Unified Modeling Language

Perché modellare il software?

Produrre software non è affatto facile: anche i sistemi apparentemente semplici tendono a raggiungere un notevole grado di complessità.

MODELLO = semplificazione della realtà

UML è stato progettato per agevolare il processo di elaborazione di modelli:

- ✓ visualizzare il sistema
- ✓ specificarne la struttura ed il comportamento
- ✓ implementarlo
- ✓ documentarne tutte le decisioni prese durante le varie fasi di lavoro

Perché modellare il software?

VISUALIZZAZIONE

UML è stato progettato per agevolare la comunicazione fra gli attori che partecipano allo sviluppo di un progetto software.

La maggior parte delle caratteristiche di un sistema si presta meglio per essere rappresentata in modo grafico piuttosto che con un testo.

UML definisce un insieme di diagrammi standard attraverso i quali ogni membro del processo di sviluppo ha la possibilità di comprendere lo stato di sviluppo del sistema, in maniera chiara ed univoca, riducendo il rischio di interpretazioni non corrette.

Perché modellare il software?

SPECIFICA

Fornire la specifica di un modello = definirlo in modo preciso, completo e privo di ambiguità.

I modelli costruiti nelle prime fasi del progetto servono a definire i requisiti dei committenti.

Addentrandosi nell'analisi è necessario arricchire i modelli iniziali per definire il sistema con maggior grado di dettaglio. Questi modelli intermedi definiscono i concetti chiave del sistema ed i meccanismi con i quali tali concetti verranno poi espressi.

I modelli più dettagliati descrivono completamente le caratteristiche del progetto.

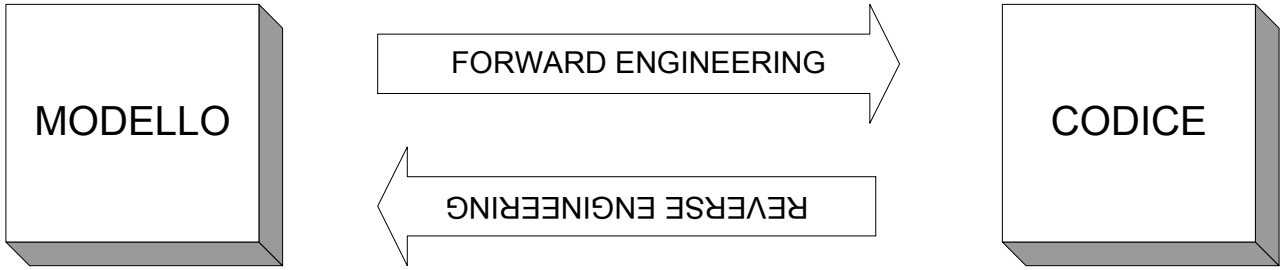
Perché modellare il software?

COSTRUZIONE

Fine ultimo di un progetto = implementazione di codice

Molti elementi di UML sono in relazione diretta con i costrutti propri dei linguaggi di programmazione ad oggetti (C++, Java).

Altri elementi di UML agevolano la progettazione di database o la progettazione di sistemi distribuiti.



Perché modellare il software?

DOCUMENTAZIONE

UML facilita il processo di stesura di documentazione di progetto.

Alcuni diagrammi costituiscono una buona base per la stesura di manuali utente

Altri diagrammi agevolano la definizione dei test del sistema

Chi non è stato coinvolto nel processo di modellazione ha la possibilità di comprendere i concetti fondamentali del sistema.

I modelli ben documentati possono essere riutilizzati, in tutto o in parte, in progetti successivi.

Un buon modello software

Cosa rende buono il modello di un software?

- Un buon modello coglie i concetti essenziali del sistema (aiuta ad andare al "cuore" del progetto)
- Un buon modello permette ai vari attori di cogliere dettagli diversi in momenti diversi
- Un buon modello, per quanto astratto, è collegato alla realtà e ne segue la sua evoluzione
- Un buon modello si presta ad agire ed interoperare con altri modelli al fine di illustrare sistemi complessi nella loro interezza.

UML: un po' di storia

1989: la modellazione software con metodologie O-O si impone all'attenzione della comunità scientifica

1991: *Jim Rumbaugh, "Object Oriented Modeling and Design" (Prentice Hall).* Definisce la Object Modeling Tecnique (OMT), primo esempio di analisi nello spazio dei problemi. Molti degli attuali diagrammi UML derivano da concetti ivi espressi.

1992: *Ivar Jacobson, "Object Oriented Software Engineering" (Addison-Wesley)* detto "libro bianco" o OOSE. Definisce il processo Objectory e per la prima volta parla di Use Cases, uno dei fondamenti di UML.

1994: *Grady Booch, "Object Oriented Analysis and Design with Applications" (Addison-Wesley).* Definisce in maniera molto rigorosa ai dettagli della costruzione di un sistema software con metodologia O-O ("metodo di Booch"). I diagrammi UML più dettagliati e più vicini alle fasi di sviluppo sono stati definiti in questo testo.

UML: un po' di storia

Fine 1994: Rumbaugh raggiunge Booch alla Rational

1995: Rational acquista la società di Jacobson

1995: nasce la versione 1.0 di UML, che viene sottoposta all'Object Management Group (OMG), ente preposto alla definizione degli standard in molte branche dell'informatica

1997: *Booch, Rumbaugh, Jacobson, UML User Guide (Addison-Wesley).*
UML 1.1 diventa il linguaggio standard per la modellazione O-O.
Gli autori vengono spesso indicati con il nickname "I tre amigos".

Oggi: UML 1.5 - è in fase di definizione UML 2.0

(ma la versione universalmente utilizzata come standard è UML 1.3)

UML sul web

Informazioni su UML possono essere reperite sul web agli indirizzi:

www.rational.com/uml

www.omg.org/uml

Definizione di UML

OMG: *Unified Modeling Language (UML) è una notazione per analizzare, specificare, visualizzare e documentare lo sviluppo dei documenti di progetto di un sistema ad oggetti.*

Più precisamente, UML è un linguaggio con specifica semantica semiformale che include sintassi astratta, rigorose regole grammaticali e semantica dinamica.

In sintesi, UML è una notazione associata ad un metamodello (ontologia) che descrive la notazione stessa.

Obiettivi di UML

Booch, Rumbaugh, Jacobson si posero precisi obiettivi nella definizione di UML, espressi nella "dichiarazione di intenti" di UML.

1. Fornire agli utenti un linguaggio di modellazione visuale, pronto all'uso ed espressivo, che permetta loro di sviluppare e di scambiarsi modelli significativi.

2. Offrire meccanismi di estensibilità e di specializzazione per estendere i concetti fondamentali

3. Essere indipendenti da particolari linguaggi di programmazione e processi di sviluppo

4. Offrire una base formale per comprendere il linguaggio di modellazione

5. Incoraggiare la crescita del mercato degli strumenti di modellazione O-O

Obiettivi di UML

6. Supportare concetti di sviluppo di livello più alto quali collaborazioni, framework, pattern e componenti

7. Integrare i processi migliori. UML integra le *best practices* in processi che abbracciano livelli di astrazione, domini, architettura, stadi del ciclo di vita, tenologie di implementazione.

Obiettivi di UML

Anche se non inizialmente previsti dagli autori, UML coglie anche altri obiettivi:

✓ Inclusione di una notazione UML più semplice nell'ambito di un linguaggio UML più completo

✓ Utilità in diverse prospettive di modellazione
Cook, Daniels, *Designing Object Systems*:

- Prospettiva essenziale, che modella concetti relativi all'azienda, senza alcun riferimento alla tecnologia di implementazione
- Prospettiva di specifica, che modella le caratteristiche del software per soddisfare i requisiti di una soluzione
- Prospettiva di implementazione, che modella la costruzione del software

✓ Corrispondenza con il codice, abbastanza stretta da consentire la reingegnerizzazione

Obiettivi di UML

- ✓ Integrazione con strumenti CASE
  - Rational Rose
  - Microsoft Visual Modeler
  - etc.

Punti di vista su un sistema

Ognuno degli attori che partecipano al processo di sviluppo software ha in mente obiettivi diversi e guarda al sistema da angolature differenti.
UML definisce 5 punti di vista sull'architettura di un sistema:

- Il punto di vista dei casi d'uso

Coinvolge utenti e sistemi esterni: descrive cosa farà il sistema senza specificare come lo farà.

- Il punto di vista del progetto

Descrive gli elementi strutturali, dinamici e comportamentali del sistema.

- Il punto di vista del processo

Descrive gli aspetti legati al flusso di controllo del sistema ed alla sua temporizzazione

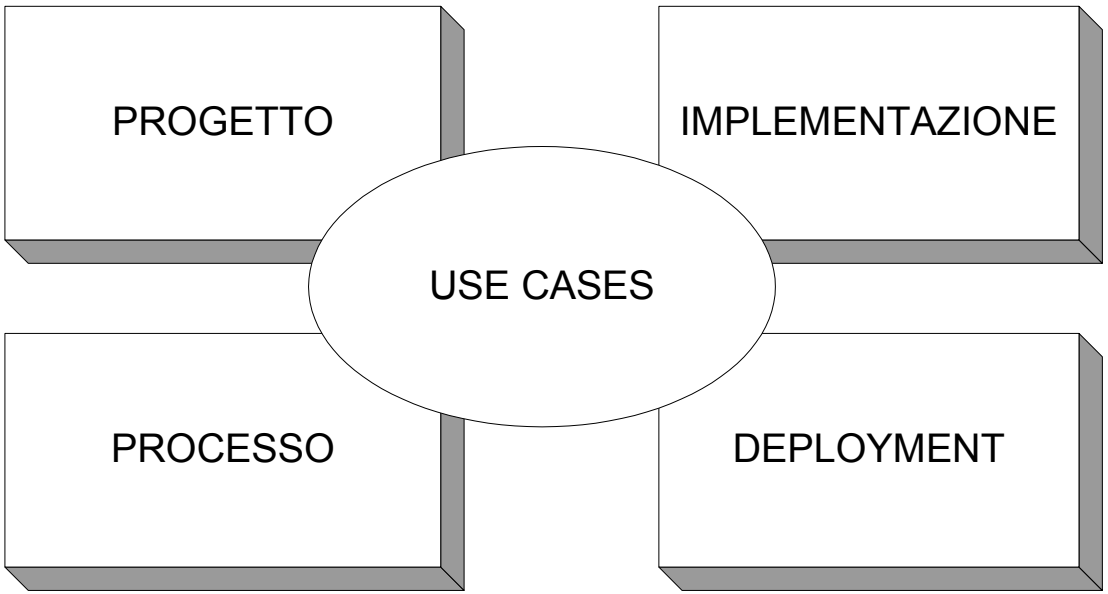
- Il punto di vista dell'implementazione

Descrive i vari componenti del sistema, prodotti dai vari attori del processo, che devono essere assemblati insieme al fine di implementare la soluzione software

Punti di vista su un sistema

- Il punto di vista del deployment

Descrive la topologia del sistema: si focalizza sulla distribuzione geografica dei vari componenti hw e sw.



Unified Process

UML è stato progettato in modo da essere indipendente dal processo di sviluppo utilizzato. Tuttavia gli ideatori di UML sono soliti fare riferimento ad un processo di sviluppo:

- Guidato dai casi d'uso (*Use cases*)
- Basato sull'architettura del sistema
- Iterativo ed incrementale

Booch, Rumbaugh, Jacobson propongono un processo di sviluppo che va sotto il nome di Rational Unified Process (RUP).

RUP prevede quattro fasi per il processo di sviluppo.

Unified Process - Avviamento

La fase di *Avviamento* definita nel RUP ha come principale scopo stabilire la fattibilità del sistema.

Le attività del gruppo di sviluppo in questa fase prevedono:

- Definizione dell'*estensione* del progetto: cosa viene incluso e cosa rimane fuori
- Definizione dell'*architettura candidata*
- Identificazione delle criticità e dei rischi principali
- Stima iniziale di costo, risorse necessarie, tempi di realizzazione, qualità del prodotto (*business plan*).

La fase di Avviamento termina con un punto denominato *Obiettivi del ciclo di vita*.

Unified Process - Elaborazione

La fase di *Elaborazione* definita nel RUP ha come principale scopo concretizzare la possibilità di realizzare il sistema alla luce dei vincoli imposti al progetto: vincoli finanziari, di tempo, legati alle risorse, etc.

Le attività del gruppo di sviluppo in questa fase prevedono:

- Raffinamento dei requisiti funzionali
- Espansione dell'architettura candidata in una architettura di base completa
- Gestione continuativa dei rischi di progetto
- Raffinamento del business plan

La fase di Elaborazione termina con un punto denominato *Architettura del ciclo di vita*.

Unified Process - Costruzione

La fase di *Costruzione* definita nel RUP ha come principale scopo la realizzazione di un sistema in grado di funzionare correttamente per un numero ristretto di utenti (detti *beta testers*).

L' attività del gruppo di sviluppo in questa fase prevede l'implementazione iterativa e incrementale del sistema, assicurandosi che:

- Il sistema rimanga sempre fattibile
- Non vengano violati i vincoli di progetto
- Ci si mantenga aderenti al business plan.

La fase di Costruzione termina con un punto denominato *Capacità iniziale di operazione*.

Unified Process - Transizione

La fase di *Transizione* definita nel RUP prevede l'effettiva consegna del sistema ai clienti.

L' attività del gruppo di sviluppo in questa fase si focalizza sulla correzione di difetti (*bug fixing*).

La fase di Transizione termina con il *Rilascio del prodotto*.

I diagrammi UML

- ✓ Diagramma delle classi (*Class*)
- ✓ Diagramma degli oggetti (*Object*)
- ✓ Diagramma dei casi di uso (*Use case*)
- ✓ Diagrammi di interazione (*Interaction*)
  - ◆ Diagramma di sequenza (*Sequence*)
  - ◆ Diagramma di collaborazione (*Collaboration*)
- ✓ Diagramma delle attività (*Activity*)
- ✓ Diagramma degli stati (*Statechart*)
- ✓ Diagramma dei componenti (*Component*)
- ✓ Diagramma della distribuzione dei componenti (*Deployment*)

Oggetti

Oggetto = rappresentazione di un elemento del problem space

Gli oggetti sono rappresentazioni di oggetti fisici o concetti logici esistenti nel mondo reale.

Esempio: sportello Bancomat

- *Carta Bancomat*
- *Conto corrente*
- *Carta moneta*
- *Scontrino*

Potremmo anche definire l'oggetto *Sportello Bancomat*...

Oggetti

Proprietà degli oggetti

Gli oggetti sono dotati di:

- *Identità* - Espressa da un nome comprensibile
- *Stato* - Include i nomi degli *attributi*, ossia delle varie proprietà che descrivono l'oggetto ed i valori di tali attributi
- *Comportamento* - Rappresentato da funzioni, dette *metodi*, che utilizzano o cambiano il valore degli attributi dell'oggetto

Concetto fondamentale OOP: un oggetto nasconde i suoi dati al resto del mondo e permette agli oggetti esterni di manipolarli solo tramite il ricorso ai suoi dati (*Encapsulation*)

Oggetti

Gli oggetti possono eseguire funzioni su se stessi: *Metodi*

Esempio: l'oggetto ContoCorrente ha come attributi

- Numero
- LimitePrelievo

E come metodi

- Preleva()
- ...

Classi

Classe = Nome collettivo di tutti gli oggetti che hanno gli stessi metodi e variabili di istanza.

Una classe è un insieme di oggetti aventi le stessa caratteristiche. Essendo una aggregazione di oggetti, confrontiamone le caratteristiche fondamentali:

- *Identità* - Come gli oggetti, una classe ha una identità espressa da un nome comprensibile ed unico all'interno del contesto di riferimento
- *Stato* - Una classe non ha uno stato, ma definisce gli attributi degli oggetti appartenenti alla classe stessa
- *Comportamento* - Una classe definisce il proprio comportamento in termini di *operazioni*, cioè di servizi che possono essere richiesti alla classe stessa. Ogni operazione di una classe è rappresentata da almeno un metodo della classe.

