embedded, in OOP spesso solo per le GUI

# Observer: UML



# Observer: Classi generiche per osservazioni

```
1 import java.util.*;
2
3 public class ESource<T> {
4
5     private final Set<EObserver<? super T>> set = new HashSet<>();
6
7     public void addEObserver(EObserver<? super T> obs){
8         this.set.add(obs);
9     }
10
11     public void notifyEObservers(T arg){
12         for (final EObserver<? super T> obs : this.set){
13             obs.update(this, arg);
14         }
15     }
16
17 }
```

```
1 public interface EObserver<T> {
2
3     public void update(ESource<? extends T> s, T arg);
4
5 }
```

# Observer: Una collezione emettitrice di eventi

```
1 import java.util.*;
2
3 public class SetWithEvents<T> extends ESource<Integer>{
4
5     // Incapsula un Set
6     private final Set<T> set = new HashSet<>();
7
8     // Ad ogni aggiunta notifica la nuova dimensione
9     public void add(T t){
10         this.set.add(t);
11         this.notifyEObservers(set.size());
12     }
13
14     public Set<T> getCopy(){
15         return Collections.unmodifiableSet(this.set);
16     }
17 }
```



# Observer: Uso della collezione

```
1 import javax.swing.*;
2
3 public class UseSetWithEvents{
4
5     public static void main(String[] args){
6         final SetWithEvents<String> set = new SetWithEvents<>();
7         set.addEObserver((s,arg)->System.out.println("Nuova dim. elenco: "+arg))
8         ;
9         // Aggiungo un osservatore che stampa a video
10        set.addEObserver((s,arg)-> {
11            if (arg > 4){
12                SwingUtilities.invokeLater(
13                    ()->JOptionPane.showMessageDialog(null, "Dim. critica"));
14            }
15        });
16        // Aggiungo un osservatore che mostra un OptionPane se > 4
17        set.add("1");
18        set.add("2");
19        set.add("3");
20        set.add("4");
21        set.add("5");
22        set.add("6");
23        System.out.println(set.getCopy());
24    }
25 }
```

# Adapter: strutturale, su classi/oggetti

## Intento/motivazione

Consente ad una classe di adattarsi all'interfaccia (diversa) richiesta da un cliente

## Esempi

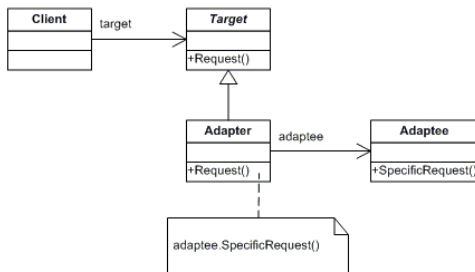
- Il metodo `Arrays.asList` (adatta un array ad una lista)
- La classe `InputStreamReader` (adatta un `InputStream` ad un `Reader`)

## Soluzione

- Si crea una nuova classe (adapter) che implementa l'interfaccia richiesta e wrappa l'oggetto di partenza (adaptee)
- L'adapter redirige opportunamente le chiamate all'adaptee (o via inheritance o via delegazione)



# Adapter: UML



## Varianti

- In Java addirittura l'Adapter può essere una inner class (istanza) per Adaptee
  - ▶ E' considerato uno degli utilizzi più rilevanti per le inner class (istanza)
  - ▶ Consente un incapsulamento ottimale dell'Adapter
- Adapter potrebbe estendere l'Adaptee invece che averne un riferimento

# Esempio: handler d'eventi locale in una GUI

```
1 public class GUI extends JFrame{
2
3     private static final long serialVersionUID = 2298666777798069846L;
4
5     public GUI(){
6         this.setSize(320,200);
7         final JButton jb = new JButton("Action!");
8         this.getContentPane().setLayout(new FlowLayout());
9         this.getContentPane().add(jb);
10        jb.addActionListener(new HandlerAdapter());
11        this.setVisible(true);
12    }
13
14    // metodo in GUI da adattare a ActionListener
15    private void handlingButtonAction(){
16        System.exit(0);
17    }
18
19    private class HandlerAdapter implements ActionListener{
20        @Override
21        public void actionPerformed(ActionEvent e) {
22            GUI.this.handlingButtonAction();
23        }
24    }
25
26    public static void main(String[] args){
27        new GUI();
28    }
```

# Outline

- 1 Ciclo di vita del software
- 2 Progettazione architetturale
- 3 Design Pattern e Progettazione di Dettaglio
- 4 Pattern già incontrati
- 5 Nuovi pattern: creazionali**

