Ingegneria del Software L-A

4. Progettazione orientata agli oggetti

Dall'OOA all'OOD

- L'OOA identifica e definisce
  - le responsabilità del sistema
  - le classi e gli oggetti

  entro i limiti del dominio del problema
- L'OOA definisce il modello statico e il modello dinamico del sistema, indipendentemente
  - dalla specifica implementazione
  - dalle modalità di interazione con l'utente
  - dalle modalità di persistenza degli oggetti

Dall'OOA all'OOD

- Per realizzare un sistema funzionante, occorre considerare anche
  - GUI
  - DB
  - *Framework*, librerie, componenti, …
  - Modifiche al modello per avere software estensibile e modulare
  - …

Dall'OOA all'OOD

- L'OOD identifica e definisce altre classi e oggetti alla luce dei seguenti principi
  - Incapsulamento
  - Flessibilità
  - Evoluzione
  - Riuso
  - Prestazioni
  - Massima indipendenza possibile da
    - Linguaggio (e ambiente) di programmazione
    - Sistema Operativo
    - Hardware
- Si noti che mediamente le classi di analisi sono solo tra l'1% e il 10% delle classi di progettazione!

Dall'OOA all'OOD

- Il paradigma ad oggetti permette
  - Sia di modellare il mondo reale
  - Sia di implementare il software

- Nella fase di progettazione
  NON occorre creare un modello differente
  da quello sviluppato nella fase di analisi
- Si utilizza un'unica notazione
  - Sia nella fase di analisi
  - Sia nella fase di progettazione

Dall'OOA all'OOD

- Durante l'OOD, il modello prodotto dall'OOA deve essere esteso al fine di modellare/progettare i quattro componenti principali di ogni singola applicazione che compone il sistema
- APPLICATION LOGIC – logica dell'applicazione e controllo degli altri componenti
- PRESENTATION LOGIC – gestione dell'interazione con l'utente a livello logico – nuovi oggetti: finestre, menù, bottoni, *toolbar*, …
- DATA LOGIC – gestione della persistenza a livello logico – nuovi oggetti: file, record, tabelle relazionali, transazioni, istruzioni SQL, …
- MIDDLEWARE – gestione dell'interazione con i (sotto)sistemi esterni e con la rete

Dall'OOA all'OOD

- Nell'OOD, i risultati dell'analisi possono essere modificati per motivi di opportunità legati all'implementazione
- Le modifiche devono essere ridotte al minimo indispensabile, per mantenere il massimo accordo possibile tra OOA e OOD
- Principali motivi per modificare il modello OOA
  - Progettazione logica
    definizione dettagliata dell'implementazione delle classi e delle loro relazioni
  - Riuso di classi e/o componenti disponibili
  - Raggruppamento di classi del modello

Dall'OOA all'OOD

- Principali motivi per modificare il modello OOA
  - Miglioramento delle prestazioni
  - Aggiunta di caratteristiche specifiche
    - Gestione persistenza
    - Gestione comunicazioni
    - Diagnostica, …
  - Supporto alla portabilità
  - Se e solo se vincolanti, caratteristiche del linguaggio di programmazione, del S.O., dell'hardware, del DBMS, ... che devono essere utilizzati
    - Livello di ereditarietà supportato
    - Api
    - …

Progettazione logica

- Progetto dello schema logico del modello
  - Tipi di dato
  - Strutture dati
  - Operazioni
- Mentre nell'analisi ci si concentra su
  cosa deve fare il sistema,
  nella progettazione logica ci si concentra su
  come deve funzionare il sistema

Progettazione logica

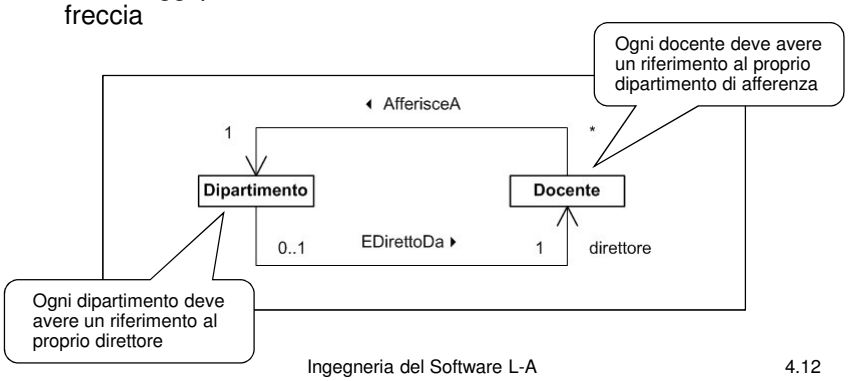
- Il passaggio dall'OOA all'OOD non è ben supportato dagli strumenti CASE:
  non c'è modo di separare le due fasi
- Due possibilità
  - L'analisi sfuma nella progettazione
    il modello OOA viene progressivamente aumentato e completato, sino alla progettazione del sistema
    ► l'analisi originale si perde
  - Il modello OOD è generato a partire dal modello OOA, ma viene mantenuto distinto
    ► l'analista deve mantenere la corrispondenza tra i due modelli, in caso di modifiche che si ripercuotono sull'analisi

