

# Ingegneria del Software L-A

## Design Pattern

### *Design Pattern*

Nel 1977, Christopher Alexander disse:

***"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 the solution a million times over, without ever doing it the same way twice"***

Parlava di costruzioni civili e di città

## *Design Pattern*

- La stessa frase è applicabile anche alla **progettazione *object-oriented***
- In questo caso, le soluzioni utilizzeranno
  - oggetti, classi e interfacce
  - invece che pareti e porte...

3

Ingegneria del Software L-A - Design Pattern

## *Design Pattern*

- **Obiettivi**
  - Risolvere problemi progettuali specifici
  - rendere i progetti *object-oriented* più flessibili e riutilizzabili
- Ogni design pattern
  - Cattura e formalizza l'**esperienza acquisita** nell'affrontare e risolvere uno specifico problema progettuale
  - Permette di **riutilizzare** tale **esperienza** in altri casi simili

4

Ingegneria del Software L-A - Design Pattern

## *Design Pattern*

- Ogni *design pattern* ha **quattro elementi essenziali**
  - un **nome** (significativo) – identifica il *pattern*
  - il **problema** – descrive quando applicare il *pattern*
  - la **soluzione** – descrive il *pattern*, cioè gli elementi che lo compongono (classi e istanze) e le loro relazioni, responsabilità e collaborazioni
  - le **conseguenze** – descrivono vantaggi e svantaggi dell'applicazione del *pattern*  
Permettono di valutare le alternative progettuali

5

Ingegneria del Software L-A - Design Pattern

## Classificazione dei *Design Pattern*

- **Pattern di creazione** (*creational pattern*)  
Risolgono problemi inerenti  
il processo di creazione di oggetti
- **Pattern strutturali** (*structural pattern*)  
Risolgono problemi inerenti  
la composizione di classi o di oggetti
- **Pattern comportamentali** (*behavioral pattern*)  
Risolgono problemi inerenti  
le modalità di interazione e di distribuzione delle  
responsabilità tra classi o tra oggetti

6

Ingegneria del Software L-A - Design Pattern

## Classificazione dei *Design Pattern*

<b><i>Pattern di creazione</i></b>	<b><i>Pattern strutturali</i></b>	<b><i>Pattern comportamentali</i></b>
Abstract Factory Builder <b>Factory Method</b> Prototype <b>Singleton</b>	<b>Adapter</b> Bridge <b>Composite</b> <b>Decorator</b> Facade <b>Flyweight</b> Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento <b>Observer</b> <b>State</b> <b>Strategy</b> <b>Template Method</b> <b>Visitor</b>

7

Ingegneria del Software L-A - Design Pattern

## *Pattern SINGLETON*

- Assicura che una classe abbia **una sola istanza** e fornisce un punto di accesso globale a tale istanza
- La classe deve:
  - tenere traccia della sua sola istanza
  - intercettare tutte le richieste di creazione, al fine di garantire che nessuna altra istanza venga creata
  - fornire un modo per accedere all'istanza unica

8

Ingegneria del Software L-A - Design Pattern

## Pattern SINGLETON

```
public class Singleton
{
    ... attributi membro di istanza ...
    private static Singleton _instance = null;
    protected Singleton()
    {
        { inizializzazione istanza }
    }
    public static Singleton GetInstance()
    {
        {
            if(_instance == null)
                _instance = new Singleton();
            return _instance;
        }
    }
    ... metodi pubblici, protetti e privati ...
}
```

9

Ingegneria del Software L-A - Design Pattern

## Pattern SINGLETON

- **Alternativa:** classe non istanziabile con soli membri statici
- Perché un *singleton*?
  - Creo il *singleton* (e quindi lo inizializzo) solo la prima volta che mi serve
  - Gli attributi membro di classe vengono inizializzati in un ordine non definito prima che il controllo passi al **main** (eccezione: costruttore statico in C#)
  - Posso specializzare il *singleton* e creare nella **GetInstance** una istanza specializzata che dipende dal contesto corrente

10

Ingegneria del Software L-A - Design Pattern

## Pattern SINGLETON

```
public static Singleton GetInstance()
{
    if(_instance == null)
        _instance = CreateInstance();
    return _instance;
}

private static Singleton CreateInstance()
{
    if(...)
        return new SubSingletonA();
    else if(...)
        return new SubSingletonB();
    else
        return new SubSingletonC();
}
```

11

Ingegneria del Software L-A - Design Pattern

## Pattern FLYWEIGHT

- Descrive come condividere oggetti “leggeri” (cioè a granularità molto fine) in modo tale che il loro uso non sia troppo costoso
- Un *flyweight* è un **oggetto condiviso** che può essere utilizzato simultaneamente ed efficientemente da più clienti (del tutto indipendenti tra loro)
- Benché condiviso, **non deve essere distinguibile da un oggetto non condiviso**
- Non deve fare ipotesi sul contesto nel quale opera

12

Ingegneria del Software L-A - Design Pattern

## Pattern FLYWEIGHT

- Distinzione tra stato intrinseco e stato estrinseco
- **Stato intrinseco**
  - **Non dipende dal contesto di utilizzo** e quindi **può essere condiviso** da tutti i clienti
  - Memorizzato nel *flyweight*
- **Stato estrinseco**
  - **Dipende dal contesto di utilizzo** e quindi **non può essere condiviso** dai clienti
  - Memorizzato nel cliente o calcolato dal cliente
  - Viene passato al *flyweight* quando viene invocata una sua operazione

