

Introduzione a UML

Modellare

- Un modello è un'**astrazione** che cattura le proprietà salienti della **realtà** che si desidera rappresentare.
- Idealizza una realtà complessa, individuandone i tratti importanti e separandoli dai dettagli, facilitandone la comprensione.
- La mente umana compie un'attività continua di modellazione, producendo schemi per comprendere e spiegare quello che viene percepito dai sensi.
- La realtà è un'*istanza* del modello.

Perché Modellare

- Per *comprendere* il soggetto in analisi.
- Per *conoscere* il soggetto in analisi (fissando ciò che si è compreso).
- Per *comunicare* la conoscenza del soggetto.

I progetti software sono complessi

- Il tipico progetto software raramente coinvolge un solo sviluppatore, e può coinvolgerne anche centinaia:
 - ▶ separare compiti e responsabilità
 - ▶ raggruppare le informazioni a diversi livelli di granularità
- Il tipico progetto software subisce un ricambio di personale nel corso della sua storia:
 - ▶ il progetto perde conoscenza
 - ▶ nuovi sviluppatori devono acquisirla
- Le caratteristiche del progetto spesso mutano col tempo:
 - ▶ necessità di comunicare con il cliente in termini chiari
 - ▶ prevedere ed adattarsi ai cambiamenti
 - ▶ stimarne l'impatto su costi, tempi e risorse di sviluppo

Come si può ragionare su questo se non si sa su cosa si sta ragionando?

I linguaggi di modellazione

- Un linguaggio di modellazione fornisce le primitive a cui ricondurre la realtà in esame.
- Permette di esprimere le **entità** che compongono un sistema complesso, le loro **caratteristiche** e le **relazioni** che le collegano.
- Nell'ambito di un progetto, il **linguaggio** di modellazione è normalmente distinto dal **processo** di sviluppo:
 - ▶ il linguaggio descrive cosa deve essere ottenuto;
 - ▶ il processo descrive i passi da intraprendere per ottenerlo;
 - ▶ insieme, linguaggio e processo definiscono un **metodo di sviluppo**.

Che cos'è UML (1)

- **UML**, Unified Modeling Language, è un linguaggio **semiformale** e **grafico** (basato su diagrammi) per specificare, visualizzare, costruire e documentare gli artefatti di un sistema software.
- Permette di:
 - ▶ modellare un dominio
 - ▶ scrivere i requisiti di un sistema software
 - ▶ descrivere l'architettura del sistema
 - ▶ descrivere struttura e comportamento di un sistema
 - ▶ documentare un'applicazione
 - ▶ generare automaticamente un'implementazione
- Gli stessi modelli UML sono quindi artefatti usati per sviluppare il sistema e comunicare con il cliente (ma anche con progettisti/sviluppatori/etc.)

Che cos'è UML (2)

- Si tratta di un linguaggio di modellazione usato per capiere e descrivere le caratteristiche di un nuovo sistema o di uno esistente.
- Indipendente dall'ambito del progetto.
- Indipendente dal processo di sviluppo.
- Indipendente dal linguaggio di programmazione (progettato per essere abbinato alla maggior parte dei linguaggi object-oriented).
- Fa parte di un metodo di sviluppo, non è esso stesso il metodo.

L come Linguaggio

- Si tratta di un vero e proprio linguaggio, non di una semplice notazione grafica.
- Un modello UML è costituito da un insieme di elementi che hanno *anche* una rappresentazione grafica.
- Il linguaggio è semiformale perché descritto in linguaggio naturale e con l'uso di diagrammi, cercando di ridurre al minimo le ambiguità.
- Ha regole *sintattiche* (come produrre modelli legali) e regole *semantiche* (come produrre modelli con un significato).

Sintassi e semantica: esempio



- Usiamo un semplice diagramma dei Casi d'Uso come esempio.
- Regola sintattica: una relazione tra un attore e un caso d'uso può opzionalmente includere una freccia.
- Regola semantica: la freccia significa che la prima interazione si svolge nel senso indicato dalla freccia.

U come Unificato

- UML rappresenta la sintesi di vari approcci metodologici fusi in un'unica entità.
- L'intento era di prendere il meglio da ciascuno dei diversi linguaggi esistenti e integrarli. Per questo motivo, UML è un linguaggio molto vasto.
- Questa è una delle critiche principali mosse a UML ("vuole fare troppe cose").

Breve storia (1)

- Agli inizi degli anni '90 vi era una proliferazione di linguaggi e approcci alla modellazione object-oriented. Mancava uno standard accettato universalmente per la creazione di modelli software.
- C'era la sensazione che la quantità di differenti soluzioni ostacolasse la diffusione dello stesso metodo object-oriented.
- Nel 1994 due esperti di modellazione della *Rational Software Corporation*, Grady Booch e James Rumbaugh, unificarono i propri metodi, **Booch** e **OMT** (Object Management Technique).
- Nel 1995 si unisce anche Ivar Jacobson con il suo **OOSE** (Object Oriented Software Engineering), dopo l'acquisizione della sua compagnia Objectory da parte di Rational.

Breve storia (2)

- Nel 1996, Booch, Rumbaugh e Jacobson, noti come i *Three Amigos*, sono incaricati da Rational di dirigere la creazione dell'Unified Modeling Language.
- Nel 1997, la specifica UML 1.0 viene presentata all'**OMG** (Object Management Group), un consorzio di grandi aziende interessate allo sviluppo di standard e tecnologie basate su oggetti.
- Nel novembre dello stesso anno, una versione arricchita, UML 1.1, viene approvata dall'OMG.
- Seguono aggiornamenti: 1.2 (1998), 1.3 (1999), 1.4 (2001), 1.5 (2003), e la major revision 2.0 (2005).
- L'ultima versione è la 2.3 rilasciata a maggio 2010.

Object-oriented

- UML non è solo un linguaggio per la modellazione, ma un linguaggio per la modellazione **orientata agli oggetti**.
- Questo include sia l'*analisi* che la *progettazione* orientata agli oggetti (OOA e OOD, rispettivamente):
 - ▶ **analisi**: capire **cosa** deve fare il sistema, senza occuparsi dei dettagli implementativi
 - ▶ **progettazione**: capire **come** il sistema raggiunge il suo scopo, come viene implementato
- UML offre strumenti di modellazione OO in entrambi gli ambiti; frammenti differenti di UML sono impiegati in diverse fasi del processo di sviluppo (anche se UML stesso non fornisce indicazioni sul suo utilizzo).

Object-oriented

- OO: un paradigma che sposta l'enfasi della programmazione dal codice verso le **entità** su cui esso opera, gli **oggetti**.
- Una rivoluzione copernicana cominciata dagli anni '60 e proseguita, lentamente, fino ad affermarsi negli anni '90.
- I principi cardine furono proposti separatamente e solo successivamente integrati in unico paradigma.
- Principi OO comunemente accettati sono: **Abstraction**, **Encapsulation**, **Inheritance**, **Polymorphism**.

Principi OO

- **Abstraction** : usare *classi* per astrarre la natura e le caratteristiche di un oggetto, che è un'istanza della propria classe di appartenenza.
- **Encapsulation** : nascondere al mondo esterno i dettagli del funzionamento di un oggetto; gli oggetti hanno accesso solo ai dati di cui hanno bisogno.
- **Inheritance** : classi possono specializzare altre classi ereditando da esse e implementando solo la porzione di comportamento che differisce.
- **Polymorphism** : invocare comportamento diverso in reazione allo stesso messaggio, a seconda di quale oggetto lo riceve.

Meta-Object Facility

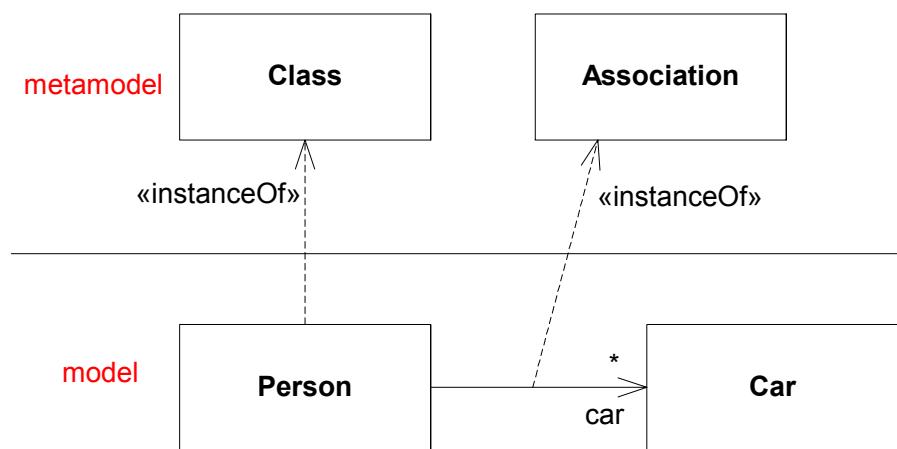
- Si può dire che in UML tutto è un oggetto.
- La relazione classe/istanza costituisce le fondamenta stessa del linguaggio.
- UML stesso, il linguaggio, è un'*istanza*!
- UML fa parte di un'architettura standardizzata per la modellazione di OMG chiamata **MOF** (Meta-Object Facility).
- Si può vedere MOF come un linguaggio per creare linguaggi, uno dei quali è UML.