# Pattern creazionali

## Pattern creazionali

- Singleton, Factory Method, Abstract Factory, Builder, Prototype

## Motivazioni generali

- Costruire correttamente oggetti o famiglie di oggetti è una responsabilità non sempre banale, e pertanto può essere meglio scorporarla dalle classi stesse
- La costruzione di un oggetto è tipicamente fuori dalla sua interfaccia d'uso, e questo quindi non consente un buon disaccoppiamento e riuso

## Una nota

- Sebbene esista una differenza tecnica chiara fra questi, è possibile ottenere soluzioni ibride: anche in rete/letteratura non c'è assoluta coerenza nella loro interpretazione

# Factories (ossia “fabbriche” di oggetti)

## Static Factory

- Una classe ha un metodo statico per generare istanze di sue (o altre) specializzazioni
- Es.: l'interfaccia `Stream`, il `Singleton` è un caso particolare..

## Simple Factory

- Come `Static Factory`, ma il metodo è non statico e in una classe separata (factory)

## Factory Method

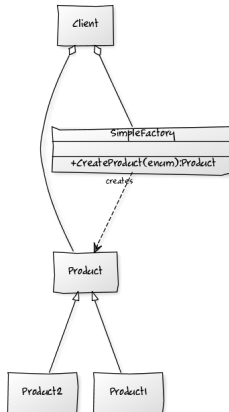
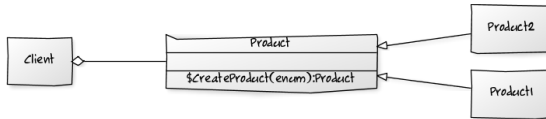
- Come `Simple Factory`, ma la costruzione avviene in sottoclassi della factory, che è un'interfaccia

## Abstract Factory

- Come `Factory Method`, ma consente di costruire più oggetti tra loro correlati



# Static e Simple Factory

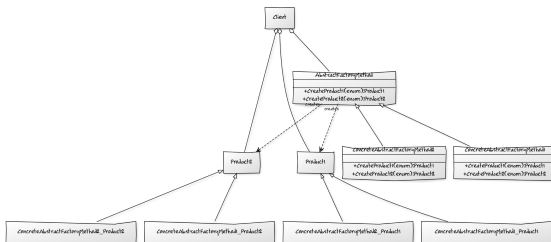
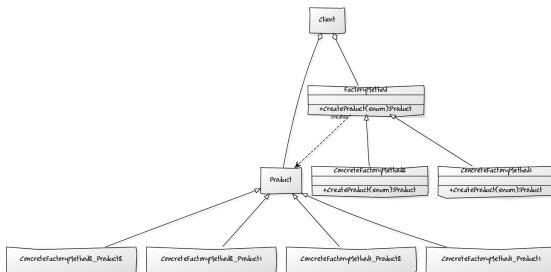


# Static Factory: Esempio Persona

```
1 public class Persona {
2
3     private String nome;
4     private String cognome;
5
6     private Persona(){}
7
8     public static Persona createPersona(String nome,String cognome){
9         final Persona p = new Persona(); // in futuro: new SpecificPersona()..
10        p.nome = nome;
11        p.cognome = cognome;
12        return p;
13    }
14
15
16    public static void main(String[] args) {
17        // TODO Auto-generated method stub
18        final Persona p = Persona.createPersona("mario", "rossi");
19
20    }
21
22 }
```



# Factory Method e Abstract Factory





# Factory Method: creazionale, su oggetti

## Intento/motivazione

Definisce una interfaccia per creare oggetti, lasciando alle sottoclassi il compito di decidere quale classe istanziare e come

## Esempi

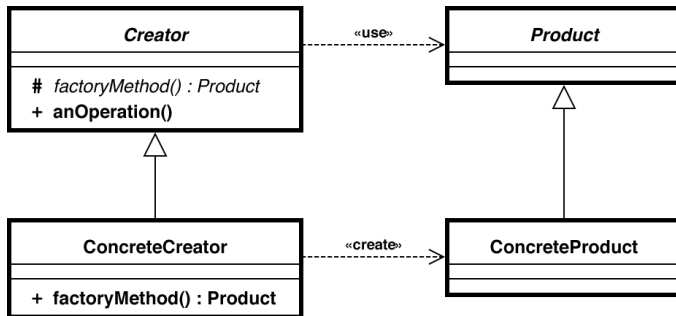
- Un framework deve creare oggetti, ma sue specializzazioni devono crearne versioni specializzate

## Soluzione

- Una interfaccia creatrice fornisce il metodo factory col compito di creare e ritornare l'oggetto
- Tale interfaccia viene poi specializzata, e incapsula la logica di creazione dell'oggetto