Un oggetto appartenente ad una classe viene detto **Istanza** della classe.

Classi

Notazione standard: *box* suddiviso in tre sezioni, contenenti
- Sezione superiore: nome della classe
- Sezione centrale: attributi
- Sezione inferiore: operazioni.

Esempio: classe ContoCorrente

Nota: è possibile omettere una o più sezioni dal box di una classe.

Classi

Alcuni “consigli” dagli autori di UML:

Each class should map to some tangible or conceptual abstraction in the domain of the end user or the implementor.
A well structured class:
- provides a crisp abstraction of something from the problem domain or the solution domain
- embodies a small, well-defined set of responsibilities and carries them out all very well
- provides a clear separation of the abstraction's specification and its implementation
- is understandable and simple, yet extensible and adaptable

Booch, Rumbaugh, Jacobson, “UML User Guide”

Relazioni tra Classi

Le relazioni fra le classi costituiscono la struttura portante del sistema in fase di progettazione. UML è progettato in modo da avere focalizzazione non tanto sulla struttura interna delle classi che costituiscono il sistema, quanto piuttosto sulle relazioni che intercorrono fra le classi del sistema.

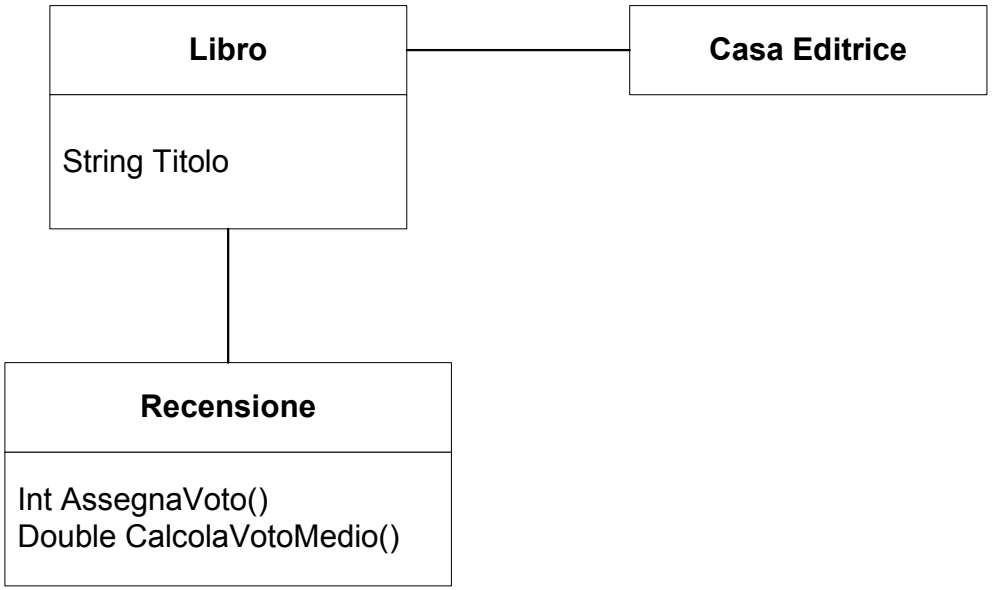
I tipi di relazioni tra classi sono:

- *Associazione*
- *Aggregazione*
- *Generalizzazione*

Relazioni tra Classi - Associazione

Una *Associazione* è una connessione strutturale fra classi.
Viene rappresentata da un semplice segmento che collega i box delle classi.

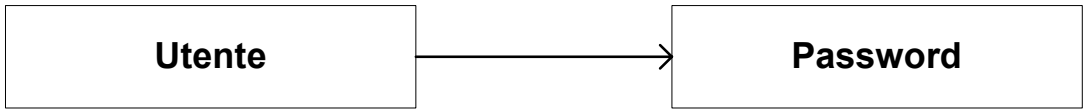
Esempio: On-line Bookshop



Relazioni tra Classi - Associazione

Nell'esempio precedente, si assume che le relazioni fra le classi siano bidirezionali: significa che ogni classe *è a conoscenza dell'esistenza dell'altra.*

E' anche possibile definire una associazione monodirezionale fra classi:



Relazioni tra Classi - Associazione

Ad una associazione fra classi è possibile associare vari tipi di dettagli, detti *Adornments.*

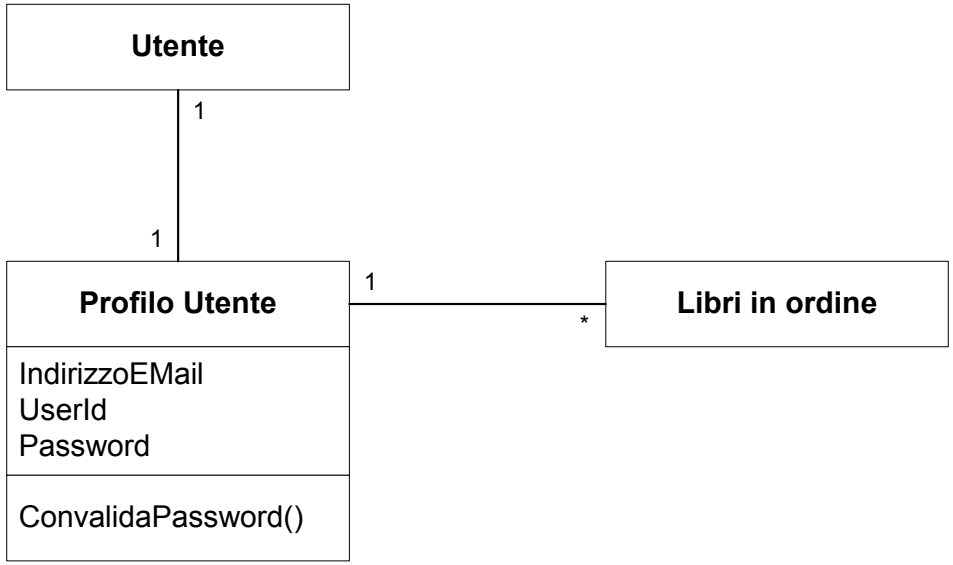
- Natura dell'associazione

Esempio: classe Autore ha scritto Libro

- Ruoli - rappresentano l'*interfaccia* che la classe mostra alle altre. All'interno di relazioni diverse, una classe può interpretare lo stasso ruolo oppure ruoli differenti.

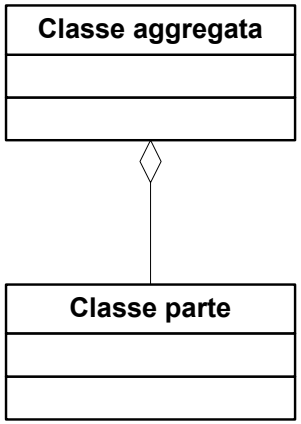
Relazioni tra Classi - Associazione

- Molteplicità dell'associazione - indica quanti oggetti di una classe possono essere presenti all'interno dell'associazione. E' rappresentata da un numero indicante la cardinalità, riportato vicino al box della classe.



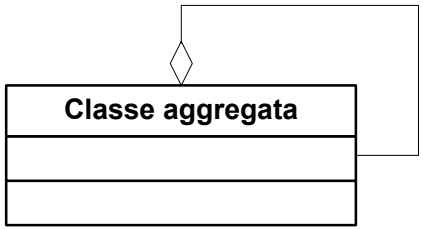
Relazioni tra Classi - Aggregazione

Una *Aggregazione* è un particolare tipo di associazione: una relazione "tutto/parte" nella quale una classe è "parte" di un'altra classe.
Graficamente viene rappresentata mediante un sgmento terminato con un rombo vuoto: la classe associata al rombo è la classe "intera" la classe all'altro estremo è la sua "parte".



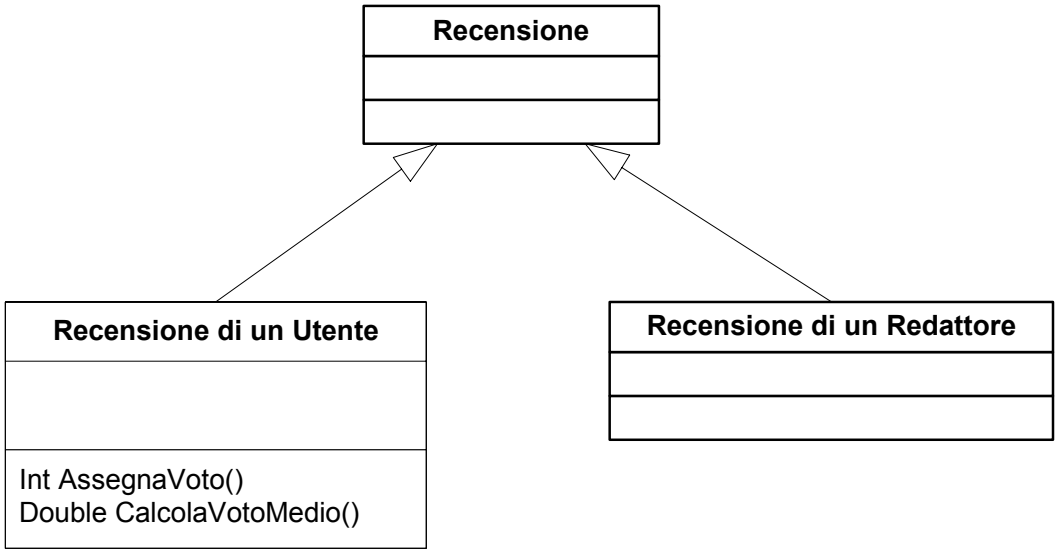
Relazioni tra Classi - Aggregazione

Una classe può anche essere costituita da una aggregazione di se stessa.
Il costrutto di auto-aggregazione è utile, ad es., in situazioni che richiedono una serie di somme successive per ottenere il totale finale.



Relazioni tra Classi - Generalizzazione

La *Generalizzazione* è una relazione fra una classe generale (detta *superclasse*) e le versioni specifiche di quella stessa classe (dette *sottoclassi*).
La generalizzazione rappresenta graficamente uno dei concetti chiave di OOP: l'ereditarietà. Una sottoclasse eredita gli attributi e le operazioni della superclasse.



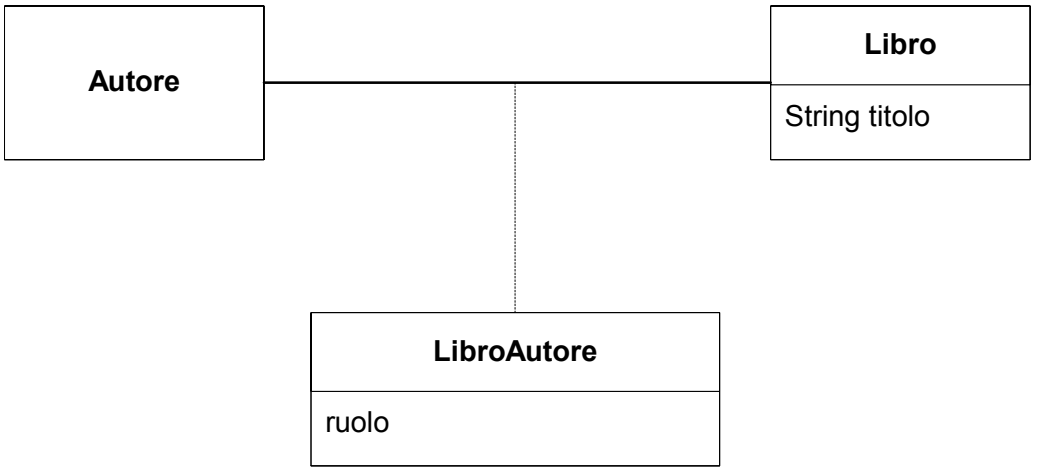
Relazioni tra Classi - Generalizzazione

Notare come nel diagramma precedente i metodi AssegnaVoto e CalcolaVotoMedio, in precedenza definiti nella classe Recensione, siano stati spostati nella classe derivata Recensione di un Utente, in quanto non si applicano alla classe Recensione di un Redattore.
La sottoclasse Recensione di un Utente rappresenta un esempio di *generalizzazione* della classe padre Recensione.

Relazioni tra Classi - Classi di Associazione

Una *Classe di Associazione* viene utilizzata per modellare le associazioni complesse che, al di là delle classi che collegano, presentano caratteristiche peculiari.
Utile nel caso di relazioni molti-a-molti (che andrebbero frammentate in più relazioni uno-a-molti).
Viene rappresentata graficamente mediante un segmento tratteggiato che collega una associazione ad una classe.

Relazioni tra Classi - Classi di Associazione



Esempio: la relazione Autore-Libro è molti-a-molti (ogni Autore può scrivere più di un Libro e ogni Libro può avere più di un Autore).
La classe di associarione LibroAutore mette in relazione un Autore con un Libro specificando il Euolo di quella relazione: l'Autore può essere l'autore principale, un collaboratore, l'editor, il traduttore, etc.

Relazioni tra Classi

Alcuni “consigli” dagli autori di UML:

- Use dependencies only when the relationship is not structural
- Use generalization only when you have an "is-a-kind-of” relationship
- Beware of introducing cyclic generalization relationships
- Use associations primarily where there are structural relationships among objects
- Keep your generalization relationships generally balanced; neither too deep nor too wide

Booch, Rumbaugh, Jacobson, “UML User Guide”

Diagramma delle Classi

Un *Diagramma delle Classi* rappresenta le classi del sistema in via di sviluppo con le varie relazioni che le coinvolgono. Costituisce lo strumento principale per rappresentare graficamente ed illustrare la struttura del sistema.

Il diagramma delle classi nella pagina successiva comprende la maggior parte delle classi e delle relazioni del sistema On-Line Bookshop.

Notare che il diagramma *non* mostra tutte le associazioni logiche presenti nel modello: ad esempio, dovrebbe esserci una relazione uno-a-molti fra Utente ed Ordine, ed anche una relazione fra Utente e Recensore.