13

Ingegneria del Software L-A - Design Pattern

## Pattern FLYWEIGHT

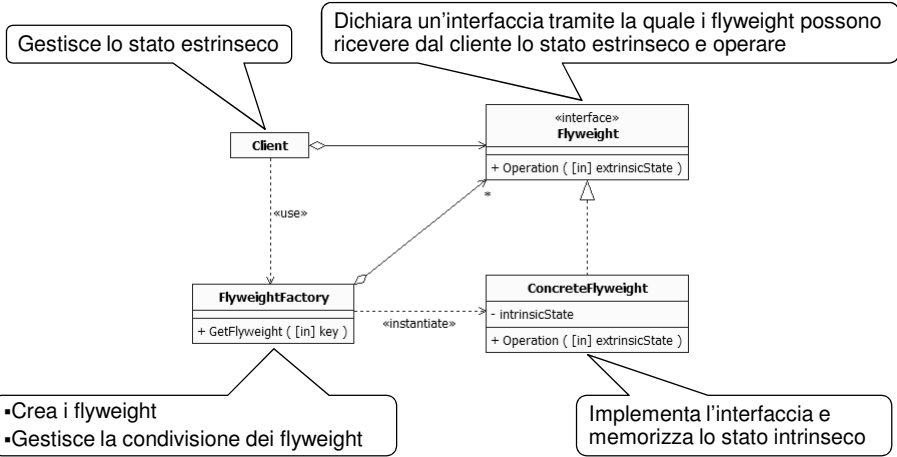
- Per assicurare una corretta condivisione, i clienti
  - non devono istanziare direttamente i *flyweight*
  - ma devono ottenerli esclusivamente tramite una **FlyweightFactory**

```
GetFlyweight (key)
{
    if(flyweights[key] not exist)
        create flyweights[key]
    return flyweights[key]
}
```

14

Ingegneria del Software L-A - Design Pattern

# Pattern FLYWEIGHT



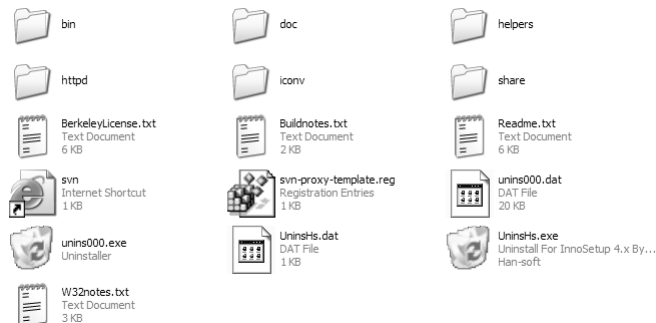
15

Ingegneria del Software L-A - Design Pattern

# Pattern FLYWEIGHT

## Esempio

- Si supponga di usare il pattern flyweight per condividere delle icone tra vari clienti



16

Ingegneria del Software L-A - Design Pattern



## Pattern FLYWEIGHT

### Esempio

- Lo **stato intrinseco** (memorizzato nel flyweight) comprenderà tutte le informazioni che i clienti devono (e possono) condividere:
  - Nome dell'icona
  - Bitmap dell'icona
  - Dimensioni originali, ...
- Lo **stato estrinseco** (memorizzato nel cliente) comprenderà il contesto in cui l'icona dovrà essere disegnata (dipendente dal singolo cliente):
  - Posizione dell'icona
  - Dimensioni richieste, ...

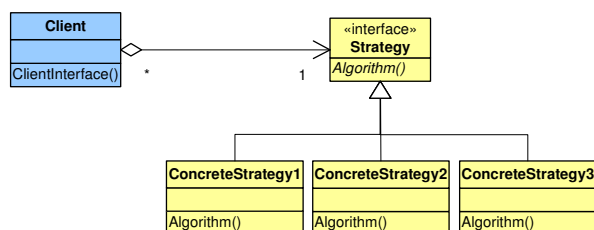
**Esempio**

17

Ingegneria del Software L-A - Design Pattern

## Pattern STRATEGY

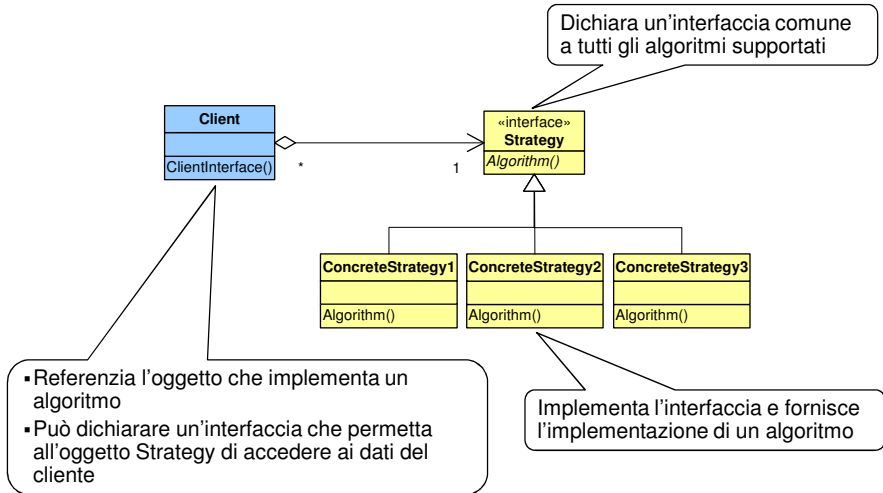
- Permette di
  - definire un insieme di algoritmi tra loro correlati,
  - incapsulare tali algoritmi in una gerarchia di classi e
  - rendere gli algoritmi intercambiabili



18

Ingegneria del Software L-A - Design Pattern

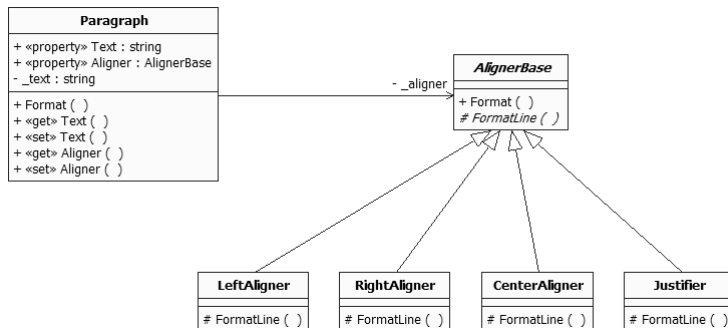
# Pattern STRATEGY



# Pattern STRATEGY

## Esempio

- **Allineamento del testo di un paragrafo**  
Esistono politiche diverse di allineamento