Progettazione logica

- Durante la progettazione logica è necessario definire
  - Tipi di dato
    che non sono stati definiti nel modello OOA
  - Determinazione della navigabilità delle associazioni tra classi e relativa implementazione
  - Strutture dati
    necessarie per l'implementazione del sistema
  - Operazioni
    necessarie per l'implementazione del sistema
  - Algoritmi
    che implementano le operazioni
  - Visibilità di classi, (attributi,) operazioni, …

Navigabilità di un'associazione

- Possibilità di spostarsi
  da un qualsiasi oggetto della classe origine
  a uno o più oggetti della classe destinazione
  (a seconda della molteplicità)
- I messaggi possono essere inviati solo nella direzione della freccia



Navigabilità di un'associazione

- A livello di analisi, le associazioni di contenimento e di aggregazione hanno una direzione precisa detti A il contenitore e B l'oggetto contenuto, è A che contiene B, e non viceversa
- A livello implementativo, un'associazione può essere
  - mono-direzionale quando da A si deve poter accedere a B, ma non viceversa
  - bi-direzionale quando da A si deve poter accedere a B e da B si deve poter accedere *velocemente* ad A

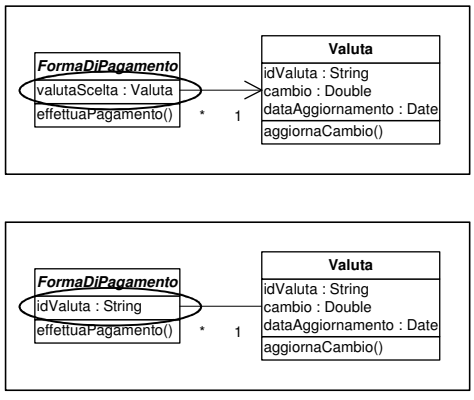
Navigabilità di un'associazione

- Dal punto di vista implementativo, la bi-direzionalità
  - è molto efficiente
  - ma occorre tenere sotto controllo la consistenza delle strutture dati utilizzate per la sua implementazione
- Negli OODBMS, è possibile definire associazioni bi-direzionali ed è lo stesso OODBMS a controllare e garantire la consistenza

Implementazione delle associazioni

- Associazioni con molteplicità 0..1 o 1..1
- Aggiungere alla classe cliente (contenitore) un attributo membro che rappresenta
  - il riferimento all'oggetto della classe fornitore
    - l'indirizzo di memoria dell'oggetto
    - l'identificatore univoco dell'oggetto (solo se persistente)
  - il valore dell'oggetto della classe fornitore (solo nel caso di composizione)

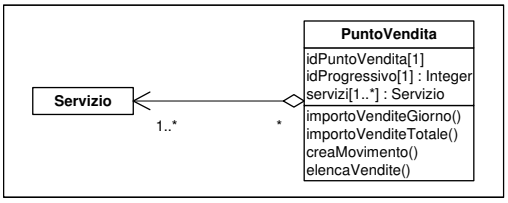
Implementazione delle associazioni

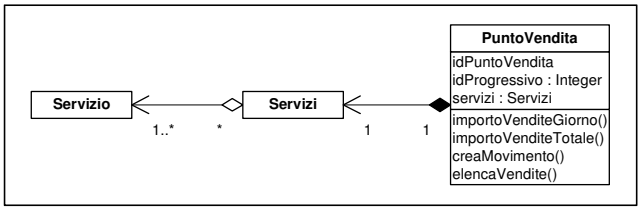


Implementazione delle associazioni

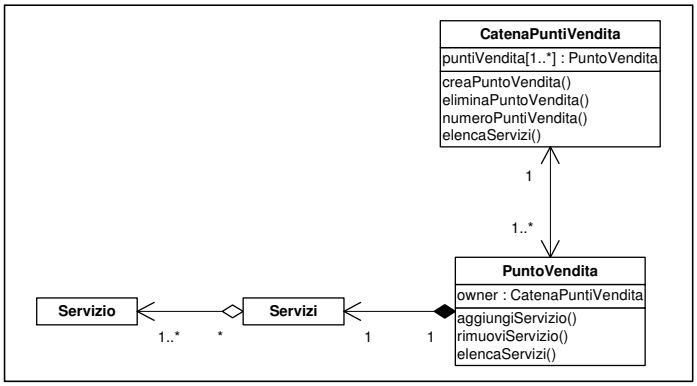
- Associazioni con molteplicità 0..* o 1..*
- Utilizzare una classe (classe contenitore) le cui istanze sono collezioni di (riferimenti a) oggetti della classe fornitore
- Aggiungere alla classe cliente un attributo che rappresenta un'istanza della classe contenitore
  - per valore, oppure
  - per riferimento
- La classe contenitore può essere
  - realizzata, oppure
  - presa da una libreria (preferibilmente)

