DIAGRAMMA DELLE CLASSI

La fase di progettazione di un prodotto o servizio sw permette di ottenere l'architettura del software.

In cosa consiste, praticamente, questa architettura?

Consiste nella descrizione delle classi che compongono il software, comprese le **responsabilità** di ciascuna classe (cosa deve fare), le **collaborazioni** fra le classi, ossia le relazioni (chiamate associazioni in UML, occhio) che ci sono fra le classi, che verranno spiegate in seguito nel dettaglio.

Uno strumento pratico che può essere utile per la progettazione delle classi è quello costituito dalle schede **CRC (Class, Responsability, e Collaboration)**. Sono dei cartoncini, ognuno dei quali rappresenta una classe, sui quali vengono scritte le seguenti informazioni: nome della classe (in alto), responsabilità (a sinistra), classi con cui collabora per portare a termine quella responsabilità (a destra). Le schede CRC sono uno strumento **informale** che aiuta nella prima stesura del diagramma delle classi. Parlo di prima stesura perché il diagramma delle classi viene generalmente realizzato due volte. Una prima volta nella fase di progettazione, in maniera approssimata (con l'indicazione degli attributi e dei soli metodi essenziali), una seconda volta dopo l'implementazione del codice, in maniera più precisa (con l'indicazione di ogni metodo, ogni parametro, con il proprio tipo ecc..), questo al fine di documentare il software.

Un esempio di schede CRC relativo ad un software per la gestione delle posizioni delle navi:

ESEMPIO

La scheda CRC relativa alla posizione di una nave registrata in una determinata data/ora ha verosimilmente il seguente aspetto:

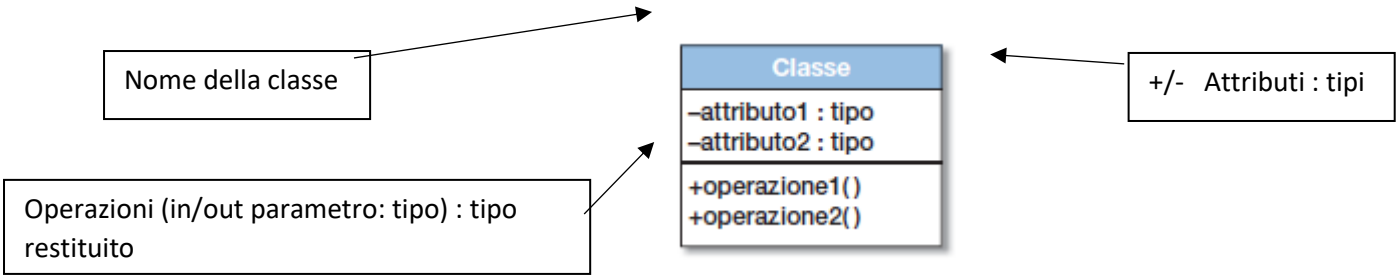
Position	
• Memorizza e consente di impostare/modificare le coordinate geografiche (latitudine/longitudine) della posizione, sia in formato decimale sia DMS. • Memorizza e consente di impostare/modificare la data e l'ora di registrazione delle coordinate geografiche. • Determina la distanza tra due posizioni.	• `Time` • `Date`

Invece la scheda CRC per un percorso inteso come una sequenza ordinata nel tempo di posizioni potrebbe essere impostata in questo modo:

Path	
• Memorizza e consente la gestione (aggiunta, eliminazione e scorrimento) della sequenza di posizioni. • Determina la lunghezza del percorso come somma delle distanze tra posizioni successive.	• `Position`

Il diagramma delle classi

Una volta individuate le classi esse vengono rappresentate nel diagramma UML delle classi. Ogni classe è rappresentata nel seguente modo:



Nel linguaggio UML i **metodi** vengono chiamati **operazioni** (ma noi li chiameremo metodi).

Attributi e metodi sono preceduti da un simbolo che indica quali classi esterne possono accedere a quell'elemento:

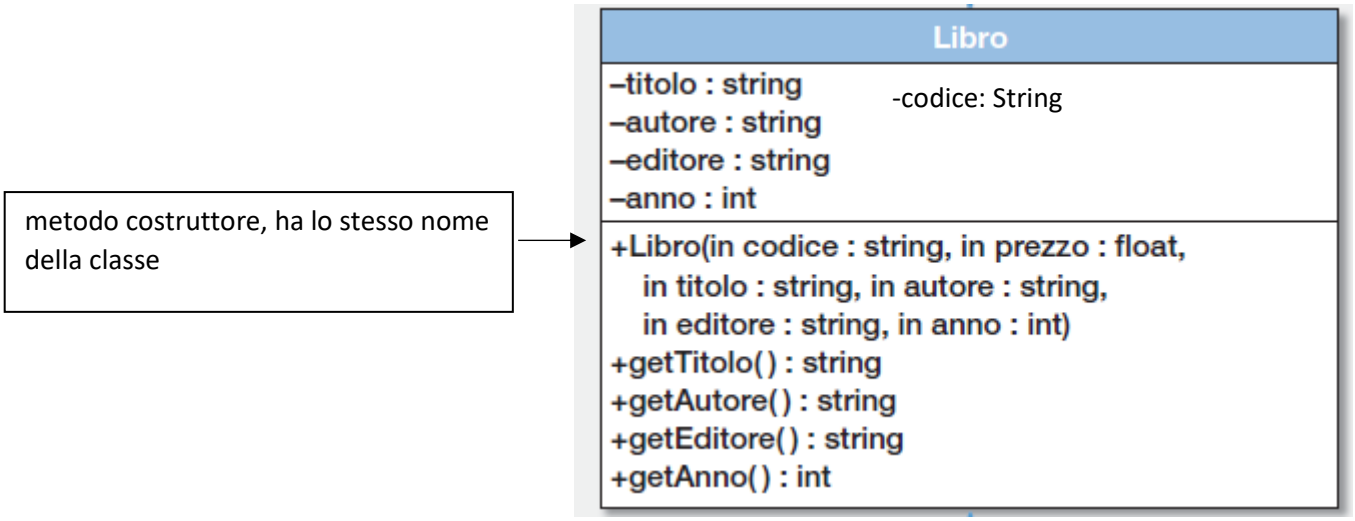
- \- Privato: attributo o operazione non è accessibile dal codice esterno alla classe
- \+ Pubblico: attributo o operazione è accessibile dal codice esterno alla classe
- \# Protetto: attributo o operazione accessibile dal codice delle classi figlie
- ~ Package: attributo o operazione accessibile dal codice di classi dello stesso package.

Per ogni attributo viene specificato il tipo di dato.

Per ogni metodo (operazione) vengono specificati il tipo di dato restituito e i parametri (con i relativi tipi di dati e con un'indicazione in/out che ne specifica il ruolo come parametro di input o di output).

