Object Oriented Programming

Prof. Ing. Loris Penserini, PhD

https://orcid.org/0009-0008-6157-0396

Informatica: Modellare il Mondo Reale

L'Informatica (o Computer Science) è la scienza che studia la gestione dell'informazione tramite procedure automatizzate. Questo è realizzabile attraverso: una consolidata teoria dell'informazione, linguaggi formali di rappresentazione, e sistemi elettronici per l'esecuzione automatica.

Attraverso specifici linguaggi di programmazione, si possono scrivere algoritmi (sequenze di istruzioni) che un calcolatore elettronico è in grado di elaborare in modo automatico al fine di fornire delle informazioni utili all'utente partendo da dati iniziali.

L'evoluzione delle tecnologie digitali nei settori della Robotica e dell'AI, spingono i ricercatori a sviluppare nuovi paradigmi di programmazione…

Programmazione

La maggior parte dei linguaggi di programmazione si basano sul «**paradigma imperativo**», in cui il programma consiste in un insieme di istruzioni dette anche «**direttive**» o «**comandi**».

La programmazione imperativa viene generalmente contrapposta a quella dichiarativa, in cui un programma consiste in un insieme di «affermazioni» (non «comandi») che l'esecutore è tenuto a considerare vere e/o rendere vere. Un esempio di paradigma dichiarativo è la programmazione logica (es. Prolog, LTL, CTL, ecc.).

Come vedremo in seguito, i concetti introdotti dalla programmazione strutturata sono alla base di numerosi altri linguaggi di programmazione, non ultimo quello orientato agli oggetti.

Modellare il Mondo Reale in OOP

Ciascun paradigma di programmazione fornisce allo sviluppatore un differente approccio concettuale per implementare il pensiero computazionale, cioè la definizione della strategia algoritmica utile per affrontare e risolvere un problema del mondo reale.

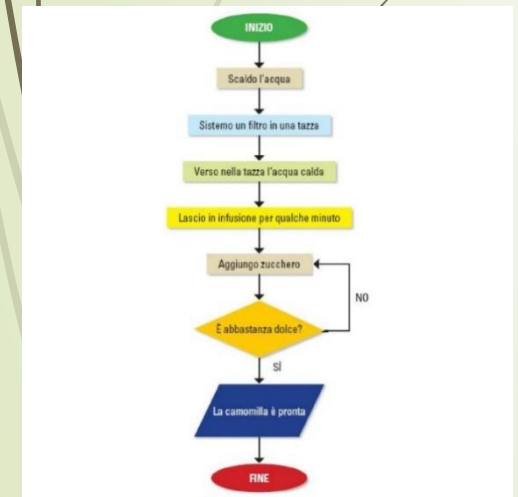
La OOP è attualmente il paradigma di sviluppo del SW più utilizzato, per questo molti linguaggi del Web che in passato nascevano non ad oggetti ora lo sono diventati, come il C → C++, il PHP dopo la ver. 5, mentre altri sono nati direttamente OO come JAVA (JSP e Servlet).

In ogni caso, per creare pagine Web dinamiche, il protocollo standard rimane sempre il **Common Gateway Interface (CGI)**, cioè, indipendentemente dal linguaggio di programmazione usato, si lascia aperta la possibilità di eseguire codice remoto (su un server) invocandolo da un client Web (browser).

Comunicare con il Calcolatore

Indipendentemente dall'hardware, dal S.O. e dai linguaggi di programmazione, queste sono le fasi necessarie per comunicare con un elaboratore.

ALGORITMO



PROGRAMMATORE

LINGUAGGIO ASSEMBLER

LINGUAGGIO MACCHINA

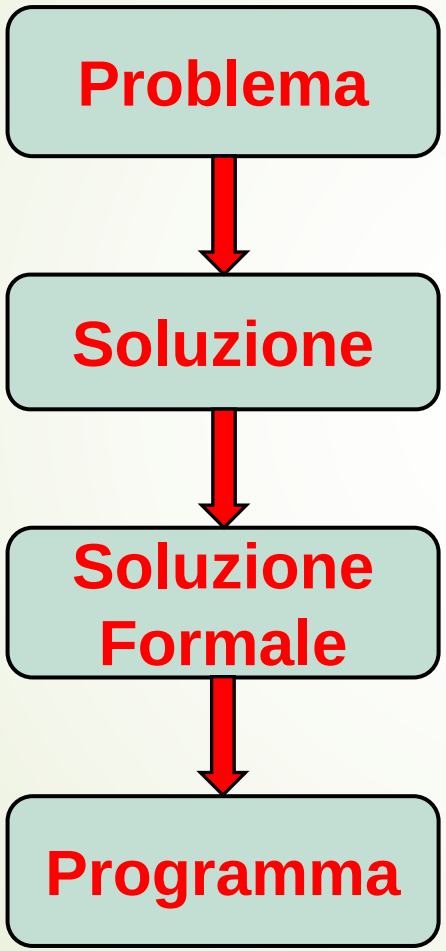
Scrivere un Algoritmo

A **design-time** un problema per essere risolto da un «elaboratore elettronico» o «esecutore» deve essere rappresentato con una formalizzazione dell'**ALGORITMO**, cioè nella logica del linguaggio formale con la quale si descrive il modello informatico di un problema del mondo reale.

L'algoritmo è definito da un insieme finito di passi (operazioni) che portano a risolvere un problema informatico, dove si hanno dei dati in ingresso (input) e si producono dei risultati (output).

La soluzione ad un problema del mondo reale può avere differenti algoritmi risolutori, tuttavia la programmazione cerca di trovare quello più efficiente.

Dall'Algoritmo al Programma



Esperienza da cui si ricava l'idea

Metodi formali e strumenti automatici per scrivere l'Algoritmo

Paradigmi di programmazione e relativi linguaggi

Algoritmo e Programma

Trovata la soluzione migliore al problema, questa viene formalizzata nell'**algoritmo** che è composto da una sequenza finita di istruzioni che sono comprensibili dall' «**esecutore**» ovvero l'elaboratore elettronico.

Infine, l'algoritmo deve essere tradotto in un linguaggio formale comprensibile all' «esecutore»: il **programma**.

Proprietà di un Algoritmo

E' composto da una sequenza ***finita*** di operazioni/istruzioni che vengono svolte dall' «esecutore» in un ***processo sequenziale***.

Ogni operazione deve produrre, se eseguita, un effetto ***osservabile*** che possa essere descritto.

Ogni operazione deve produrre lo stesso effetto ogni volta che viene eseguita a partire dagli stessi input e dalle stesse condizioni iniziali (***determinismo***).