Implementazione delle associazioni

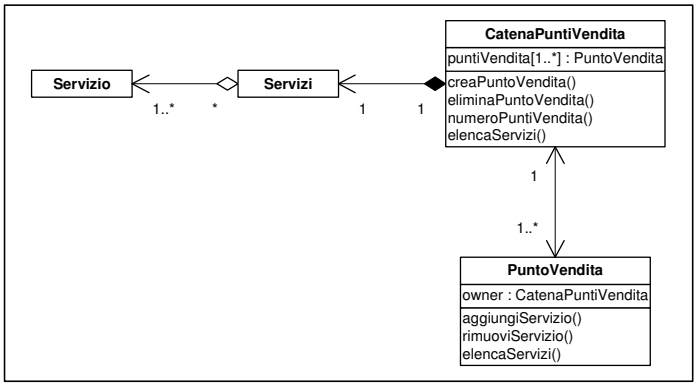




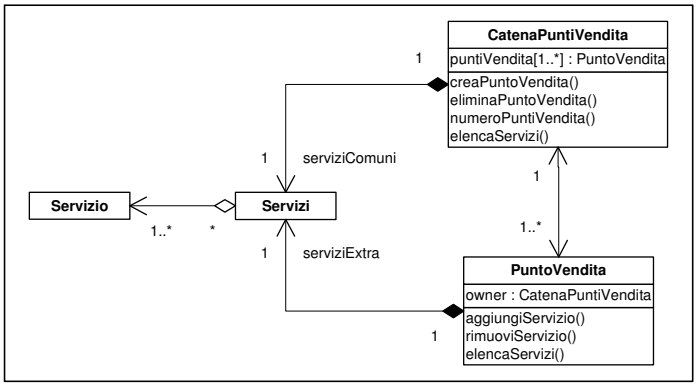
Implementazione delle associazioni



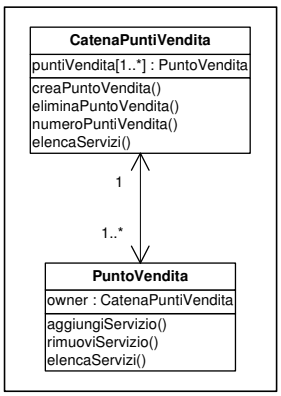
Implementazione delle associazioni

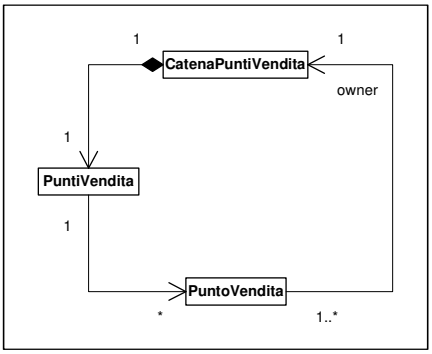


Implementazione delle associazioni



Implementazione delle associazioni





Classi contenitore

- Una classe contenitore (o semplicemente contenitore) è una classe le cui istanze contengono oggetti di altre classi
- Se gli oggetti contenuti sono in numero fisso e non è richiesto un particolare ordine, può essere sufficiente un vettore predefinito del linguaggio
- Se gli oggetti contenuti sono in numero variabile o hanno un ordine da mantenere, allora un vettore predefinito non basta e occorre una classe contenitore
- Esempi di classi contenitore sono
  - Vettori, *stack*, liste, alberi, …

Classi contenitore

- Funzionalità minime di una classe contenitore
  - Memorizzare (e quindi tenere insieme) gli oggetti della collezione
  - Aggiungere un oggetto alla collezione
  - Togliere un oggetto dalla collezione
  - Trovare un oggetto in una collezione
  - Enumerare (iterare su) gli oggetti della collezione
- I contenitori possono essere classificati in funzione
  - del modo in cui contengono gli oggetti
  - dell'omogeneità o eterogeneità di tali oggetti

Contenimento per valore

- L'oggetto contenuto (fornitore)
  - viene memorizzato nella struttura dati del contenitore (cliente)
  - esiste solo in quanto contenuto fisicamente in un altro oggetto
- Quando un oggetto deve essere inserito in un contenitore, viene duplicato
- La distruzione del contenitore comporta la distruzione degli oggetti contenuti

Contenimento per riferimento

- L'oggetto contenuto esiste per conto proprio
- L'oggetto contenuto può essere in più contenitori contemporaneamente
- Quando un oggetto viene inserito in un contenitore, non viene duplicato
  ma ne viene memorizzato solo il riferimento
- La distruzione del contenitore non comporta la distruzione degli oggetti contenuti
- Se un oggetto contenuto viene cancellato e il contenitore non ne viene avvisato, sorgono grossi problemi