Il metamodello UML

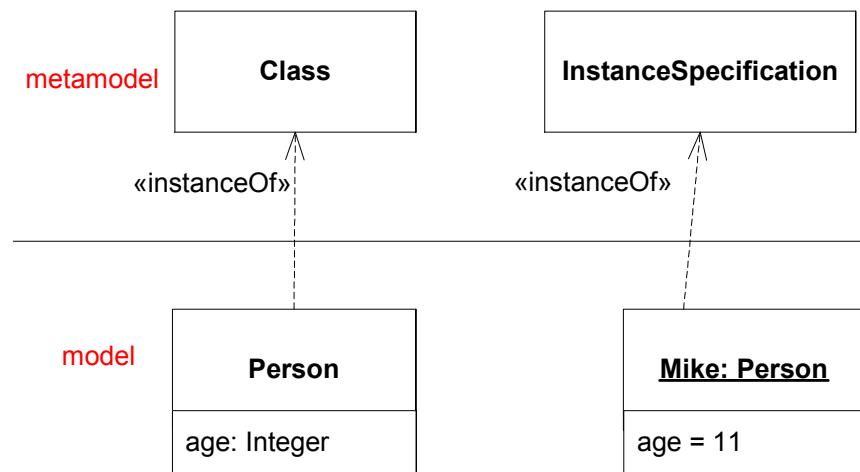
- MOF ha 4 livelli: M0, M1, M2 e M3. Ogni livello è un'istanza di un elemento del livello superiore.
 - ▶ un elemento di M0 è la realtà da modellare
 - ▶ un elemento di M1 è un modello che descrive la realtà
 - ▶ un elemento di M2 è un modello che descrive modelli (*metamodello*); **UML è qui**
 - ▶ un elemento di M3 è un modello che descrive metamodelli (*meta-metamodello*); **MOF è qui**
- OMG usa MOF per definire altri linguaggi oltre a UML.
- Tutti i linguaggi basati su MOF e i modelli con essi prodotti possono essere serializzati e scambiati tramite lo standard **XMI** (XML Metadata Interchange).

Semplificando...

- Si può dire che UML è definito tramite un modello UML (o piuttosto, usando un modello M3 che usa primitive comuni a UML).
- In qualunque momento, un oggetto al livello Mx è un'istanza di uno del livello superiore.
- Il meta-metamodello di livello M3 è progettato per essere istanza di se stesso, quindi non esiste M4.
- **Chi usa UML crea modelli di livello M1**; tuttavia, è bene sapere che esistono anche i livelli superiori.



M1 e M2 a confronto (usando un diagramma UML).



Altro esempio con M1 (modello utente) e M2 (metamodello UML)

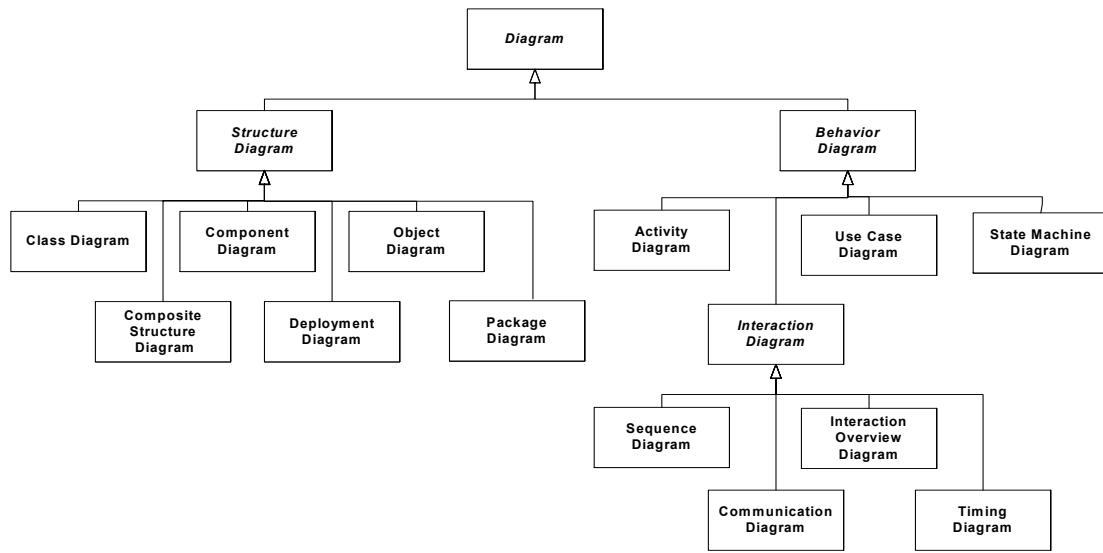
I diagrammi e le viste

- Un **diagramma** è la **rappresentazione grafica di una parte del modello**.
- Fornisce una **vista** di un sistema o una sua parte, cioè ne mette in risalto diverse proprietà.
- 4+1 viste (Kruchten, 1995):
 - ▶ **Logical:** mette in risalto la scomposizione logica del sistema tramite classi, oggetti e loro relazioni
 - ▶ **Development:** mostra l'organizzazione del sistema in blocchi strutturali (package, sottosistemi, librerie, ...)
 - ▶ **Process:** mostra i processi (o thread) del sistema in funzione, e le loro interazioni
 - ▶ **Physical:** mostra come il sistema viene installato ed eseguito fisicamente
 - ▶ **Use case:** (+1) la vista che agisce da 'collante' per le altre; spiega il funzionamento desiderato del sistema
- UML 2 definisce 13 diagrammi (contro i 9 di UML 1.x), divisi in due categorie:
 - ▶ **Structure diagrams:** come è fatto il sistema; forniscono le viste Logical, Development e Physical
 - ▶ **Behavior diagrams:** come funziona il sistema; forniscono le viste Process e Use case

Structure	Behavior
Class diagram Object diagram Package diagram* Composite Structure diagram* Component diagram Deployment diagram	Use Case diagram Activity diagram State Machine diagram Sequence diagram Communication diagram Interaction Overview diagram* Timing diagram*

* : non esiste in UML 1.x

Tassonomia dei 13 diagrammi di UML 2

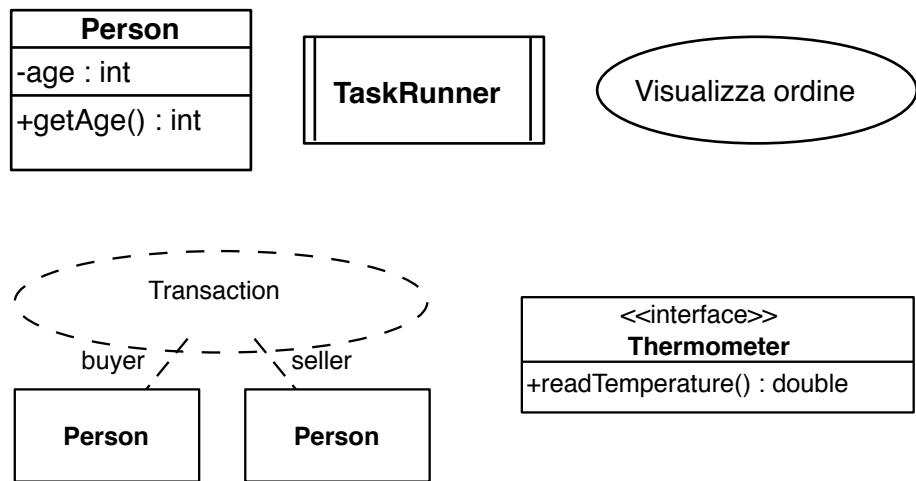


Entità UML

- UML prevede diversi tipi di entità che possono essere organizzati in quattro categorie:
 - ▶ Strutturali
 - ▶ Comportamentali
 - ▶ Informative
 - ▶ Raggruppamento e contenimento
- Segue una panoramica delle entità principali ma andremo in dettaglio quando analizzeremo i diversi diagrammi

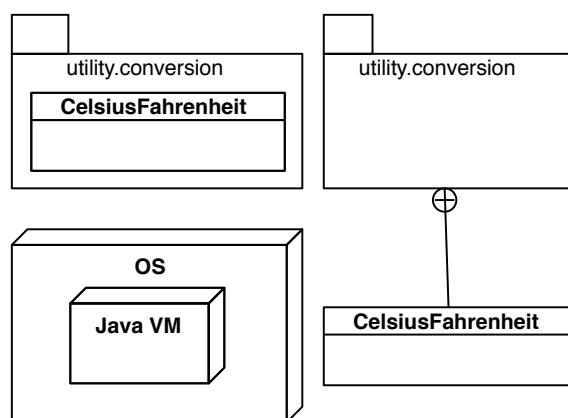
UML: entità strutturali

- Definiscono le "cose" (*classificatori, things*) del modello
- Alcuni esempi: classi, classi attive, use-case, collaborazioni, interfacce



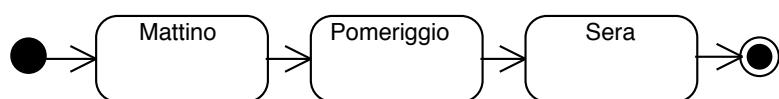
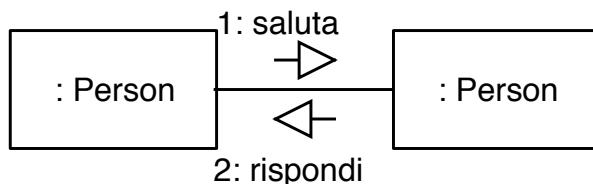
UML: entità di raggruppamento e contenimento

- I *package* raggruppano altri elementi e forniscono loro un *namespace* (che permette poi di identificare ogni elemento con il suo nome).
- In UML, moltissimi elementi possono contenere altri elementi al loro interno, formando una struttura gerarchica, rappresentabile graficamente in vari modi.