Le fasi del ciclo di vita del software

Nell'ingegneria del software si spiegano le fasi di sviluppo di un'applicazione software come fosse un prodotto con un suo «ciclo di vita».

Il modello più noto di sviluppo del software è il «**modello a cascata**» (*waterfall model*) in cui ci sono una serie di fasi a cascata e ciascuna fase riceve ingressi dalla fase precedente e produce uscite, che sono a loro volta ingressi per la fase seguente.

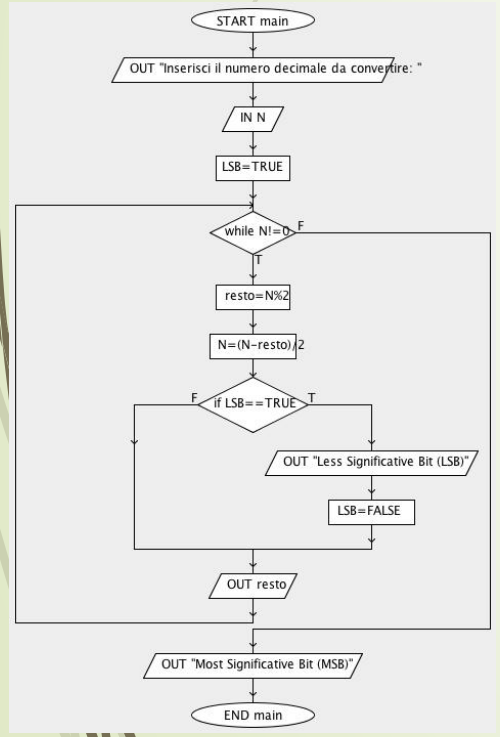
- **Studio di fattibilità**
- **Analisi e specifica dei requisiti**
- **Progettazione**
- **Programmazione/Implementazione e test**
- **Installazione, Integrazione e test di sistema**
- **Manutenzione**

Altri modelli si sono evoluti nel tempo: «**modello trasformazionale**» e «**modello a spirale**».

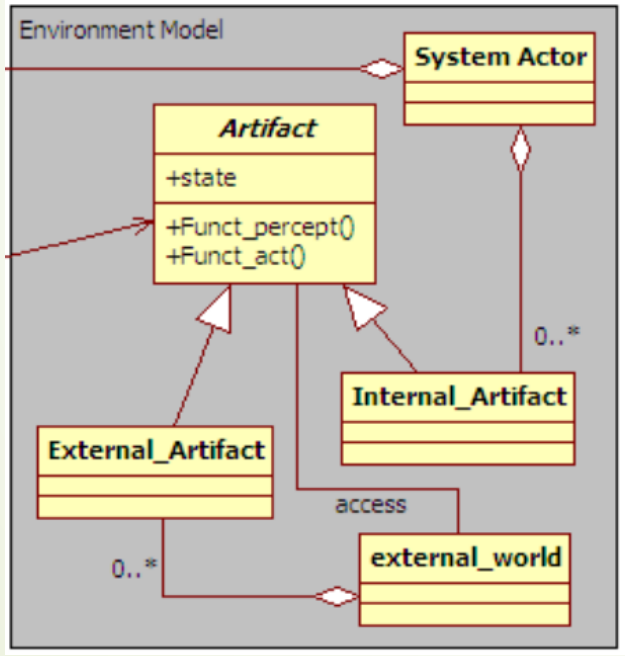
Paradigmi di programmazione…

Alcuni principali paradigmi di programmazione e livelli di astrazione del pensiero computazionale.

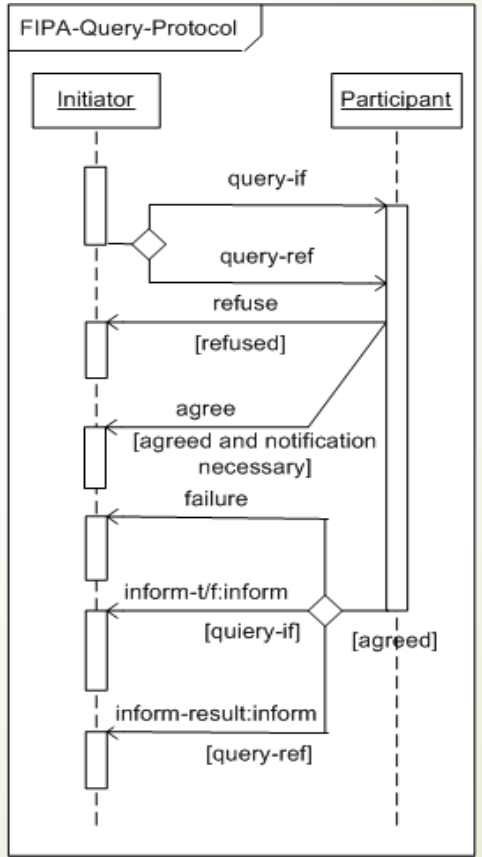
Flow-Chart (Structured Prog.)



UML (Object Oriented Prog.)