Gli eventuali attributi e metodi statici vengono indicati sottolineati

Ecco un esempio di rappresentazione di una classe in un diagramma delle classi UML. La classe in questione è la classe "Libro" .



Osservazioni importanti:

- Le classi hanno lettera iniziale maiuscola
- le operazioni (metodi) e gli attributi hanno lettera iniziale minuscola e, se il nome è formato da più parole, le parole successive alla prima hanno lettera iniziale maiuscola (esempio: getTitolo).
- Esiste sempre una classe "costruttore" che, quando viene invocato "istanzia" un oggetto di quella classe. I costruttori possono essere più di uno.

Associazioni fra classi

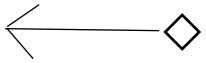
Trattiamo separatamente le associazioni raggruppandole nel seguente modo:

A. **Aggregazione:** si ha quando la classe C1 ha un attributo o più attributi (ad esempio un vettore di elementi) di tipo C2.

B. **Composizione**: è quasi uguale all' aggregazione, anche la composizione si ha quando la classe C1 ha un attributo o più attributi (ad esempio un vettore di elementi) di tipo C2. Vedremo più avanti la differenza.

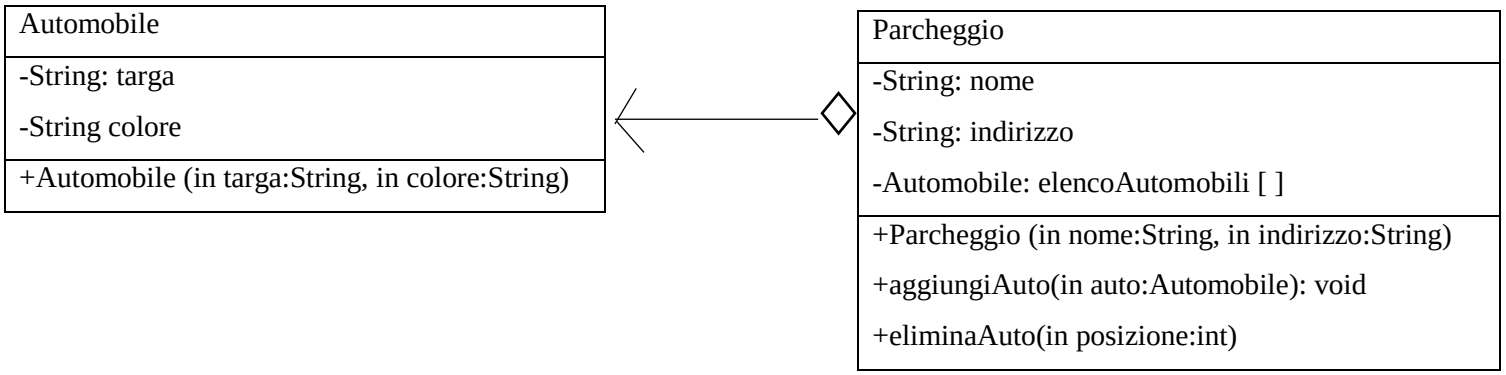
C. **Generalizzazione:** si ha quando una classe C1 è figlia di una classe C2.

Spieghiamo la differenza fra **aggregazione** e **composizione**.

Aggregazione (simbolo:  rombo vuoto)

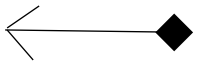
L'aggregazione è una relazione parte/tutto fra due classi dove una classe (la classe "tutto") ha un attributo che è un'istanza dell'altra classe (classe "parte"). Il rombo indentifica la parte del tutto. Si ha **un'aggregazione** quando, nel software **possono esistere istanze della classe "tutto" nelle quali non sono presenti elementi della classe "parte".**

Esempio:



La classe parcheggio avrà come attributo un array di automobili. Ogni elemento dell'array contiene un oggetto di tipo automobile che rappresenta un'automobile parcheggiata nel parcheggio.

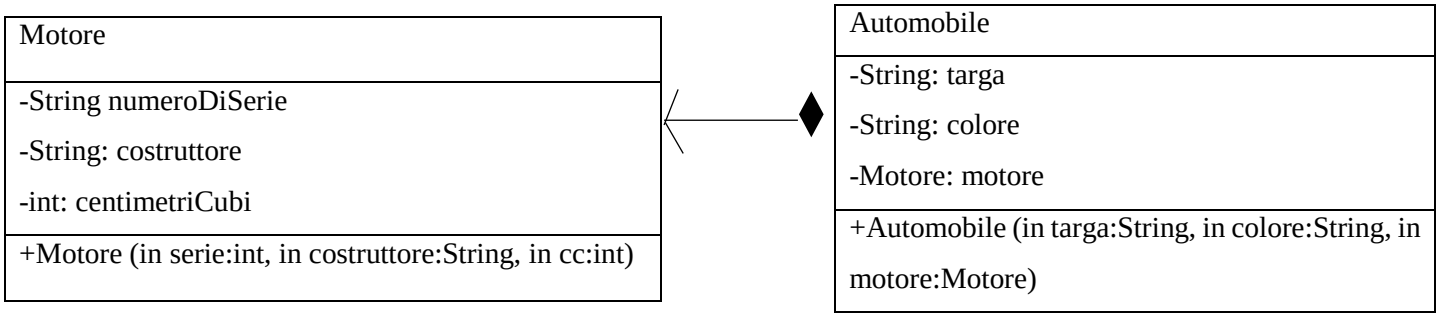
Un parcheggio aggrega zero o più automobili, ma possono esistere oggetti di tipo parcheggio **in cui non vi sono automobili**, ad esempio perché è vuoto. Per questo l'associazione è di **aggregazione** (e non di composizione).

Composizione (simbolo:  rombo pieno)

Si ha una relazione parte/tutto fra due classi dove una classe "tutto" ha uno o più attributi che soni un'istanza della classe "parte". Il rombo indentifica la classe "tutto". Le istanze della classe "parte" esistono, nel software, solamente come componenti della parte "tutto". Un'istanza della classe "tutto" in cui manca l'istanza della classe "parte" è descritta in maniera incompleta. E' una relazione più forte rispetto alla aggregazione.

Esempio:

modifichiamo la precedente classe Automobile in modo da descriverla meglio aggiungendo un attributo per indicare le caratteristiche del motore montato



Ogni istanza di automobile è composta da vari attributi (targa, colore ecc.) fra i quali un'istanza della classe motore. Eliminando l'attributo motore, la classe Automobile risulta descritta in maniera incompleta.