## *Pattern STRATEGY*

### Esempio

- **AlignerBase**
  - suddivide il testo in linee (**Format**)
  - delega alle sue sottoclassi l'allineamento delle singole linee (**FormatLine**)
- **Paragraph** utilizza i servizi di un "Aligner" specificato dinamicamente *run-time*
- È possibile realizzare gli "Aligner" utilizzando il pattern *flyweight*

**Esempio**

21

Ingegneria del Software L-A - Design Pattern

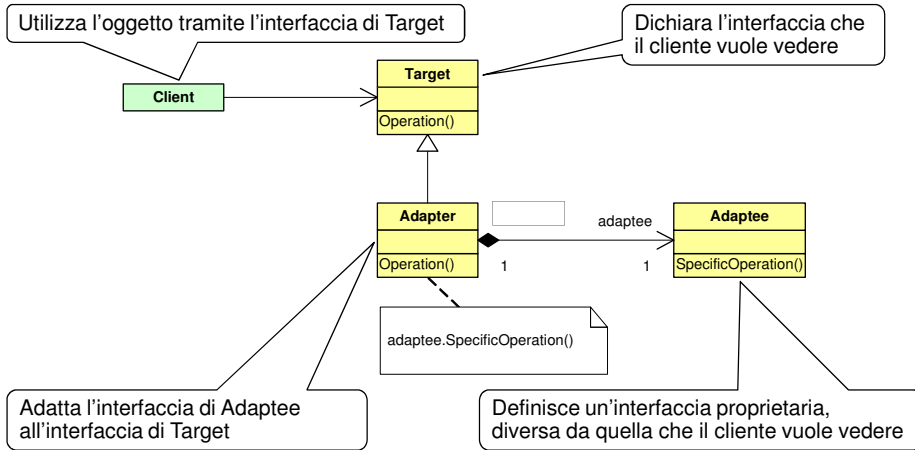
## *Pattern ADAPTER*

- Converte l'interfaccia originale di una classe nell'interfaccia (diversa) che si aspetta il cliente
- Permette a classi che hanno interfacce incompatibili di lavorare insieme
- Si usa quando
  - **si vuole riutilizzare una classe esistente e**
  - la sua interfaccia non è conforme a quella desiderata
- Noto anche come *wrapper*

22

Ingegneria del Software L-A - Design Pattern

## Pattern ADAPTER



23

Ingegneria del Software L-A - Design Pattern

## Pattern ADAPTER

- In C++, si potrebbe scrivere:  

```
class Adapter : public Target, private Adaptee
```
- Cioè ereditare l'interfaccia di **Target** e l'implementazione di **Adaptee**
- In un linguaggio in cui non è ammesso ereditare l'implementazione, conviene utilizzare la composizione

**Esempio**

24

Ingegneria del Software L-A - Design Pattern

## *Pattern DECORATOR*

- Permette di **aggiungere responsabilità** a un oggetto dinamicamente
- Fornisce un'**alternativa flessibile alla specializzazione**
  - In alcuni casi, le estensioni possibili sono talmente tante che per poter supportare ogni possibile combinazione, si dovrebbe definire un numero troppo elevato di sottoclassi

25

Ingegneria del Software L-A - Design Pattern

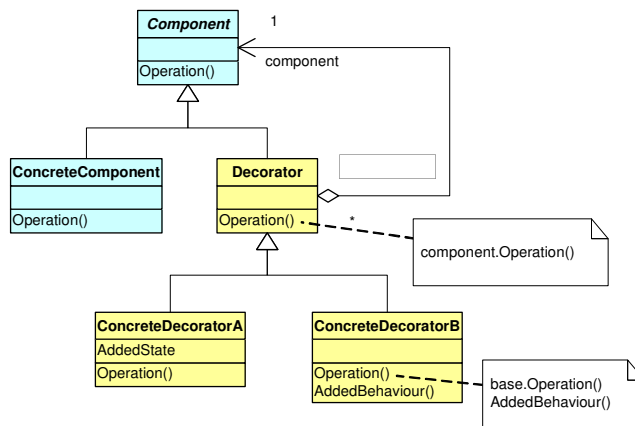
## *Pattern DECORATOR*

- TextBox
  - BorderTextBox
  - FilledTextBox
  - VerticalTextBox
  - BorderFilledTextBox
  - BorderVerticalTextBox
  - BorderFilledVerticalTextBox
  - FilledVerticalTextBox
- E se volessi
  - 2 o più bordi
  - Cambiare il font
  - ...

26

Ingegneria del Software L-A - Design Pattern

## Pattern DECORATOR



27

Ingegneria del Software L-A - Design Pattern

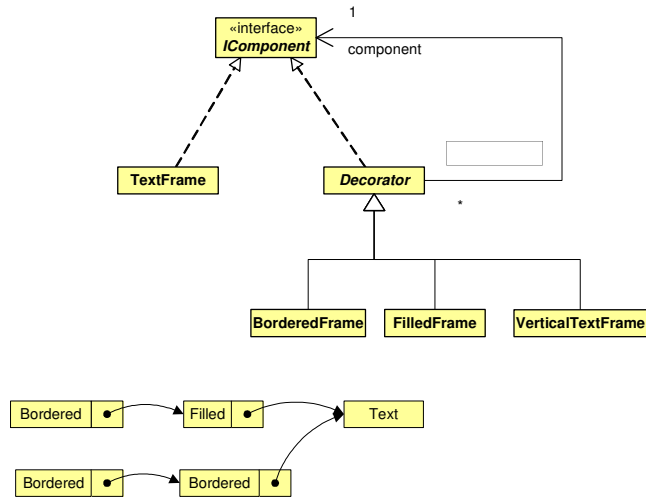
## Pattern DECORATOR

- **Component (interfaccia o classe astratta)**
  - Dichiarare l'interfaccia di tutti gli oggetti ai quali deve essere possibile aggiungere dinamicamente responsabilità
- **ConcreteComponent**
  - Definisce un tipo di oggetto al quale deve essere possibile aggiungere dinamicamente responsabilità
- **Decorator (classe astratta)**
  - Mantiene un riferimento a un oggetto di tipo Component e definisce un'interfaccia conforme all'interfaccia di Component
- **ConcreteDecorator**
  - Aggiunge responsabilità al componente referenziato