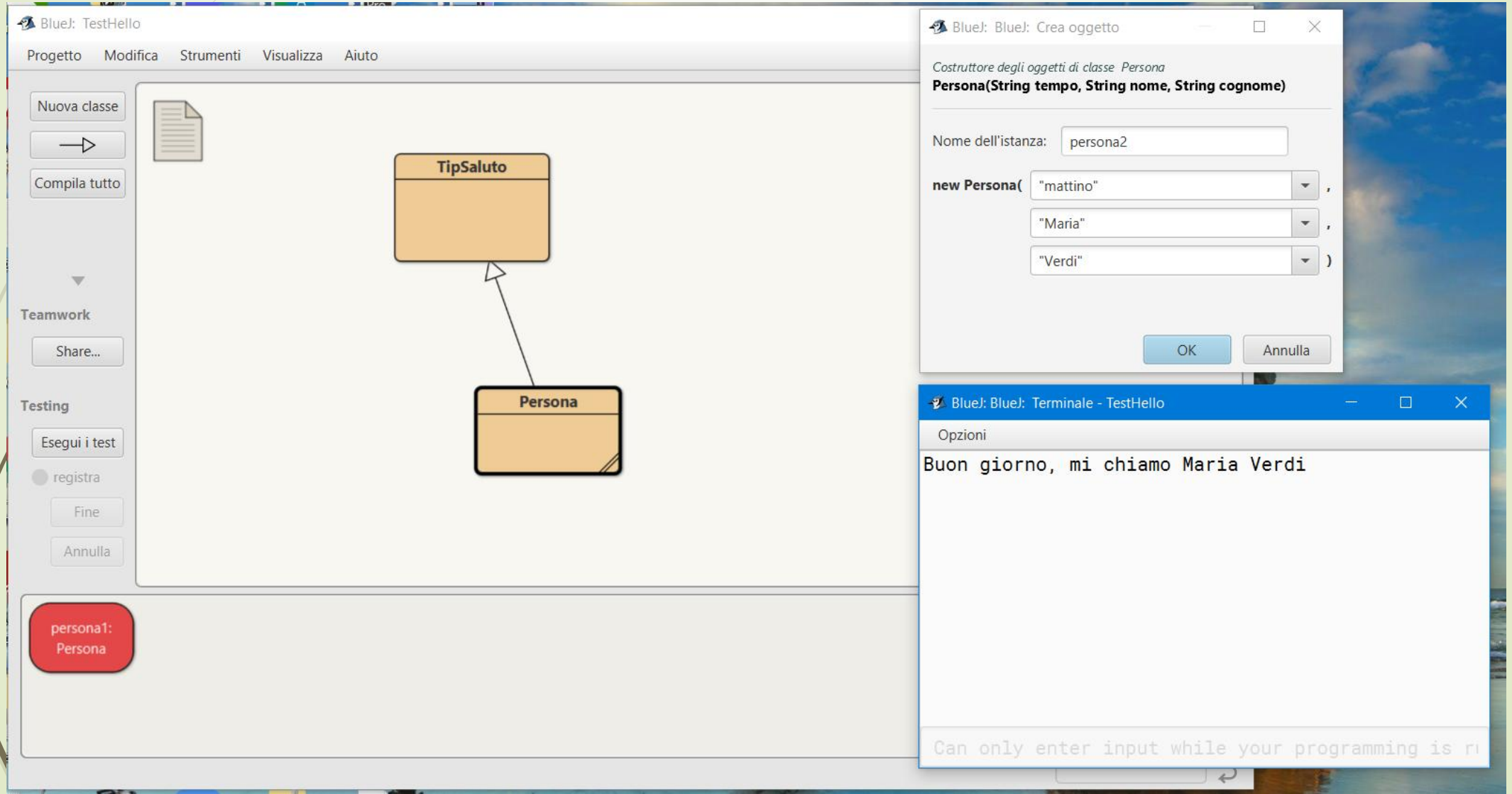
Agent-UML (Agent Oriented Prog.)



Scratch/Blockly (Block based Prog.)

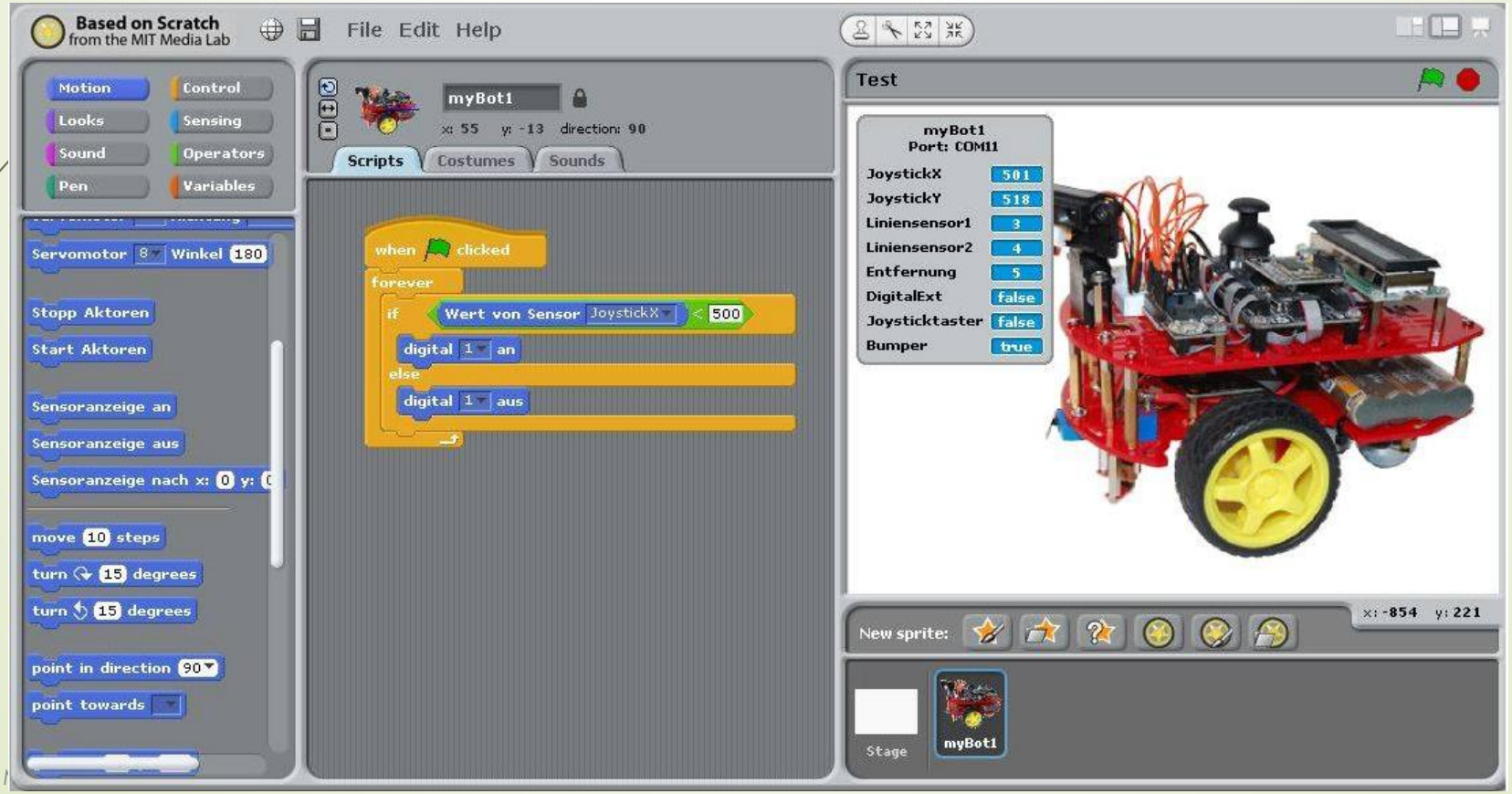
Ambienti visuali per OOP…

BlueJ è uno dei primi ambienti visuali per OOP…



Nella Robotica

Semplificare l'integrazione di sensori e microcontrollori di un robot attraverso l'utilizzo di «moduli» preconfezionati che si possono incastrare come puzzle per realizzare algoritmi efficienti.