Nella rappresentazione grafica delle due associazioni la freccia rappresenta la **navigabilità** ossia che dalla classe di partenza sarà possibile accedere a metodi della classe di arrivo**, la freccia parte dalla classe che contiene come attributo una o più istanze della classe verso cui arriva la freccia.**

Non è sempre semplice distinguere una aggregazione da una composizione. Ci si deve chiedere: ha ancora senso di esistere nel software la classe "parte" se viene distrutta la parte "tutto"? In caso affermativo la relazione è una aggregazione altrimenti composizione.

Le associazioni di composizione e aggregazione sono dette anche associazioni "***has a"*** (**ha un**) perché in tale associazione ***una classe "ha un" oggetto dell'altra classe***.

Se non si riesce a distinguere aggregazione da composizione non perdiamoci troppo tempo, comunque il codice che genereranno in Java sarà lo stesso. L'importante è distinguerle chiaramente dalla generalizzazione (che vedremo dopo).

Molteplicità delle associazioni:

La molteplicità indica, nelle relazioni di aggregazione e composizione, il numero di **oggetti (istanze) minimo e massimo** con cui una classe partecipa alla associazione. I possibili valori di molteplicità cono:

1	Esattamente una istanza
N	Esattamente N istanze
1…*	1 o più istanze (si legge 1-star)
0…*	0 o più istanze (si legge 0-star)
1…N	1 o più istanze ma al massimo N
0…N	0 o più istanze ma al massimo N

L'utilizzo della molteplicità verrà illustrato nell'esempio seguente.

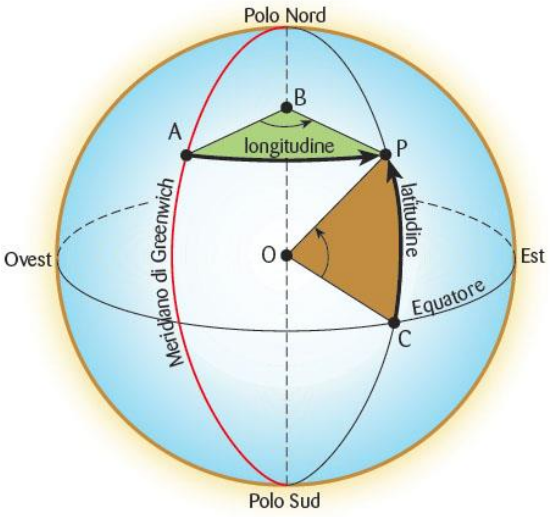
Esempio diagramma delle classi

Ipotizziamo di voler realizzare un software che consenta di rappresentare il percorso di una nave nel tempo. Il percorso della nave è costituito da una sequenza di "posizioni" della nave sulla superficie terrestre, rilevate nel tempo (ad esempio ogni 10 secondi).

Ogni posizione rilevata è caratterizzata da:

- una data/ora di rilevazione
- dalle coordinate (latitudine e longitudine) in cui si trova la nave in quella data/ora. La latitudine e la longitudine sono degli angoli, quindi possono essere espresse sia come numeri decimali (double) sia come coordinate DMS (Degree Minutes, Seconds).

La classe Path rappresenta un percorso (della nave) ed ha come attributi un array di oggetti istanza della classe Position che rappresentano le varie posizioni.



Come si misurano le coordinate geografiche

La latitudine varia da 0° (all'equatore) a 90° (ai poli), e il valore della latitudine è seguito dall'indicazione:

- Nord (abbreviato con N), se il punto si trova nell'emisfero settentrionale;

- Sud (abbreviato con S), se il punto si trova nell'emisfero meridionale.

La longitudine varia da 0° (sul meridiano di Greenwich) a 180°, a cui segue l'indicazione:

- Est (abbreviato E), se il punto è ad est del meridiano di Greenwich;

- Ovest (abbreviato O oppure W), se il punto è ad ovest di tale meridiano.

Unità di misura delle coordinate geografiche

Per come sono state definite, latitudine e longitudine sono grandezze angolari e quindi vengono misurate in gradi. Per esprimerne la misura si può utilizzare:

- il sistema sessagesimale, e quindi indicare la misura in gradi, primi e secondi;

- il grado decimale, ossia esprimere la misura dell'angolo con un unico numero decimale.

Per esempio le coordinate del Colosseo a Roma sono, nel sistema sessagesimale:

- latitudine N 41° 53’ 24’’
- longitudine E 12° 29’ 32’’

Per convertirle in gradi decimali, i gradi rimangono uguali, ai gradi si sommano i primi divisi per 60 e i secondi divisi per 60*60

- latitudine N 41+53/60 + 24/(60*60)= 41.89
- longitudine E 12+29/60+ 32/(60*60)= 12.4922222222

La conversione da grado decimale a grado sessagesimale si ottiene moltiplicando la parte decimale per 60 (la parte intera ottenuta indica i primi) e moltiplicando la parte decimale dei primi per 60 (la parte intera indica i secondi

- latitudine N 41.89 → 0,89*60= 53.4 → 0.4*60 = 24
- longitudine E 12.4922222222 → 0.4922222222*60= 29.53333333 → 0.53333333*60= 32

gradi | primi | secondi

Un primo esempio di diagramma delle classi potrebbe essere il seguente: cerchiamo insieme di dare un significato agli attributi e ai metodi presenti.