Contenimento di oggetti omogenei ed eterogenei

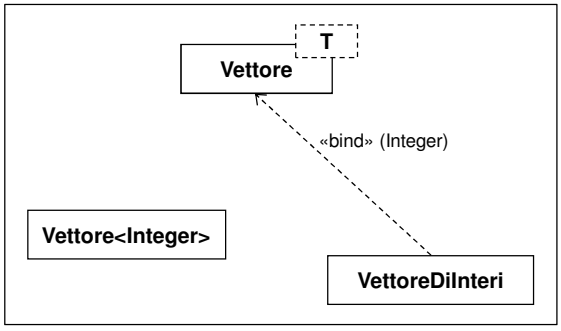
- Un contenitore può contenere una collezione di
  - oggetti omogenei, cioè tutti dello stesso tipo
  - oggetti eterogenei, cioè di tipo diverso
- Per implementare contenitori di oggetti omogenei (sia per valore, sia per riferimento) sono ideali le classi generiche (*template*)
- Per implementare contenitori di oggetti eterogenei (solo per riferimento) è necessario usare l'ereditarietà e sfruttare la proprietà che un puntatore alla superclasse radice della gerarchia può puntare a un'istanza di una qualunque sottoclasse

Contenimento di oggetti omogenei

- Classi generiche (*template*)
- Il tipo degli oggetti contenuti viene lasciato generico e ci si concentra sugli algoritmi di gestione della collezione di oggetti
- Quando serve una classe contenitore di oggetti appartenenti a una classe specifica, è sufficiente istanziare la classe generica, specificando il tipo desiderato

Classi generiche

- Classi in cui uno o più **tipi** (e/o valori) sono **parametrici**
- Ogni istanza di una classe generica costituisce una **classe indipendente** – non esiste un legame di ereditarietà



Classe generica Stack (C#)

```
public class Stack<T>
{
  private T[] _array;
  private int _size;
  private const int _defaultCapacity = 4;
  private static T[] _emptyArray = new T[0];

  public Stack()
  {
    _array = _emptyArray;
    _size = 0;
  }

  public int Count
  {
    get { return _size; }
  }
```

Classe generica Stack (C#)

```
public void Push(T item)
{
  if (_size == _array.Length)
  {
    T[] destinationArray = new
      T[(_array.Length == 0) ? _defaultCapacity :
        (2 * _array.Length)];
    Array.Copy(_array, 0, destinationArray, 0, _size);
    _array = destinationArray;
  }
  _array[_size++] = item;
}
```


Classe generica Stack (C#)

```
public T Peek()
{
  if (_size == 0)
    throw new InvalidOperationException("_size == 0");
  return _array[_size - 1];
}

public T Pop()
{
  if (_size == 0)
    throw new InvalidOperationException("_size == 0");
  T local = _array[--_size];
  _array[_size] = default(T);
  return local;
}
```

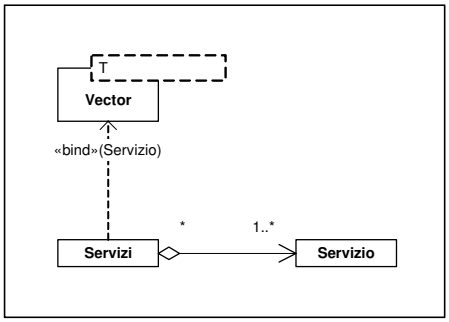
Classe generica Stack (C#)

```
  public void Clear()
  {
    //  Sets a range of elements in the System.Array
    //  to zero, to false, or to null,
    //  depending on the element type.
    Array.Clear(_array, 0, _size);
    _size = 0;
  }
}  //  Stack<T>
```


Classe generica Stack (C#)

```
…
Stack<int> s1;
Stack<double> s2;
Stack<DateTime> s3;
Stack<Stack<int>> s4;
…
for (int j = 1; j <= 20; j++)
  s1.Push(j);
…
while (s1.Count > 0)
{
  int v = s1.Pop();
  //  utilizzo di v
}
```

Contenimento di oggetti omogenei



Contenimento di oggetti eterogenei

- È necessario utilizzare l'ereditarietà
- La classe contenitore può essere generica, ma solo per quanto attiene la gestione dei riferimenti agli oggetti contenuti
- Nei linguaggi come Smalltalk, Java e C#, tutte le classi derivano da una sola classe radice
- In C#, la classe radice è **object** (System.Object)

