CC BY NC SA

DESIGN PATTERN STRUTTURALI

INGEGNERIA DEL SOFTWARE

Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea in Informatica, A.A. 2017 – 2018

rcardin@math.unipd.it

DESIGN PATTERN STRUTTURALI

Relazioni tra	Campo di applicazione		
	Creational (5)	Structural (7)	Behavioral (11)
Class	Factory method	Adapter (Class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento Observer State Strategy Visitor

Architetturali
Model view controller

INTRODUZIONE

- Scopo dei *design pattern* strutturali
  - Affrontare problemi che riguardano la composizione di classi e oggetti
  - Consentire il ri-utilzzo degli oggetti esistenti fornendo agli utilizzatori un'interfaccia più adatta
    - Integrazioni fra librerie / componenti diverse
  - Sfruttano l'ereditarietà e l'aggregazione

ADAPTER

- Scopo
  - Convertire l'interfaccia di una classe in un'altra.

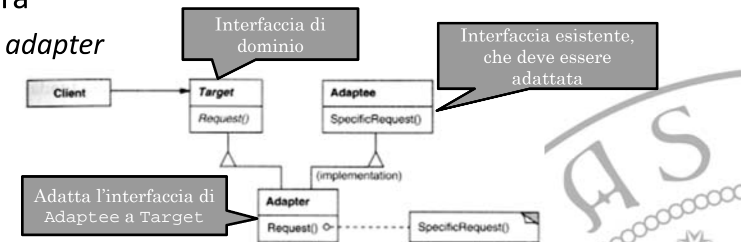
- Motivazione
  - Spesso i *toolkit* non sono riusabili
    - Non è corretto (e possibile) modificare il *toolkit*!
  - Definiamo una classe *(adapter)* che adatti le interfacce.
    - Per ereditarietà o per composizione
    - La classe *adapter* può fornire funzionalità che la classe adattata non possiede

ADAPTER

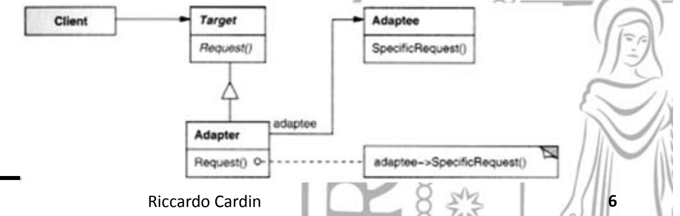
- Applicabilità
  - Riutilizzo di una classe esistente, non è conforme all'interfaccia *target*
  - Creazione di classi riusabili anche con classi non ancora analizzate o viste
  - Non è possibile adattare l'interfaccia attraverso ereditarietà (*Object adapter*)

ADAPTER

- Struttura
  - *Class adapter*



  - *Object adapter*



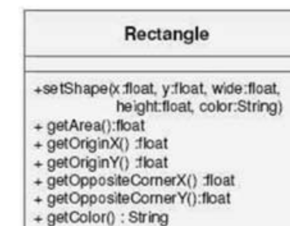
ADAPTER

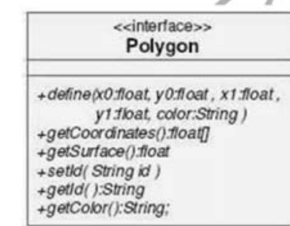
- Conseguenze
  - *Class adapter*
    - Non funziona quando bisogna adattare una classe e le sue sottoclassi
    - Permette all'Adapter di modificare alcune caratteristiche dell'Adaptee
  - *Object adapter*
    - Permette ad un Adapter di adattare più tipi (Adaptee e le sue sottoclassi)
    - Non permette di modificare le caratteristiche dell'Adaptee
    - Un oggetto *adapter* non è sottotipo dell'*adaptee*