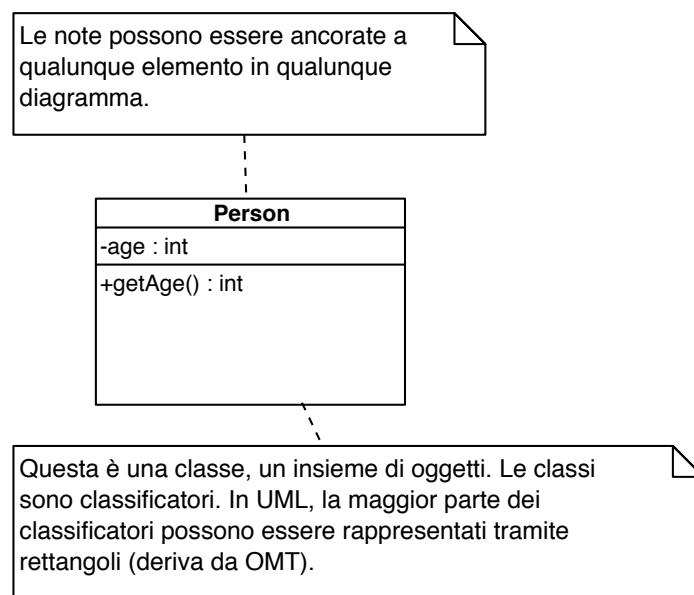
UML: entità comportamentali

- Descrivono il "behavior": interazioni, collaborazioni (communication), scambi di messaggi, transizioni di stato, etc.



Entità informative

- Uno degli scopi principali della modellazione è la leggibilità: un diagramma non leggibile e informativo serve a poco...
- Le *note UML* non hanno effetti sul modello ma migliorano la leggibilità.



Relazioni

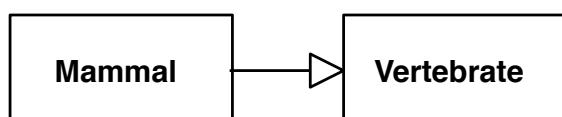
- Gli elementi del modello possono essere collegati da **relazioni**.
- Rappresentate graficamente tramite linee.
- Possono avere un nome.
- Quattro sottotipi fondamentali:
 - ▶ **Association**
 - ▶ **Generalization**
 - ▶ **Dependency**
 - ▶ **Realization**

Association e generalization

- Un'associazione descrive l'esistenza di un nesso tra le istanze di classificatori (*things*) e ha varie caratteristiche, alcune opzionali.

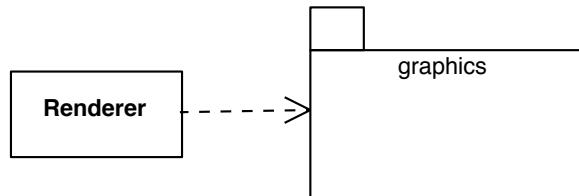


- La generalizzazione è una relazione tassonomica da un elemento specializzato verso un altro, più generale, dello stesso tipo.
 - ▶ Il figlio è sostituibile al genitore dovunque appaia, e ne condivide struttura e comportamento.

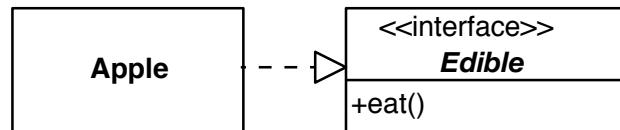


Dipendenze e realizzazioni

- Una dipendenza è una relazione semantica: indica che il *client* dipende, semanticamente o strutturalmente, dal *supplier* (variazioni alla specifica del supplier possono cambiare quella del client).



- Anche la realizzazione è una relazione semantica: il supplier fornisce una specifica, il client la realizza (es: implementazione di interfacce, templates, etc.)



Le frecce in UML

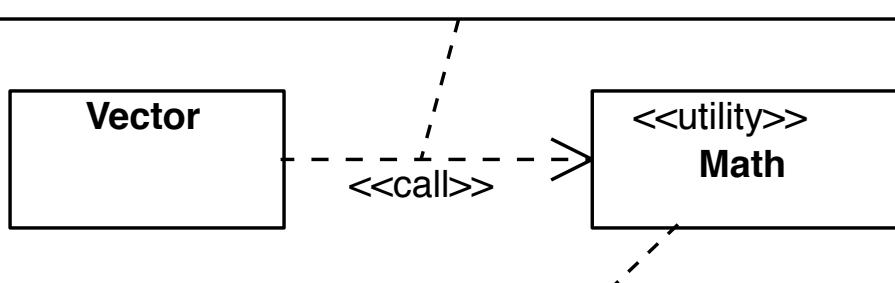
- Un metodo mnemonico per ricordarsi il verso di tutte le frecce in UML è il seguente:
- In UML, tutte le frecce vanno **da chi sa verso chi non sa** (dell'esistenza dell'altro).
- In una generalizzazione, il figlio sa di estendere il genitore, ma il genitore non sa di essere esteso.
- In una dipendenza, chi dipende sa da chi dipende, ma non vice-versa.
- In una realizzazione, chi implementa conosce la specifica, ma non il contrario.

Stereotipi

- Un'altra primitiva di UML comune ad ogni diagramma.
- Rende un diagramma più informativo arricchendo la semantica dei costrutti UML.
- Uno stereotipo è una parola chiave tra virgolette e abbinata ad un elemento del modello.
- Es. «import», «utility», «interface».

Stereotipi in un diagramma delle classi

Questa dipendenza tra Vector e Math ha lo stereotipo «call»; significa che operazioni di Vector invocano operazioni di Math.



Lo stereotipo «utility» indica che questa è una classe utility, cioè una collezione di variabili globali e operazioni statiche usate da altre parti del sistema.

Stereotipi come estensioni di UML

- Gli stereotipi forniscono significato aggiuntivo ai costrutti UML.
- Possono essere usati per adattare UML a particolari ambiti e piattaforme di sviluppo.
- Stereotipi, vincoli e regole aggiuntivi vengono raccolti in *profili*, che costituiscono uno dei principali meccanismi di estensione di UML.

OCL (1)

- **Object Constraint Language** (OCL) è un linguaggio, approvato e standardizzato da OMG, per la specifica di vincoli. Si usa assieme ad UML e a tutti i linguaggi dell'architettura MOF.
- Non è obbligatorio, ma aggiunge rigore formale al modello.
- Specifica condizioni che devono essere soddisfatte dalle istanze di una classe: i vincoli sono espressioni booleane considerate *true*.
- Un tool UML può usare OCL
 - ▶ per validare il modello
 - ▶ per generare codice

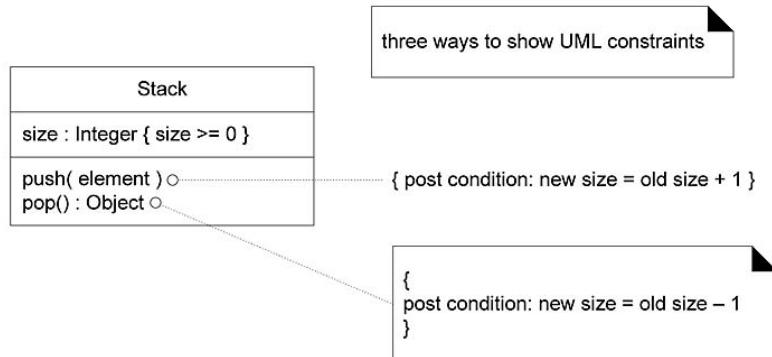
OCL (2)

- Un **vincolo** OCL opera in un determinato **contesto**, specificando proprietà soddisfatte da tutte le istanze di quel contesto.
- Il contesto può essere una classe, un suo attributo o una sua operazione.
- I vincoli hanno un tipo che descrive l'ambito della loro **validità**.
- **context Car inv:**
`fuel>=0`
- Questo vincolo si applica alla classe Car ed è un invarianto (sempre valido).

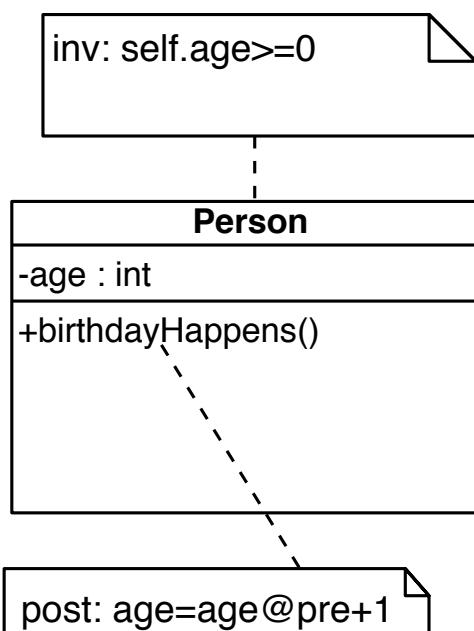
Tipi di vincoli in OCL

- **inv:** Invariante, sempre valido nel contesto.
- **pre:** Nel contesto di un'operazione, una precondizione per la sua esecuzione.
- **post:** Nel contesto di un'operazione, una postcondizione vera dopo l'esecuzione.
- **body:** Definisce una query nel contesto.
- **init:** Definisce il valore iniziale nel contesto.
- **derive:** Definisce un attributo derivato del contesto.

OCL nei diagrammi (1)



OCL nei diagrammi (2)



(In una postcondizione, '@pre' permette di accedere al valore precedente di un attributo.)