Diagramma delle Classi

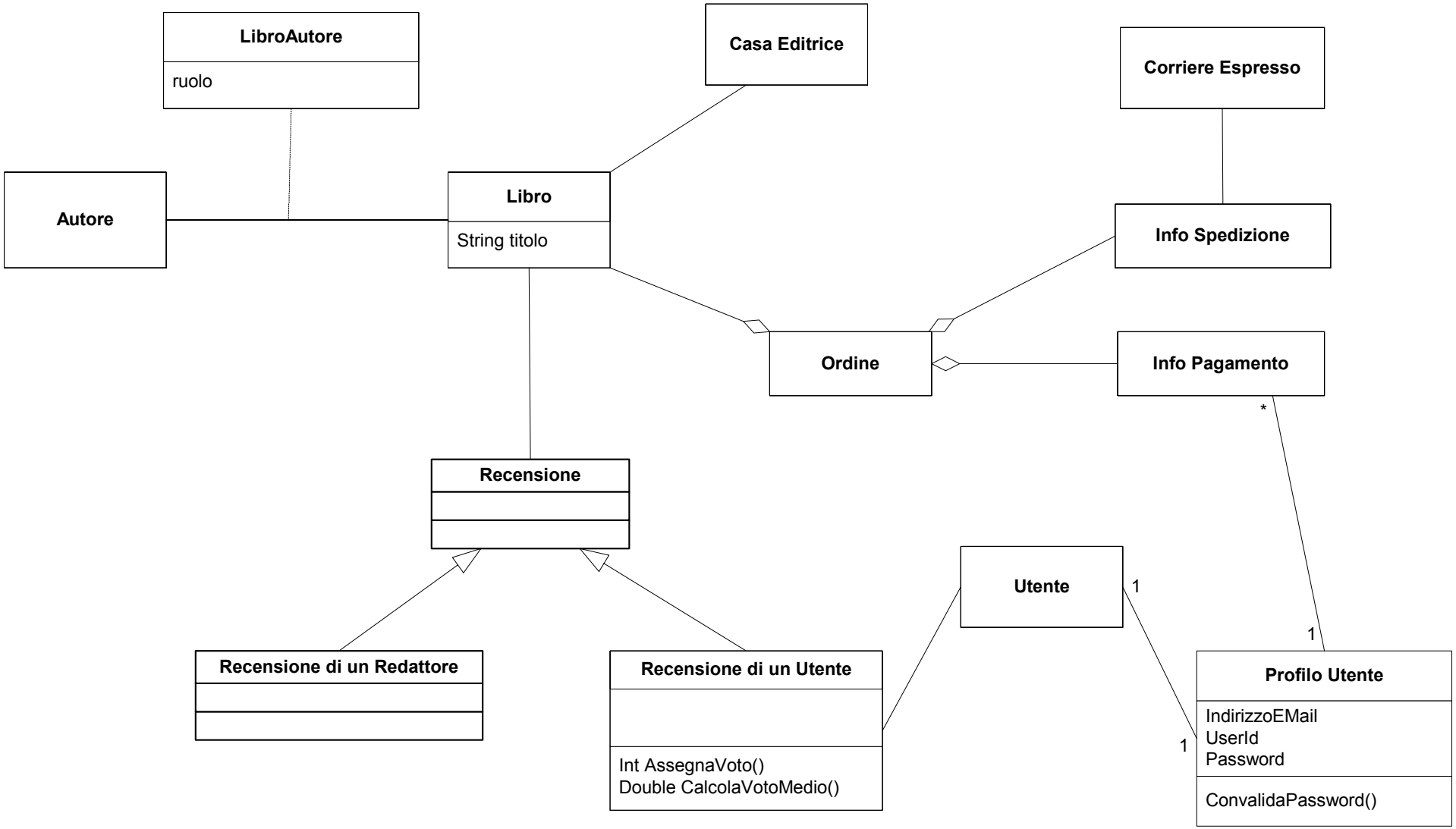
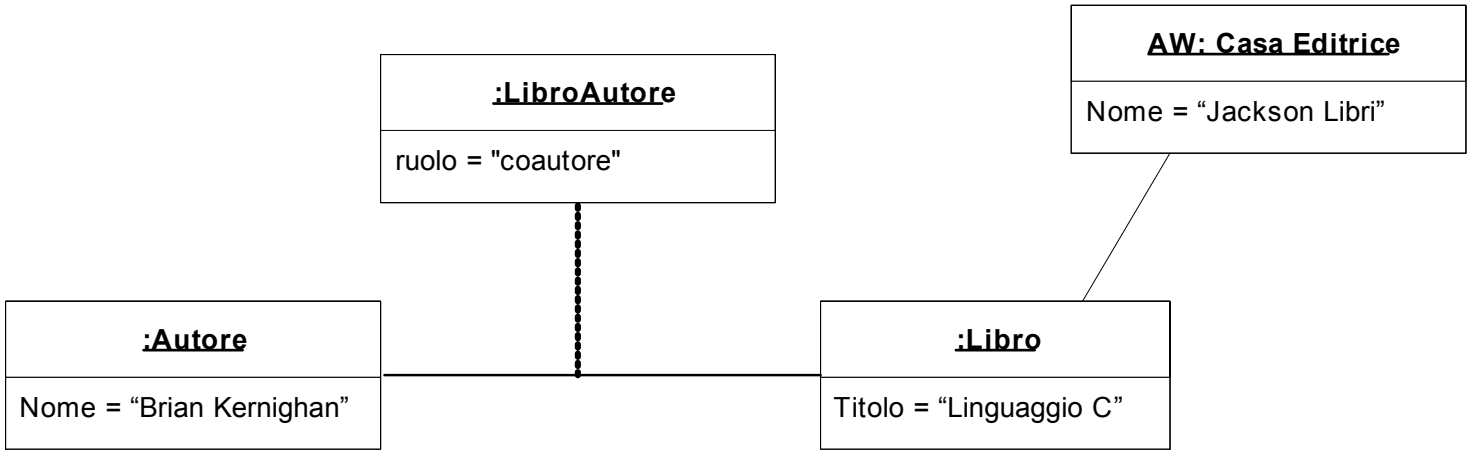


Diagramma degli Oggetti

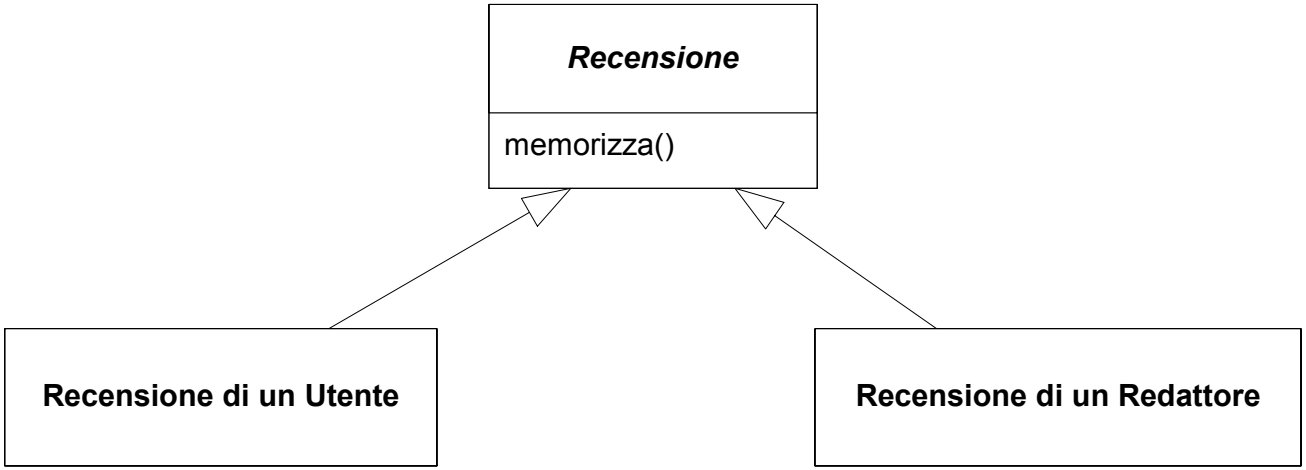
I *Diagrammi degli Oggetti* hanno la stessa notazione UML dei diagrammi delle classi, ma con alcune differenze:

- Nella sezione superiore del box, il nome della classe a cui l'oggetto appartiene viene specificato dopo un segno di due punti; se necessario, il nome proprio dell'oggetto (istanza) viene rappresentato prima dei due punti.
- Il nome dell'oggetto è sottolineato
- Ogni attributo di una classe ha un valore specifico per ogni oggetto appartenente a quella classe (istanza).



Classi astratte

Si definisce *Classe astratta* una classe che non può avere istanze.
Nei diagrammi UML, una classe astratta viene rappresentata scrivendone il nome in corsivo.
E' una estensione del concetto di generalizzazione; la superclasse è completamente generica, dal momento che non contiene l'implementazione di nessun metodo: l'implementazione è a carico delle classi derivate.



Classi astratte

Nota bene: le classi derivate devono implementare i metodi definiti nella classe astratta!
E' uno dei concetti alla base del meccanismo di ereditarietà.

Dipendenza fra classi

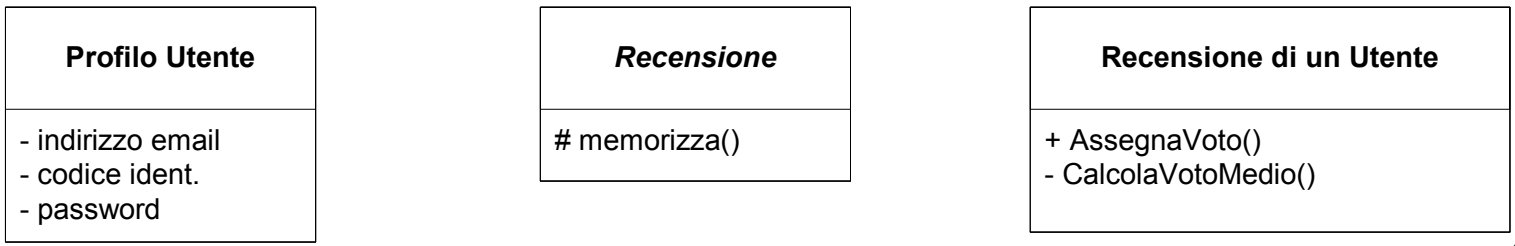
Si ha relazione di *Dipendenza* fra due classi quando il cambiamento in una classe implica il cambiamento nell'altra.
Nella notazione UML viene rappresentata con una freccia tratteggiata.

Visibilità

Il concetto O-O di encapsulation prevede che gli oggetti siano organizzati in modo tale che essi siano descritti insieme alle relative operazioni.
Esistono più livelli di accesso agli elementi interni di una classe.

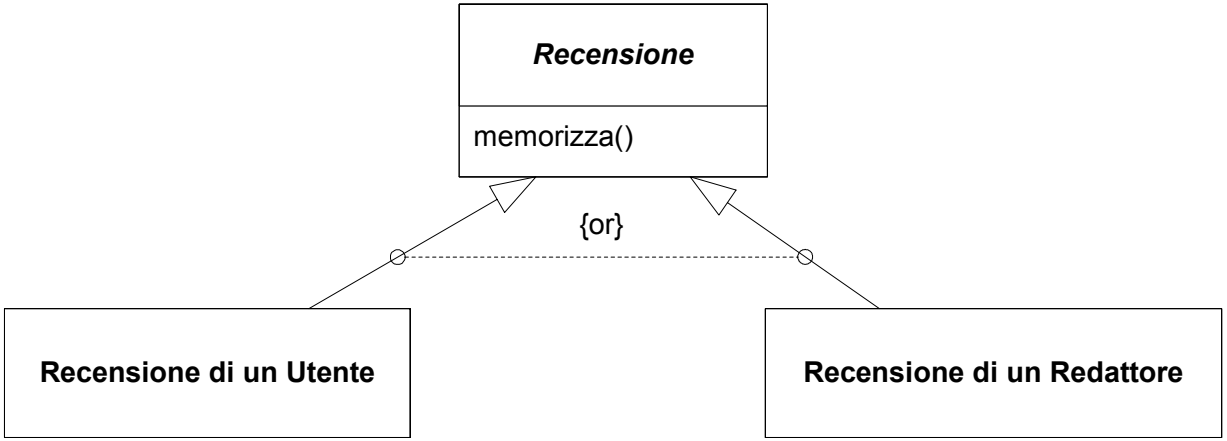
Data una classe, UML supporta tre livelli di visibilità:

- *Visibilità pubblica* (segno **+**): un oggetto di qualsiasi altra classe può utilizzare l'elemento della classe data
- *Visibilità protetta* (segno **#**): solo gli oggetti appartenenti alla classe data o ad una sottoclasse possono utilizzare l'elemento della classe data
- *Visibilità privata* (segno **-**): solo gli oggetti appartenenti alla classe data possono utilizzare l'elemento



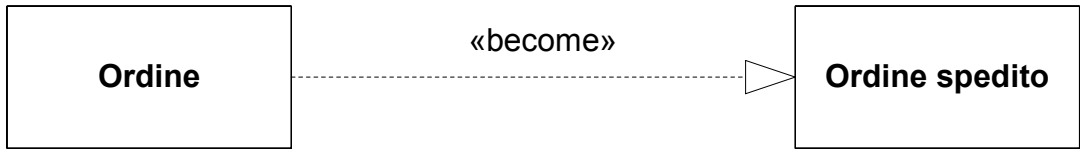
Vincoli

Un *Vincolo* è un costrutto UML utilizzato per estendere la semantica di un elemento del modello.
I vincoli vengono utilizzati nei diagrammi UML per specificare delle condizioni che devono essere sempre soddisfatte da un elemento del modello.
I vincoli possono essere applicati a attributi di una classe o a relazioni fra classi. Sono rappresentati ponendo un'espressione fra parentesi graffe.

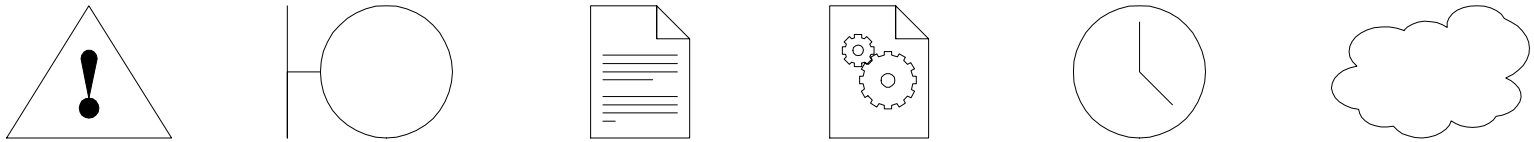


Stereotipi

Uno *Stereotipo* rappresenta una estensione al vocabolario di base di UML, al fine di rappresentare costrutti di modellazione non definiti in UML, ma simili ad elementi UML già esistenti.
Uno stereotipo è indicato con un nome tra doppie virgolette. UML fornisce circa 40 stereotipi, ma è possibile definire stereotipi personalizzati.

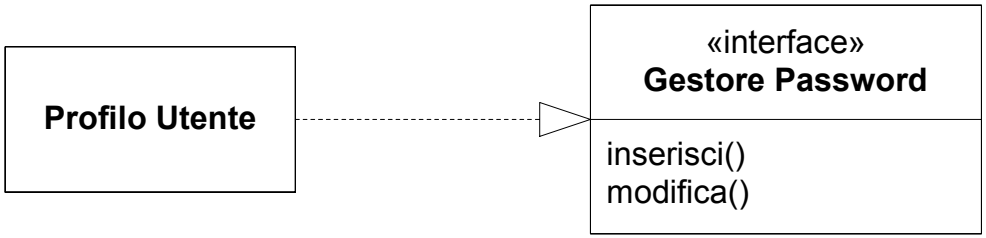


UML fornisce anche delle icone per rappresentare alcuni stereotipi:

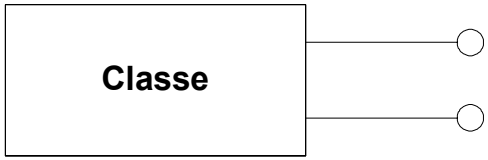


Stereotipi - Interfacce

Uno dei concetti fondamentali di OOP prevede la separazione di interfaccia ed implementazione di una classe.
In UML il concetto viene rappresentato utilizzando lo stereotipo *interface*:



UML definisce anche una notazione alternativa detta "lollipop":



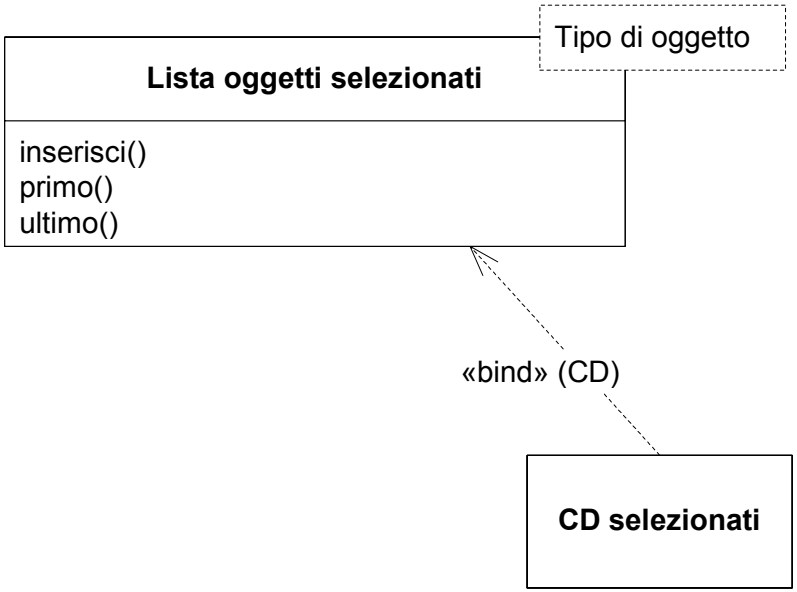
Stereotipi - Template

Un *Template* di classi è un costrutto che rappresenta una famiglia di potenziali classi. Un template presenta un insieme di parametri formali; è possibile definire una classe della famiglia legando al template un set di tali parametri formali.
Si tratta di un concetto derivato direttamente dal C++, che definisce la keyword template ed usa estensivamente tale concetto nella Standard Template Library (STL).