ADAPTER

- Esempio

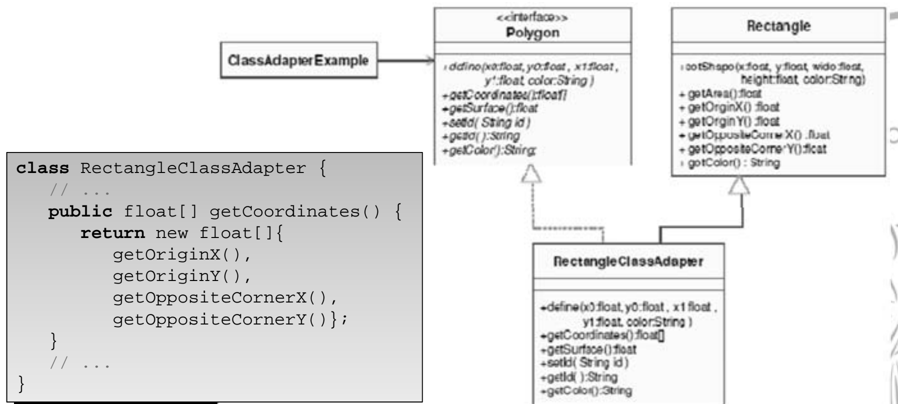
Convertire (adattare) una vecchia classe `Rectangle` ad una nuova interfaccia `Polygon`.





ADAPTER

- Esempio
  - *Class adapter*

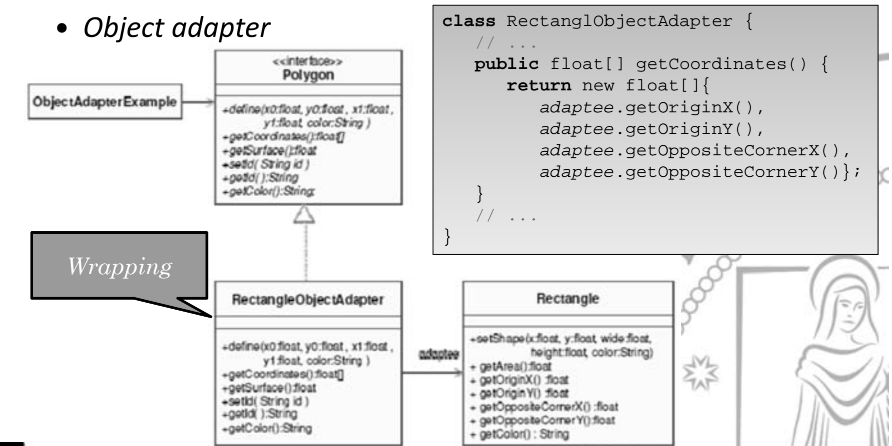



```
class RectangleClassAdapter {
   // ...
   public float[] getCoordinates() {
      return new float[]{
         getOriginX(),
         getOriginY(),
         getOppositeCornerX(),
         getOppositeCornerY()};
   }
   // ...
}
```


ADAPTER

- Esempio
  - *Object adapter*

```
class RectanglObjectAdapter {
   // ...
   public float[] getCoordinates() {
      return new float[]{
         adaptee.getOriginX(),
         adaptee.getOriginY(),
         adaptee.getOppositeCornerX(),
         adaptee.getOppositeCornerY()};
   }
   // ...
}
```



ADAPTER

- Esempio
  - `Scala`: classi implicite

```
trait Log {
  def warning(message: String)
  def error(message: String)
}

final class Logger {
  def log(level: Level, message: String) { /* ... */ }
}

implicit class LoggerToLogAdapter(logger: Logger) extends Log {
  def warning(message: String) { logger.log(WARNING, message) }
  def error(message: String) { logger.log(ERROR, message) }
}

val log: Log = new Logger()
```

Scala utilizza il costruttore per eseguire la conversione implicita

ADAPTER

- Esempio
  - `Javascript`: ...non ci sono classi, ma oggetti...

```
// Adaptee
AjaxLogger.sendLog(arguments);
AjaxLogger.sendInfo(arguments);
AjaxLogger.sendDebug(arguments);

var AjaxLoggerAdapter = {
    log: function() {
        AjaxLogger.sendLog(arguments);
    },
    info: function() {
        AjaxLogger.sendInfo(arguments);
    },
    debug: function() {
        AjaxLogger.sendDebug(arguments);
    },
    ...
};
window.console = AjaxLoggerAdapter;
```

Uso funzioni e *namespace* per simulare le classi e gli oggetti

ADAPTER

- Implementazione
  - Individuare l'insieme minimo di funzioni *(narrow)* da adattare
    - Più semplice da implementare e manutenere
    - Utilizzo di operazioni astratte
  - Diverse varianti strutturali alternative
    - (Client – Target) + Adapter
    - Client + Target + Adapter