⇒ ..spesso frainteso con static o simple factory

# Factory Method: UML



# Factory Method: Esempio Persona e FactoryPersona

```
1 import java.util.Optional;
2
3 public interface Person {
4
5     String getName();
6
7     String getSurname();
8
9     Optional<String> getCity();
10
11     int getYear();
12
13 }
```

```
1 public interface PersonFactory {
2
3     Person createBasic(String name, String surname);
4
5     Person createAdvanced(String name, String surname, String city);
6
7 }
```



# Abstract Factory: creazionale, su oggetti

## Intento/motivazione

Definisce una interfaccia per creare famiglie di oggetti tra loro correlati o dipendenti, lasciando alle sottoclassi il compito di decidere quali classe istanziare e come

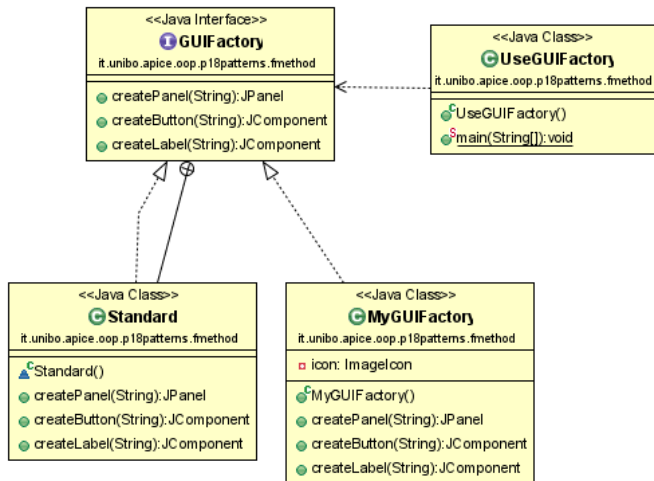
## Esempi

- Un framework deve creare una famiglia di oggetti, ma sue specializzazioni devono crearne versioni specializzate

## Soluzione

- Una interfaccia creatrice fornisce i metodi factory col compito di creare e ritornare gli oggetti
- Tale interfaccia viene poi specializzata, e incapsula la logica di creazione degli oggetti

# Abstract Factory: An example



# Abstract Factory: Esempio GUIFactory

```
1 import javax.swing.*;  
2  
3 public interface GUIFactory {  
4     // 3 factory methods  
5     JPanel createPanel(String text);  
6  
7     JComponent createButton(String text);  
8  
9     JComponent createLabel(String text);  
10 }
```

# Abstract Factory: Specializzazione MyGUIFactory

```
1 import javax.swing.*;
2
3 public class MyGUIFactory implements GUIFactory{
4
5     private static final Font MY_FONT = new Font("Garuda",Font.BOLD,12);
6     private static final Color MY_COLOR = new Color(255,255,0);
7
8     @Override
9     public JPanel createPanel(String text) {
10         final JPanel p = new JPanel();
11         p.setBorder(new javax.swing.border.TitledBorder(text));
12         return p;
13     }
14
15     @Override
16     public JComponent createButton(String text) {;
17         final JButton b = new JButton(text);
18         b.setFont(MY_FONT);
19         b.setBackground (MY_COLOR);
20         return b;
21     }
22
23     @Override
24     public JComponent createLabel(String text) {
25         final JLabel j = new JLabel(text);
26         j.setFont(MY_FONT);
27         return j;
28     }
```

# Abstract Factory: Uso di GUIFactory

```
1 import java.util.*;
2
3 public class UseGUIFactory {
4
5     public static void main(String[] args) {
6
7         //System.out.println(Arrays.toString(GraphicsEnvironment.
8             getLocalGraphicsEnvironment().getAllFonts()));
9         //final GUIFactory factory = new MyGUIFactory();
10        final GUIFactory factory = new Standard();
11
12        final JFrame frame = new JFrame("Testing GUIFactory");
13        frame.setSize(640,480);
14        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15
16        final JPanel panel = factory.createPanel("main");
17        panel.add(factory.createLabel("Label.."));
18        panel.add(factory.createButton("Button.."));
19        frame.getContentPane().add(panel);
20        frame.setVisible(true);
21    }
22 }
```





# Variante dell'interfaccia con defaults

```
1 public interface GUIFactory {
2
3     // 3 factory methods, in questo caso con implementazione di default
4     default JPanel createPanel(String text){
5         return new JPanel();
6     }
7     default JComponent createButton(String text){
8         return new JButton(text);
9     }
10    default JComponent createLabel(String text){
11        return new JLabel(text);
12    }
13
14    // Torna una implementazione di default
15    public static GUIFactory createStandard(){
16        return new GUIFactory(){};
17    }
18
19 }
```