Paradigmi per Sistemi Complessi

Il software di oggi è più complesso che in passato, perché gli si richiede di funzionare in modo autonomo in ambienti dinamici, aperti e imprevedibili. Le metodologie di ingegneria del software che si ispirano ai sistemi autonomi o *self-adaptive* (*SAS*), offrono una soluzione promettente per far fronte al problema di sviluppo di sistemi complessi.

Uno dei paradigmi di programmazione più recente è quello orientato agli agenti: **AOP** (*Agent Oriented Programming*) e il relativo **AOSE** (*Agent Oriented Software Engineering*):

- **AOP: Jade**, **Jadex**, **Jake**, **3APL**, **Alan**, **JEAP**, **…**

[Morandini et al., 2008] [Penserini et al., 2007b] [Pagliarecci et al., 2007] [Panti et al., 2003]

- **AOSE: MaSE**, **GAIA**, **Tropos4AS**,**...**

[Penserini et al., 2007a] [Morandini et al., 2009]

Le fasi del «ciclo di vita» nella metodologia Tropos4AS sono un ibrido tra mod. trasformazionale (adotta approccio MDA) e mod. a spirale (si reitera sulle fasi precedenti) [Morandini et al., 2017]

Dal Sorgente al Programma Eseguibile

IL CALCOLATORE NON E' IN GRADO DI COMPRENDERE DIRETTAMENTE UN LINGUAGGIO AD ALTO LIVELLO, cioè un linguaggio utile a semplificare il lavoro del programmatore.

Per cui, il testo del programma scritto in un linguaggio di programmazione di alto livello, detto programma **sorgente** (source), deve essere tradotto in **linguaggio macchina** per poter essere eseguito dall'elaboratore.

Esempio in Assembler

Questo è il linguaggio con il quale si comunica con lo specifico processore del PC in uso. Per esempio, la famiglia dei Motorola (es. MC68000, …) e la famiglia degli INTEL (es. 8086, …, 80386, …, Pentium, ecc.)

Esempio:

```
if (D0 == 5)
      D1++;
D2 = D0;
```

Linguaggio di Alto Livello (C, C++, Java, …)

```
        CMPI.L  #5,D0
        BNE     SKIP
        ADDQ.L  #1,D1
SKIP    MOVE.L  D0,D2
```

Assembly del Processore Motorola 68000

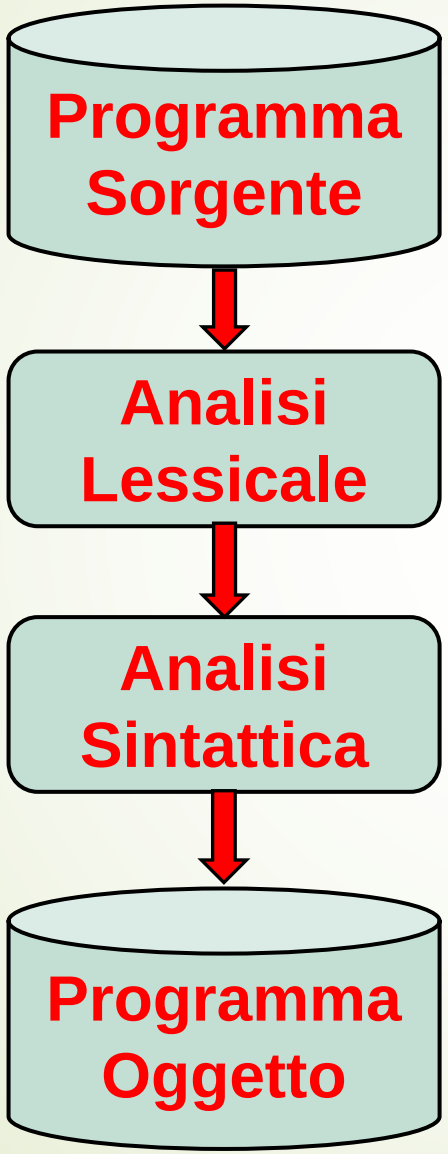
Tradurre il Programma

Il processo di traduzione può essere di due tipologie:

Interpretazione: un'applicazione che prende il nome di interprete considera il testo sorgente, istruzione per istruzione, e lo traduce mentre lo esegue; su questo principio lavorano linguaggi quali il Basic, PHP, Python e altri linguaggi per la gestione di basi di dati e per le applicazioni Web.

Compilazione: un'applicazione (compilatore) che trasforma l'intero programma sorgente in linguaggio macchina, memorizzando in un file il risultato del proprio lavoro. Così un programma compilato una sola volta, può essere eseguito senza bisogno di altri interventi, quante volte si vuole. Il risultato della compilazione si chiama programma oggetto (object). Linguaggi come: C/C++ e Java.

Interpretazione e Compilazione



Testo del programma

Controllo della correttezza dei termini e dei simboli

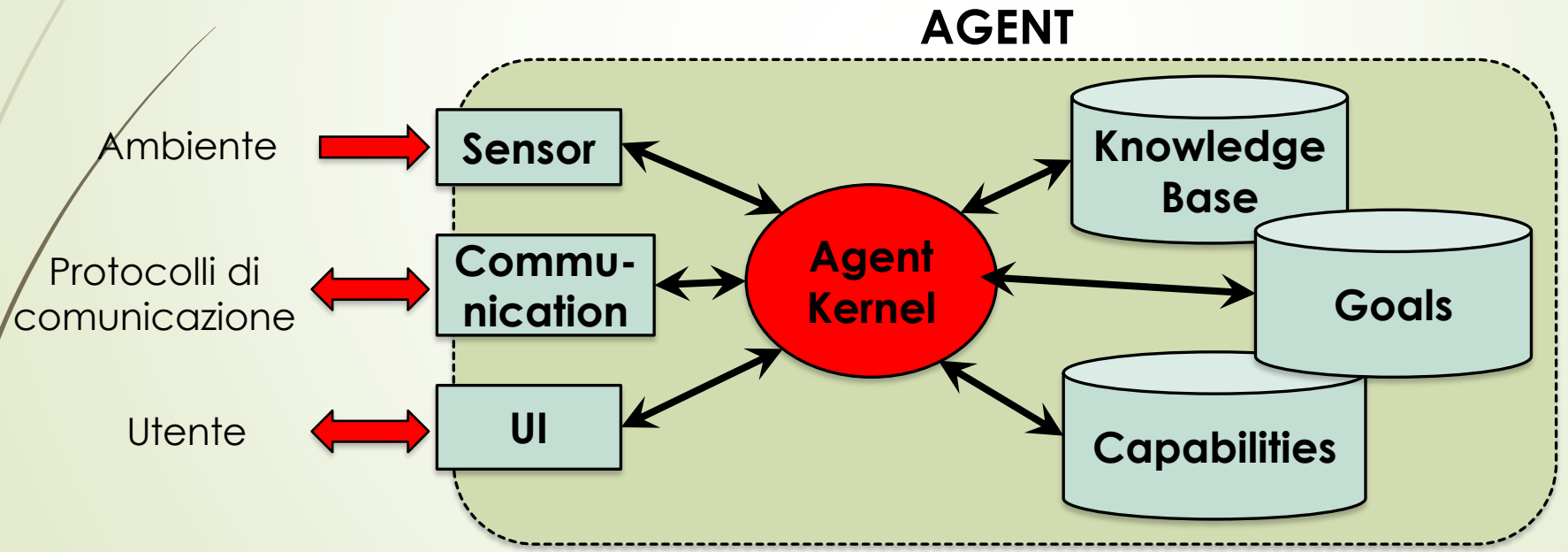
Controllo della correttezza dei costrutti sintattici