DECORATOR

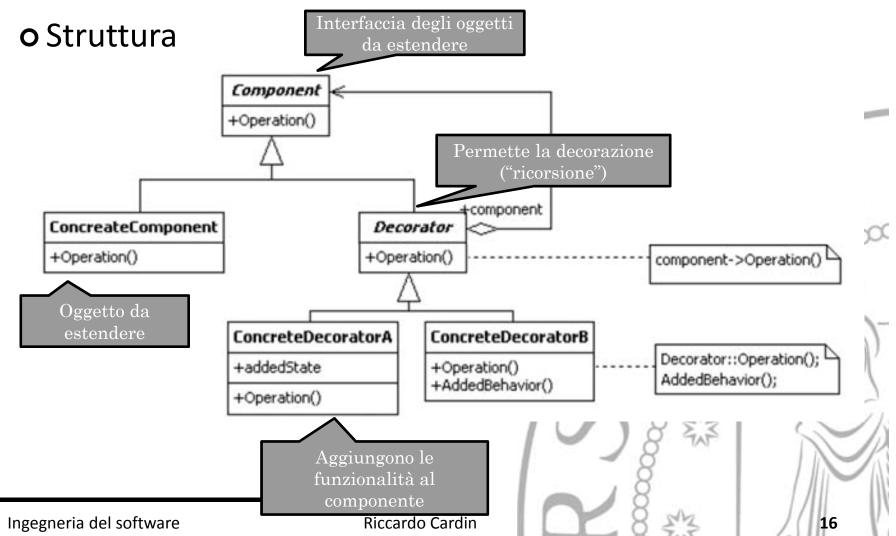
- Scopo
  - Aggiungere responsabilità a un <u>oggetto</u> dinamicamente
- Motivazione
  - Il *Decorator* ingloba un componente in un altro oggetto che aggiunge la funzionalità
    - Il *subclassing* non può essere sempre utilizzato
    - Funzionalità aggiunte prima o dopo l'originale

DECORATOR

- Applicabilità
  - Aggiungere funzionalità dinamicamente ad un oggetto in modo trasparente
  - Funzionalità che possono essere "circoscritte"
  - Estensione via *subclassing* non è possibile
    - Esplosione del numero di sottoclassi
    - Non disponibilità della classe al *subclassing*

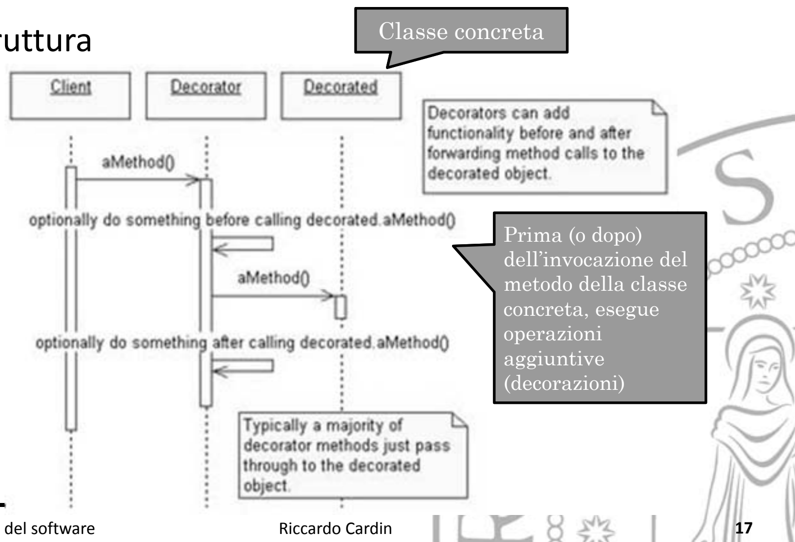
DECORATOR

- Struttura



DECORATOR

- Struttura



DECORATOR

- Conseguenze
  - Maggiore flessibilità della derivazione statica
  - Evita classi "agglomerati di funzionalità" in posizioni alte delle gerarchia
    - La classi componenti diventano più semplici
    - *Software as a Service* (SaaS)
  - Il decoratore e le componenti non sono uguali
    - Non usare nel caso in cui la funzionalità si basi sull'identità
  - Proliferazione di piccole classi simili
    - Facili da personalizzare, ma difficili da comprendere e testare.