La notazione UML prevede tre elementi per definire i template:
• Il box di classe ha un rettangolo tratteggiato in alto a dx, contenente i parametri formali del template
• La classe creata a partire dal template (detta *classe legata*) viene raffigurata con il consueto box
• Una relazione di dipendenza con stereotipo <<bind>> e indicazione dei valori dei parametri formali parte dalla classe legata e termina nella classe template.

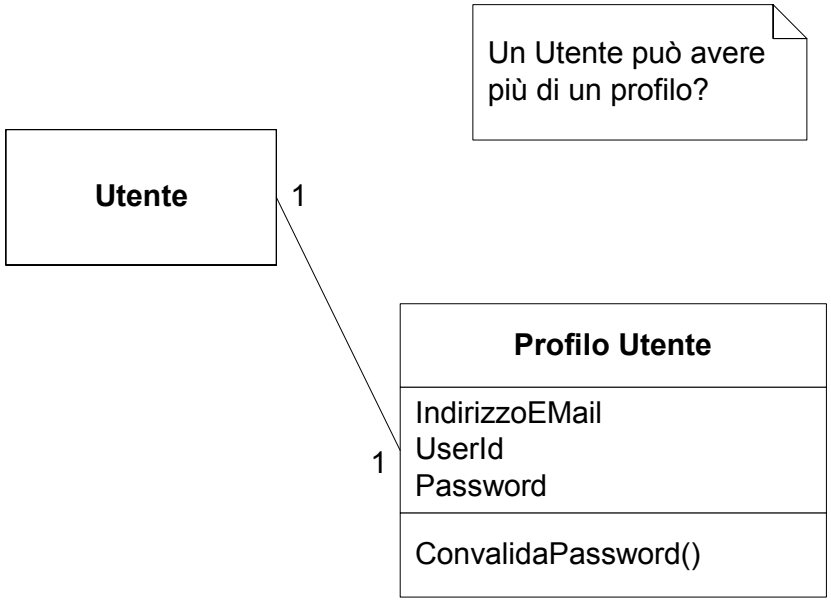
Stereotipi - Template

Nell'esempio della On-line Bookshop:



Note

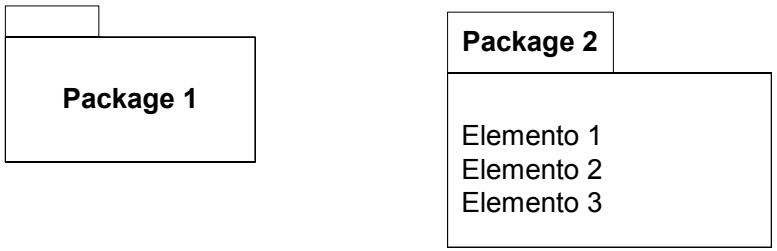
Un ulteriore *Adornment* dei diagrammi UML è costituito dalle *Note*.
Le note in UML sono rappresentate con box di testo con l'angolo superiore dx ripiegato. Possono comparire in qualsiasi diagramma e non hanno alcun impatto sul sistema in via di sviluppo.



Package

Un *Package* è un raggruppamento di elementi che fanno parte del modello.
La notazione UML per i package prevede un box dotato di tag.

Esistono due notazioni alternative:



Package

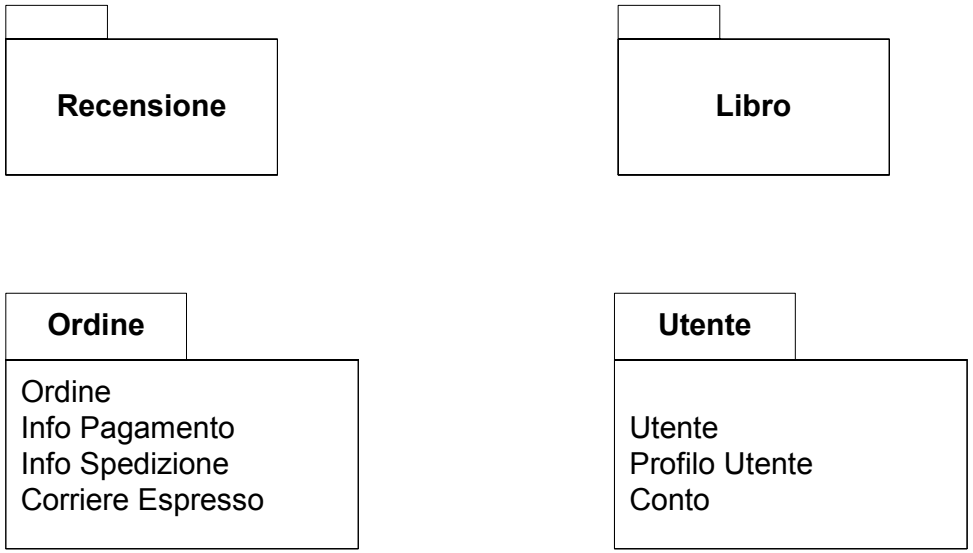
I package servono a raggruppare concettualmente elementi correlati del modello. Sono molto utili per suddividere il modello in sottosistemi, anche al fine di suddividere il lavoro su più team di sviluppo.

Un package può contenere elementi diversi: solo classi, classi e diagrammi, … Unica regola: un elemento di un modello UML può appartenere ad un solo package.

Nota: un modello UML può essere visto come un package contenente altri package!

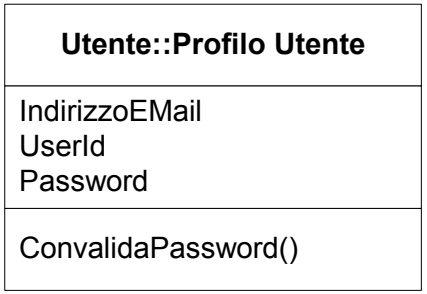
Package

Modello On-line Bookshop rappresentato con package:



Package

Nota: in qualche caso il diagrama delle classi può riportare lo *scope* del package a cui la classe appartiene.
In questo caso il nome completo della classe viene detto *Path Name* della classe.



Use Cases

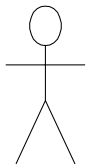
Il processo di identificazione dei requisiti permette ai committenti del progetto di accordarsi con progettisti sulle funzionalità del sistema in via di sviluppo.

Gli *Use Cases* rappresentano la specifica UML per l'identificazione dei requisiti del sistema.

Definizione di Jacobson: "sequenza comportamentalmente correlata di transazioni in un dialogo con un sistema"

Use Cases - Attori

Gli Use Cases prevedono dei ruoli interpretati dagli *Attori* del sistema.



Attore

Un Attore dunque rappresenta:

- Un ruolo interpretato da un utente nell'interazione con il sistema, oppure
- Un'entità, come ad es. un altro sistema o un database, all'esterno del sistema stesso.

Use Cases

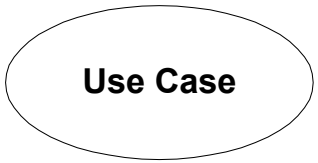
Un utente può ricoprire più ruoli di attore; in altri termini, può prendere parte a più use cases.

Definizione: *Use Case* = sequenza di azioni compiute da un attore all'interno del sistema al fine di ottenere un determinato scopo.

In altre parole, uno use case definisce cosa farà il sistema in conseguenza a determinate operazioni dell'utente, senza specificare come lo farà.

L'insieme degli attori di un sistema include qualsiasi entità che scambi informazioni con il sistema stesso.

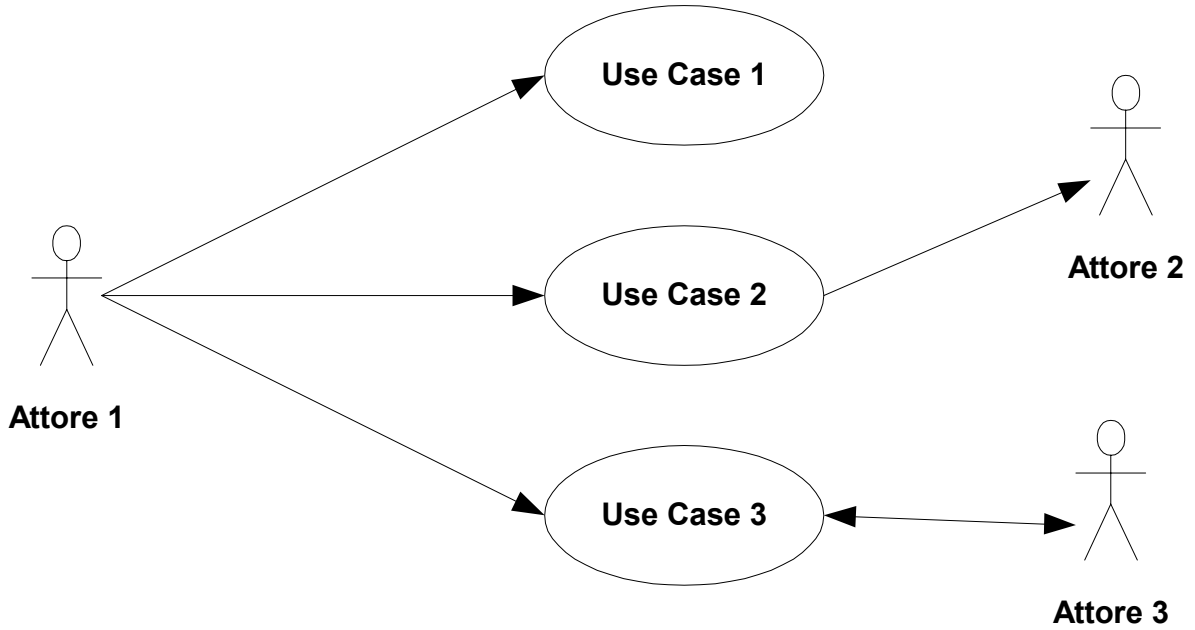
L'insieme di tutti gli use cases di un sistema esprime i requisiti funzionali espressi dal committente del progetto.



Diagrammi di Use Cases

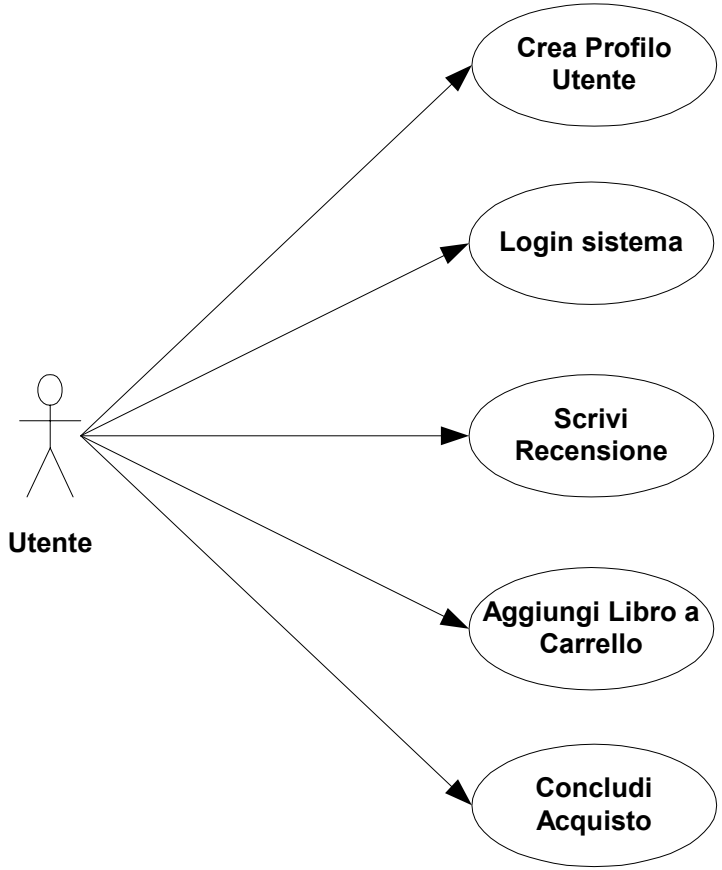
Per convenzione, nei diagrammi di use cases:

- L'attore che esegue un determinato caso d'uso viene raffigurato sul lato sx del diagramma
- Lo use case viene posto al centro del diagramma
- Altri attori coinvolti nello use case vengono posti sul lato dx



Diagrammi di Use Cases

Esempio On-line Bookshop



Flussi di Eventi

La descrizione di uno use case prevede la definizione di uno o più *Flussi di eventi*, che rappresentano la sequenza di azioni dell'attore e risposte del sistema a tali azioni, che si susseguono durante il processo di interazione fra l'utente ed il sistema.

In ogni sistema si delineano due tipologie di flussi di eventi:

- Il *Flusso di eventi Principale* (o *Corso d'azione Base*) descrive il fluire del processo di interazione che si verifica in circostanze normali, ossia in mancanza di errori da parte degli attori o del sistema.
Ogni use case deve avere un flusso di eventi principale.

- Il *Flusso di eventi Eccezionale* (o *Corso d'azione Alternativo*) è un percorso attraverso lo use case che descrive una condizione di errore, o quantomeno una situazione "limite" che si verifica raramente.
Normalmente ogni use case ha più flussi di eventi eccezionali (nei quali si esplicitano i comportamenti più interessanti del sistema…).

Flussi di Eventi - Esempio

Esempio: use case **Login** nel modello della On-line Bookshop.

L'Utente clicca sul pulsante Login nella Home Page.
Il sistema visualizza la pagina di accesso al sistema.
L'utente digita il suo codice identificativo e la sua password, poi clicca OK.
Il sistema convalida i dati inseriti dall'utente confrontandoli con quelli presenti nel Profilo Utente, e in caso positivo fa tornare l'utente alla Home Page.

Esercizio: nella On-line Bookshop, definire

✓ use case **Scrivi Recensione Utente**

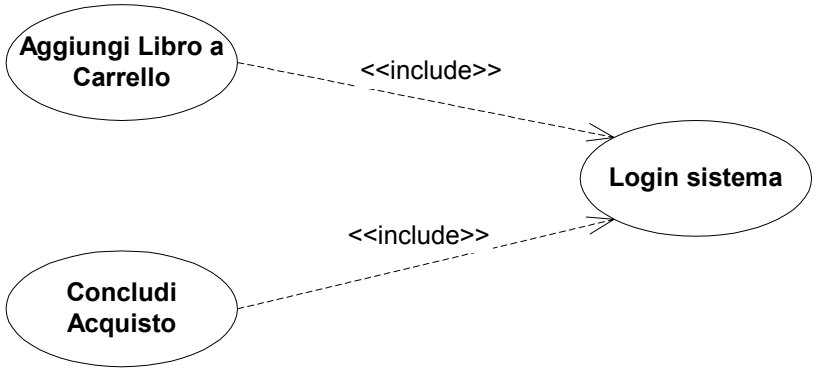
Relazioni fra Use Cases - Inclusione

Spesso uno use case complesso viene suddiviso in casi più semplici.

UML fornisce tre costrutti che permettono di definire le relazioni fra use cases, al fine di fattorizzare il comportamento comune e definire cammini alternativi.

In una relazione di *Inclusione* uno use case include esplicitamente il comportamento di un altro use case in un punto specifico del corso d'azione.