Programma tradotto in linguaggio macchina

In IA: programmare sistemi autonomi

Esempio di uno «smart agent» con architettura BDI (*Beliefs – Desires – Intentions*).

Nel 1996 nasce in Svizzera la Foundation for Intelligent Physical Agents (FIPA) che nel 2005 entrò a far parte degli standard della IEEE Computer Society.



Perché OOP per lo sviluppo di SW

Ecco alcuni vantaggi di pensare ad un algoritmo in termini di Oggetti:

- Astrarre dalla complessità del codice per concentrarsi maggiormente sulle similitudini tra gli oggetti software e gli oggetti del mondo reale che si vogliono modellare.
- Pensare al comportamento (behavior) di un oggetto software come al suo analogo reale: causalità e relativi effetti.
- Notevole aumento della possibilità di riuso del codice, con conseguente aumento della produttività di qualità:
  - Maggiore stabilità delle applicazioni;
  - Facilità nell'aggiornare mantenere il codice

La «classe»

Nella OOP, la **classe** è il modulo autocontenuto che definisce il progetto (parziale o intero) dell'algoritmo che si vuole realizzare. E' caratterizzata da:

- Proprietà o variabili (in PHP la tipizzazione del dato non è obbligatoria)
- Funzioni o metodi, che contengono la logica dell'algoritmo
- Il nome della classe deve coincidere con quella del file
- I blocchi delimitati dalle parentesi graffe determinano lo scope o campo d'azione delle proprietà
- Lo stato di una classe dipende dai valori di inizializzazione delle sue proprietà nel momento in cui viene caricata in memoria

La classe assomiglia al progetto della casa, ma non è la casa!

Esempio concettuale di «classe» in OOP

//classe
Persona
- nome
- cognome
- età
- Interessi
- pagina di saluto

Buongiorno sono «nome» + «cognome»

specializzare

generalizzare

//sottoclassi
Studente
- nome
- cognome
- età
- interessi
- indirizzo_studi
- pagina di saluto_stu

Buongiorno sono «nome» + «cognome», e frequento «indirizzo_studi»

Insegnante
- nome
- cognome
- età
- interessi
- materia
- pagina di saluto_ins

Buongiorno sono «nome» + «cognome» e insegno «materia»

Cosa è un «oggetto»?

Nella OOP il concetto di «oggetto» è :

- Un processo in esecuzione in memoria
- Una applicazione che fa uso di funzioni/metodi appartenenti a classi diverse (librerie diverse)
- Una applicazione alla quale si può dare uno stato iniziale che ne determina il comportamento (behavior) iniziale.
- Una applicazione che interagisce con il mondo esterno determinando il comportamento che più si adatta per quello scenario

Differenza tra «classe» e «oggetto»

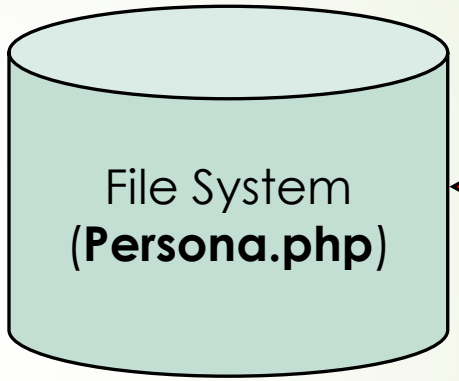
DESIGN time

//classe: è un file
Persona
- nome
- cognome
- età
- Interessi
- pagina di benvenuto

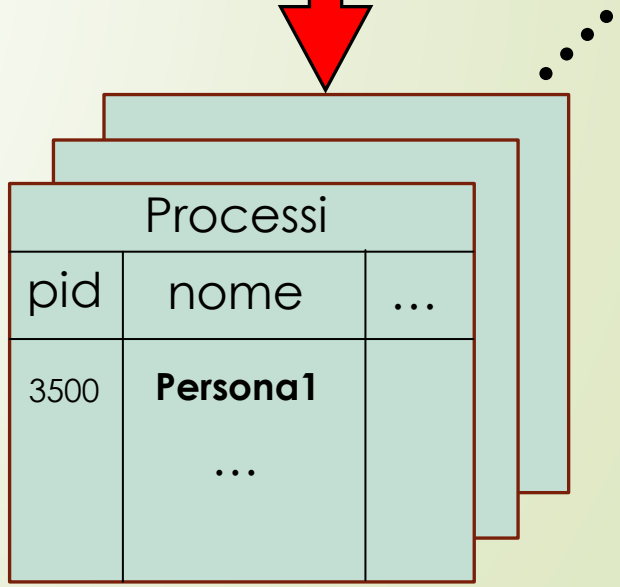
//oggetto: è un processo
Persona1
- nome → Mario
- Cognome → Rossi
- Età → 50
- Interessi → robotica
- pagina di benvenuto → Ciao Mario Rossi

RUN time

Memoria di massa



RAM



Proprietà fondamentali in OOP

Tutti i linguaggi OOP forniscono sempre queste tre proprietà:

- **Ereditarietà**
- **Incapsulamento**
- **Polimorfismo**

EREDITARIETA'

Ereditarietà

In PHP (come in Java) **non** è prevista l'ereditarietà multipla come invece è possibile nel C++, per cui in PHP una classe può al più ereditare da una sola altra classe.

In PHP, dalla versione 5.4, sono state aggiunte delle funzioni per ovviare a questa limitazione (i trait) che vedremo più avanti**.**

Tuttavia, per bravi programmatori OOP, l'ereditarietà singola non è considerata una limitazione ma un vantaggio per progettare SW efficiente e modulare.