28

Ingegneria del Software L-A - Design Pattern

## Pattern DECORATOR



29

Ingegneria del Software L-A - Design Pattern

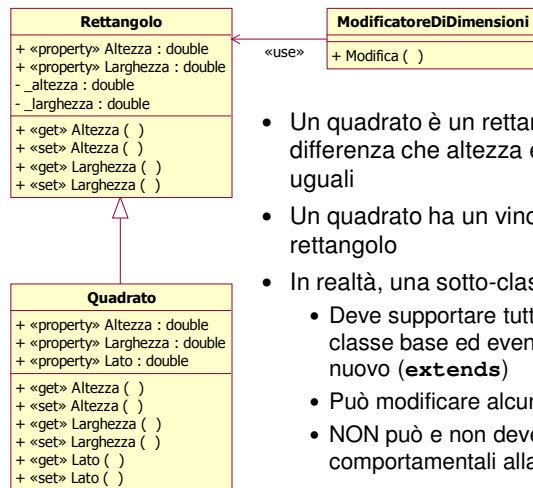
## Ereditarietà dinamica

- Una sotto-classe deve sempre essere una **versione più specializzata** della sua super-classe (o classe base)
- Un buon test sul corretto utilizzo dell'ereditarietà è che sia valido il **principio di sostituibilità di Liskov**:  
“*B è una sotto-classe di A se e solo se ogni programma che utilizzi oggetti di classe A può utilizzare oggetti di classe B senza che il comportamento logico del programma cambi*”
- Perché ciò sia valido, è necessario che:
  - le **pre-condizioni** di tutti i metodi della sotto-classe siano **uguali o più deboli**
  - le **post-condizioni** di tutti i metodi della sotto-classe siano **uguali o più forti**
  - ogni metodo ridefinito nella sotto-classe deve mantenere la **semantica** del metodo originale

30

Ingegneria del Software L-A - Design Pattern

# Ereditarietà dinamica



## Esempio S1

- Un quadrato è un rettangolo con la sola differenza che altezza e larghezza devono essere uguali
- Un quadrato ha un vincolo in più rispetto al rettangolo
- In realtà, una sotto-classe
  - Deve supportare tutto il comportamento della classe base ed eventualmente aggiungerne di nuovo (**extends**)
  - Può modificare alcuni aspetti del comportamento
  - NON può e non deve aggiungere vincoli comportamentali alla classe base!

31

Ingegneria del Software L-A - Design Pattern

# Ereditarietà dinamica

- Il metodo **Modifica** della classe **ModificatoreDiDimensioni**
  - funziona correttamente su un **Rettangolo**
  - ma NON funziona correttamente su un **Quadrato**
- Quindi non è possibile passare un'istanza di **Quadrato** dove è prevista un'istanza di **Rettangolo** (il principio di sostituzione di Liskov è violato!)
- **Conclusione:** un quadrato NON è un rettangolo perché pone dei nuovi vincoli al concetto di rettangolo
- Come possiamo tenere conto di ciò che il rettangolo e il quadrato hanno in comune?

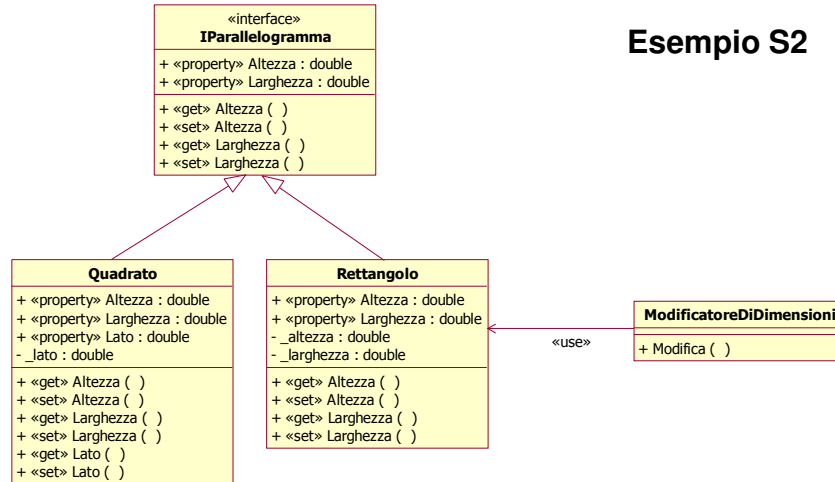
32

Ingegneria del Software L-A - Design Pattern



# Ereditarietà dinamica

## Esempio S2



33

Ingegneria del Software L-A - Design Pattern

# Ereditarietà dinamica

- Cosa intendiamo esattamente per Rettangolo e per Quadrato?
- **Rettangolo**: parallelogramma i cui angoli sono retti
- **Parallelogramma**: quadrilatero i cui lati opposti sono paralleli tra loro
- **Quadrilatero**: poligono avente quattro lati e quattro angoli
  - Quadrilateri notevoli sono il quadrato, il rettangolo, il parallelogramma, il rombo e il trapezio
- **Poligono**: figura geometrica limitata da una linea poligonale chiusa
- **Rombo**: parallelogramma equilatero in cui gli angoli adiacenti sono diversi tra loro
- **Quadrato**: parallelogramma equilatero ed equiangolo