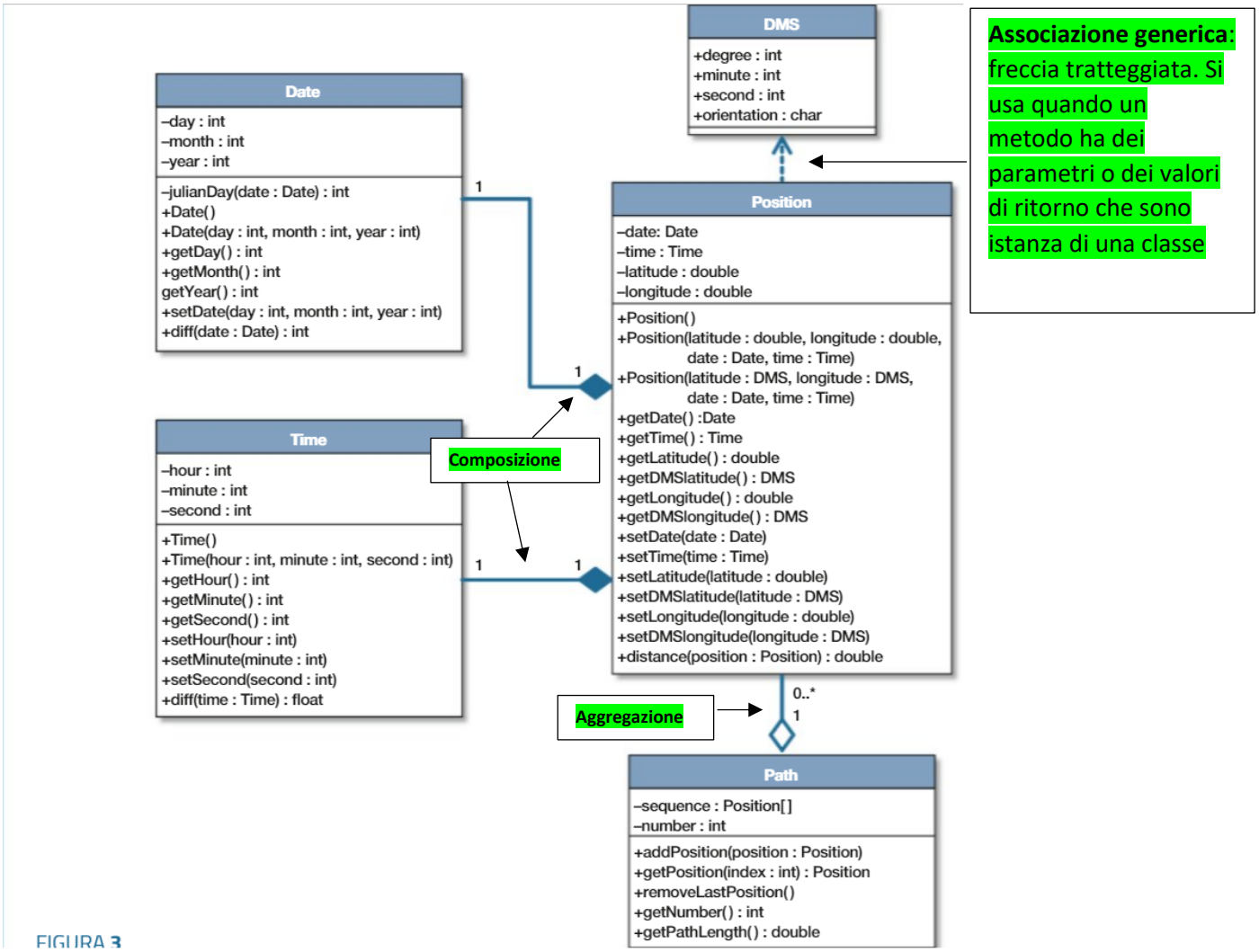


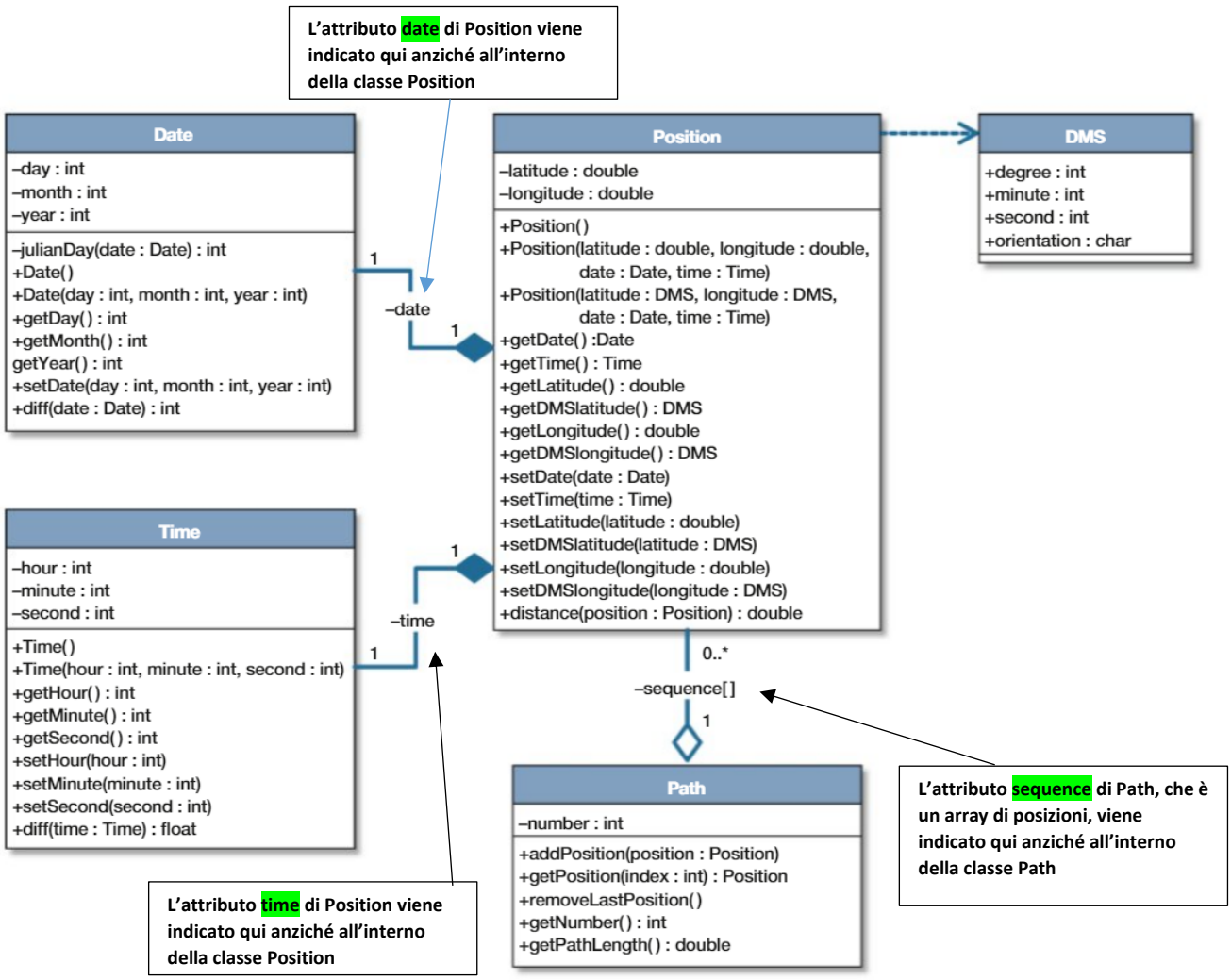
FIGURA 3

Perché le associazioni di Date e Time con Position sono Composizioni? Perché Date e Time sono dei componenti fondamentali per la posizione, senza di esse mancherebbe una certa quantità di informazione alla classe Position. Quindi Date e Time sono in relazione di **Composizione** con la classe Position.