La classe Persona

```php
class Persona {
    //Proprietà
    public $nome = "";
    public $cognome = "";
    public $eta = "";
    public $interessi = "";
    public $saluto = "";

    //costruttore
    public function __construct($nome,$cognome,$eta,$interessi) {
        //inizializzazione
        $this->nome = $nome;
        $this->cognome = $cognome;
        $this->eta = $eta;
        $this->interessi = $interessi;
        $this->saluto = "Buongiorno sono ".$nome." ".$cognome;
    }

    //metodo che restituisce la pagina di saluto
    public function getPagBenvenuto() {
        $this->saluto = "Buongiorno sono ".$this->nome." ". $this->cognome;
        return $this->saluto;
    }
}
```

La classe Studente

L'operatore «extends»

```php
class Studente extends Persona {
    public $ind_studio = "";

    //metodo per inserire info specifiche per lo studente
    public function setIndStudio($ind_studio) {
        $this->ind_studio = $ind_studio;
    }

    public function getPagBenvenutoStud() {
        $saluto_persona = $this->getPagBenvenuto();
        return $saluto_persona.", e frequento ".$this->ind_studio;
    }
}
```

Le Istanze di classi sono Oggetti

```php
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <?php
        include 'Persona.php';
        //require_once 'Persona.php';
        include 'Studente.php';

        $nome = "Loris";
        $cognome = "Penserini";
        $eta = "50";
        $interessi = "DRONI";

        //CREO L'OGGETTO "Personal"
        $Personal = new Persona($nome, $cognome, $eta, $interessi);
        $Studente1 = new Studente($nome, $cognome, $eta, $interessi);
        $Studente1->setIndStudio("Sistemi Informativi Aziendali");

        echo "SALUTO DI PERSONA_1: <br>".$Personal->getPagBenvenuto();
        echo "<br><br>SALUTO DI STUDENTE_1: <br>".$Studente1->getPagBenvenutoStud();

        ?>
    </body>
</html>
```

Costruttori di classe

Ogni classe dovrebbe avere il metodo costruttore, poiché definisce lo stato iniziale dell'oggetto associato alla classe. Cioè il metodo costruttore crea un punto di partenza nell'esecuzione del codice dell'oggetto.

Allora nella classe Studente da quale blocco di codice si inizia?

Project work 1

Riutilizzate il codice del progetto appena presentato. E con modifiche minimali, aggiungere la classe «Dirigente» (utilizzando come guida la classe Studente), poi dalla pagina index.php lanciare un'istanza e accodare a video il relativo saluto:

output

```
SALUTO DI PERSONA_1:
Buongiorno sono Loris Penserini

SALUTO DI STUDENTE_1:
Buongiorno sono Mario Rossi, e frequento Sistemi Informativi Aziendali

SALUTO DI DIRIGENTE_1:
Buongiorno sono Giovanna Rossini, e dirigo la scuola IIS POLO3 FANO
```

Project work (Dirigente.php)

La classe Dirigente (come Studente) eredita dalla classe Persona, ovviamente al contrario di Studente i metodi di Dirigente si specializzano per questo ruolo.

```php
class Dirigente extends Persona {
    public $scuola = "";

    //metodo per inserire info specifiche per lo studente
    public function setScuola($scuola) {
        $this->scuola = $scuola;
    }

    public function getPagBenvenutoDir() {
        $saluto_persona = $this->getPagBenvenuto();
        return $saluto_persona.", e dirigo la scuola ".$this->scuola;
    }
}
```

Project work (index.php)

```php
include 'Persona.php';
//require_once 'Persona.php';
include 'Studente.php';
include 'Dirigente.php';

//Simula una lettura dati che può avvenire tramite FORM HTML,
//tramite DB, tramite lettura file, ecc.
$nome1 = "Loris";
$cognome1 = "Penserini";
$eta1 = "50";
$interessi1 = "DRONI";

$nome2 = "Mario";
$cognome2 = "Rossi";
$eta2 = "40";
$interessi2 = "Pesca";

$nome3 = "Giovanna";
$cognome3 = "Rossini";
$eta3 = "40";
$interessi3 = "Opera";

//CREO GLI OGGETTI "Personal", "Studente1" e "Dirigente1"
$Personal = new Persona($nome1, $cognome1, $eta1, $interessi1);
$Studente1 = new Studente($nome2, $cognome2, $eta2, $interessi2);
$Studente1->setIndStudio("Sistemi Informativi Aziendali");
$Dirigente1 = new Dirigente($nome3, $cognome3, $eta3, $interessi3);
$Dirigente1->setScuola("IIS POLO3 FANO");

echo "SALUTO DI PERSONA_1: <br>".$Personal->getPagBenvenuto();
echo "<br><br>SALUTO DI STUDENTE_1: <br>".$Studente1->getPagBenvenutoStud();
echo "<br><br>SALUTO DI DIRIGENTE_1: <br>".$Dirigente1->getPagBenvenutoDir();
```

Sono state cerchiate le modifiche necessarie al file «index.php»

Project work 2

Riutilizzate il codice del progetto appena presentato. Inserire nella classe «Studente» i metodi necessari per leggere le proprietà: nome, cognome, età e indirizzo di studio.

Project work 3

Riutilizzate il codice della classe «Studente» per fare «overriding» del metodo costruttore, cioè inserire il metodo costruttore e verificare come viene eseguito con priorità rispetto al costruttore della classe padre («Pesona»).

Interfacce

Una interfaccia rappresenta una «definizione di progetto» per una classe, cioè ne dichiara i metodi fondamentali che la classe corrispondente dovrà possedere obbligatoriamente.

Per cui a **design-time** si forniscono allo sviluppatore delle linee guida, utili a sviluppare classi specifiche senza dover avere la visione completa del progetto d'insieme.

Regole principali:

- I metodi dichiarati nelle interfacce devono sempre essere **public**, e perciò anche quelli implementati all'interno della classe.
- Una classe può implementare più interfacce contemporaneamente, utilizzando come separatore la virgola.
- Una interfaccia può sostenere **l'ereditarietà multipla** (extends).
- Le interfacce non possono contenere proprietà, ma **solo metodi**.
- Nell'interfaccia **non c'è costruttore**.