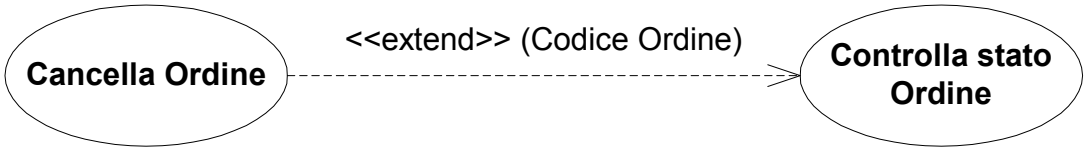
Viene usato al fine di circoscrivere comportamenti che si ripetono in più punti del sistema, e che come tali si ripeterebbero in più use cases.



Relazioni fra Use Cases - Estensione

In una relazione di *Estensione* uno use case base include implicitamente il comportamento di un altro use case in uno o più punti specifici del corso d'azione, detti *Punti di estensione*.

Viene usato al fine di delineare comportamenti opzionali o che si verificano solo in determinate circostanze.

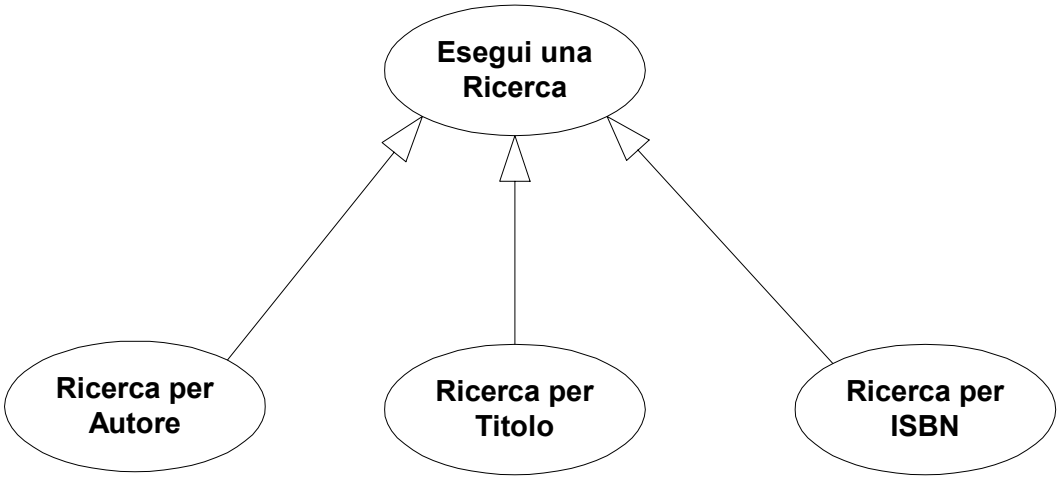


Nota: abbiamo fatto uso degli stereotipi <<include>> e <<extend>>.

Relazioni fra Use Cases - Generalizzazione

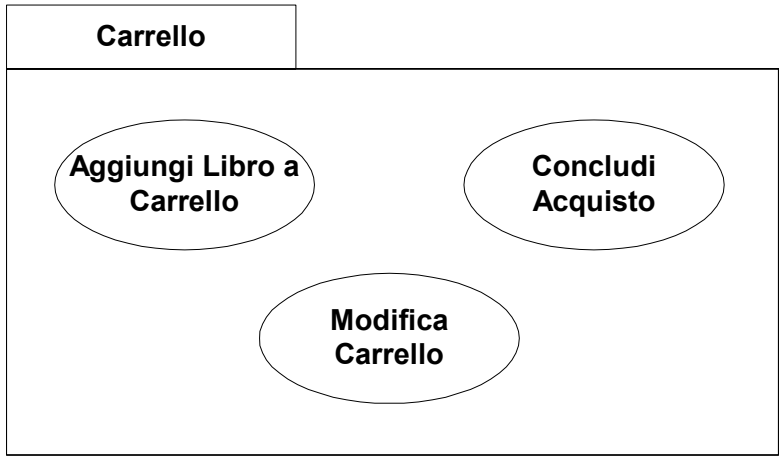
Con la relazione di *Generalizzazione* fra use cases, analogamente al caso delle classi, uno use case padre definisce il comportamento ereditato dai suoi figli, i quali possono ridefinire quel comportamento o aggiungerne altri.

Analogamente alla generalizzazione fra classi, viene rappresentata con una freccia con il triangolo pieno.



Package e Use Cases

I package definiti in precedenza possono contenere anche use cases (oltre che classi).



Esercizio:

✓ Definire il package **Gestione Profilo Utente**

Use Cases - Alcune riflessioni

- La creazione di rapida di prototipi (*Fast Prototyping*) è una tecnica universalmente comprovata. Gli use cases permettono a committenti e progettisti di costruire rapidamente una simulazione del sistema in via di sviluppo.
- Per reingegnerizzare un sistema già esistente, gli use cases possono essere desunti dalla documentazione di progetto già disponibile. Utili anche in caso di *Reverse Engineering*.
- I diagrammi di use cases costituiscono un'ottima base di partenza per realizzare la documentazione utente del sistema. Per esempio, ad ogni package può corrispondere un capitolo.
- Gli use cases sono molto utili in fase di testing (*Black-box Test*).

Analisi di robustezza

L'*Analisi di Robustezza* consiste nall'analisi del testo di uno use case, nell'identificazione di un primo insieme di oggetti che vi prenderanno parte e nella classificazione di tali oggetti in base alle loro caratteristiche.

All'interno del RUP, l'analisi di robustezza è associata al Workflow di Analisi.

Parte integrante dell'analisi di robustezza è la definizione delle *Classi di Analisi*, che costituiscono il nucleo del modello di analisi.

Esistono tre tipi di classi di analisi:

- Classi di confine (Boundary Classes)
- Classi entità
- Classi di controllo

Analisi di robustezza - Oggetti Confine

Un *Oggetto Confine* (o *Boundary Object*) è un elemento con il quale interagisce un attore in uno Use Case.

Se l'attore è un essere umano, un oggetto confine può essere un elemento di interazione grafica con il sistema, ad es. un dialog box.

Se l'attore è un database o un sistema esterno, un oggetto confine può essere una procedura di interfaccia fra i due sistemi.

Analisi di robustezza - Oggetti Entità e Oggetti di Controllo

Un *Oggetto Entità* contiene informazioni persistenti, come ad es. dati memorizzati in un database, oppure dati temporanei come lo stato di una finestra o il risultato di una ricerca.

Un *Oggetto di Controllo* viene utilizzato per rappresentare concetti come il coordinamento e la sequenza di eventi, oppure interazioni che coinvolgono più oggetti entità in presenza di segnali provenienti da oggetti confine.

Gli oggetti di controllo contengono un pezzo di logica dell'applicazione.

Analisi di robustezza - Regole

✓ Un oggetto confine non può comunicare con un altro oggetto confine: è necessario inserire un oggetto di controllo. Può però esistere una relazione di aggregazione fra oggetti confine, come nel caso precedente.

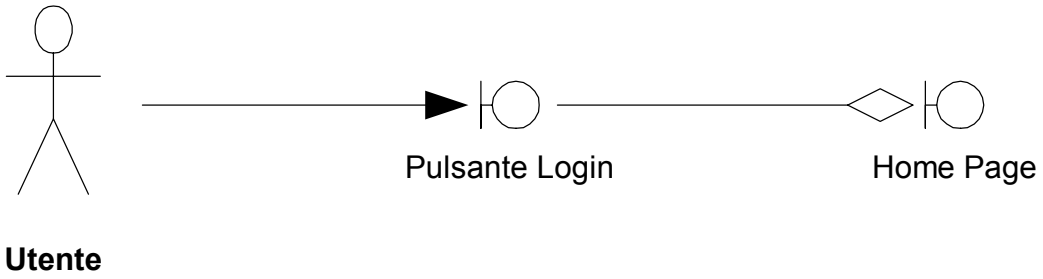
✓ Un oggetto di confine non può comunicare un oggetto entità: è necessario inserire un oggetto di controllo.

Analisi di robustezza - Esempio

Riprendiamo lo use case Login nel modello della On-line Bookshop.

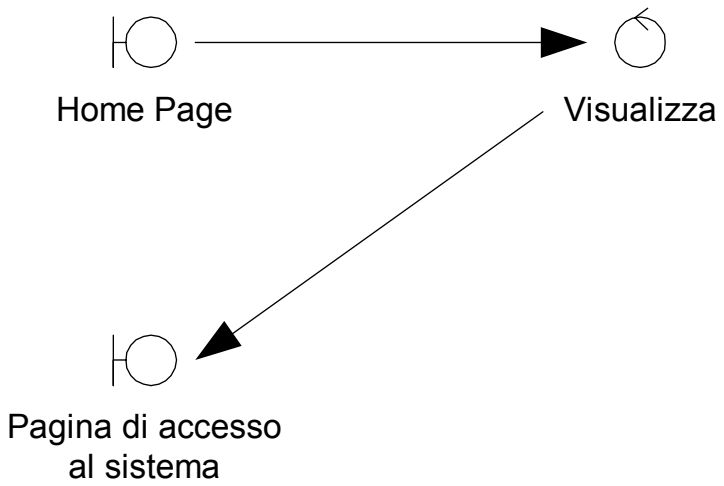
L'Utente clicca sul pulsante Login nella Home Page.
Il sistema visualizza la pagina di accesso al sistema.
L'utente digita il suo codice identificativo e la sua password, poi clicca OK.
Il sistema convalida i dati inseriti dall'utente confrontandoli con quelli presenti nel Profilo Utente, e in caso positivo fa tornare l'utente alla Home Page.

1. L'Utente clicca sul pulsante Login nella Home Page



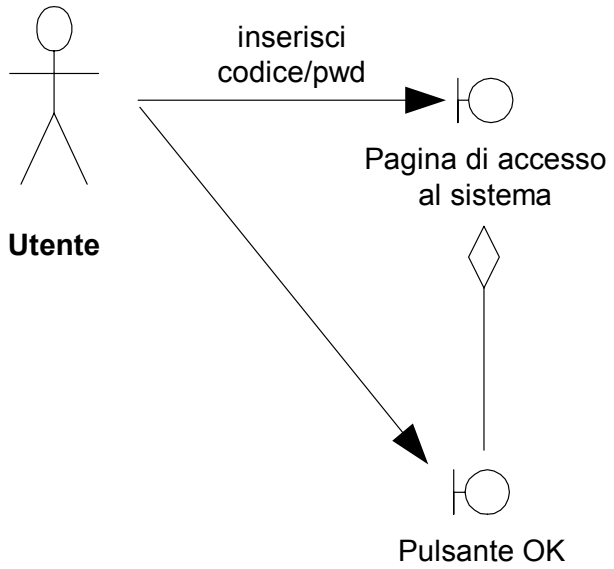
Analisi di robustezza - Esempio

2. Il sistema visualizza la pagina di accesso al sistema



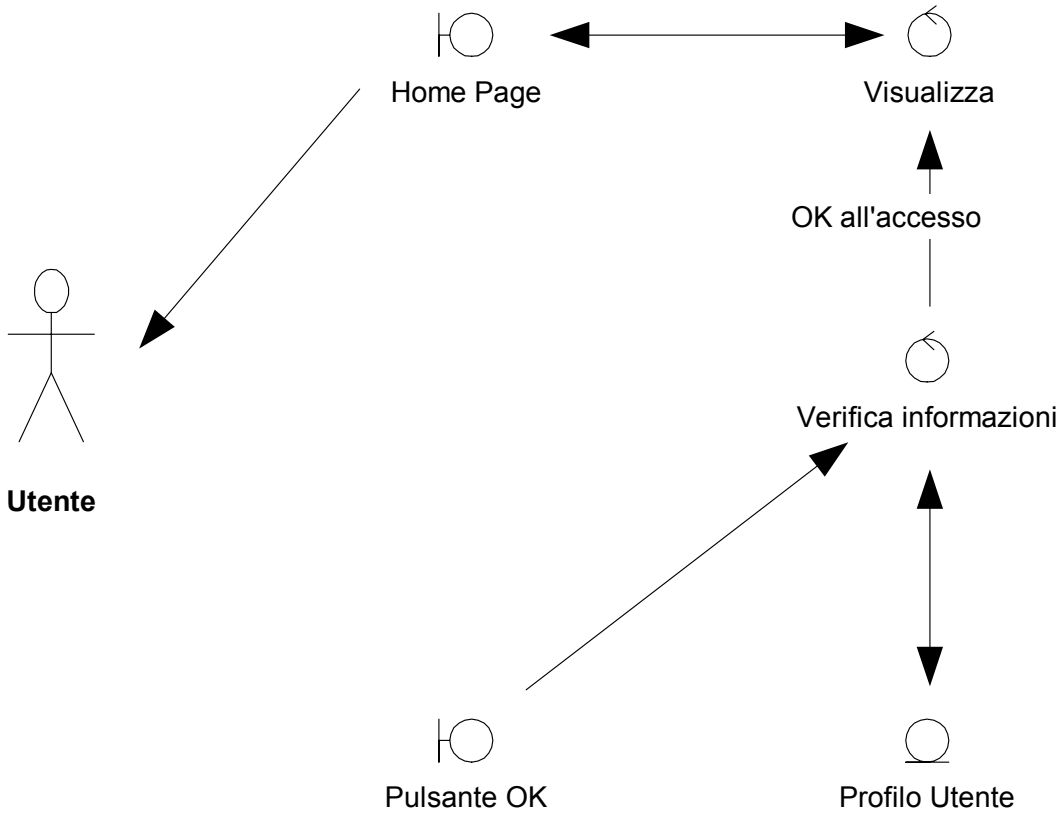
Analisi di robustezza - Esempio

3. L’utente digita il suo codice identificativo e la sua password, poi clicca OK

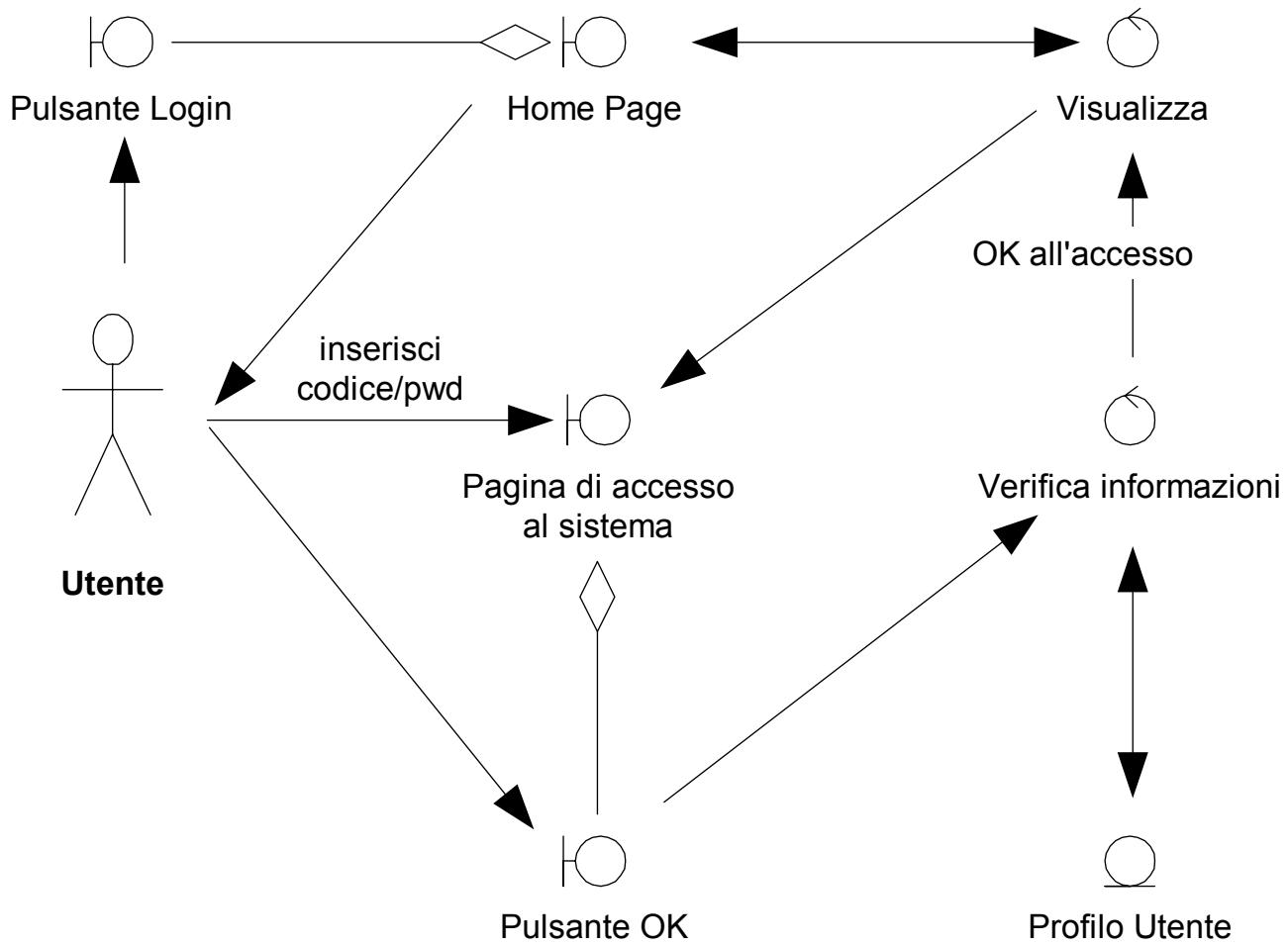


Analisi di robustezza - Esempio

4. Il sistema convalida i dati inseriti dall'utente confrontandoli con quelli presenti nel Profilo Utente, e in caso positivo fa tornare l'utente alla Home Page.