Conclusioni

- Modellare è indispensabile in qualunque progetto non banale.
- UML è un linguaggio general-purpose per la modellazione.
- Non è perfetto, ma è potente e costituisce uno standard diffuso ed accettato.
- Costruire un buon modello è difficile.

Alcuni tool

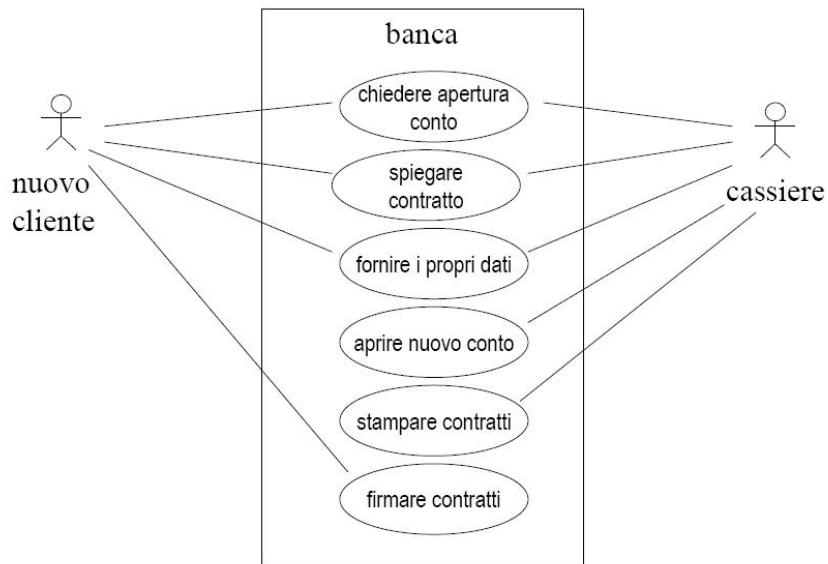
- *ArgoUML*: <http://argouml.tigris.org/>
- *Papyrus*: <http://eclipse.org/papyrus>
- *Umbrello*: <http://uml.sourceforge.net/>
- *Eclipse*: www.eclipse.org/uml2/
- ...

Il diagramma dei casi d'uso

Il diagramma dei casi d'uso

- Si tratta di un diagramma che esprime un comportamento, desiderato o offerto.
- L'oggetto esaminato è solitamente un sistema o una sua parte
- Individua:
 - ▶ **chi** o che cosa ha a che fare con il sistema (**attore**)
 - ▶ **che cosa** l'attore può fare (**caso d'uso**).
- Tipicamente è il primo tipo di diagramma ad essere creato in un processo o ciclo di sviluppo, nell'ambito dell'analisi dei requisiti.

Un esempio: conto in banca



I casi d'uso

- I casi d'uso esistono da prima dei diagrammi UML!
- Proposti da Ivar Jacobson nel 1992 per il metodo Objectory
- In realtà la tecnica era già consolidata: *studio degli scenari di operatività degli utilizzatori di un sistema*
- In altre parole:
 - ▶ i modi in cui il sistema può essere utilizzato
 - ▶ le funzionalità che il sistema mette a disposizione dei suoi utilizzatori

Casi d'uso e requisiti funzionali

- I **requisiti funzionali** specificano **cosa** deve essere fatto.
- Sono indipendenti dalla tecnologia, dall'architettura, dalla piattaforma, dal linguaggio di programmazione.
- I requisiti **non-funzionali** specificano **vincoli aggiuntivi**, ad esempio:
 - ▶ performance
 - ▶ scalabilità
 - ▶ tolleranza ai guasti
 - ▶ dimensione degli eseguibili
- I **casi d'uso** modellano i **requisiti funzionali**.

Caso d'uso

- **Specifica cosa** ci si aspetta da un sistema
- **Nasconde come** il sistema lo implementa
- E' una sequenza di azioni (con varianti) che producono un risultato **osservabile da un attore**
- Si usa per
 - ▶ Descrivere i requisiti (analisi iniziale)
 - ▶ Convalidare l'architettura e verificare il sistema

Un passo indietro: i desiderata

- I *desiderata* sono ciò che il cliente desidera.
- Formalizzare i **desiderata** in **requisiti** è una delle più importanti sfide aperte dell'ingegneria del software.
- La specifica errata o incompleta delle richieste del cliente è una delle cause principali del fallimento dei progetti software.
- Il diagramma e la modellazione dei casi d'uso aiutano l'interazione con il cliente e migliorano l'estrazione dei requisiti (funzionali).

Esempio di desiderata: vanno bene?

Cliente:

- vorrei vendere i manufatti che realizzo...
- non vorrei solo un mercato locale...
- mi piacerebbe che gli acquirenti potessero visionare un catalogo da cui scegliere...
- vorrei gestire gli ordini da qualunque posto perché viaggio molto...

Che cosa viene fatto? Da chi?

Riformulati meglio...

Cliente:

- vorrei avere la possibilità di creare un catalogo dei miei manufatti...
- vorrei un catalogo liberamente consultabile da chiunque...
- vorrei organizzare i manufatti raccogliendoli in categorie...
- vorrei che gli interessati all'acquisto potessero inviarmi un ordine, che io provvederò ad evadere previa una qualche forma di registrazione...

Estrarre i requisiti

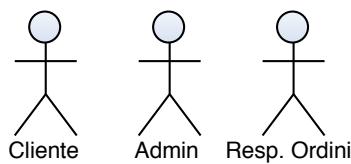
Chi interagisce con il sistema (**attori**)?

- Clienti
- Amministratori del negozio online
- Reparto ordini

Cosa fanno (**casi d'uso**)?

- Il cliente si registra, consulta il catalogo ed effettua acquisti
- L'amministratore organizza il catalogo, che è diviso in categorie
- Il reparto ordini riceve ordini da evadere
- Il cliente sceglie il tipo di pagamento

Attori



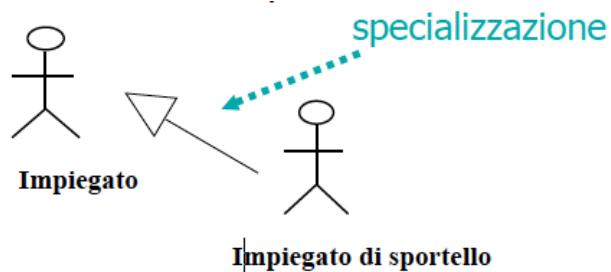
- Un attore specifica un ruolo assunto da un utente o altra entità che interagisce col sistema nell'ambito di un'unità di funzionamento (caso d'uso).
- Un attore è **esterno** al sistema.
- **Non è necessariamente umano:** oggetto fisico, agente software, condizioni ambientali, etc.
- Gli attori eseguono casi d'uso: prima si cercano gli attori, poi i loro casi d'uso!

Come individuare gli attori

- Per individuare un attore è necessario individuare chi/cosa interagisce col sistema e con quale ruolo.
- Domande utili:
 - ▶ Chi/cosa usa il sistema?
 - ▶ Che ruolo ha chi/cosa interagisce col sistema?
 - ▶ In quale parte dell'organizzazione è utilizzato il sistema?
 - ▶ Chi/cosa avvia il sistema?
 - ▶ Chi supporterà e manterrà il sistema?
 - ▶ Altri sistemi interagiscono col sistema?
 - ▶ Ci sono funzioni attivate periodicamente? es. *backup*
 - ▶ Chi/cosa ottiene o fornisce informazioni dal sistema?
 - ▶ Un attore ha diversi ruoli? Lo stesso ruolo è assegnato a più attori?

Specializzazione degli attori

- Gli attori possono essere specializzati e connessi ad altri attori tramite relazioni di *generalization*
- Attori specializzati usano le stesse funzionalità degli attori che generalizzano ed eventualmente alcune funzionalità specifiche.



Caso d'uso UML (1)



- La specifica di una sequenza di azioni, incluse eventuali sequenze alternative e/o di errore che un sistema (o sottosistema) può eseguire interagendo con attori esterni.
- È qualcosa che un attore vuole il sistema faccia:
 - ▶ È descritto dal punto di vista dell'attore
 - ▶ È causato da un'azione di un attore
- Il nome (etichetta) dovrebbe essere basato su un verbo o su un sostantivo che esprime un avvenimento.

Caso d'uso UML (2)

- Un caso d'uso è sempre iniziato da un attore: in UML, un evento è sempre legato all'entità che lo ha generato.
- L'attore che inizia un caso d'uso è detto *primario*, gli altri attori che interagiscono nell'ambito di quel caso d'uso sono *secondari*.
- Un caso d'uso è un *classificatore dotato di comportamento*: può essere specificato da diagrammi di stato, interazione, sequenza, avere pre- e post-condizioni, ...
- Può ammettere al suo interno *varianti* al comportamento principale.

Come individuare un caso d'uso

- Individuare i casi d'uso è un'operazione **iterativa**.
- Per individuare i casi d'uso possono essere utili le seguenti domande:
 - ▶ Ciascun attore che funzioni si aspetta?
 - ▶ Il sistema gestisce (archivia/fornisce) informazioni? Se sì quali sono gli attori che provocano questo comportamento?
 - ▶ Alcuni attori vengono informati quando il sistema cambia stato?
 - ▶ Gli attori devono informare il sistema di cambiamenti improvvisi?
 - ▶ Alcuni eventi esterni producono effetti sul sistema?
 - ▶ Quali casi d'uso manutengono il sistema?
 - ▶ I requisiti funzionali sono tutti coperti dai casi d'uso?