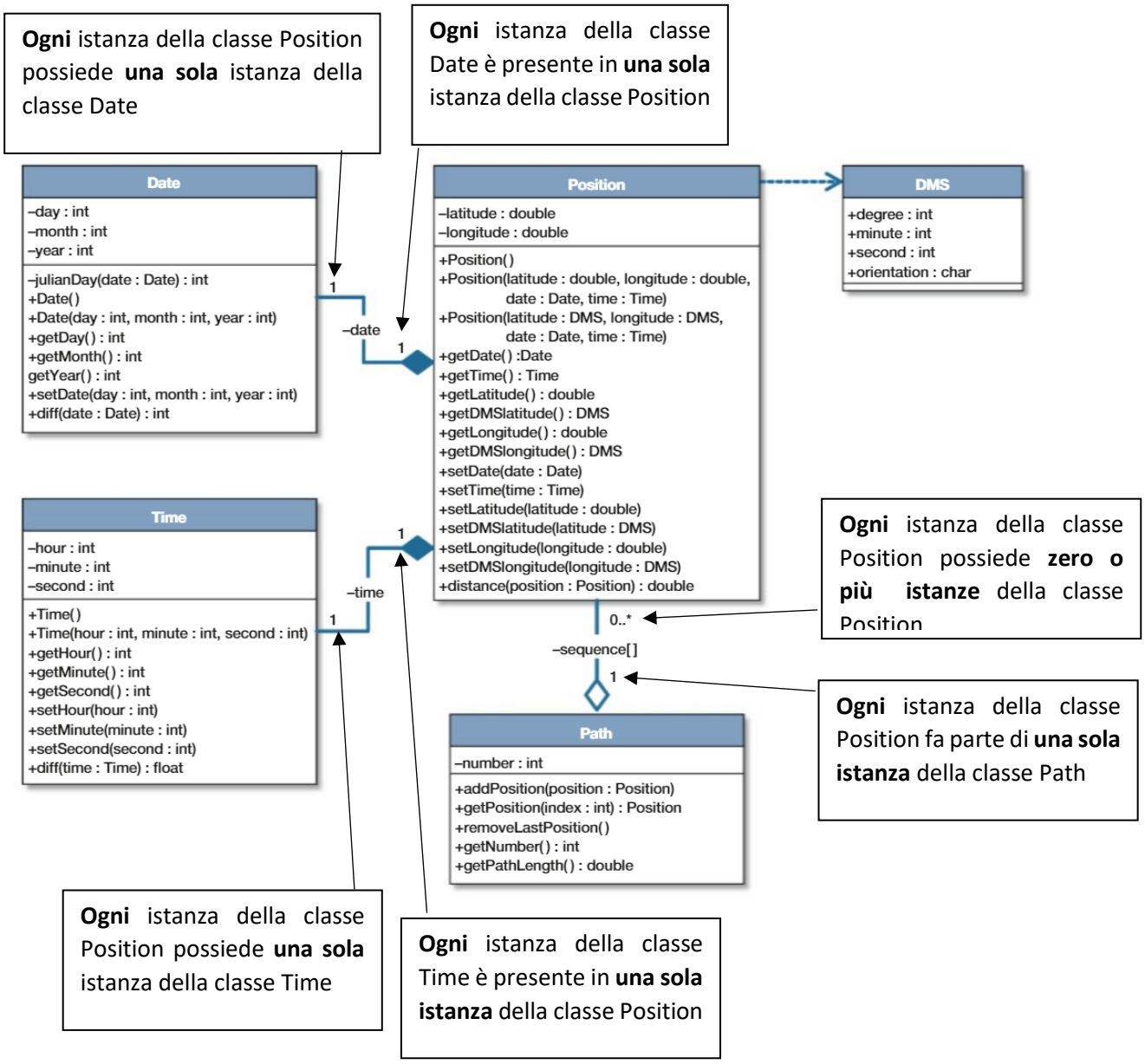
Perchè l'associazione di Position con Path è un'aggregazione?

Perché la classe Path può esistere anche senza posizioni (ad esempio prima che la nave inizi il viaggio), quindi fra la classe path e la classe Position vi è una relazione di **aggregazione**. Si può dire che un'istanza di **Path** è **un'aggregazione** di istanze **Position.**

OSSERVAZIONE: In realtà, lo standard del linguaggio UML prevede che nel rappresentare le associazioni di **aggregazione** e **composizione** gli attributi che realizzano le associazioni (nell'esempio gli attributi date, time e sequence) non vengano indicato nell'elenco degli attributi all'interno della classe ma sulla la freccia che indica l'associazione. Quindi il diagramma precedente è più corretto nel seguente modo:



I valori di molteplicità sono indicati nel diagramma sulla freccia che rappresenta l'associazione. Le molteplicità vengono lette nel seguente modo:



La regola di lettura è la seguente: il valore di **molteplicità viene messa sulla classe di "arrivo"** dell'associazione e si legge così': *ogni classe di partenza ha molteplicità classe di arrivo*, si legge in entrambe le direzioni.