34

Ingegneria del Software L-A - Design Pattern

## Ereditarietà dinamica

- Cosa intendiamo **esattamente** per Rettangolo e per Quadrato **nella nostra applicazione**?
- **Ipotesi**: abbiamo a che fare esclusivamente con parallelogrammi
- 1. **Lati e angoli NON sono modificabili**
  - Definisco quattro classi concrete che derivano dalla classe astratta **Parallelogramma** (o implementano **IParallelogramma**): **Rettangolo, Quadrato, Rombo, ParallelogrammaGenerico**
  - Uso una **factory** che in base ai valori dei lati e degli angoli mi istanzia un rettangolo (che NON deve avere i lati uguali), un quadrato, un rombo o un parallelogramma generico

35

Ingegneria del Software L-A - Design Pattern

## Ereditarietà dinamica

2. **Lati e angoli sono modificabili**
  - Definisco un'unica classe concreta **Parallelogramma** le cui istanze possono comportarsi a seconda del loro stato come: un rettangolo, un quadrato, un rombo, o un parallelogramma generico

36

Ingegneria del Software L-A - Design Pattern

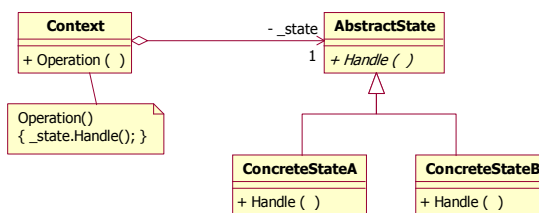
## Ereditarietà dinamica

- Come può un oggetto cambiare comportamento, al cambiare del suo stato?
- 1<sup>a</sup> possibilità: **si cambia la classe dell'oggetto *run-time*** – nella maggior parte dei linguaggi di programmazione a oggetti, questo non è possibile (inoltre, è meglio che un oggetto non possa cambiare classe durante la sua esistenza – la classe di un oggetto deve basarsi sulla sua essenza e non sul suo stato)
- 2<sup>a</sup> possibilità: **si utilizza il *pattern State*** che usa un **meccanismo di delega**, grazie al quale l'oggetto è in grado di comportarsi **come se** avesse cambiato classe

37

Ingegneria del Software L-A - Design Pattern

## Pattern STATE

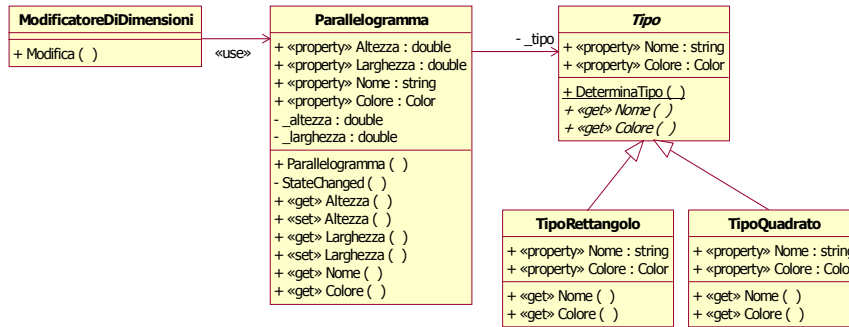


- Localizza il comportamento specifico di uno stato e suddivide il comportamento in funzione dello stato
- Le classi concrete contengono la logica di transizione da uno stato all'altro
- Permette anche di emulare l'ereditarietà multipla

38

Ingegneria del Software L-A - Design Pattern

# Pattern STATE



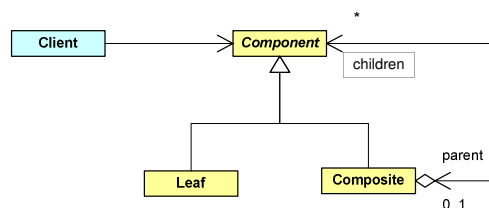
## Esempio S3

39

Ingegneria del Software L-A - Design Pattern

# Pattern COMPOSITE

- Permette di comporre oggetti in una **struttura ad albero**, al fine di rappresentare una **gerarchia di oggetti contenitori-oggetti contenuti**
- Permette ai clienti di **trattare in modo uniforme oggetti singoli e oggetti composti**

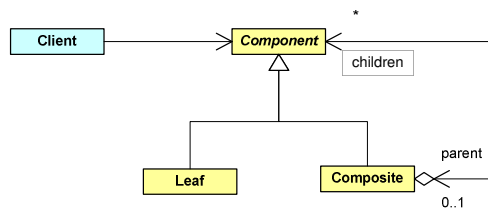


40

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

- **Component** (classe astratta)
  - Dichiarare l'interfaccia
  - Realizzare il comportamento di *default*
- **Client**
  - Accede e manipola gli oggetti della composizione attraverso l'interfaccia di **Component**

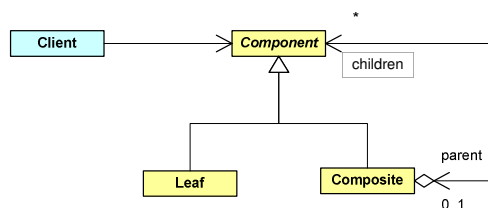


41

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

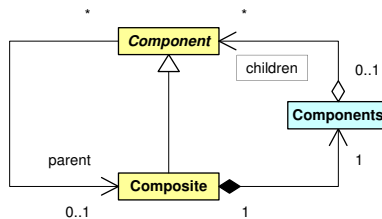
- **Leaf**
  - Descrive oggetti foglia – cioè senza figli
  - Definisce il comportamento di tali oggetti
- **Composite**
  - Descrive oggetti che hanno figli – contenitori
  - Definisce il comportamento di tali oggetti



42

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE



- Il contenitore dei figli deve essere un attributo di **Composite** e può essere di qualsiasi tipo (*array*, lista, albero, tabella *hash*, ...)