Come recuperare informazioni su un caso d'uso

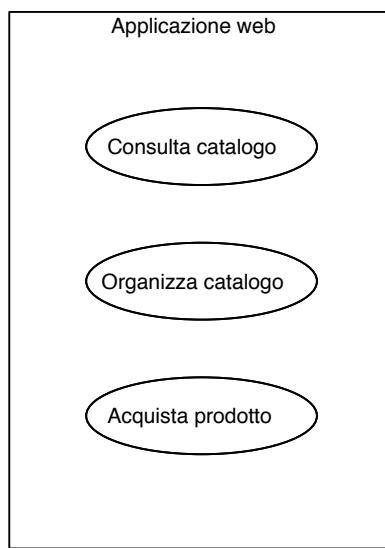
- Interviste con gli esperti del dominio (*desiderata ben strutturati*)
- Bibliografia del dominio del sistema
- Sistemi già esistenti
- Conoscenza personale del dominio

Come descrivere un caso d'uso

- Descrivere in modo generico e sequenziale il flusso di eventi di un caso d'uso
 - ▶ Descrivere la precondizione (stato iniziale del sistema)
 - ▶ Elencare la sequenza di passi
- Descrivere le interazioni con gli attori e i messaggi scambiati
- Aggiungere eventuali punti di estensione
- Descrivere in modo chiaro, preciso e breve

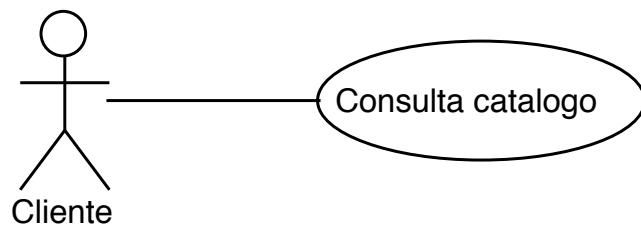
[ci torneremo più avanti]

Elementi del diagramma: sistema



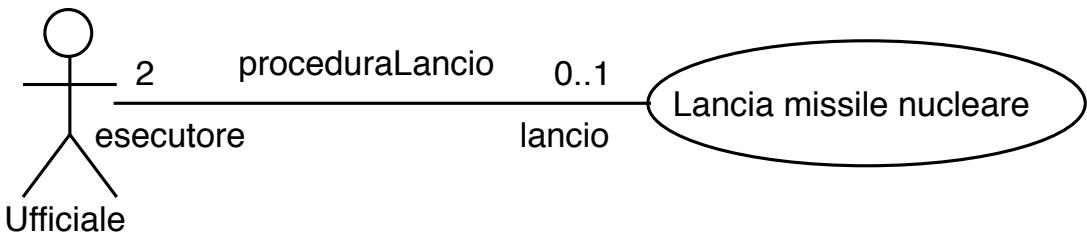
- Delimita l'argomento del diagramma, specificando i confini del sistema.

Elementi del diagramma: associazione (1)



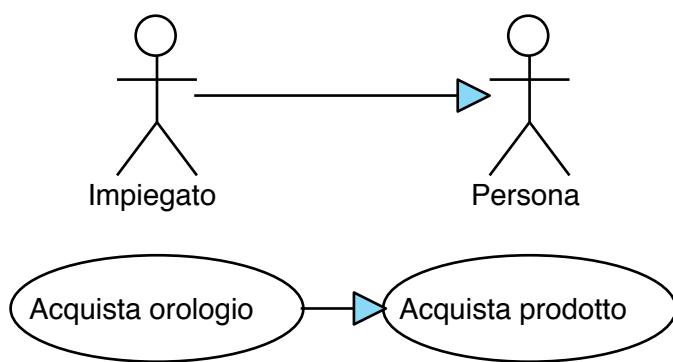
- Collega gli attori ai casi d'uso.
- Un attore si può associare solo a casi d'uso, classi e componenti (che verranno eventualmente associati ad altri casi d'uso, classi e componenti).
- Un caso d'uso non dovrebbe essere associato ad altri casi d'uso riguardanti lo stesso argomento.

Elementi del diagramma: associazione (2)



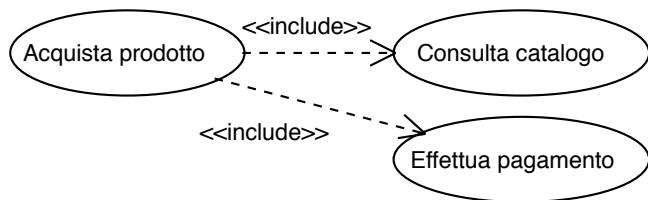
- Alcune caratteristiche opzionali comuni a tutte le associazioni in UML:
 - ▶ nome
 - ▶ molteplicità
 - ▶ ruoli

Elementi del diagramma: generalizzazione



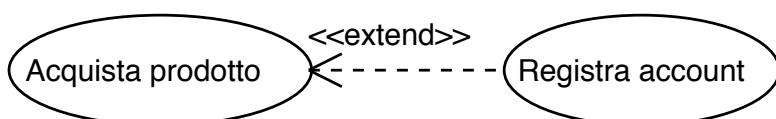
- Collega un attore o caso d'uso ad un altro più generale.
- Il figlio può sostituire il genitore dovunque questi appaia.

Elementi del diagramma: include



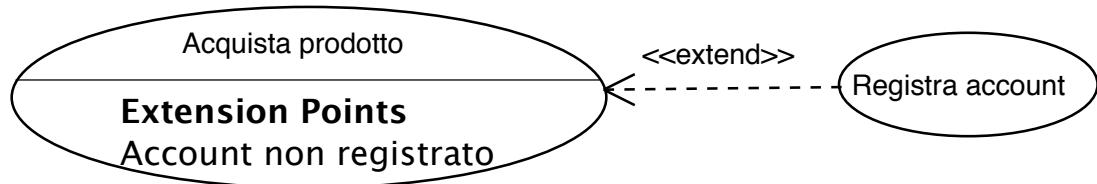
- Una **dipendenza** tra casi d'uso; il caso incluso fa parte del comportamento di quello che lo include.
- L'inclusione **non è opzionale** ed avviene in ogni istanza del caso d'uso.
- La corretta esecuzione del caso d'uso che include dipende da quella del caso d'uso incluso.
- Non si possono formare cicli di include.
- Usato per **riutilizzare parti comuni** a più casi d'uso.

Elementi del diagramma: extend



- Una **dipendenza** tra casi d'uso (notare il verso della freccia).
- Il caso d'uso che estende (client) specifica un incremento di comportamento a quello esteso (supplier).
- Si tratta di comportamento **supplementare** ed **opzionale** che gestisce **casi particolari** o non standard.
- *Diverso* da una generalizzazione tra casi d'uso:
 - ▶ in una generalizzazione, entrambi i casi d'uso sono ugualmente significativi
 - ▶ in un extend, il client non ha necessariamente senso se preso da solo

Extension points

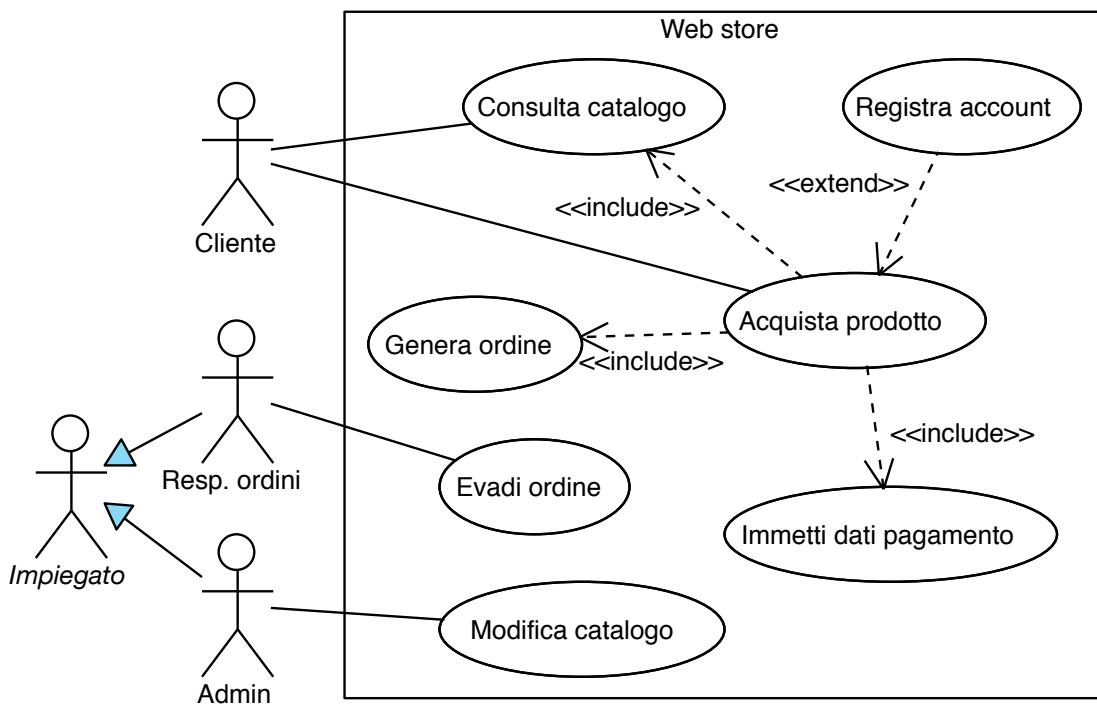


- Un caso d'uso raggiunto da almeno un extend può opzionalmente visualizzare i propri extension points.
- Specifica i **punti** e/o **condizioni** dell'esecuzione in cui il comportamento viene esteso.
- Se gli extension points sono molti, alcuni tool possono supportare la rappresentazione a rettangolo (i casi d'uso sono classificatori).

include vs. extend

- Include specifica comportamento **obbligatorio**.
- Extend specifica comportamento **supplementare** (varianti).
- Nell'include la freccia va dal caso d'uso che include verso quello incluso.
- Nell'extend la freccia va dal caso d'uso che estende verso quello esteso.
- Sono entrambi costrutti utili, ma non se ne deve abusare, o la leggibilità ne risente.