Contenimento di oggetti eterogenei

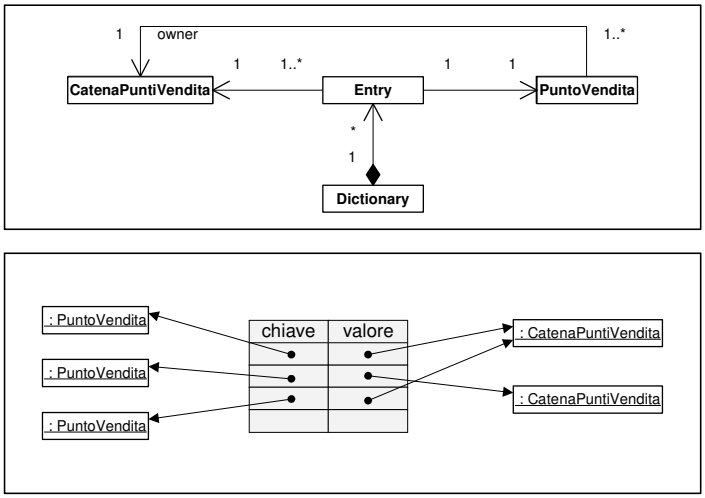
- In alcuni linguaggi (ad es. Java),
  i valori primitivi NON DERIVANO dalla classe radice,
  pertanto una classe contenitore
  - Non può contenere valori primitivi eterogenei
    ad esempio, se 120 non deriva dalla classe radice
    è necessario usare new Integer(120)
- In altri linguaggi (ad es. C#),
  i valori primitivi DERIVANO dalla classe radice,
  pertanto una classe contenitore
  - Può contenere valori primitivi eterogenei (*boxing*)

  Esempio 1

Implementazione delle associazioni

- Un modo alternativo per implementare un'associazione tra due oggetti è tramite un dizionario
- Un dizionario è un tipo particolare di contenitore, che associa due oggetti: la chiave e il rispettivo valore
- La chiave
  - Può essere un oggetto qualsiasi
    non necessariamente una stringa o un intero
  - Deve essere unica
- Il dizionario, data una chiave, ritrova in modo efficiente
  il valore ad essa associato

Implementazione delle associazioni



Identificazione degli oggetti

- Un oggetto (contenitore o no) può contenere un riferimento univoco a un altro oggetto
- Come è possibile identificare univocamente un oggetto per poterlo associare ad un altro?
- Nel caso di strutture dati interamente contenute nello spazio di indirizzamento dell'applicazione, un oggetto può essere identificato univocamente mediante il suo indirizzo (logico) di memoria

Identificazione degli oggetti

- Nel caso di database o di sistemi distribuiti, ad ogni oggetto deve essere associato un identificatore univoco persistente tramite il quale deve essere possibile risalire all'oggetto stesso, sia che risieda in memoria, su disco o in rete
- L'identificatore univoco è un attributo che al momento della creazione dell'oggetto viene inizializzato con:
  - un valore generato automaticamente dal sistema
  - il valore della chiave primaria di una tabella relazionale, …
- Il nome di tale attributo potrebbe essere
  - idDocente
  - idStudente, …

Identificazione degli oggetti
Un esempio reale

- La tecnologia COM (MS) permette a un'applicazione di trovare, caricare e utilizzare *run-time* i componenti necessari per la sua esecuzione
- Ogni componente è memorizzato in una DLL (*Dynamic Link Library*) – un file locale o remoto
- Quando l'applicazione ha bisogno di un componente, il sistema deve essere in grado di localizzare la DLL che contiene quel particolare componente

Identificazione degli oggetti
Un esempio reale

- L'indipendenza dalla collocazione fisica non consente di utilizzare un indirizzo fisico (*pathname*)
- Pertanto, deve essere utilizzato un meccanismo di indirizzamento logico che permetta di identificare univocamente il file che contiene il componente
- Si utilizzano degli identificatori globali (**GUID** = *Globally Unique Identifier*)

Identificazione degli oggetti
Un esempio reale

- Il concetto di GUID è stato introdotto, con un nome leggermente diverso (UUID = *Universally Unique Identifier*), dall'OSF (*Open Software Foundation*) nelle specifiche DCE (*Distributed Computing Environment*)
- In DCE gli UUID vengono utilizzati per identificare i destinatari delle chiamate di procedura remota (RPC)

Identificazione degli oggetti
Un esempio reale

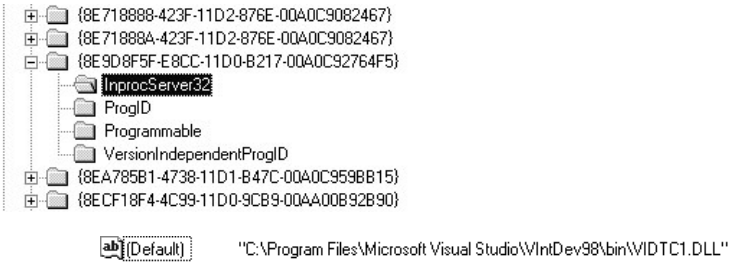
- Un GUID è un numero di 128 bit (16 byte) generato in modo da garantire l'unicità nello spazio e nel tempo e viene rappresentato così:
  `{32bb8320-b41b-11cf-a6bb-0080c7b2d682}`
- COM utilizza diversi tipi di GUID
- Il tipo più importante di GUID serve a identificare le classi di componenti:
  ogni classe di componenti COM è caratterizzata da un proprio identificatore che viene chiamato **CLSID** (*Class Identifier*)