43

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

- **Riferimento esplicito al genitore (*parent*)**
  - Semplifica l'attraversamento e la gestione della struttura
  - L'attributo che contiene il riferimento al genitore e la relativa gestione devono essere posti nella classe **Component**
- **Invariante**

Tutti gli elementi che hanno come *parent* lo stesso componente devono essere (gli unici) figli di quel componente

  - incapsulare l'assegnamento di *parent* nei metodi **Add** e **Remove** della classe **Composite**, **oppure**
  - incapsulare le operazioni di **Add** e **Remove** nella set dell'attributo *parent* della classe **Component**

44

Ingegneria del Software L-A - Design Pattern

## *Pattern COMPOSITE*

```
public class Composite : Component
{
    ...
    public void Add(Component aChild)
    {
        if(aChild.Parent != null)
            throw new ArgumentException(...);
        _children.Add(aChild);
        aChild._parent = this;
    }
    ...
}
```

45

Ingegneria del Software L-A - Design Pattern

## *Pattern COMPOSITE*

```
public class Composite : Component
{
    ...
    public void Remove(Component aChild)
    {
        if(aChild.Parent != this)
            throw new ArgumentException(...);
        if(!_children.Contains(aChild))
            throw new ArgumentException(...);
        _children.Remove(aChild);
        aChild._parent = null;
    }
    ...
}
```

46

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

```
public class Component
{
    ...
    public Composite Parent
    {
        get { return _parent; }
        set
        {
            if (value != _parent)
            {
                if (_parent != null)
                    _parent.Remove(this);
                if (value != null)
                    value.Add(this);
            }
        }
    }
    ...
}
```

47

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

- **Massimizzazione dell'interfaccia Component**
  - Un obiettivo del pattern Composite è quello di fare in modo che **il cliente veda solo l'interfaccia di Component** ► **in Component devono essere inserite tutte le operazioni che devono essere utilizzate dai clienti** - nella maggior parte dei casi, **Component** definisce una realizzazione di default che le sotto classi devono ridefinire
  - Alcune di queste operazioni possono essere prive di significato per gli oggetti foglia (**Add, Remove, ...**)

48

Ingegneria del Software L-A - Design Pattern



## *Pattern COMPOSITE*

- **Trasparenza**

Dichiaro tutto al livello più alto, in modo che il cliente possa trattare gli oggetti in modo uniforme ma... **il cliente potrebbe cercare di fare cose senza senso**, come aggiungere figli alle foglie

- Se scegliamo la trasparenza

- **Add e Remove** devono avere una **realizzazione di default che genera un'eccezione**
- dovremmo disporre di un modo per verificare se è possibile aggiungere figli all'oggetto su cui si vuole agire

49

Ingegneria del Software L-A - Design Pattern

## *Pattern COMPOSITE*

```
// Il cliente conosce solo Component

Component parent =
    ComponentFactory.CreateInstance (...);
...
Component child =
    ComponentFactory.CreateInstance (...);
...
// Prima di inserire un figlio,
// occorre controllare se è possibile
if(parent.IsComposite())
    parent.Add(child);
```

50

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

- **Sicurezza**

Tutte le operazioni sui figli vengono messe in **Composite** – a questo punto, qualsiasi invocazione sulle foglie genera un errore in fase di compilazione ma... **il cliente deve conoscere e gestire due interfacce differenti**

- Se scegliamo la sicurezza

- dobbiamo disporre di un modo per verificare se l'oggetto su cui si vuole agire è un **Composite**

51

Ingegneria del Software L-A - Design Pattern

## Pattern COMPOSITE

```
// Il cliente conosce Component e Composite
```

```
Component child = ComponentFactory.CreateComponent (...);  
Composite parent1 = ComponentFactory.CreateComposite (...);  
parent1.Add(child);
```

```
...
```

```
Component parent2 = ComponentFactory.CreateComponent (...);  
// Errore di compilazione  
parent2.Add(child);  
// Prima di inserire un figlio,  
// occorre controllare se è possibile e fare un cast  
if(parent2 is Composite)  
    ((Composite) parent2).Add(child);
```

52

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

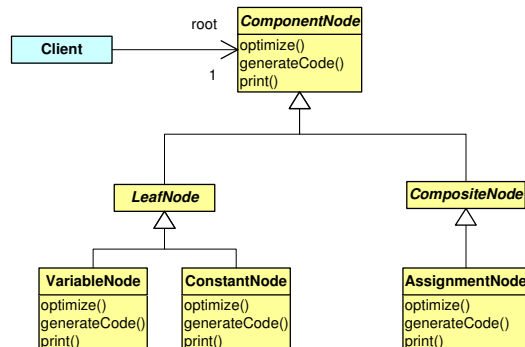
- Permette di **definire una nuova operazione** da effettuare su gli elementi di una struttura, **senza dover modificare le classi degli elementi coinvolti**
- Ad esempio, si consideri la rappresentazione di un programma come "**abstract syntax tree**" (AST) – un albero i cui nodi descrivono elementi sintattici del programma
- Su tale albero **devono poter essere effettuate molte operazioni di tipo diverso**
  - Controllare che tutte le variabili siano definite
  - Eseguire delle ottimizzazioni
  - Generare il codice macchina
  - Stampare l'albero in un formato leggibile
  - ...