Analisi di robustezza - Esempio



Analisi di robustezza - Esempio

Il diagramma rappresenta il corso d'azione base dello use case Accedi al sistema. L'analisi di robustezza prevede che vengano disegnati anche tutti i corsi d'azione alternativi.

Esercizio:

✓ Disegnare il corso d'azione alternativo corrispondente al caso in cui la verifica informazioni dia esito "Accesso fallito"

✓ Disegnare il diagramma di robustezza dello use case Scrivi recensione Utente.

Diagrammi di Interazione

UML definisce due tipi di diagrammi, detti *Diagrammi di Interazione*, che sono associati all'analisi di robustezza.

I diagrammi di interazione sono ideali per modellare l'evoluzione di un singolo use case.

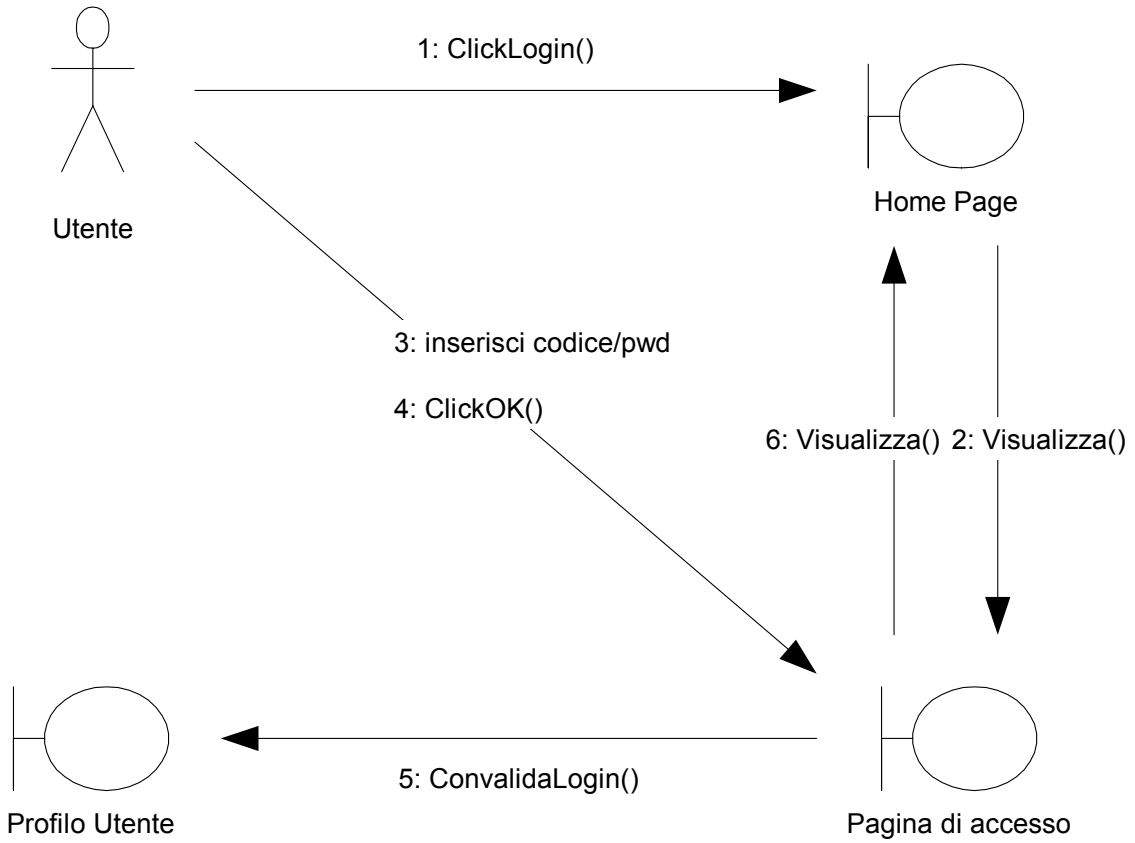
Esistono in due forme:

- *Diagramma di Collaborazione* - si focalizza sulle relazioni statiche fra oggetti e sull'invio dinamico di messaggi fra oggetti
- *Diagramma di Sequenza* - presenta un'organizzazione più rigida e si focalizza sulla sequenza temporale delle interazioni fra gli oggetti

I due diagrammi contengono le stesse informazioni, tanto che dall'uno si può derivare l'altro.

Diagramma di Collaborazione

Riprende i concetti di base dell'analisi di robustezza, con in più l'indicazione dell'ordine temporale con cui avviene lo scambio di messaggi.



Messaggi e Azioni

Si definisce *Messaggio* una comunicazione fra due oggetti (o all'interno di uno stesso oggetto) che dà origine ad una attività.

Un'attività implica una o più *Azioni*, che consistono in istruzioni eseguibili aventi come risultato il cambiamento di valore di uno o più attributi di un oggetto, o la restituzione di uno o più valori di ritorno all'oggetto che ha inviato il messaggio.

Esistono 5 tipi di azioni esplicitamente supportate da UML:

- Azione di Chiamata
- Azione di Risposta (o di Ritorno)
- Azione di Creazione
- Azione di Distruzione
- Azione di Invio di Segnali

Azioni di Chiamata e Risposta

Un'*Azione di Chiamata* consiste nell'invocazione di un metodo di un oggetto. Un oggetto può eseguire un'azione di chiamata su altri oggetti, ma anche su se stesso.

Un'*Azione di Risposta* consiste nella restituzione di un valore in risposta ad un'azione di chiamata.

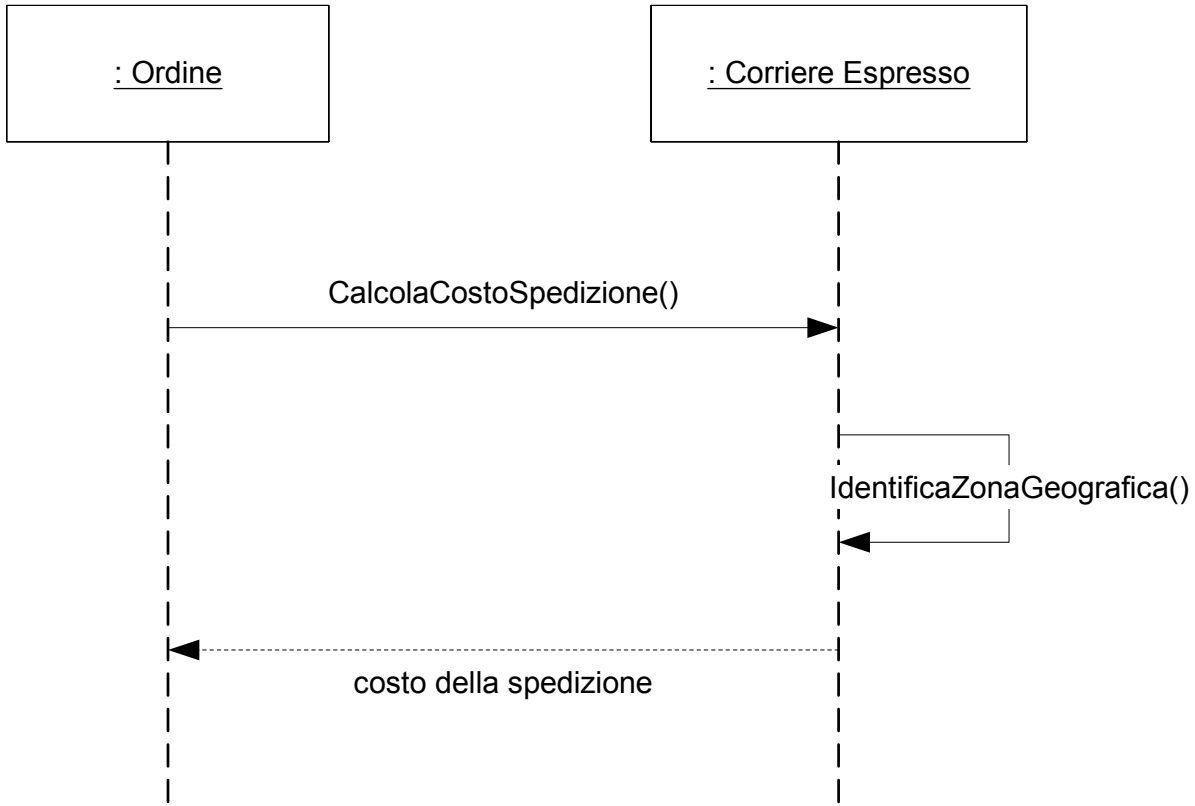
Per sua natura il meccanismo di chiamata e risposta è sincrono:

- il chiamante si aspetta che il ricevente sia pronto ad accogliere il segnale che gli viene inviato
- il chiamante, prima di procedere con altre attività, rimane in attesa di una risposta da parte dell'oggetto chiamato.

Non deve obbligatoriamente esserci corrispondenza 1-1 fra chiamata e risposta: se la risposta è deducibile dal contesto, si possono anche indicare azioni di chiamata senza le relative azioni di risposta.

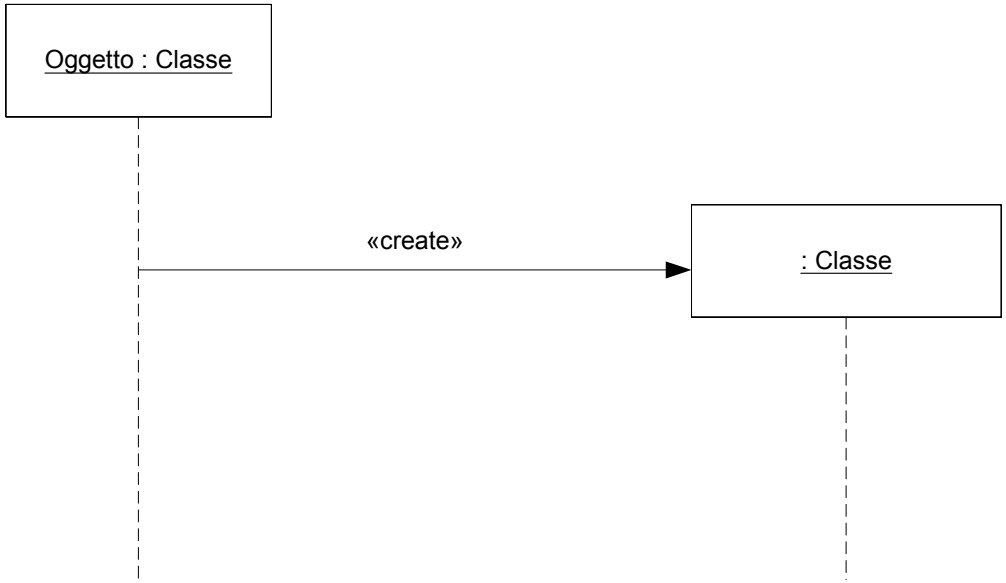
Azioni di Chiamata e Risposta

Rappresentazione grafica delle azioni di chiamata e risposta secondo lo standard UML:



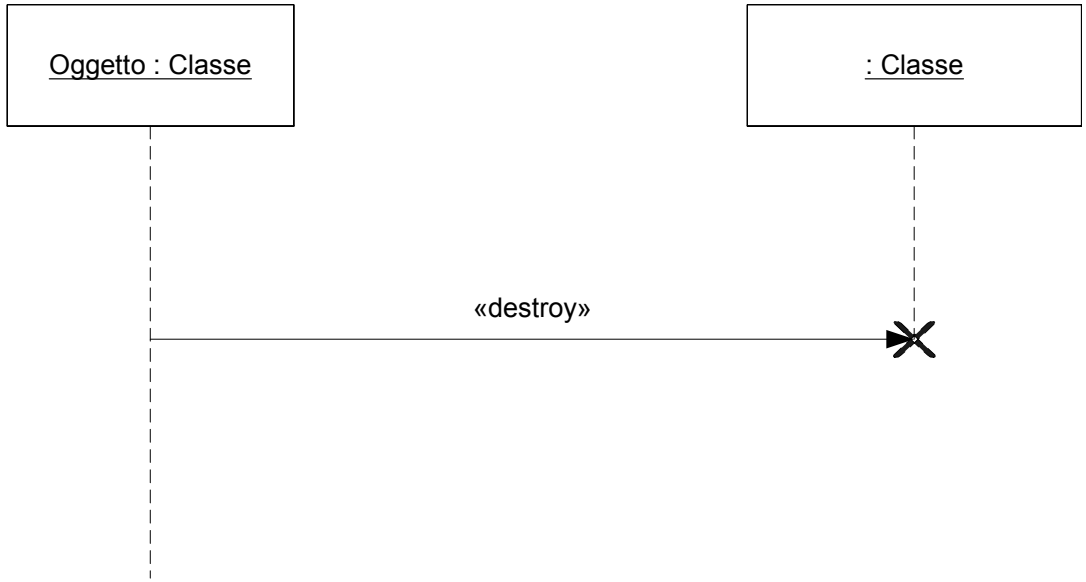
Azioni di Creazione e Distruzione

Un'*Azione di Creazione* crea un oggetto, ossia un'istanza di una classe.



Azioni di Creazione e Distruzione

Un'*Azione di Distruzione* distrugge un oggetto.
Un oggetto può eseguire un'azione di distruzione su un altro oggetto, ma anche su se stesso.



Le linee verticali sono dette *Lifelines* degli oggetti del sistema.

Azioni di Invio di Segnali

Un'*Azione di Invio* invia un segnale ad un oggetto. Un *Segnale* è un elemento di comunicazione asincrona fra oggetti.

Meccanismo *throw/catch*: un oggetto lancia un segnale ad un altro oggetto (che lo "riceve al volo"), e non si aspetta alcuna risposta da esso.

Le eccezioni sono il tipo più comune di segnali: non a caso throw e catch sono keywords del C++ per quanto riguarda la gestione delle eccezioni.

Diagramma di Sequenza

Il *Diagramma di Sequenza* si focalizza sull'ordinamento temporale dei messaggi che vengono scambiati fra gli oggetti.