DECORATOR

- Esempio

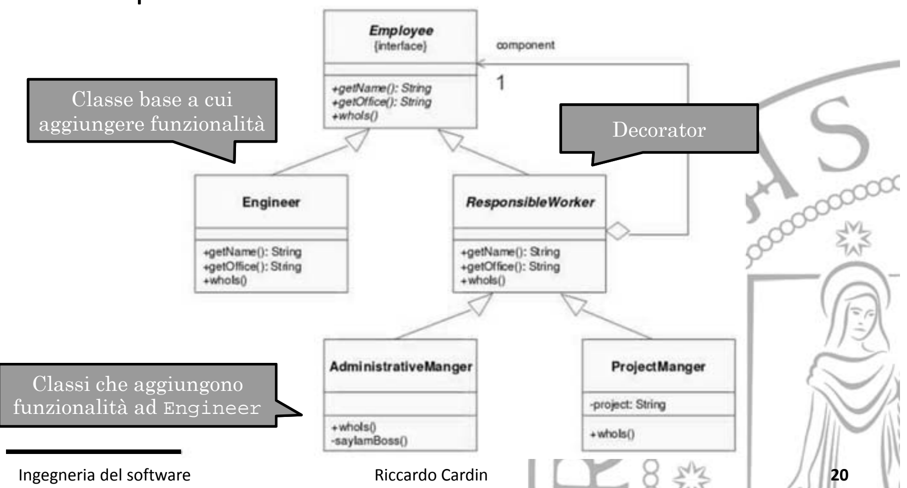
Si possiede un modello di gestione di oggetti che rappresentano gli impiegati (`Employee`) di una azienda. Il sistema vuole prevedere la possibilità di "promuovere" gli impiegati con delle responsabilità aggiuntive (e adeguato stipendio :P).

Ad esempio, da impiegato a capoufficio (`AdministrativeManager`) oppure da impiegato a capo progetto (`ProjectManager`).

Nota: Queste responsabilità non si escludono tra di loro.

DECORATOR

- Esempio



DECORATOR

- Esempio
  - ORM is an Offensive Anti-Pattern
    http://www.yegor256.com/2014/12/01/orm-offensive-anti-pattern.html
    - Buon esempio di *decorator* utilizzato come *cache* (può essere visto anche come esempio di *proxy pattern*)

DECORATOR

- Esempio
  - `Javascipt`: gli oggetti sono dei "dizionari" di valori

1

```
function Engineer( name, office){
   //...
   this.whois = function() { console.log("I'm an engineer") ;  }
}
var projectManager = new Engineer("Riccardo", "Development");
projectManager.whois = function() { console.log("I'm the boss!"); }
```

2

```
// What we're going to decorate
function Engineer() {
   //...
}
/* Decorator 1 */
function AdministrativeManager(engineer) {
   var v = engineer.whois();
   engineer.whois = function() {
      return v + console.log(" and I'm the super boss too!");
   }
}
```

Multiple decorator

DECORATOR

- Esempio
  - `Scala`: *mixin*

```
trait Employee {
  // ...
  def whois(): String
}

class Engineer(name: String, office: String) extends Employee
{ /* ... */ }

trait ProjectManager extends Employee {
  abstract override def whois() {
    // super rappresenta il mixin a sinistra
    super.whois(buffer)
    println("and I'am a project manager too!")
  }
}
new Engineer("Riccardo","Development") with ProjectManager
```

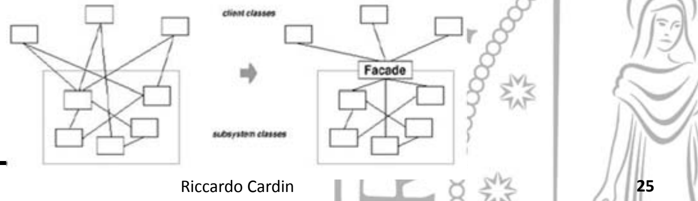
Decorator con static binding

DECORATOR

- Implementazione
  - Interfaccia del decoratore DEVE essere conforme a quella del componente
  - Omissione della classe astratta del decoratore
    - … grandi gerarchie di classi già presenti …
  - Mantenere "leggera" (*stateless*) l'implementazione del *Component*
  - Modifica della "pelle" o della "pancia"?
    - Decorator: quando le componenti sono "leggere"
    - Strategy: quando le componenti hanno un'implementazione corposa
      - Evita decoratori troppo "costosi" da manutenere.