# Builder: creazionale, su oggetti

## Intento/motivazione

Definisce una strategia separata per la creazione step-by-step di un oggetto

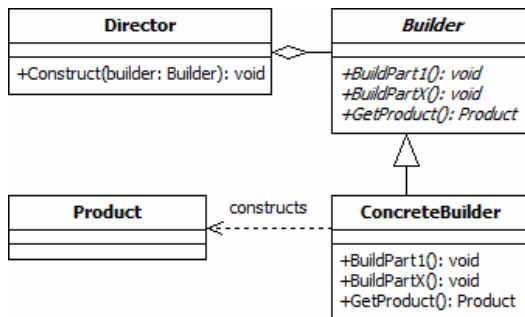
## Esempi

- `Stream.Builder`, `StringBuilder`, `StringBuffer`
- Una pipeline per gli stream assomiglia ad un builder per un iteratore

## Soluzione

- Una classe separata ha i metodi per settare le varie proprietà dell'oggetto da costruire, e quando tutto è pronto si chiama un metodo di building
- Spesso è comodo usare una interfaccia “fluent”, ossia dove i setter tornano `this`
- Comodo per evitare la proliferazione di costruttori in una classe
- Comodo per costruire oggetti immutabili complessi
- Dove possibile/utile, lasciare i pochi parametri obbligatori nel costruttore del builder

# Builder: UML



# Esempio: StringBuilder

```
1  /*
2   * StringBuilder is an optimised builder of (immutable) strings
3   * StringBuffer is to be used if thread safety is needed
4   */
5
6  public class UseStringBuilder {
7
8      public static void main(String[] args) {
9
10         List<Integer> l = Arrays.asList(10,20,30,40,50,60);
11         String s = "[";
12         for (int i: l){
13             s = s + i + ",";
14         }
15         s = s + "]";
16         System.out.println(s);
17
18
19         final StringBuilder sb = new StringBuilder();
20         sb.append("Costruzione di stringa :");
21         sb.append(10);
22         sb.append(20);
23         sb.append(30);
24         sb.append(40);
25         sb.append(50);
26         String s2 = sb.toString();
27         System.out.println(s2);
28     }
```

# Esempio: un builder per persone (1/2)

```
1 import java.util.Optional;
2
3 public class Person {
4
5     final private String name;
6     final private String surname;
7     final private Optional<String> city;
8     final private int year;
9
10    private Person(String name, String surname, Optional<String> city, int year) {
11        super();
12        this.name = name;
13        this.surname = surname;
14        this.city = city;
15        this.year = year;
16    }
```



## Esempio: un builder per persone (2/2)

```
1 public static class Builder{
2
3     private String name;
4     private String surname;
5     private Optional<String> city = Optional.empty();
6     private Optional<Integer> year = Optional.empty();
7
8     public Builder(String name, String surname){
9         this.name = name;
10        this.surname = surname;
11    }
12
13    public Builder city(String s){
14        this.city = Optional.ofNullable(s);
15        return this;
16    }
17
18    public Builder year(int i){
19        this.year = Optional.of(i).filter(k->k>=1900 || k<=2015);
20        return this;
21    }
22
23    public Person build() throws IllegalStateException{
24        if (this.name == null || this.surname == null || !this.year.isPresent()){
25            throw new IllegalStateException("");
26        }
27        return new Person(this.name, this.surname, this.city, this.year.get());
28    }
29
30 }
31
32 }
```

# Esempio: UsePersonBuilder

```
1  /*
2   * Builder implementato come member class e stile "fluent"
3   */
4
5  public class UsePersonBuilder {
6
7      public static void main(String[] args) {
8
9          final Person p = new Person.Builder("Mirko", "Viroli")
10              .city("Cesena")
11              .year(1973)
12              .build();
13
14          // Creazione errata, intercettata dalla logica del Builder
15          final Person p2 = new Person.Builder("Gino", "Bianchi")
16              //.year(2016)
17              .build();
18
19      }
20 }
```



## Riferimenti

- Composite: meccanismo di composizione ad albero di componenti
- Memento: per consentire forme di Undo/Redo di modifiche
- Lightweight: per gestire un pool di oggetti per ottimizzare performance
- Visitor: per incapsulare la logica di visita di un albero
- Command: per rappresentare in un argomento il tipo di comando da eseguire