Secondo la notazione standard UML, nei diagrammi di sequenza:

• Gli oggetti sono allineati nella parte superiore del diagramma; possono essere rappresentati mediante boxes oppure mediante le stesse icone dei diagrammi di robustezza.

• Ogni oggetto presenta una Lifeline

• Un *Box di Attivazione* o *Focus of Control*, rappresentato con un piccolo rettangolo lungo la lifeline, indica il periodo temporale durante il quale l'oggetto ha il controllo del flusso. L'indicazione dei box di attivazione nei diagrammi di sequenza è opzionale.

• I messaggi mostrano le azioni eseguite dagli oggetti su loro stessi o sugli altri oggetti.

Diagramma di Sequenza

Diagramma di sequenza dello use case **Login** del modello di On-Line Bookshop :

1. L'Utente clicca sul pulsante Login nella Home Page

viene così rappresentata nel diagramma di sequenza:

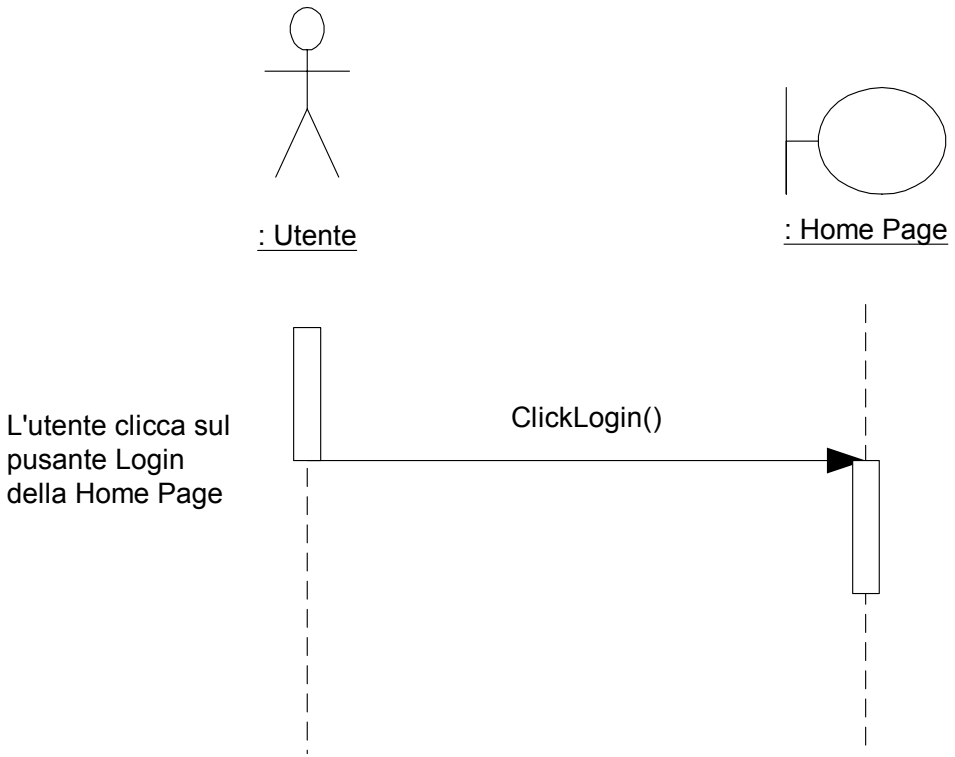


Diagramma di Sequenza

Notare che:

- Si è scelta la rappresentazione con le icone del diagramma di robustezza.
- Non compare il pulsante di login. La relazione di aggregazione fra pulsante di login e home page è stata collassata nella sola home page.
Questa pratica è comune nei diagrammi di sequenza, al fine di aumentarne la leggibilità.
- Il testo dello use case è riportato nella parte sx del diagramma
- Si è scelto di rappresentare i box di attivazione.

Diagramma di Sequenza

2. Il sistema visualizza la pagina di accesso al sistema

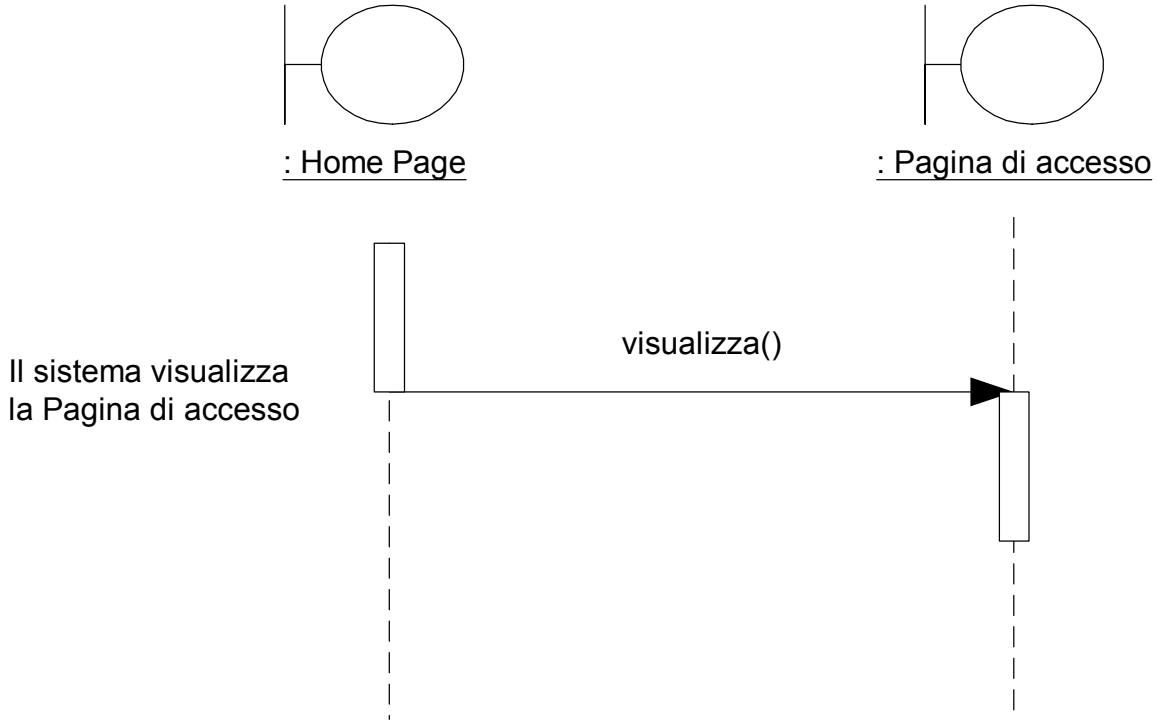


Diagramma di Sequenza

3. L’utente digita il suo codice identificativo e la sua password, poi clicca OK

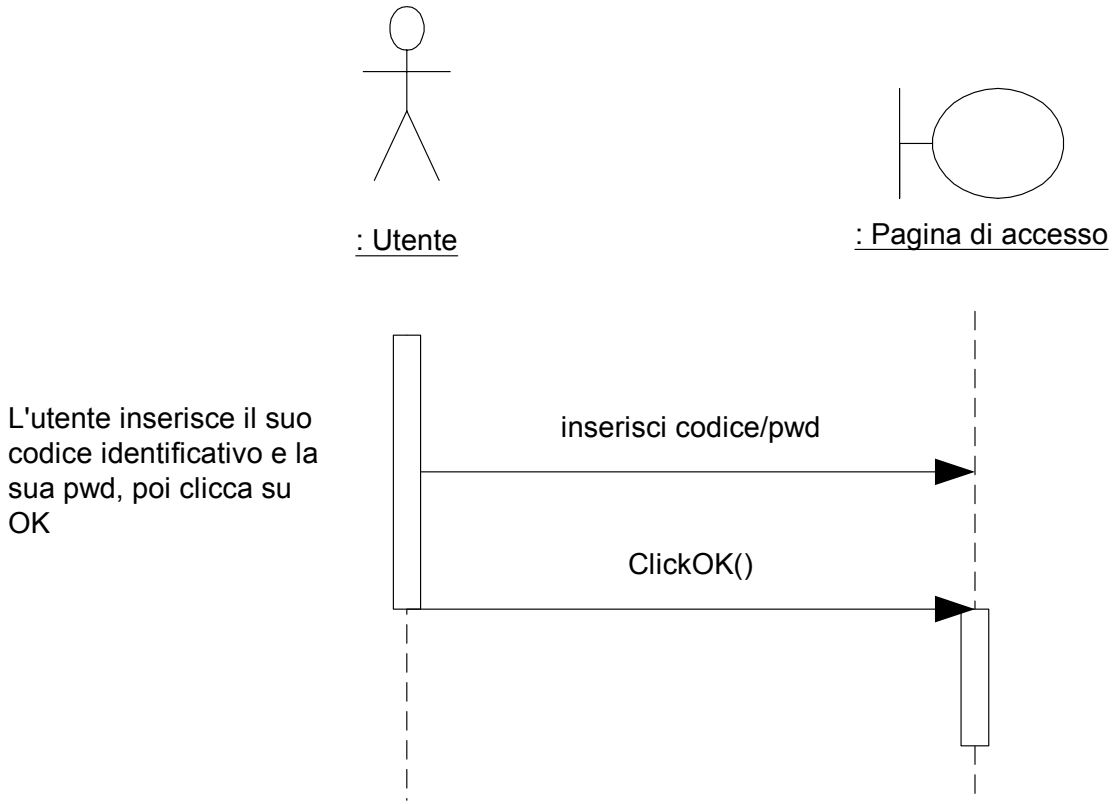
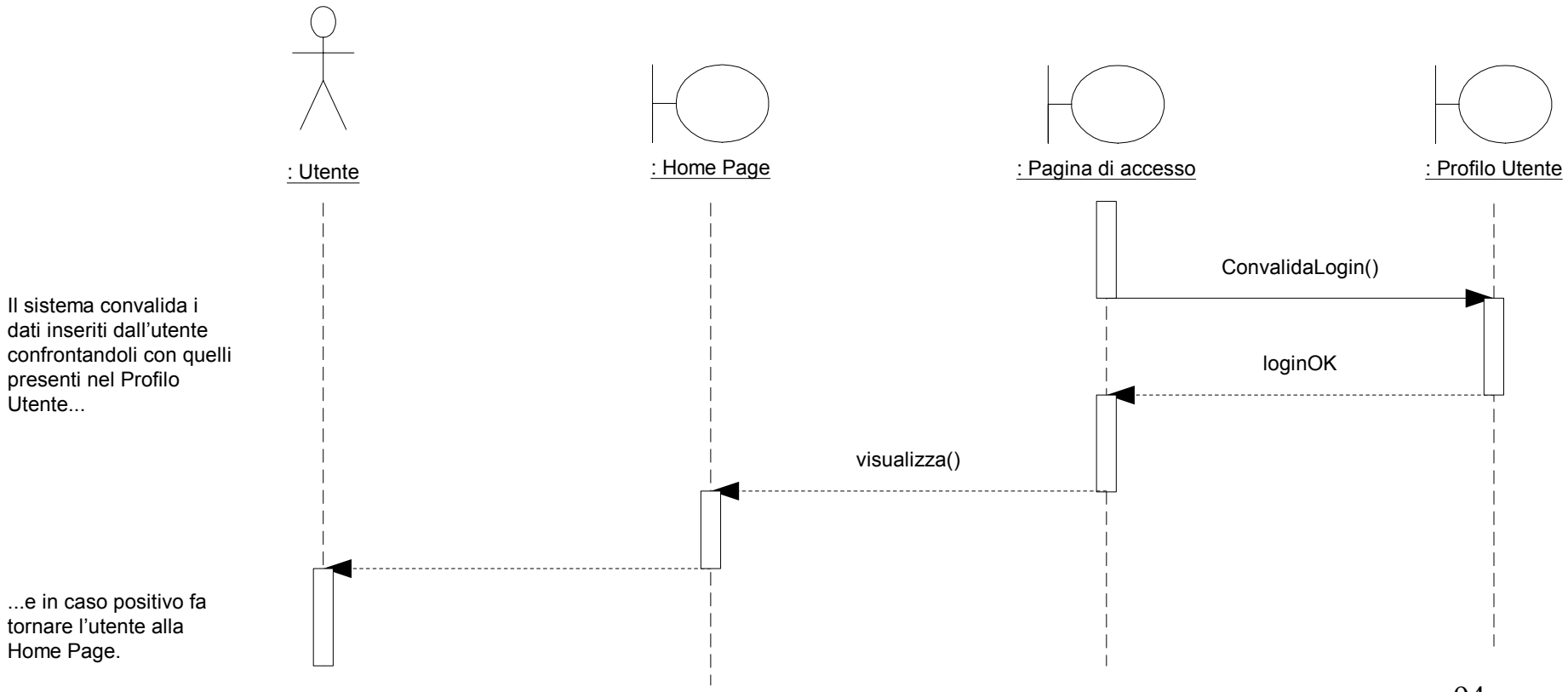
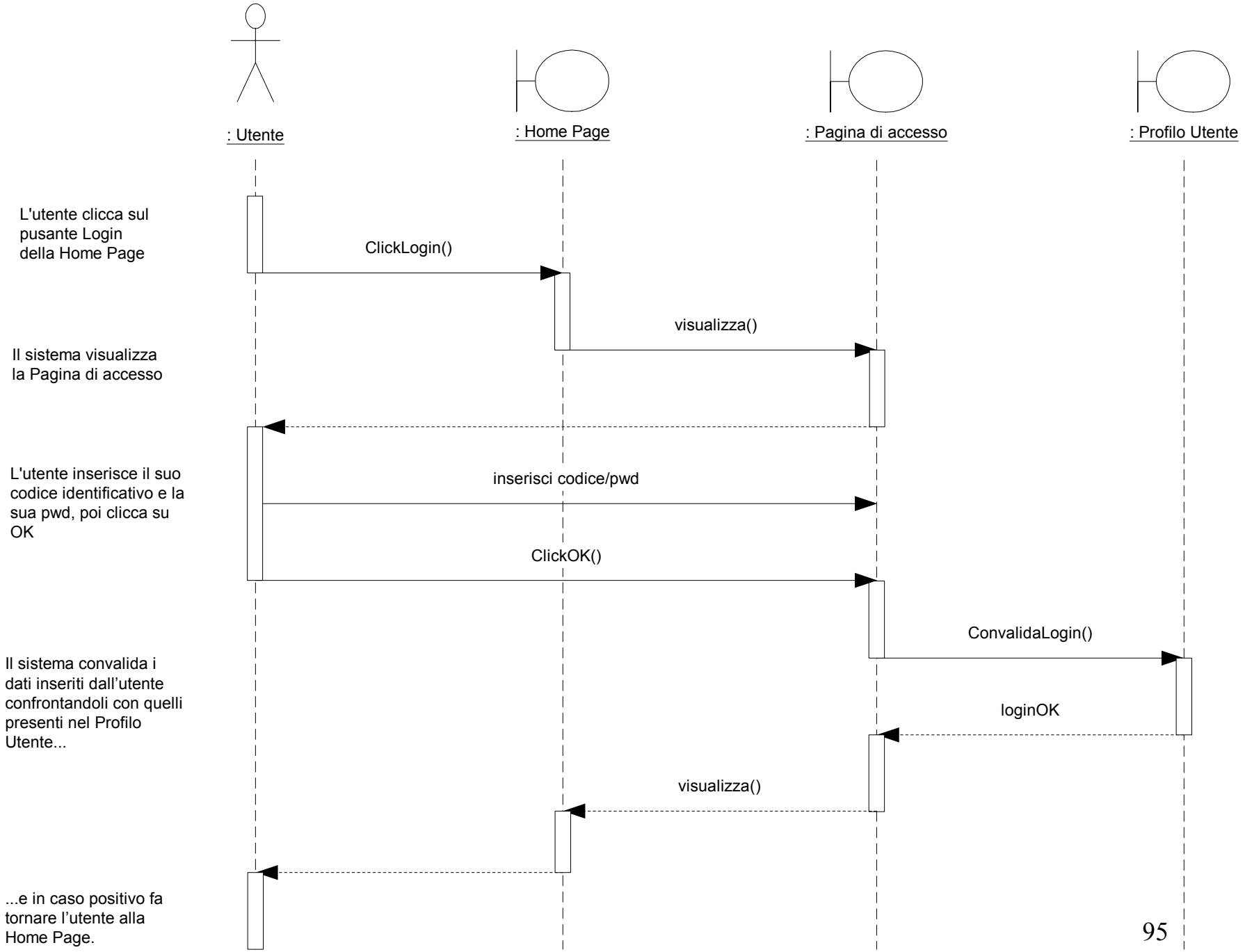


Diagramma di Sequenza

4. Il sistema convalida i dati inseriti dall'utente confrontandoli con quelli presenti nel Profilo Utente, e in caso positivo fa tornare l'utente alla Home Page.