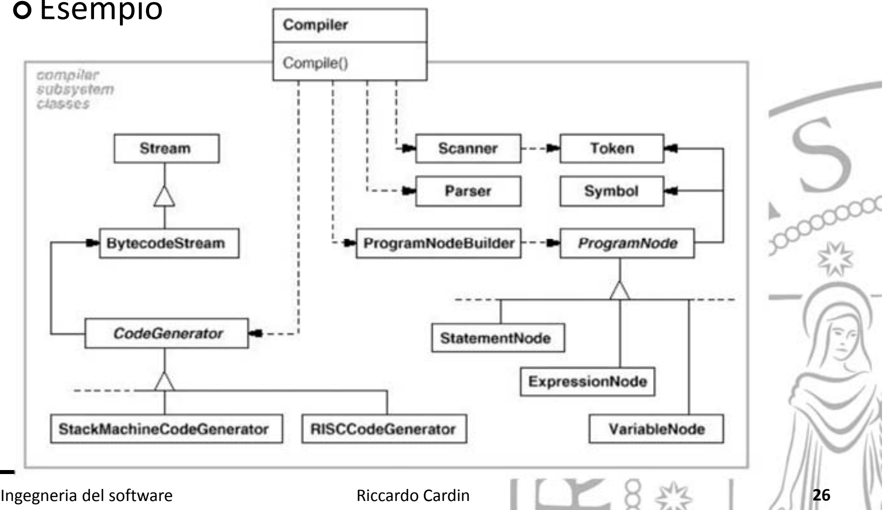
Facade

- Scopo
  - Fornire un'interfaccia unica semplice per un sottosistema complesso
- Motivazione
  - Strutturazione di un sistema in sottosistemi
    - Diminuisce la complessità del sistema, ma aumenta le dipendenze tra sottosistemi
    - L'utilizzo di un Facade semplifica queste dipendenze
      - Ma non nasconde le funzionalità *low-level*



Facade

- Esempio

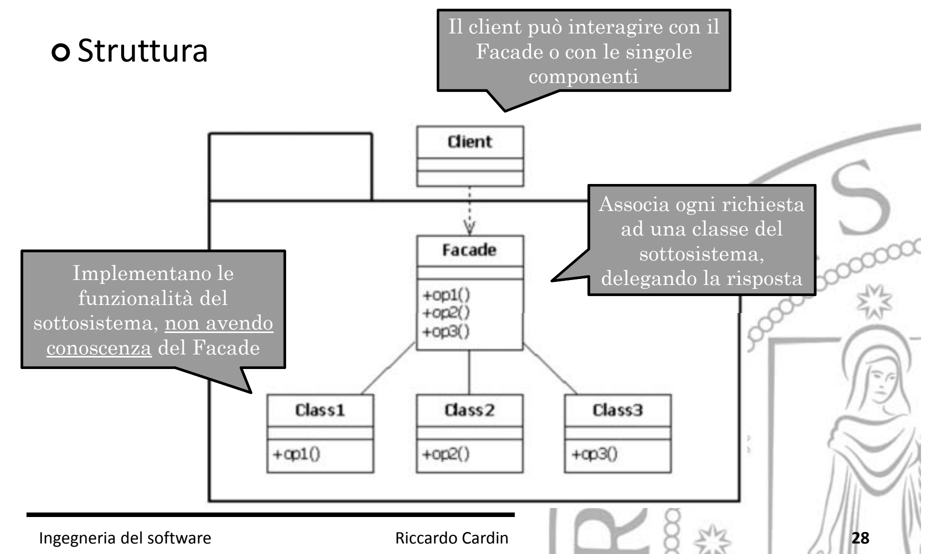


Facade

- Applicabilità
  - Necessità di una singola interfaccia semplice
    - Design pattern tendono a generare tante piccole classi
    - Vista di *default* di un sottosistema
  - Disaccoppiamento tra sottosistemi e *client*
    - Nasconde i livelli fra l'astrazione e l'implementazione
  - Stratificazione di un sistema
    - Architettura *Three tier*