53

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

Per l'AST utilizziamo il *pattern Composite*



54

Ingegneria del Software L-A - Design Pattern

## *Pattern VISITOR*

- In seguito potremmo voler effettuare **altri tipi di operazioni**
  - controllare che le variabili siano state inizializzate prima dell'uso
  - ristrutturare automaticamente il programma
  - calcolare varie metriche
  - ...
- Se distribuiamo le operazioni sui vari tipi di nodo, otteniamo un sistema che è difficile da
  - capire
  - mantenere
  - modificare

55

Ingegneria del Software L-A - Design Pattern

## *Pattern VISITOR*

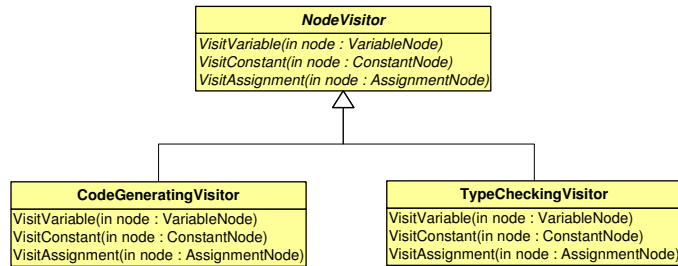
- La soluzione è quella di eliminare le singole operazioni dall'AST (la cui responsabilità principale è quella di rappresentare un programma sotto forma di albero)
- **Tutto il codice relativo ad un singolo tipo di operazione** (ad es. generazione del codice) viene raccolto in **una singola classe**
- I nodi dell'AST devono **accettare la visita** delle istanze di queste nuove classi (**visitor**)
- Per aggiungere un **nuovo tipo di operazione**, è sufficiente progettare una **nuova classe**

56

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- Il *Visitor* deve dichiarare un'operazione per ogni tipo di nodo concreto

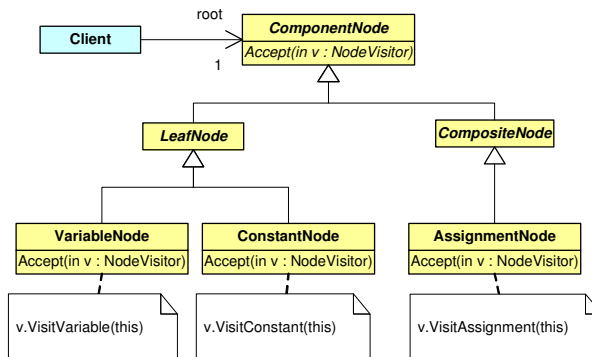


57

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- Ogni nodo deve dichiarare un'operazione per accettare un generico *visitor*

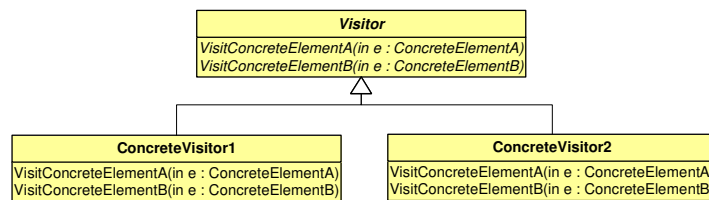


58

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Visitor** (classe astratta o interfaccia)
  - Dichiara un metodo **Visit** per ogni classe di elementi concreti
- **ConcreteVisitor**
  - Definisce tutti i metodi **Visit**
  - Globalmente **definisce l'operazione da effettuare sulla struttura** e (se necessario) ha un proprio stato

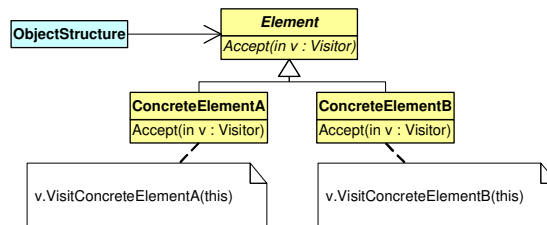


59

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Element** (classe astratta o interfaccia)
  - Dichiara un metodo **Accept** che accetta un Visitor come argomento
- **ConcreteElement**
  - Definisce il metodo **Accept**



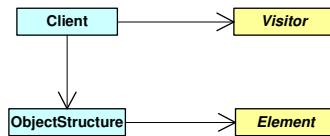
60

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **ObjectStructure**

- Può essere realizzata come *Composite* o come normale collezione (*array*, lista, ...)
- Deve poter enumerare i suoi elementi
- Deve dichiarare un'interfaccia che permetta a un cliente di far visitare la struttura a un *Visitor*



61

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Facilita l'aggiunta di nuove operazioni**

- È possibile aggiungere nuove operazioni su una struttura esistente, semplicemente aggiungendo un nuovo *visitor* concreto
- Senza il *pattern Visitor*, sarebbe necessario aggiungere un metodo ad ogni classe degli elementi della struttura
- Ogni *Visitor* concreto
  - Raggruppa i metodi necessari ad eseguire una data operazione
  - Nasconde i dettagli di come tale operazione debba essere eseguita

62

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Incapsulamento**
  - Ogni **Visitor** deve essere in grado di accedere allo stato degli elementi su cui deve operare
- **È difficile aggiungere una nuova classe ConcreteElement**
  - Per ogni nuova classe **ConcreteElement** è necessario inserire un nuovo metodo **Visit** in tutti i **Visitor** esistenti
    - ▶ **la gerarchia Element deve essere** poco o per nulla modificabile – cioè essere **stabile**

63

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Visita di elementi non correlati**
  - Non è necessario che tutti gli elementi da visitare derivino da una classe comune

```
VisitClasseA(ClasseGerarchiaA a);
VisitClasseB(ClasseGerarchiaB b);
```
- **Stato**
  - Durante l'operazione ogni **Visitor** può modificare il proprio stato – ad esempio, per accumulare dei valori o altro