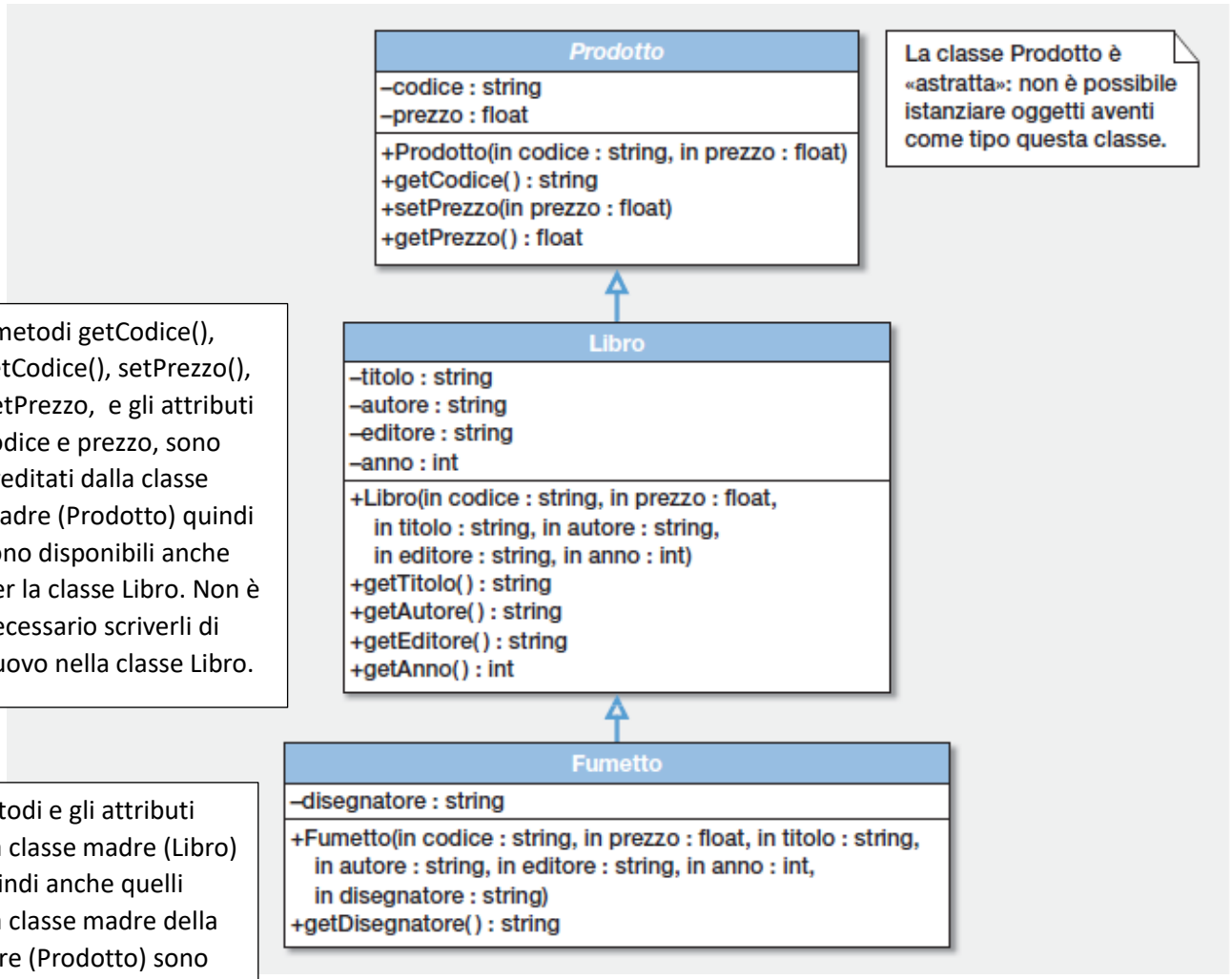
Generalizzazione

Il concetto di generalizzazione è stato incontrato anche nei diagrammi UML dei casi d'uso. La generalizzazione fra classi rappresenta in ULM il concetto di ereditarietà della OOP. La classe figlia eredita tuti i metodi e gli attributi dalla classe madre. La classe figlia può avere ulteriori attributi e metodi.

La generalizzazione viene indicata anche come associazione ***"is a"*** **(è un)**, perché la classe figlia, possedendo tutti metodi e attributi della madre "e' anche una classe madre" (nell'esempio successivo: un libro **è un** prodotto)

La generalizzazione si rappresenta con una freccia con la punta triangolare vuota all'interno.

Nel seguente esempio vengono rappresentati dei libri che sono i prodotti venduti in un negozio. La classe Libro è figlia della classe Prodotto (ne eredita i metodi, gli attributi ed e' caratterizzata da ulteriori metodi e attributi). A sua volta la classe Fumetto è figlia della classe Libro.



I metodi getCodice(), setCodice(), setPrezzo(), getPrezzo, e gli attributi codice e prezzo, sono ereditati dalla classe madre (Prodotto) quindi sono disponibili anche per la classe Libro. Non è necessario scriverli di nuovo nella classe Libro.

I metodi e gli attributi della classe madre (Libro) e quindi anche quelli della classe madre della madre (Prodotto) sono ereditati anche dalla classe Fumetto.

Realizzare il refactoring

Il refactoring è una riscrittura del codice "*già funzionante"* per migliorarlo dal punto di vista della aderenza alle specifiche della OOP (massima indipendenza frale classi e massima riusabilità del codice), tali specifiche migliorano notevolmente la manutenibilità del codice. Una tecnica di refactoring consiste nell'individuare, fra varie classi, quali hanno dei metodi in comune, e quindi raggruppare i metodi in una **superclasse** (di cui le classe precedentemente individuate diverranno figlie). Questo consente di scrivere una sola volta il codice dei metodi utilizzati in diverse classi

Abbiamo visto come rappresentare le classi e le loro relazioni. Per realizzare un diagramma delle classi bisogna:

1. Individuare le classi
2. Identificare le responsabilità
3. Identificare le relazioni fra le classi

Le tre cose non vengono svolte in sequenza ma in maniera iterativa, in progetti complessi non verranno individuate tutte le classi subito, magari quando si individuano le responsabilità nasce l'esigenza di definire altre classi, oppure ci si accorge che quello che era stato considerato un attributo in realtà è modellizzato meglio come classe, allora si apportano modifiche e così via… per questo serve la matita! Bisogna spesso cancellare e riscrivere!

Come individuare le classi?

1) Il primo passo da fare per creare il diagramma delle classi è quello di individuare quali "concetti" potranno diventare delle classi

- Dei concetti reali (libro, biblioteca, automobile)
- I ruoli assunti da una persona: (studente, impiegato, docente..)
- Gli eventi: (prestito, assunzione, partenza, arrivo, esame, noleggio, spesa, trasfrimento..)

Un primo approccio è quello di individuare i **sostantivi** presenti nella descrizione dei requisiti, i sostantivi possono essere **classi** o **attributi**.

Sono classi se:

- sono descritti da altre informazioni (es Automobile è descritta dalla targa e dal colore)

Sono attributi se:

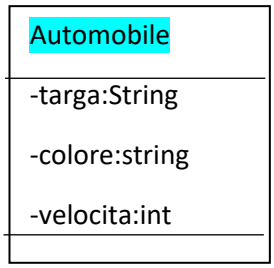
- Descrivono delle proprietà intrinseche di altri oggetti (delle classi) (ad esempio: colore)

Un sostantivo può identificare in alcuni casi un attributo in altri una classe, dipende dal suo significato all'interno del sistema software che si vuole realizzare. Ad esempio:

"Le automobili sono caratterizzate da targa, colore e velocità"

Automobile è una classe

colore è un attributo

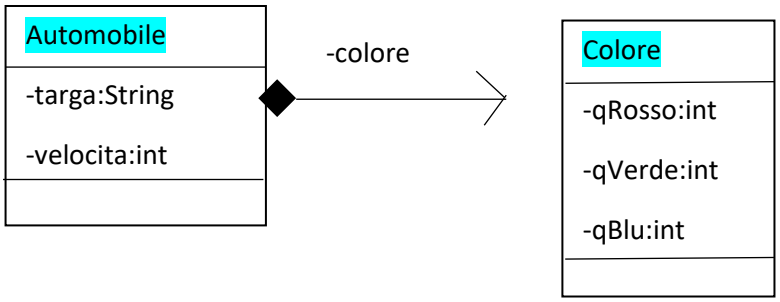


"Le automobili sono caratterizzate da targa, velocità e colore. Il colore è il risultato della somma di tre componenti, rosso, verde e blu ciascuna presa in una specifica quantità espressa con un valore che va da 0 a 255"

Automobile è una classe (targa e colore sono suoi attributi)

Colore è una classe (quantità di rosso, quantità di verde, quantità di blu sono suoi attributi)

Le due classi sono in una associazione di composizione



Come individuare le operazioni (metodi) e le relazioni?