Identificazione degli oggetti
Un esempio reale

- Disponendo di un CLSID, un'applicazione può chiedere alla funzione di sistema **`CoCreateInstance`** di creare un istanza del componente e di restituire un riferimento nel spazio di indirizzamento dell'applicazione stessa
- Il database di sistema di Windows (*registry*) mantiene una corrispondenza tra CLSID ed entità fisiche (DLL, EXE) che contengono l'implementazione dei componenti (*server*)

Identificazione degli oggetti
Un esempio reale

- **`CoCreateInstance`** provvede a
  - Reperire il *server* tramite il *registry*
  - Caricarlo in memoria (se non è già presente)
  - Creare un'istanza e restituirne un riferimento



Identificazione degli oggetti
Un esempio reale

- In .NET esiste la classe **`System.Guid`** che permette di gestire istanze di GUID
- Ad esempio, per ottenere un nuovo GUID, è sufficiente invocare il metodo statico **`Guid.NewGuid()`** che, ovviamente, restituisce un **`System.Guid`**
- Altri metodi e operatori permettono di confrontare GUID

Controllo della visibilità

- I tipi di accessi a un membro di una classe (attributo o metodo) possono essere di tre tipi:
  - *public*, ovvero visibili dall'esterno (+)
  - *private*, ovvero non visibili dall'esterno (-)
  - *protected*, ovvero visibili all'esterno solo dalle sottoclassi (#)
- È anche possibile rendere un membro di una classe o una intera classe visibile solo all'interno del *package* che contiene la classe (visibilità di *package*, *internal* in .NET)
- Infine, dichiarando una classe A *friend* di una classe B, ai metodi della classe A è concessa la visibilità completa di tutti i membri della classe B (C++)

Controllo della visibilità

Regole

- Dichiarare *private* (o *protected*) tutti gli attributi membro
- Dichiarare *public* solo i metodi (le operazioni) che devono essere visibili all'esterno
- Dichiarare *private* o *protected* tutti gli altri metodi
- Non abusare della dichiarazione *friend* (C++)

Interfaccia vs Classe astratta

- Spesso, molte classi concettualmente eterogenee presentano
  - un insieme di operazioni simili
  - in alcuni casi, un insieme di attributi simili

  es. gestione della persistenza
- In questi casi, è conveniente
  - Definire e realizzare un protocollo comune a tali classi
  - Definire un'interfaccia o una classe di generalizzazione comune dalla quale derivano tutte le suddette classi

Interfaccia vs Classe astratta

- Un'interfaccia
  - Non può essere istanziata
  - Non contiene alcuna implementazione – le classi derivate devono realizzare tutte le funzionalità
  - Non può contenere uno stato
  - Non può contenere attributi e metodi (e proprietà ed eventi) statici (a parte eventuali costanti comuni)

Interfaccia vs Classe astratta

- Una classe astratta
  - Non può essere istanziata
  - Può essere implementata completamente, parzialmente o per niente – le classi derivate:
    - devono realizzare le funzionalità non implementate
    - possono fornire una realizzazione alternativa a quelle implementate
  - Può contenere uno stato (comune a tutte le sottoclassi)
  - Può contenere attributi e metodi (e proprietà ed eventi) statici

Interfaccia vs Classe astratta

- Un'interfaccia
  - Deve descrivere una funzionalità semplice implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra di loro) ad esempio, una classe può essere
    - Clonabile (se implementa **`ICloneable`**),
    - Vista come una lista (se implementa **`IList`**), …
  - Deve essere stabile
    se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate

Interfaccia vs Classe astratta

- Una classe astratta
  - Può descrivere una funzionalità anche complessa comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro) - ad esempio, **`System.Enum`** fornisce le funzionalità di base di tutti i tipi enumerativi
  - Può fornire un'implementazione di *default* semplificando la programmazione delle sottoclassi
  - Può essere modificata quando si aggiunge un metodo a una classe astratta già in uso, è possibile fornire un'implementazione di *default*, in modo tale da non dover modificare le sottoclassi

Interfaccia vs Classe astratta

- Un'interfaccia può "ereditare"
  - Da 0+ interfacce
- Una classe astratta può "ereditare"
  - Da 0+ interfacce
  - Da 0+ classi (astratte e/o concrete)
    - **minimo 1** classe, nel caso esista una classe radice di sistema (ad es., **`System.Object`**)
    - **massimo 1** classe, nel caso in cui non sia ammessa l'ereditarietà multipla