64

Ingegneria del Software L-A - Design Pattern



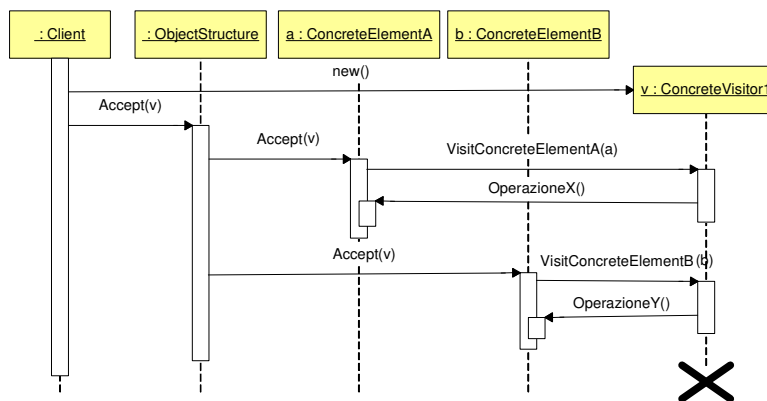
## Pattern VISITOR

```
public class CompositeElement : Element
{
    ...
    private List<Element> _children;
    ...
    public override void Accept(Visitor visitor)
    {
        foreach (Element aChild in _children)
            aChild.Accept(visitor);
        visitor.VisitCompositeElement(this);
    }
    ...
}
```

65

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR



66

Ingegneria del Software L-A - Design Pattern

## Pattern VISITOR

- **Double dispatch**

- L'operazione che deve essere effettuata **dipende dal tipo di due oggetti**
  - il *visitor*
  - l'elemento
- **Accept** è un'operazione di tipo **double dispatch**

**Esempio + EsempioDecorator**

67

Ingegneria del Software L-A - Design Pattern

## Pattern OBSERVER

- **Context**

- A change in one object (the **subject**) will sometimes require other objects (**observers**) to be updated
- This relationship can be explicitly coded in the subject, but this requires knowledge about, how the observers should be updated
- The result is that the objects become intertwined (**closely coupled**) and **can't easily be reused**

- **Solution**

- Create a loosely-bound one-to-many relationship between an object and others that depend on it
- A change in the object will result in the others **receiving a notification**, enabling them to update themselves accordingly

68

Ingegneria del Software L-A - Design Pattern

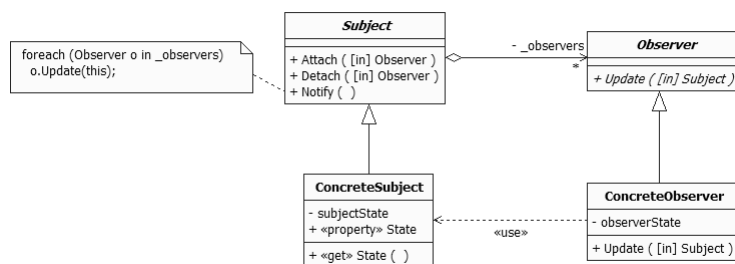
## Pattern OBSERVER

- There are two mechanisms which can be used to implement a loose coupling between a subject and its observers
- **Dependency** allows an observer to register an **interest in ALL aspects of another object**
  - the observer then has to sort out what actually changed at runtime so it can do something sensible in response
- **Events** allow an observer to register an **interest in a specific aspect of another object (publisher)** and even request that a particular message is routed to them
  - when the publisher triggers an event the routing is automatic

69

Ingegneria del Software L-A - Design Pattern

## Pattern OBSERVER



70

Ingegneria del Software L-A - Design Pattern

## Il Pattern *Model / View / Controller (MVC)*

- Utilizzato per realizzare le interfacce utenti in *Smalltalk-80*
- Permette di suddividere un'applicazione, o anche la sola interfaccia dell'applicazione, in tre parti
  - **Modello** elaborazione / stato
  - **View** (o *viewport*) *output*
  - **Controller** *input*

71

Ingegneria del Software L-A - Design Pattern

## Il Pattern *Model / View / Controller (MVC)*

### **Modello**

- Gestisce un insieme di dati logicamente correlati
- Risponde alle interrogazioni sui dati
- Risponde alle istruzioni di modifica dello stato
- Genera un evento quando lo stato cambia
- Registra (in forma anonima) gli oggetti interessati alla notifica dell'evento
- In Java, deve estendere la classe `java.util.Observable`

72

Ingegneria del Software L-A - Design Pattern

## Il *Pattern* *Model / View / Controller (MVC)*

### **View**

- Gestisce un'area di visualizzazione, nella quale presenta all'utente una vista dei dati gestiti dal modello
  - Mappa i dati (o parte dei dati) del modello in oggetti visuali
  - Visualizza tali oggetti su un (particolare) dispositivo di output
- Si registra presso il modello per ricevere l'evento di cambiamento di stato
- In Java, deve implementare l'interfaccia `java.util.Observer`

73

Ingegneria del Software L-A - Design Pattern

## Il *Pattern* *Model / View / Controller (MVC)*

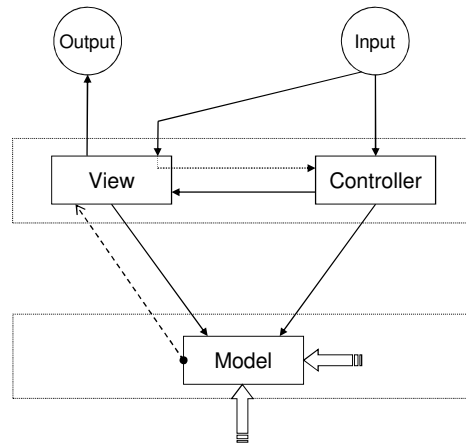
### **Controller**

- Gestisce gli input dell'utente (mouse, tastiera, ...)
- Mappa le azioni dell'utente in comandi
- Invia tali comandi al modello e/o alla *view* che effettuano le operazioni appropriate
- In Java, è un *listener*

74

Ingegneria del Software L-A - Design Pattern

## Il Pattern Model / View / Controller (MVC)



**Esempio**

75

Ingegneria del Software L-A - Design Pattern