Esempio diagramma



Sono sufficienti questi diagrammi?

- Spesso nasce l'esigenza di abbinare i diagrammi dei casi d'uso a **specifiche testuali** più formali.
- I diagrammi dei casi d'uso non sono adatti a mostrare:
 - ▶ la sequenza temporale delle operazioni
 - ▶ lo stato del sistema e degli attori prima e dopo l'esecuzione del caso d'uso
- Manca la parte più importante: la descrizione (specifica) dei casi d'uso!

Specifiche del caso d'uso

- Ogni caso d'uso ha un **nome** e una **specificità**.
- La specifica è composta da:
 - ▶ **precondizioni:** condizioni che devono essere vere prima che il caso d'uso si possa eseguire
 - ▶ **sequenza degli eventi:** i passi che compongono il caso d'uso
 - ▶ **postcondizioni:** condizioni che devono essere vere quando il caso d'uso termina l'esecuzione

Esempio specifica caso d'uso

<i>Nome del caso d'uso</i>	Caso d'uso: PagamentoIVA
<i>Identificatore univoco</i>	ID: UC1
<i>Gli attori interessati dal caso d'uso</i>	Attori: Tempo, Fisco
<i>Lo stato del sistema prima che il caso d'uso possa iniziare</i>	Precondizioni: 1. Si è concluso un trimestre fiscale
<i>I passi del caso d'uso</i>	Sequenza degli eventi: 1. Il caso d'uso inizia quando si conclude un trimestre fiscale. 2. Il sistema calcola l'ammontare dell'IVA dovuta al Fisco. 3. Il sistema trasmette un pagamento elettronico al Fisco.
<i>Lo stato del sistema quando l'esecuzione del caso d'uso è terminata</i>	Postcondizioni: 1. Il Fisco riceve l'importo IVA dovuto.

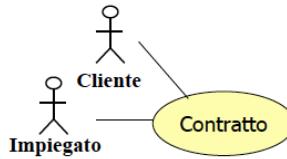
Sequenza degli eventi

- Un **elenco di azioni** che definisce il caso d'uso nella sua **completezza**.
- Il caso d'uso si considera eseguito solo se l'esecuzione arriva fino alla fine.
- Un'azione è sempre iniziata da un attore oppure dal sistema (in UML, gli eventi sono sempre legati a chi li crea).
 - ▶ **Passo iniziale:** 1. Il caso d'uso inizia quando <attore> <azione>...
 - ▶ **Passi successivi:** <numero>. Il <attore/sistema> <azione>

Sequenza (corretta) di eventi

- ~~Incomincia quando si seleziona la funzione 'ordina libro'~~
- Il caso d'uso inizia quando il cliente seleziona la funzione 'ordina libro'
- ~~Vengono inseriti i dati del cliente~~
- Il cliente inserisce nel form il suo nome e indirizzo
- Il sistema verifica i dati del Cliente

Un altro esempio di sequenza di eventi



- **Nome:** Contratto
- **Precondizione:** l'impiegato è connesso
- **Flusso principale degli eventi**
 - ▶ L'impiegato inserisce il nome cliente e numero di conto
 - ▶ Controlla la loro validità
 - ▶ Inserisce il numero di azioni da comprare e ID azienda quotata
 - ▶ Il sistema determina il prezzo
 - ▶ Il sistema controlla il limite
 - ▶ Il sistema manda l'ordine in Borsa
 - ▶ L'impiegato memorizza il numero di conferma

Flusso degli eventi

- Ogni caso d'uso:
 - ▶ Ha una sequenza di transizioni (eventi) normale o di base
 - ▶ Può avere varie sequenze alternative
 - ▶ Ha varie sequenze di transazioni eccezionali per la gestione di errori o casi particolari
- Per descrivere correttamente il flusso di eventi quindi:
 - ▶ Descrivere solo gli eventi relativi al caso d'uso, e non quel che avviene in altri casi d'uso
 - ▶ Descrivere come e quando il caso d'uso inizia e finisce
 - ▶ Descrivere il flusso base degli eventi
 - ▶ Descrivere ogni flusso alternativo

E in UML?

- UML usa parole chiave per esprimere queste variazioni alla sequenza principale.
- Parola chiave **Se**: indica una ramificazione della sequenza degli eventi nel caso si verifichi una condizione.
- Ripetizioni all'interno di una sequenza:
 - ▶ parola chiave **Per** (For)
 - ▶ parola chiave **Fintantoché** (While)
- È bene non eccedere con le ramificazioni!

Ramificazioni e sequenze alternative

- Facciamo un esempio partendo dal caso d'uso 'aggiorna carrello' di un negozio on-line.
- Possibili ramificazioni, dopo aver aggiunto un articolo al carrello:
 - ▶ se il cliente richiede una nuova quantità il sistema aggiorna la quantità di quell'articolo
 - ▶ se il client seleziona rimuovi articolo il sistema elimina quell'articolo dal carrello
- Le sequenze alternative sono invece ramificazioni che non possono essere espresse utilizzando il **Se**.
 - ▶ Ad esempio dovute a condizioni che si possono verificare in un qualunque momento: il cliente abbandona la pagina del carrello

Esempio ramificazioni

Caso d'uso: AggiornaCarrello
ID: UC2
Attori: Cliente
Precondizioni: 1. Il contenuto del carrello è visibile
Sequenza degli eventi: 1. Il caso d'uso inizia quando il Cliente seleziona un articolo nel carrello. 2. Se il Cliente seleziona "rimuovi articolo" 2.1 Il Sistema elimina l'articolo dal carrello. 3. Se il Cliente digita una nuova quantità 3.1 Il Sistema aggiorna la quantità dell'articolo presente nel carrello
Postcondizioni: 1. Il contenuto del carrello è stato aggiornato
Sequenza alternativa 1: 1. In qualunque momento il Cliente può abbandonare la pagina del carrello
Postcondizioni:

Come individuare sequenze alternative?

- Ad ogni passo della sequenza degli eventi principale, cercare:
 - ▶ alternative all'azione eseguita in quel passo
 - ▶ errori possibili nella sequenza principale
 - ▶ interruzioni che possono avvenire in qualunque momento della sequenza principale
- Sequenze alternative abbastanza complesse possono essere descritte separatamente.
 - ▶ stessa sintassi, si sostituisce 'caso d'uso' con 'sequenza degli eventi alternativa'
 - ▶ il primo passo può indicare il punto della sequenza principale da cui si proviene

Esempio: apertura conto corrente

- ① il cliente si presenta in banca per aprire un nuovo c/c
- ② l'addetto riceve il cliente e fornisce spiegazioni
- ③ se il cliente accetta fornisce i propri dati
- ④ l'addetto verifica se il cliente è censito in anagrafica
- ⑤ l'addetto crea il nuovo conto corrente
- ⑥ l'addetto segnala il numero di conto al cliente

Varianti:

- ③ (a) se il cliente non accetta il caso d'uso termina
- ③ (b) se il conto va intestato a più persone vanno forniti i dati di tutte
- ④ (a) se il cliente (o uno dei diversi intestatari) non è censito l'addetto provvede a registrarlo

Esempio: apertura conto corrente (2)

Il punto 5 (crea conto) può essere ulteriormente dettagliato:

- ① l'addetto richiede al sistema la transazione di inserimento nuovo conto
- ② il sistema richiede i codici degli intestatari
- ③ l'addetto fornisce i codici al sistema
- ④ il sistema fornisce le anagrafiche corrispondenti, e richiede le condizioni da applicare al conto
- ⑤ l'addetto specifica le condizioni e chiede l'inserimento
- ⑥ il sistema stampa il contratto con il numero del conto

Varianti:

- ③ (a) se il sistema non riconosce il cliente, o se fornisce un'anagrafica imprevista, l'addetto può effettuare correzioni o terminare l'inserimento

Scenari

- Uno **scenario** rappresenta una particolare interazione tra uno o più attori e il sistema.
- Non contiene ramificazioni o sequenze alternative: è semplicemente la **cronaca di un'interazione** vera o verosimile.
 - ▶ il concetto risulta più immediato pensando agli attori in maniera concreta: non un generico cliente, ma Alice o Bob
- Una definizione alternativa: un caso d'uso è un insieme di scenari possibili se un utente prova a raggiungere un dato risultato

Scenario principale e scenari secondari

- Corrispondono ad esecuzioni della **sequenza principale** e di quelle **alternative**, rispettivamente.
- Strategie per limitare il numero, potenzialmente enorme, degli scenari secondari:
 - ▶ documentare solo quelli considerati più importanti
 - ▶ se ci sono scenari secondari molto simili, se ne documenta uno solo, se necessario aggiungendo annotazioni per spiegare come gli altri scenari differiscano dall'esempio.

Un esempio completo: conto in banca

- Vedi PDF allegato.