Interfacce: esempio

Prendiamo in considerazione l'esempio precedente e creiamo una interfaccia per il saluto…

```
interface Saluto {
    public function getPagBenvenuto();
}
```

«Implementare» una Interfaccia

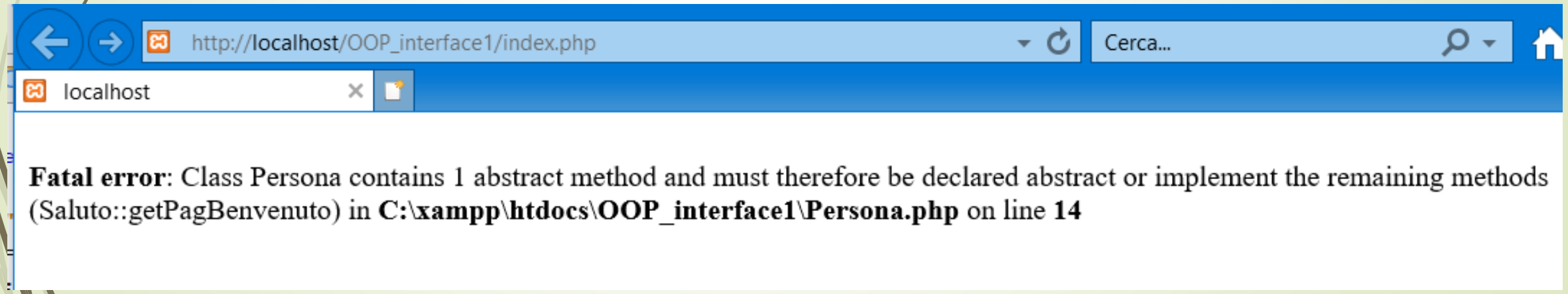
```
class Persona implements Saluto{
    //Proprietà
    public $nome = "";
    public $cognome = "";
    public $eta = "";
    public $interessi = "";
    public $saluto = "";

    //costruttore
    public function __construct($nome,$cognome,$eta,$interessi) {
        //inizializzazione
        $this->nome = $nome;
        $this->cognome = $cognome;
        $this->eta = $eta;
        $this->interessi = $interessi;
        $this->saluto = "Buongiorno sono ".$nome." ".$cognome;
    }

    //metodo che restituisce la pagina di saluto
    public function getPagBenvenuto() {
        $this->saluto = "Buongiorno sono ".$this->nome." ". $this->cognome;
        return $this->saluto;
    }
}
```

Project work 3

Nella classe «Persona» provate a commentare il metodo «getPagBenvenuto()»… cosa accade?

Soluzione

Aver commentato il metodo di cui sopra, viola una delle proprietà delle interfacce: la classe che implementa un'interfaccia ne DEVE definire i metodi dichiarati. Per cui si genera l'errore seguente:



Classi Astratte

La classe astratta, è una classe in cui uno o più metodi non sono implementati. Per cui, da una classe astratta **non** si può creare un oggetto.

Similmente alle interfacce, servono a definire delle **linee guida per creare le sottoclassi** che ne ereditano le proprietà. Cioè una sottoclasse che eredita da una classe astratta, ne deve implementare i metodi astratti, se vuole istanziare oggetti.

Nel caso una sottoclasse, che eredita da una classe astratta, non implementa tutti i metodi di quest'ultima, deve essere anch'essa dichiarata astratta.

- Possono contenere istruzioni, proprietà e metodi come le normali classi
- Possono utilizzare i modificatori di visibilità come le normali classi
- Possono avere il costruttore
- Vale l'ereditarietà singola come per le classi

Classi Astratte: esempio

Utilizziamo come guida per la creazione della classe «Studente» una classe astratta…

```php
abstract class StudentePrototipo{
    public $ind_studio = "";

    /*
     * Metodi per inserire/leggere info specifiche per lo studente.
     * Questi metodi sono definiti per intero.
     */
    public function setIndStudio($ind_studio) {
        $this->ind_studio = $ind_studio;
    }

    public function getIndStudio(){
        return $this->ind_studio;
    }

    /*
     * Metoto non implementato, per cui dichiarato come "abstract"
     */
    public abstract function getPagBenvenutoStud():string;
}
```

Classi Astratte: esempio con «extends»

Implementando la classe «Studente» sfruttiamo le linee guida della classe astratta «StudentePrototipo»…

```php
class Studente extends StudentePrototipo {

    public $nome = "";
    public $cognome = "";
    public $eta = "";
    public $interessi = "";
    public $saluto = "";

    //costruttore
    public function __construct($nome,$cognome,$eta,$interessi) {
        //inizializzazione
        $this->nome = $nome;
        $this->cognome = $cognome;
        $this->eta = $eta;
        $this->interessi = $interessi;
        $this->saluto = "Buongiorno sono ".$nome." ".$cognome;
    }

    public function getPagBenvenutoStud():string{
        return $this->saluto.", uno studente, e frequento ".parent::getIndStudio();
    }
}
```

Classi Astratte: utilizzo

```php
<?php
    include 'StudentePrototipo.php';
    include 'Studente.php';

    $nome = "Loris";
    $cognome = "Penserini";
    $eta = "50";
    $interessi = "DRONI";

    /*
     * CREO L'OGGETTO "Studente1"
     */
    $Studente1 = new Studente($nome, $cognome, $eta, $interessi);
    $Studente1->setIndStudio("Sistemi Informativi Aziendali");

    echo "<br><br>SALUTO DI STUDENTE_1: <br>".$Studente1->getPagBenvenutoStud();
?>
```

Project work 4

Nella classe astratta «StudentePrototipo» provate a cambiare il modificatore di visibilità del metodo «getIndStudio()» dal valore attuale «public» al valore «protected»… infine testare anche «private»

cosa accade?

Project work 4: soluzione

Nella classe astratta «StudentePrototipo» provate a cambiare il modificatore di visibilità del metodo «getIndStudio()» dal valore attuale «public» al valore «protected»… infine testare anche «private»

cosa accade?

Soluzione

Con «protected» tutto dovrebbe funzionare come prima, perché?

Poiché il metodo «getIndStudio()» della classe «StudentePrototipo» è utilizzato da una sua classe figlio.

Con «private» si genera un errore…

INCAPSULAMENTO

INCAPSULAMENTO

E' una proprietà della OOP che facilita a seguire la filosofia dell'ingegneria del software dell'usabilità e della modularità delle applicazioni.

In pratica, nella OOP con l'utilizzo dell'incapsulamento, si 'aggregano' all'interno di un oggetto i dati e le azioni che lo interessano, che diventano quindi elementi **visibili e gestiti** solo dall'oggetto stesso (o da una sua classe), mentre si rendono **pubblici** solo metodi (o attributi) che servono all'oggetto per poter essere riutilizzato in altri contesti applicativi.

Quindi, si parla spesso di limitare l'accesso diretto agli elementi di un oggetto, ovvero di occultamento delle informazioni (*information hiding*).