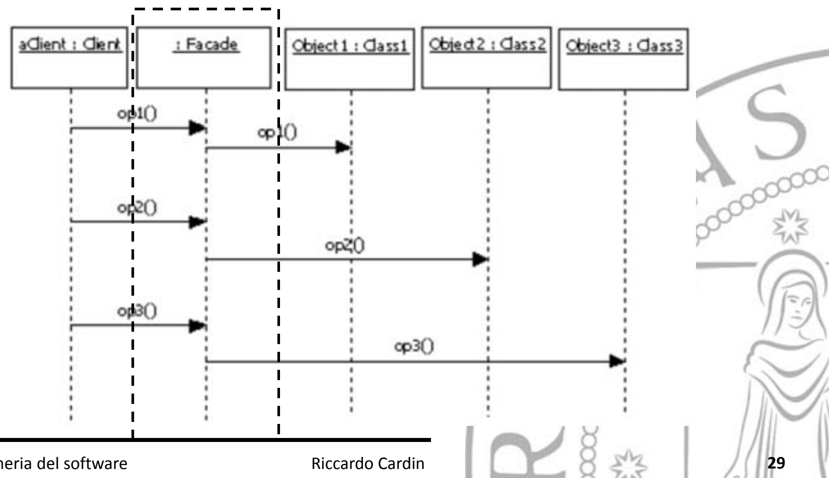
Facade

- Struttura



FACADE

- Struttura



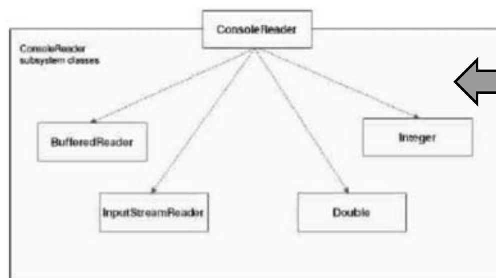
FACADE

- Conseguenze
  - Riduce il numero di classi del sottosistema con cui il *client* deve interagire
  - Realizza un accoppiamento lasco tra i sottosistemi e i *client*
    - Eliminazione delle dipendenze circolari
    - Aiuta a ridurre i tempi di compilazione e di *building*
  - Non nasconde completamente le componenti di un sottosistema
  - *Single point of failure*
  - Sovradimensionamento della classe Facade

FACADE

- Esempio

All'interno di un'applicazione si vuole consentire la lettura da tastiera di tipi di dati diversi (es. interi, float, stringhe, ecc).



Il pattern *Facade* permette l'utilizzo di una classe `ConsoleReader` che espone i metodi di lettura e incapsula le regole degli effettivi strumenti di lettura.

FACADE

- Esempio
  - `Javascript`: utilizzato spesso con *module pattern*

```
var module = (function() {
    var _private = {
        i:5,
        get : function() {console.log( "current value:" + this.i);},
        set : function( val ) {this.i = val;},
        run : function() {console.log( "running" );},
    };

    return {
        facade : function( args ) {
            _private.set(args.val);
            _private.get();
            if ( args.run ) {
                _private.run();
            }
        }
    };
}());
```

Metodi privati

Metodo pubblico

FACADE

- Esempio
  - `Scala`: *mixin*

```
trait ToolA {
  //stateless methods in ToolA
}
trait ToolB {
  //stateless methods in ToolB
}
trait ToolC {
  //stateless methods in ToolC
}

object facade extends ToolA with ToolB with ToolC
```

Versione modificata del *facade*, dove si hanno a disposizione tutti i metodi delle classi del sottosistema

FACADE

- Implementazione
  - Classe Facade come classe astratta
    - Una classe concreta per ogni "vista" (implementazione) del sottosistema
  - Gestione di classi da più sottosistemi
  - Definizione d'interfacce "pubbliche" e "private"
    - Facade nasconde l'interfaccia "privata"
      - *Module pattern* in Javascript
  - *Singleton pattern*: una sola istanza del Facade

PROXY

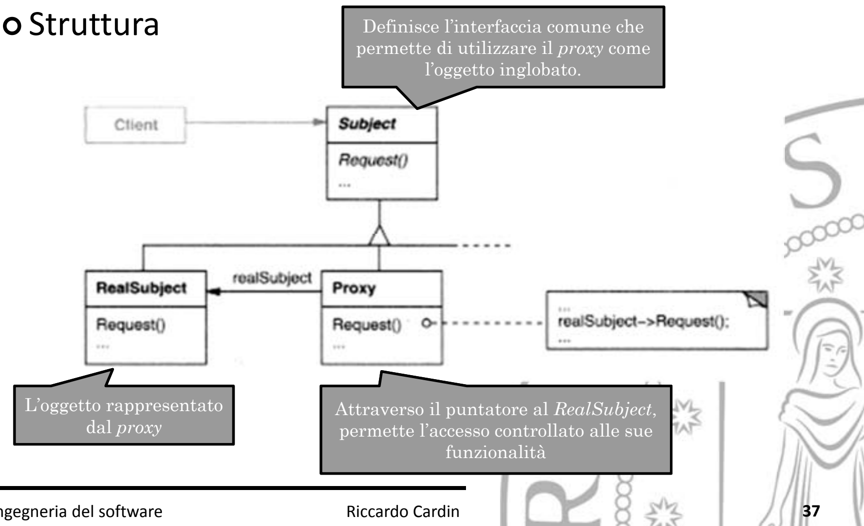
- Scopo
  - Fornire un surrogato di un altro oggetto di cui si vuole controllare l'accesso
- Motivazione
  - Rinviare il costo di creazione di un oggetto all'effettivo utilizzo (*on demand*)
  - Il *proxy* agisce come l'oggetto che ingloba
    - Stessa interfaccia
  - Le funzionalità dell'oggetto "inglobato" vengono accedute attraverso il *proxy*
    - ...o senza l'accesso vero e proprio (*virtual proxy*)