Un suggerimento è quello di individuare le operazioni cercando i verbi nella descrizione dei requisiti, ad esempio: "l'automobile deve poter accelerare e decelerare" indica che sarà necessario creare due metodi: "accelera" e "decelera", i quali conterranno il codice che consente, quando invocati, di modificare l'attributo velocità di un'istanza della classe Automobile.

Altri metodi che devono generalmente essere presenti sono i metodi che consentono di impostare i valori degli attributi (metodi setter: setColore, setTarga…) o di ottenere il valore degli attributi (metodi getter: getTarga, getColore..);

Una classe C1 può essere in relazione con un'altra classe C2 per uno dei seguenti motivi:

- Un attributo di C1 è un'istanza di C2 (Esempio: il colore e l'automobile nel secondo esempio)
- C1 vuole poter rimuovere/aggiungere elementi di un'altra classe in una struttura dati (ad esempio in un array) Ad esempio nella classe "Parcheggio" dell'esempio precedentemente illustrato, la classe Parcheggio avrà come attributo un array di automobili, quindi la classe Parcheggio avrà dei metodi per la gestione delle automobili: "aggiungiAutomobile", "RimuoviAutomobile" " getAutomobile" ecc..

Esempio Expense –manager: la progettazione del software

Per individuare le classi che compongono l'applicazione Expense manager realizziamo le seguenti schede CRC. Scriviamo all'interno delle schede:

- cosa rappresenta la classe
- cosa deve fare
- le classi con cui collabora

OSSERVAZIONE: Poiché abbiamo trattato in informatica la classe LocalDate, utilizzeremo tale classe per rappresentare le date. Essendo una classe "già pronta" non la riportiamo nel diagramma delle classi, questo renderà il diagramma più leggibile.

Progetto	
Rappresenta una collezione (ad esempio un array) di spese e di trasferimenti. Permette di aggiungere/eliminare/modificare le spese e i trasferimenti al progetto. Permette di visualizzare tutte le spese e i trasferimenti oppure solo alcune (quelle con determinati pagatori o partecipanti o destinatari ecc.) di un progetto. Permette di visualizzare il bilancio di un progetto mostrando chi deve ricevere denaro e chi deve versare denaro. Permette di esportare in formato html o CSV i dati visualizzati. Permette di salvare su file i dati (spese e trasferimenti).	• Spesa • Trasferimento

Spesa	
Rappresenta una spesa sostenuta da un partecipante al progetto. Permette di memorizzare chi ha sostenuto la spesa, quando è stata sostenuta, chi sono i partecipanti alla spesa, l'ammontare in euro della spesa	

Trasferimento	
Rappresenta una trasferimento di denaro da parte di un partecipante al viaggio in favore di un altro partecipante. Permette di memorizzare chi ha è il pagatore, chi è il beneficiario, quando è stato effettuato il trasferimento, l'ammontare in euro del trasferimento.	

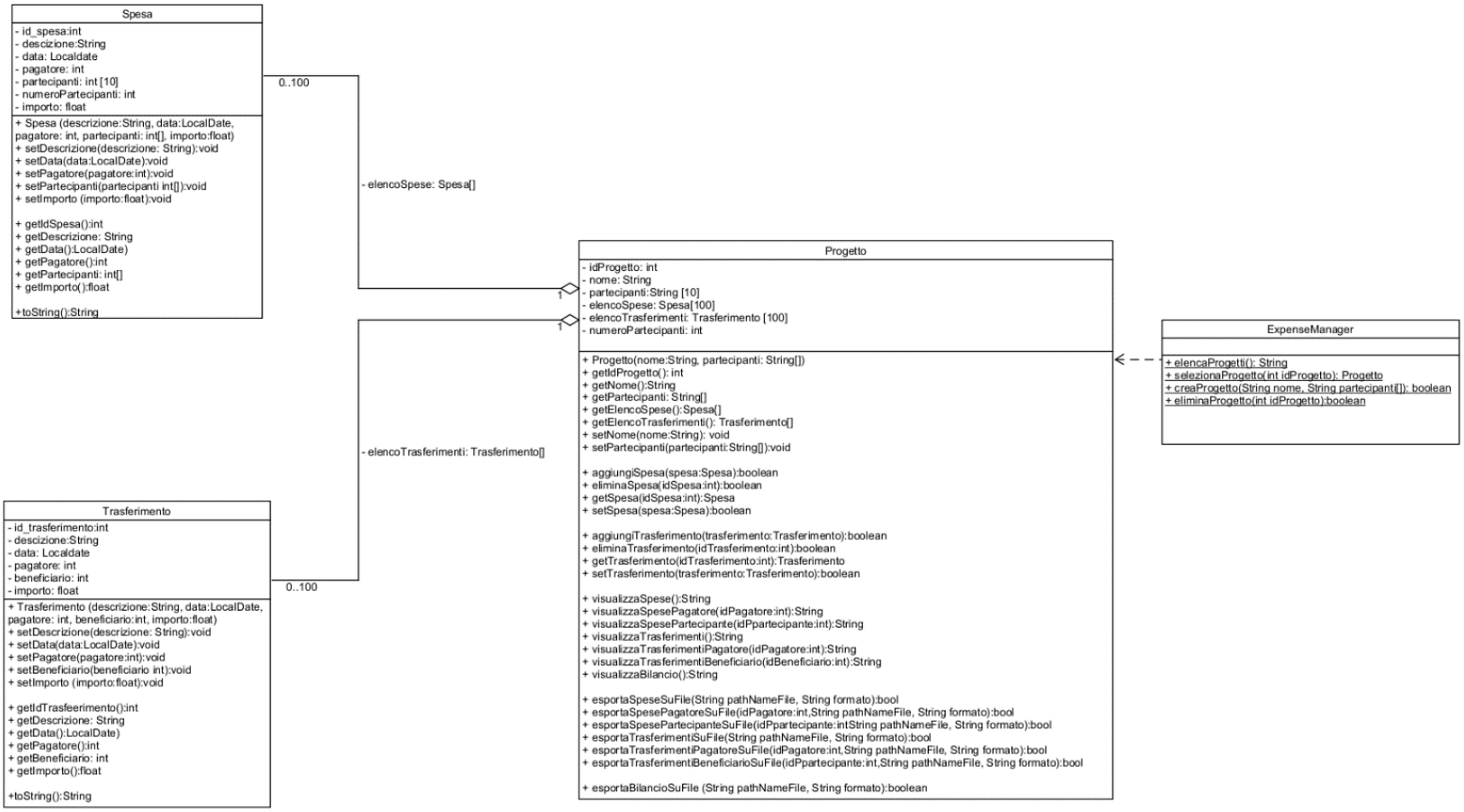
ExpenseManager	
Consente di visualizzare l'elenco dei progetti presenti. Consente di selezionare, eliminare, aggiungere un progetto.	• Progetto

Realizziamo il diagramma delle classi individuando gli attributi e i metodi e le relazioni per ciascuna classe.