Modificatori di accesso/visibilità

Per cui in tutti i linguaggi OOP troverete questi operatori:

```
public $nome; // visibile ovunque nello script
protected $email; // visibile nella superclasse e nella sottoclasse
private $password; // Visibile solo nella classe
```

I modificatori sono applicabili pure ai metodi/funzioni.

Esempio di «incapsulamento»

Riprendiamo l'esempio precedente ma operiamo a livello di «design-time» un cambio di strategia di utilizzo della stessa applicazione: gli attributi della classe padre non devono essere accessibili dall'esterno, e solo la classe figlio può invocarne il metodo del saluto…

```php
<?php
include 'Persona.php';
//require_once 'Persona.php';
include 'Studente.php';

$nome = "Loris";
$cognome = "Penserini";
$eta = "50";
$interessi = "DRONI";

//CREO L'OGGETTO "Personal"
//$Personal = new Persona($nome, $cognome, $eta, $interessi);
$Studente1 = new Studente($nome, $cognome, $eta, $interessi);
$Studente1->setIndStudio("Sistemi Informativi Aziendali");

//Commentato poiché utilizzando "protected" su getPagBenvenuto() si
//considera tale metodo di Persona accessibile solo dalle sue sottoclassi.
//echo "SALUTO DI PERSONA_1: <br>".$Personal->getPagBenvenuto();
echo "<br><br>SALUTO DI STUDENTE_1: <br>".$Studente1->getPagBenvenutoStud();
```

Esempio di «incapsulamento»

Gli attributi sono visibili solo da dentro la classe: «**private**»

Il metodo getPagBenvenuto() diventa «**protected**»

```php
class Persona {
    //Proprietà di INCAPSULAMENTO: gli attributi della classe sono
    //visibili/accessibili solo da dentro la classe stessa.
    private $nome = "";
    private $cognome = "";
    private $eta = "";
    private $interessi = "";
    private $saluto = "";

    //costruttore
    public function __construct($nome,$cognome,$eta,$interessi) {
        //inizializzazione
        $this->nome = $nome;
        $this->cognome = $cognome;
        $this->eta = $eta;
        $this->interessi = $interessi;
        $this->saluto = "Buongiorno sono ".$nome." ".$cognome;
    }

    /* metodo che restituisce la pagina di saluto
     * Notare la forma di INCAPSULAMENTO adottata: il metodo sarà accessibile solo
     * dalle classi figlie (e ovviamente dalla classe padre)
    */
    protected function getPagBenvenuto() {
        $this->saluto = "Buongiorno sono ".$this->nome." ". $this->cognome;
        return $this->saluto;
    }
}
?>
```

Esempio di «incapsulamento»

Gli attributi sono visibili solo da dentro la classe: «**private**»

Metodi accessibili dall'esterno: «**public**»

```php
class Studente extends Persona {
    private $ind_studio = "";

    /*
     * Questa classe NON ha il costruttore, per cui usa quello della superclasse
     * Se aggiungete questo costruttore che segue, si farà overriding...
     *
    public function __construct($nome,$cognome,$eta,$interessi) {
        //inizializzazione
        $this->nome = "xxx";
        $this->cognome = "yyy";
        $this->eta = "aaaa";
        $this->interessi = "zzz";
    }*/

    //metodo per inserire info specifiche per lo studente
    public function setIndStudio($ind_studio) {
        $this->ind_studio = $ind_studio;
    }

    public function getPagBenvenutoStud() {
        $saluto_persona = $this->getPagBenvenuto();
        return $saluto_persona.", e frequento ".$this->ind_studio;
    }
}
?>
```

Project work 5

Riutilizzate il codice del progetto appena presentato. Applicate le modifiche alla pagina «index.php» affinché si istanzi la classe «Persona» e si richiami il metodo del saluto…

Deve comparirvi il seguente errore…

Fatal error: Uncaught Error: Call to protected method Persona::getPagBenvenuto() from global scope in C:\xampp\htdocs\OOP_Incaps\index.php:29 Stack trace: #0 {main} thrown in **C:\xampp\htdocs\OOP_Incaps\index.php** on line **29**

Project work 5: spiegazione

Riutilizzate il codice del progetto appena presentato. Applicate le modifiche alla pagina «index.php» affinché si istanzi la classe «Persona» e si richiami il metodo del saluto…

Deve comparirvi il seguente errore…

Fatal error: Uncaught Error: Call to protected method Persona::getPagBenvenuto() from global scope in C:\xampp\htdocs\OOP_Incaps\index.php:29 Stack trace: #0 {main} thrown in **C:\xampp\htdocs\OOP_Incaps\index.php** on line **29**

Soluzione

Aver utilizzato il modificatore di visibilità «protected», sul metodo «getPagBenvenuto» della classe «Persona», implica che dall'esterno (da «index.php») non possiamo utilizzarlo.

Project work 6

Riutilizzate il codice del progetto appena presentato. Applicate i metodi del project work 2, per la classe «Persona» (getNome, getCognome, ecc.), poi dalla classe «Studente» implementate il metodo «getStato()» che legge tutte le informazioni e le restituisce al chiamante.

Includere classi in una pagina…

I due file: Persona.php e Studente.php vengono completamente inclusi nella pagina, cioè il loro codice resta disponibile per tutta l'esecuzione.

```php
<?php
include 'Persona.php';
//require_once 'Persona.php';
include 'Studente.php';

$nome = "Loris";
$cognome = "Penserini";
$eta = "50";
$interessi = "DRONI";

//CREO L'OGGETTO "Personal"
//$Personal = new Persona($nome, $cognome, $eta, $interessi);
$Studentel = new Studente($nome, $cognome, $eta, $interessi);
$Studentel->setIndStudio("Sistemi Informativi Aziendali");

//Commentato poiché utilizzando "protected" su getPagBenvenuto() si
//considera tale metodo di Persona accessibile solo dalle sue sottoclassi.
//echo "SALUTO DI PERSONA_1: <br>".$Personal->getPagBenvenuto();
echo "<br><br>SALUTO DI STUDENTE_1: <br>".$Studentel->getPagBenvenutoStud();
```

Tecniche per includere…

Esistono diverse istruzioni in PHP per includere codice di altre classi nelle pagine php. Ecco le caratteristiche fondamentali di queste funzioni:

«**include** … » se il file non viene trovato tenta di continuare ad eseguire il codice

«**require** … » se il file non viene trovato, termina l'esecuzione

«include_once …» come sopra con la differenza che prima controlla che il file non sia già stato incluso

«require_once …» come sopra con la differenza che prima controlla che il file non sia già stato incluso

GRAZIE!