Errori tipici sui diagrammi

- Diagrammi di flusso invece di casi d'uso: un caso d'uso è una sequenza di azioni, non una singola azione!
- Nome del caso d'uso che appare più volte nel diagramma
- Le frecce tra i casi d'uso non sono tratteggiate (- - - - - >) o etichettate «*extend*» o «*include*»
- «*extend*»: la freccia va dal caso che descrive l'evento alternativo al caso standard
- «*include*»: la freccia va dal caso chiamante al caso che descrive le azioni da includere

Errori tipici sugli scenari

- Assenza di precondizioni
- Mancata connessione alla rappresentazione grafica
- Nomi diversi per le stesse entità nelle rappresentazioni grafica e testuale
- Flusso eccezionale: mancanza di indicazioni nel flusso principale del punto in cui va controllata la condizione eccezionale

Riassumendo

- Analizzare il materiale contenente i requisiti
- Creare un glossario di progetto con parole chiave e una breve descrizione
- Capire chi sono gli attori
- Estrarre i casi d'uso più evidenti
- Cominciare ad organizzare i casi d'uso in un diagramma
- Modellare i casi d'uso come sequenze di eventi
- Raffinare progressivamente se necessario

Conclusioni

- Il diagramma UML dei casi d'uso è un tool per la modellazione del comportamento di un sistema.
- Descrive gli attori che interagiscono con il sistema, cosa fanno, e cosa ottengono dal sistema.
- A questo punto non interessa sapere come il sistema fornisca il comportamento richiesto.

Esercizi

Esercizio gestione biblioteca

- Viene richiesto un sistema che permetta al bibliotecario e a un utente di effettuare ricerche di libri. Il bibliotecario deve poter effettuare il prestito e gestire la restituzione del libro. Un utente deve restituire il libro entro una certa data. Se il prestito risulta scaduto per la prima volta il sistema emette un avviso, se è la seconda volta il bibliotecario registra e stampa una multa. L'utente a questo punto può decidere se pagare la multa subito oppure no. Il sistema deve permettere la registrazione del pagamento.
- Disegnare il diagramma dei casi d'uso e descrivere le sequenze relative.

Esercizio gestione del personale

- Viene richiesto un sistema che permetta la gestione del personale di un'azienda. Per accedere alle operazione bisogna autenticarsi. Le operazioni possibili saranno la modifica dei dati dell'impiegato, la semplice visualizzazione dei suoi dati, e la cancellazione dei dati.
- Disegnare il diagramma dei casi d'uso e descrivere le sequenze relative.

Esercizio Ricerca di un prodotto

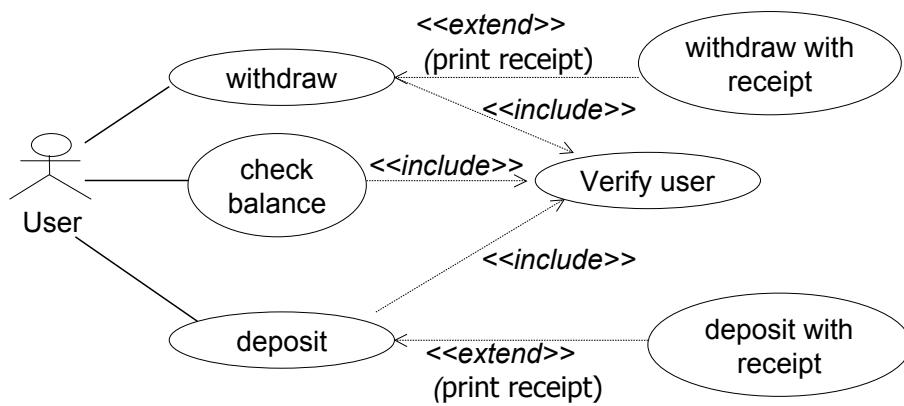
- Una libreria ha la necessità di un sistema che le permetta di far effettuare ricerche al suo personale. All'interno della libreria vengono venduti anche dvd video, e cd musicali.
- Disegnare il diagramma dei casi d'uso e descrivere le sequenze relative.

Esempio casi d'uso UML

Esempio: gestione conto corrente

- Il sistema usa uno sportello Bancomat
- L'utente deve poter depositare assegni
- L'utente deve poter ritirare contante
- L'utente deve poter chiedere il saldo
- L'utente deve poter ottenere una ricevuta se lo richiede. La ricevuta riporta il tipo di transazione, la data, il numero di conto, la somma, ed il nuovo saldo
- Dopo ciascuna transazione viene visualizzato il nuovo saldo

Soluzione



Soluzione

Use case: withdraw

Precondition: User has selected withdraw option

Main flow:

- Include (Verify user)
- Prompt user for amount to withdraw
- Check available funds of user
- Check available money of ATM
- Remove amount from account
- Give money
- (print receipt)
- Print current balance

Exceptional flow

- If not sufficient funds or money available, prompt user for lower amount

Soluzione

Use case: deposit

Precondition: User has selected deposit option

Main flow:

- Include (Verify user)
- Prompt user for amount of deposit
- Open slot
- Get check
- (print receipt)
- Print (balance + deposited amount)

Soluzione

Use case: check balance

Precondition: User has selected balance option

Main flow:

- Include (Verify user)
- Print balance

Soluzione

Use case: verify user

Precondition: none

Main flow:

- User enters ID card
- User enters PIN number
- System checks validity of card and number

Exceptional flow:

- If combination is not valid, reject user

Soluzione

Use case: withdraw with receipt

Precondition: User has selected withdraw option and print receipt option

...

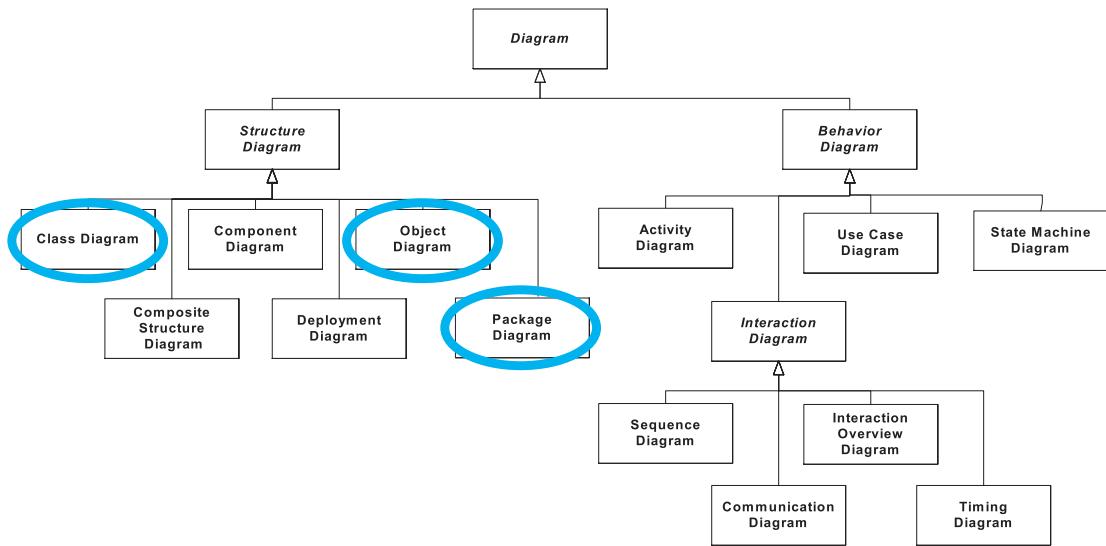
Use case: deposit with receipt

Precondition: User has selected deposit option and print receipt option

...

I diagrammi di struttura

Tassonomia dei diagrammi UML 2



A che punto siamo

- Conosciamo i casi d'uso di un sistema
- Abbiamo steso una specifica dei requisiti
- Abbiamo sequenze di eventi
- Abbiamo un glossario di termini di progetto

Fasi di sviluppo

- Requisiti (decidere cosa deve essere fatto)
- **Analisi (dare struttura ai requisiti)**
- **Progettazione (decidere come implementare il sistema analizzato)**
- Implementazione
- Test

Analisi

- Si trovano le **entità** coinvolte
- Si studiano i legami tra le entità (**relazioni** tra le entità)
- Si sta individuando la struttura del sistema

Analisi vs. Progettazione

- L'analisi **modella** i concetti chiave del **dominio** del problema.
- La progettazione **adatta** il modello di analisi e lo **completa** affinché diventi **implementabile**.

In altre parole...

- L'analisi è vicina al problema.
- La progettazione è vicina alla soluzione.

Dal punto di vista di UML, non si usano primitive o diagrammi diversi, ma gli stessi tipi di diagramma con **diversi livelli di dettaglio** (i diagrammi di analisi sono più ‘astratti’ di quelli di progettazione).

Classi e oggetti

Costituiscono i ‘mattoni’ della struttura di un sistema.

- **Oggetto:** Entità discreta, con confini ben definiti, che incapsula stato e comportamento; un’istanza di una classe.
- **Classe:** Descrittore di un insieme di oggetti che condividono gli stessi attributi, operazioni, metodi, relazioni e comportamento.

Uno dei primi compiti dell’analisi è quello di modellare il dominio del problema in **classi di analisi**.

Queste verranno poi trasformate in **classi di progettazione** adatte all’implementazione.

Come procedere: analisi

- Estrarre un insieme di classi di analisi dalla specifica del problema (ne parleremo tra poco)
- Ragionare su queste **classi**: quali **attributi** e quali **operazioni** devono fornire?
- Stendere una mappa delle classi e delle loro **relazioni**.
- Modellare la dinamica delle classi con i *diagrammi di comportamento*.
- Procedere per **raffinamenti successivi** fino a quando il modello rappresenta efficacemente il dominio del problema.