Per leggere meglio il diagramma è opportuno scaricare ed aprire il file "Expense manager.uxf" leggibile con il software "UMLet". Il file si trova su moodle.

Si osservi nel diagramma sottostante che la classe Progetto è un'aggregazione di Spese e di Trasferimenti.

Si osservi che la classe ExpenseManager non ha attributi ed è costituita da soli metodi statici. Questo ci permette di capire che tale classe non è un "elenco dei progetti", essa è statica quindi quindi non verranno istanziati oggetti da questa classe. Essa è una sorta di "libreria" i cui metodi sono come delle funzioni invocabili direttamente dalla classe (astratti). I metodi della classe Expense Manager saranno invocati all'interno della Main Class e consentiranno di creare, selezionare, eliminare un progetto e di elencare i progetti.



Allo scopo di verificare che tutti i requisiti funzionali stabiliti in fase di analisi siano stati presi in considerazione **nella fase di progettazione**, è opportuno integrare la tabella dell'elenco dei requisiti aggiungendo una colonna in cui indicare quali classi e quali metodi implementano ogni singolo requisito funzionale.

Analisi dei requisiti

L'analisi dei requisiti del nostro software porta a individuare i seguenti requisiti:

N	Tipologia	Priorità	Definizione	Classe (metodo)
1	**Funzionale**	**MUST**	**L'utente crea un progetto definendone il nome e i partecipanti**	**ExpenseManager** creaProgetto () **Progetto** Progetto()
2	**Funzionale**	**MUST**	**L'utente elimina un progetto**	**ExpenseManager** eliminaProgetto()
3	**Funzionale**	**MUST**	**L'utente registra una nuova spesa per un progetto specificando la data, l'importo, il pagatore, i partecipanti alla spesa (può darsi che una spesa riguardi solo alcuni partecipanti al progetto)**	**Progetto** aggiungiSpesa() **Spesa** Spesa()
4	**Funzionale**	**MUST**	**L'utente elimina una spesa di un progetto**	**Progetto** eliminaSpesa()

5	**Funzionale**	**MUST**	**L'utente aggiorna una spesa modificandone i dati**	**Progetto** setSpesa() **Spesa()** setDescrizione() setData() setPagatore() setPartecipanti() setImporto()
6	**Funzionale**	**MUST**	**L'utente visualizza tutte le spese di un progetto**	**Progetto()** visualizzaSpese() **Spesa()** toString()
7	**Funzionale**	**MUST**	**L'utente registra un nuovo trasferimento per un progetto specificando la data, l'importo, il pagatore, il beneficiario (colui che riceve il denaro)**	**Progetto** aggiungiTrasferimento() **Trasferimento** Trasferimento()
8	**Funzionale**	**MUST**	**L'utente elimina un trasferimento di un progetto**	**Progetto** eliminaTrasferimento()
9	**Funzionale**	**MUST**	**L'utente aggiorna un trasferimento modificandone i dati**	**Progetto** setTrasferimento() **Trasferimento()** setDescrizione() setData() setPagatore() setBeneficiario() setImporto()
10	**Funzionale**	**MUST**	**L'utente visualizza tutti i trasferimenti di un progetto**	**Progetto()** visualizzaTrasferimenti() **Trasferiemento()** toString()
11	**Funzionale**	**MUST**	**L'utente visualizza tutte le spese effettuate da uno specifico pagatore con i partecipanti**	**Progetto()** visualizzaSpesePagatore() **Spese()** toString()
12	**Funzionale**	**MUST**	**L'utente visualizza tutte le spese con uno specifico partecipante e il relativo pagatore e gli altri partecipanti**	**Progetto()** visualizzaSpesePartecipante() **Spese()** toString()

13	**Funzionale**	**MUST**	**L'utente visualizza tutti i trasferimenti effettuati da uno specifico pagatore con il relativo beneficiario**	**Progetto()** visualizzaTrasferimentiPagatore() **Trasferimento()** toString()
14	**Funzionale**	**MUST**	**L'utente visualizza tutti i trasferimenti effettuati a favore di uno specifico beneficiario con il relativo pagatore**	**Progetto()** visualizzaTrasferimentiBeneficiario() **Spese()** toString()
15	**Funzionale**	**MUST**	**L'utente visualizza il bilancio di un progetto che evidenzia chi, fra i partecipanti, deve dare o ricevere denaro a quali altri partecipanti al progetto ("Chi deve pagare e chi deve ricevere denaro")**	Progetto() **visualizzaBilancio()**
16	**Funzionale**	**MUST**	**L'utente salva i dati su file**	**Progetto()** esportaSpeseSuFile() esportaTrasferimentiSuFile() **Spesa()** getIdSpesa() getDescrizione() getData() getPagatore() getPartecipanti() getImporto() **Trasferimento()** getIdTrasferimento() getDescrizione() getData() getPagatore() getBeneficiario() getImporto()
17	**Tecnologico**		**I dati vengono salvati su un file separato per ciascun progetto.**	
18	**Non Funzionale**	**MAY**	**L'applicazione salva i dati relativi alle attività anche in caso di interruzione anomala del funzionamento (con salvataggi periodici ogni volta che vengono modificati i dati)**	

19	**Funzionale**	**SHOULD**	**L'utente può esportare i dati visualizzati in formato CSV**	**Progetto()** esportaSpesePagatoreSuFile() esportaSpesePartecipanteSuFile() esportaTrasferimentiPagatoreSuFile() esportaTrasferimentibeneficiarioSuFile() esportaBilancioSuFile() **Spesa()** getIdSpesa() getDescrizione() getData() getPagatore() getPartecipanti() getImporto() **Trasferimento()** getIdTrasferimento() getDescrizione() getData() getPagatore() getBeneficiario() getImporto()
20	**Funzionale**	**MAY**	**L'utente può esportare i dati visualizzati in formato HTML**	**Progetto()** esportaSpesePagatoreSuFile() esportaSpesePartecipanteSuFile() esportaTrasferimentiPagatoreSuFile() esportaTrasferimentibeneficiarioSuFile() esportaBilancioSuFile() **Spesa()** getIdSpesa() getDescrizione() getData() getPagatore() getPartecipanti() getImporto() **Trasferimento()** getIdTrasferimento() getDescrizione() getData() getPagatore() getBeneficiario() getImporto()

21	**Tecnologico**	**MUST**	**L'applicazione deve essere eseguibile sui PC con SO Windows e Linux**	