PROXY

- Applicabilità
  - Remote proxy
    - Rappresentazione locale di un oggetto che si trova in uno spazio di indirizzi differente
      - Classi *stub* in Java RMI
  - Virtual proxy
    - Creazione di oggetti complessi *on-demand*
  - Protection proxy
    - Controllo degli accessi (diritti) all'oggetto originale
  - Puntatore "intelligente"
    - Gestione della memoria in Objective-C

PROXY

- Struttura

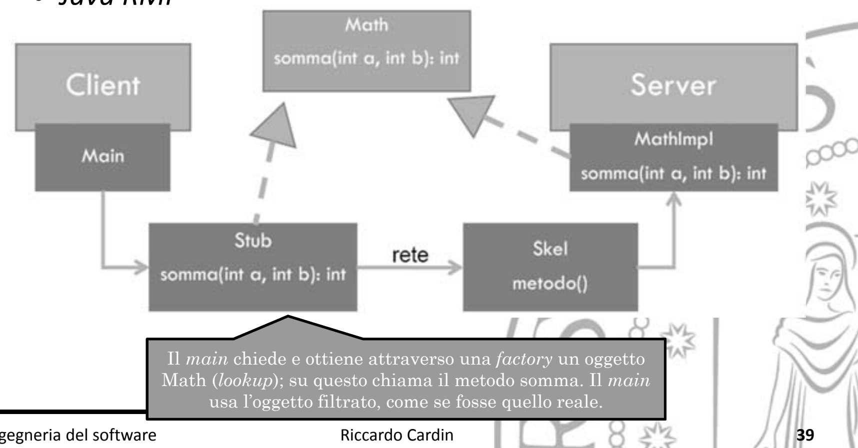


PROXY

- Conseguenze
  - Introduce un livello di indirezione che può essere "farcito"
    - *Remote proxy*, nasconde dove un oggetto risiede
    - *Virtual proxy*, effettua delle ottimizzazioni
    - *Protection proxy*, definisce ruoli di accesso alle informazioni
  - *Copy-on-write*
    - La copia di un oggetto viene eseguita unicamente quando la copia viene modificata.

PROXY

- Esempio
  - *Java RMI*



PROXY

- Implementazione
  - Implementazione "a puntatore"
    - Overload operatore –> e * in C++
  - Alcuni *proxy* , invece, agiscono in modo differente rispetto alle operazioni
    - In Java costruzione tramite *reflection* (Spring, H8...)
  - *Proxy* per più tipi ...
    - ... *subject* è una classe astratta ...
    - ... ma non se il proxy deve istanziare il tipo concreto!
  - Rappresentazione del *subject* nel *proxy*

RIFERIMENTI

- Design Patterns, Elements of Reusable Object Oriented Software, GoF, 1995, Addison-Wesley
- Design Patterns http://sourcemaking.com/design_patterns
- Java DP
  http://www.javacamp.org/designPattern/
- Exploring the Decorator Pattern in Javascript http://addyosmani.com/blog/decorator-pattern/
- Design Patterns in Scala http://pavelfatin.com/design-patterns-in-scala
- Implicit Classeshttp://docs.scala-lang.org/overviews/core/implicit-classes.html
- Create Facade by combining scala objects/traits http://stackoverflow.com/questions/14905473/create-facade-by-combining-scala-objects-traits
- Ruby Best Practices http://blog.rubybestpractices.com/posts/gregory/060-issue-26-structural-design-patterns.html

GITHUB REPOSITORY

https://github.com/rcardin/swe