Interfaccia vs Classe astratta

- Un'interfaccia
  - Non può gestire la creazione delle istanze delle classi che la implementano
    La creazione deve essere effettuata
    - dai costruttori delle suddette classi
    - da (un'istanza di) una classe non correlata, la cui unica funzionalità è la creazione di istanze di altre classi (***factory***)

Interfaccia vs Classe astratta

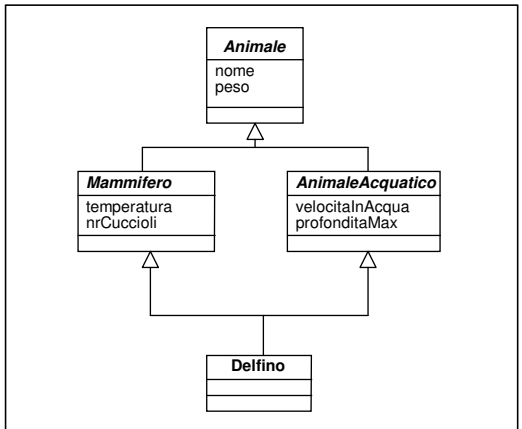
- Una classe astratta
  - Può gestire la creazione delle istanze delle sue sottoclassi
    La creazione può essere effettuata
    - come per l'interfaccia, ma anche
    - da un metodo statico della classe astratta (*factory*)

Modifiche per utilizzare il livello di ereditarietà supportato

- Se esistono strutture con ereditarietà multipla
- Se il linguaggio di programmazione non ammette l'ereditarietà multipla

- È necessario convertire le strutture con ereditarietà multipla in strutture con solo ereditarietà semplice

Modifiche per utilizzare il livello di ereditarietà supportato

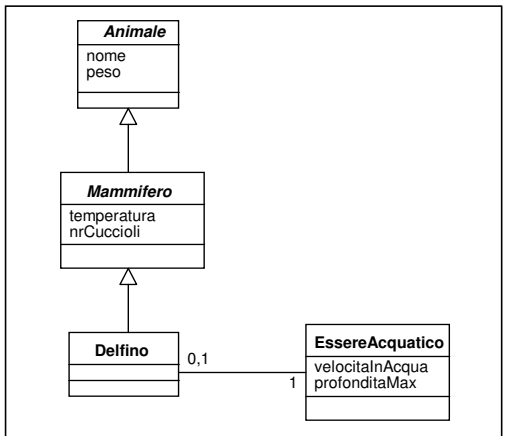


Ereditarietà multipla e ripetuta

Modifiche per utilizzare il livello di ereditarietà supportato

- 1ª possibilità
  - Scegliere la più significativa tra le superclassi ed ereditare esclusivamente da questa
  - Tutte le altre superclassi diventano possibili “ruoli” e vengono connesse con relazioni di contenimento
- In questo modo, le caratteristiche delle superclassi escluse vengono incorporate nella classe specializzata tramite contenimento e non tramite ereditarietà

Modifiche per utilizzare il livello di ereditarietà supportato

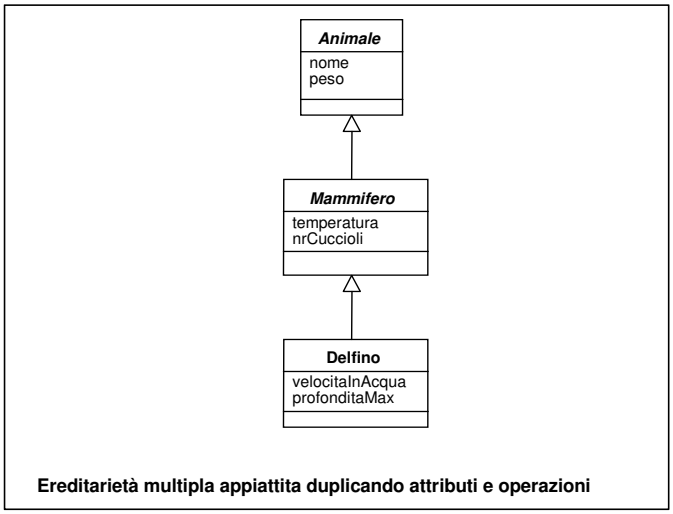


Ereditarietà multipla che diventa associazione

Modifiche per utilizzare il livello di ereditarietà supportato

- 2ª possibilità
  - Appiattire tutto in una gerarchia semplice
- In questo modo, una o più relazioni di ereditarietà si perdono e gli attributi e le operazioni corrispondenti devono essere ripetuti nelle classi specializzate