: Utente
: Home Page
: Pagina di accesso
: Profilo Utente
L'utente clicca sul pusante Login della Home Page
ClickLogin()
visualizza()
Il sistema visualizza la Pagina di accesso
L'utente inserisce il suo codice identificativo e la sua pwd, poi clicca su OK
inserisci codice/pwd
ClickOK()
ConvalidaLogin()
Il sistema convalida i dati inseriti dall'utente confrontandoli con quelli presenti nel Profilo Utente...
loginOK
visualizza()
...e in caso positivo fa tornare l'utente alla Home Page.

Eventi

Un *Evento* è un qualcosa che accade ed ha rilevanza per un oggetto.

UML supporta la rappresentazione di 4 tipi di eventi:

- *Segnale* - comunicazione asincrona fra oggetti
- *Evento di chiamata* - comunicazione sincrona durante la quale un oggetto invoca un metodo di un altro oggetto, oppure un proprio metodo.
- *Evento temporale* - avviene dopo un periodo di tempo specificato
- *Evento di cambiamento* - avviene quando è verificata una particolare condizione.

Il modo con cui un oggetto risponde ad un particolare evento dipende, in generale, dallo stato dell'oggetto nel momento in cui riceve l'evento.

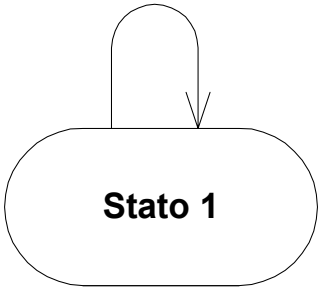
Stati e Transizioni

Uno *Stato* è una condizione nella quale un oggetto, durante il suo ciclo di vita, può venirsi a trovare per un periodo finito di tempo.

Una *Transizione* è il passaggio di un oggetto da uno stato (detto stato *origine*) ad un altro (detto stato *destinazione*).
Può avvenire in corrispondenza di un evento (detto *trigger*), oppure può avvenire incondizionatamente (in questo caso la transizione è detta *triggerless*).

Stati e Transizioni

E' anche possibile che stato origine e stato destinazione coincidano: in questo caso si ha un *autoanello*.



Una *Condizione di guardia* è un'espressione booleana che deve valere True affinchè una transizione possa scattare.

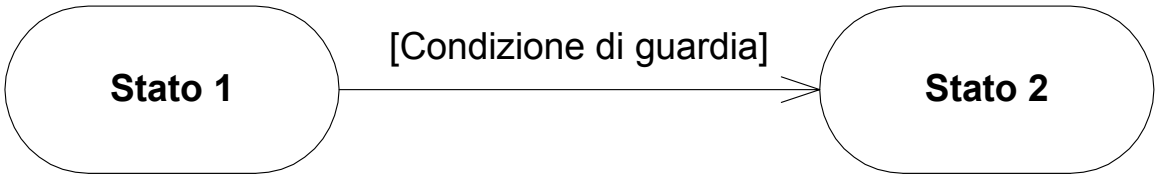


Diagramma degli Stati

Il *Diagramma degli Stati* rappresenta la macchina a stati di un oggetto, ossia la combinazione dei seguenti elementi:

- Gli stati che un oggetto può assumere durante il suo ciclo di vita
- Gli eventi a cui un oggetto può rispondere
- Le possibili risposte che l'oggetto può fornire a quegli eventi
- Le transizioni che avvengono fra gli stati dell'oggetto.

Il diagramma degli stati UML rappresenta un costrutto ideale per la modellazione di un oggetto che presenta queste caratteristiche:

- Ammette un numero limitato di valori
- Ha delle restrizioni sulle transizioni fra questi valori.

Diagramma degli Stati

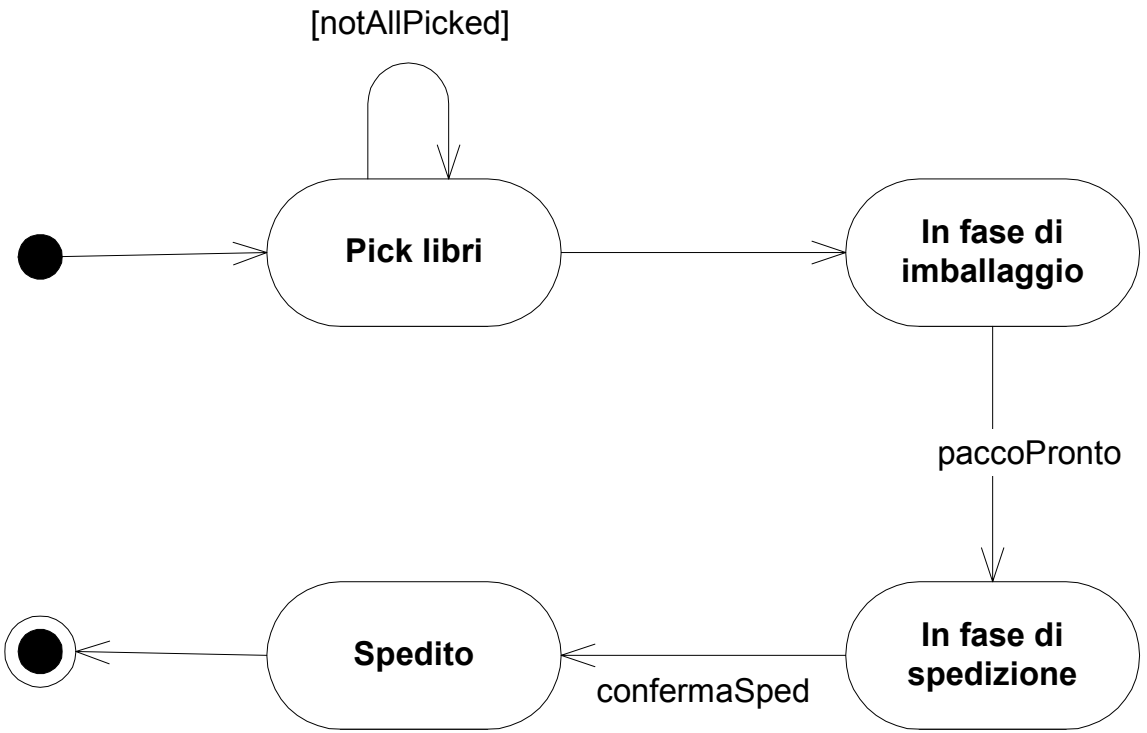
Oltre alle notazioni già viste per stati e transizioni, esistono:

- Stato iniziale
- Stato finale
- Azione di entrata (entry action) - eseguita dall'oggetto subito dopo essere passato in uno stato. Indicata con la dicitura *entry / nome_azione*.
- Azione di uscita (exit action) - eseguita da un oggetto subito prima di uscire da uno stato. Indicata con la dicitura *exit / nome_azione*.

Stato iniziale Stato finale

Diagramma degli Stati

Esempio: spedizione libri della On-Line Bookshop



Stati Compositi

Gli stati visti fino adesso sono detti *semplici*.

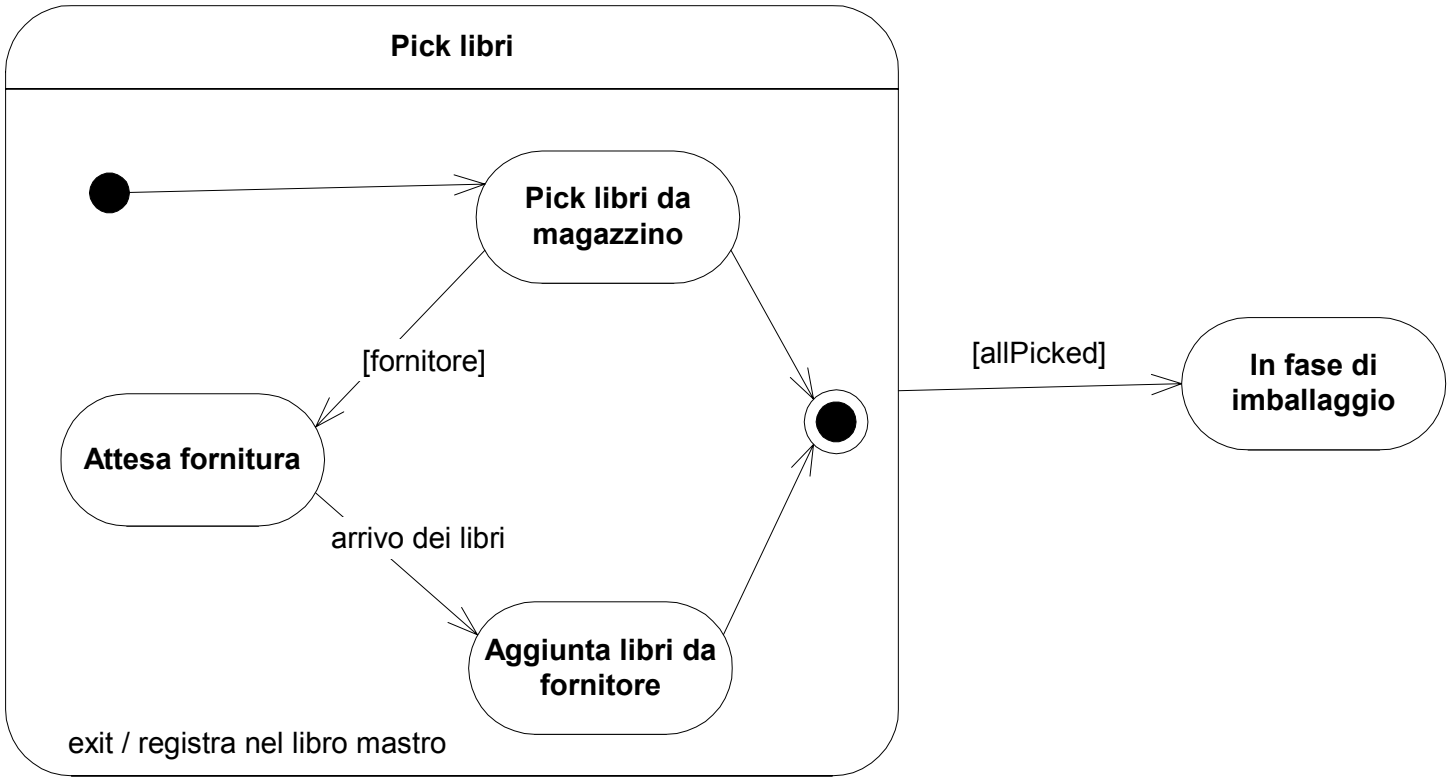
UML consente anche l'annidamento degli stati: gli *stati compositi* sono stati che contengono altri stati, detti *sottostati*.

Esistono tre tipi di sottostati:

- Sottostati sequenziali
- History states
- Sottostati concorrenti

Stati Compositi - Sottostati sequenziali

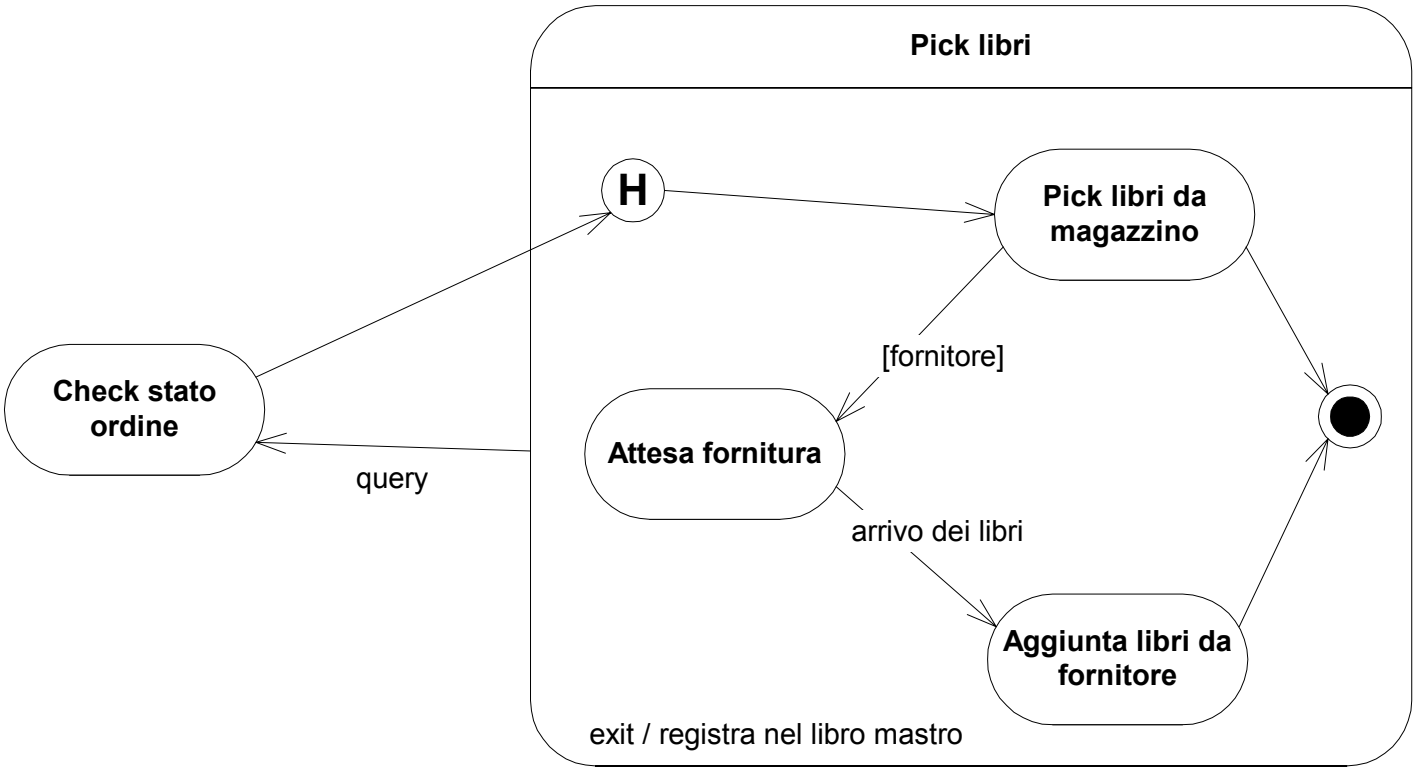
Se un oggetto, in un determinato momento, può trovarsi solo in uno dei sottostati di uno stato composito, questi sono detti *sottostati sequenziali*.



Stati Compositi - History states

Nell'esempio precedente, quando un oggetto compie una transizione verso uno stato composito, inizia dal flusso dal sottostato iniziale.

Talvolta è necessario interrompere il flusso all'interno di uno stato composito e successivamente riprenderlo nel punto in cui è stato interrotto.
Per questo scopo si fa uso di uno *History state*.



Stati Compositi - Sottostati concorrenti

Se un oggetto, in un determinato momento, può trovarsi in più sottostati di uno stato composito, questi sono detti *sottostati sequenziali*.