Modifiche per utilizzare il livello di ereditarietà supportato



Ereditarietà multipla appiattita duplicando attributi e operazioni

Miglioramento delle prestazioni

- Il software con le prestazioni migliori
  - Fa la cosa giusta “abbastanza velocemente” (cioè, soddisfacendo i requisiti e/o le attese del cliente)
  - Pur rimanendo entro costi e tempi preventivati
- Per migliorare la velocità percepita può bastare
  - La memorizzazione di risultati intermedi
  - Un’accurata progettazione dell’interazione con l’utente (ad es. utilizzando *multi-threading*)
- Un traffico di messaggi molto elevato tra oggetti può invece richiedere dei cambiamenti per aumentare la velocità

Miglioramento delle prestazioni

- Di norma, la soluzione è che un oggetto possa accedere direttamente ai valori di un altro oggetto (aggirando l’incapsulamento!)
  - Utilizzare metodi *inline*
  - Utilizzare la dichiarazione *friend*
  - Combinare insieme due o più classi
- Questo tipo di modifica deve essere presa in considerazione solo dopo che tutti gli altri aspetti del progetto sono stati soggetti a misure e modifiche
- L’unico modo per sapere se una modifica contribuirà in modo significativo a rendere il software “abbastanza veloce” è tramite le misure e l’osservazione

Supporto al componente DATA LOGIC

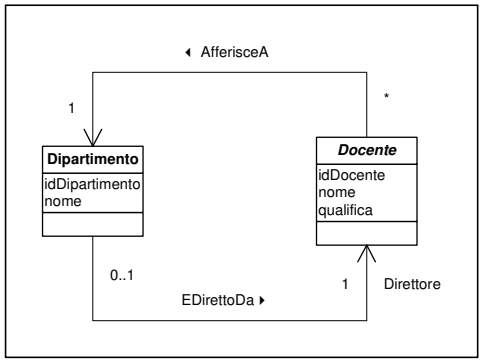
- Gestione della persistenza
- È necessario aggiungere un supporto alla persistenza - in particolare, gli oggetti persistenti devono essere in grado di
  - salvare il proprio stato
  - ripristinare il proprio stato
- Altre possibilità per gestire gli oggetti persistenti:
  - utilizzare un "*Object-Oriented Database Management System*" (OODBMS)
  - utilizzare la tecnica di "*Object-Relational Mapping*" (ORM) – Hibernate e NHibernate

Supporto al componente DATA LOGIC

- Ad ogni classe persistente aggiungere
  - Un attributo (*id*) che permetta di identificare in modo univoco le istanze della classe
  - Un'operazione (*virtual*) per definire come gli oggetti debbano salvare il proprio stato
  - Un'operazione (*virtual*) per definire come gli oggetti debbano ripristinare il proprio stato

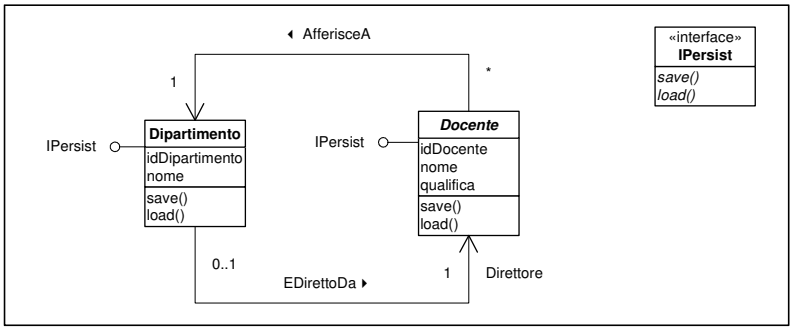
Supporto
al componente DATA LOGIC

- L'identificare univoco è fondamentale per poter salvare e ripristinare i riferimenti tra oggetti



Supporto
al componente DATA LOGIC

- Tutte le classi persistenti dovrebbero
  - derivare dalla stessa classe o
  - implementare la stessa interfaccia



Progettazione del componente DATA LOGIC

- **DATA LOGIC** deve contenere la logica necessaria per gestire la persistenza dei dati
- In particolare, deve:
  - Fornire l'infrastruttura per la memorizzazione e il ritrovamento degli oggetti su/da un sistema di gestione dati (**DATA MANAGER**)
  - Isolare l'**APPLICATION LOGIC** dal **DATA MANAGER**, sia quest'ultimo
    - un semplice gestore di file
    - un database relazionale
    - un database orientato agli oggetti
    - …

Progettazione del componente DATA LOGIC

- Con tale approccio, ogni oggetto
  - saprà come salvare e ripristinare il proprio stato (in quanto responsabile della propria *serializzazione*)
  - non conoscerà nulla dei dettagli di come tali operazioni verranno effettuate (responsabilità del **DATA LOGIC**)
- La progettazione del componente **DATA LOGIC** comprende il progetto
  - dello schema dei dati (struttura dei file o schema del database)
  - delle operazioni che agiscono sui dati