Come procedere: progettazione

- Si parte dal modello di analisi che contiene classi abbastanza generiche, e lo si raffina.
- I costrutti più **astratti** di UML vengono trasformati in altri più **concreti** che possono essere implementati in un linguaggio di programmazione OO.
- Finalmente si considerano i **vincoli di piattaforma** e **linguaggio**, e i requisiti **non funzionali**.
- Le classi di analisi si trasformano in classi di progettazione (non c'è corrispondenza 1 a 1)
- Ancora una volta si procede per **raffinamenti successivi**.
- Il risultato è un modello pronto per l'implementazione.

Come estrarre le classi di analisi

- Una classe di analisi modella un concetto o entità del problema: se la specifica dei casi d'uso è buona i concetti basilari sono già in evidenza.
- I candidati più probabili sono nomi che compaiono nella specifica e nella documentazione.
- Una ragione in più per tenere un glossario di progetto: le parole nel glossario sono spesso candidati ideali per diventare classi di analisi.
- Le classi di analisi non sopravviveranno necessariamente alla progettazione.
- Due metodi molto diffusi per trovare le classi di analisi:
 - ▶ analisi nome-verbo
 - ▶ analisi CRC

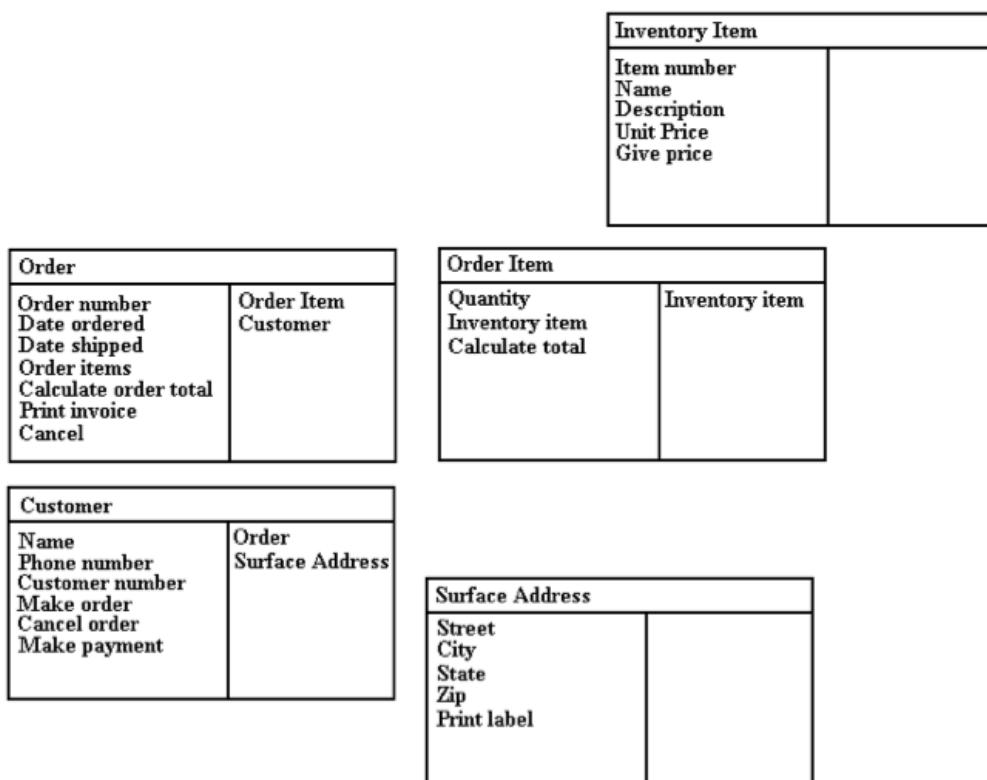
Analisi nome-verbo

- Si analizza tutta la documentazione disponibile, selezionando nomi e verbi.
 - ▶ I **nomi**: (es: conto corrente) sono i potenziali candidati per divenire classi o attributi.
 - ▶ I **predicati nominali**: (es: numero del conto corrente) sono i potenziali candidati per divenire classi o attributi.
 - ▶ I **verbi**: (es: aprire) sono potenziali candidati a divenire responsabilità di classe.
- Notate che ancora non parliamo di UML!

Analisi CRC

- Class-Responsibilities-Collaborators
- Si usano post-it divisi in tre sezioni chiamate proprio in questo modo.
- Si tratta di un metodo di brainstorming di gruppo che coinvolge sviluppatori, esperti, committenti.
- Si individuano i nomi delle classi, un insieme ristretto di responsabilità (cose che la classe sa/fa) e di classi collaboratori (alle quali viene richiesto comportamento/informazione).
- Le schede sono piazzate su un tavolo, la loro vicinanza fisica rispecchia quella logica.
- Si procede iterativamente.
- Usato in congiunzione con analisi nome-verbo.

Esempio CRC



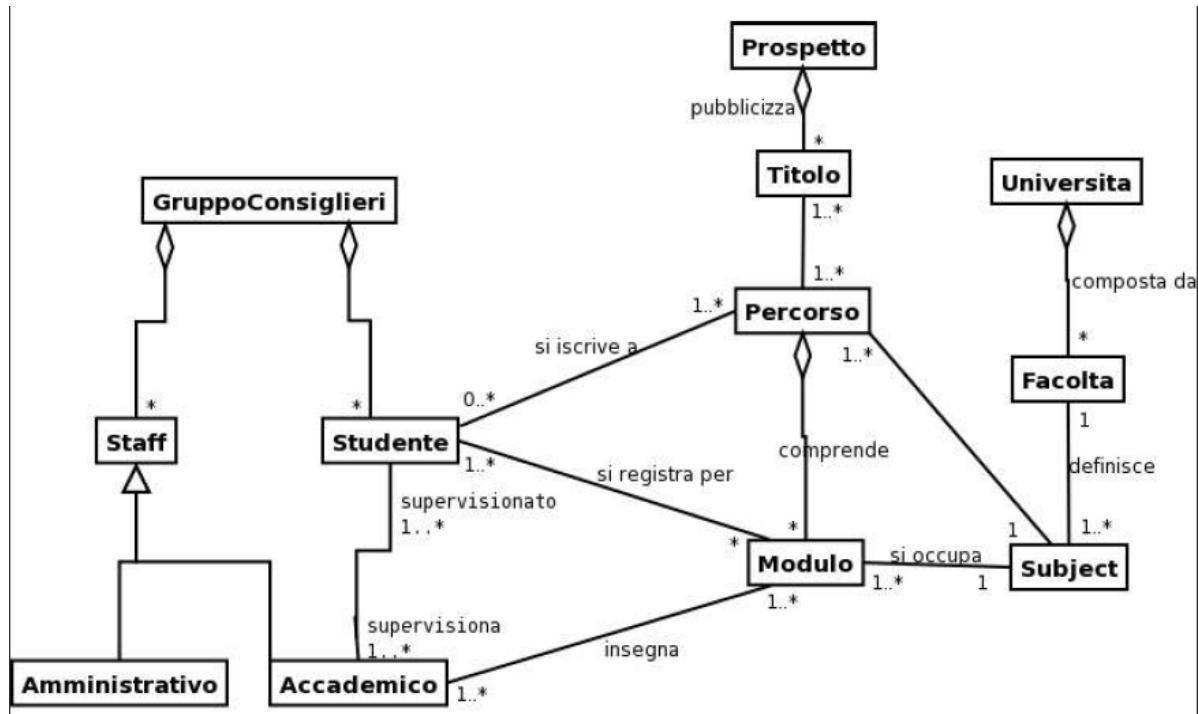
Esempio Classi di Analisi (1/2)

- De Montfort University (DMU) offre corsi di laurea modulari, e altri tipi di percorsi formativi ciascuno dei quali porta al conseguimento di un titolo di riconoscimento.
- Ogni titolo di riconoscimento è pubblicizzato nel prospetto informativo di DMU
- Ogni percorso comprende differenti moduli
- Gli studenti di un percorso seguono fino a 8 moduli all'anno
- Alcuni titoli sono ‘congiunti’, ad esempio uno studente può iscriversi a due differenti percorsi (come ‘contabilità’ e ‘ragioneria’)

Esempio Classi di Analisi (1/2)

- De Montfort University (DMU) offre corsi di laurea modulari, e altri tipi di **percorsi formativi** ciascuno dei quali porta al conseguimento di un **titolo** di riconoscimento.
- Ogni titolo di riconoscimento è pubblicizzato nel **prospetto** informativo di DMU
- Ogni percorso **comprende** differenti **moduli**
- Gli **studenti** di un percorso **seguono** fino a 8 moduli all'anno
- Alcuni titoli sono ‘congiunti’, ad esempio uno studente **può iscriversi** a due differenti percorsi (come ‘contabilità’ e ‘ragioneria’)

Soluzione De Montfort University



Esempio Classi di Analisi (2/2)

- La DMU è composta di 6 Facoltà
- Ogni facoltà definisce un numero di soggetti ('contabilità', 'ragioneria', etc.) ciascuno dei quali si occupa di differenti percorsi.
- Il consiglio di Facoltà è composto da studenti e da staff accademico o amministrativo
- Lo staff accademico insegna un numero arbitrario di moduli
- Lo staff accademico supervisiona diversi studenti, ciascuno dei quali segue un percorso formativo
- Alcuni rappresentanti dello staff amministrativo sono consiglieri ma non insegnano

Esempio Classi di Analisi (2/2)

- La DMU è composta di 6 Facoltà
- Ogni facoltà definisce un numero di soggetti ('contabilità', 'ragioneria', etc.) ciascuno dei quali si occupa di differenti percorsi.
- Il consiglio di Facoltà è composto da studenti e da staff accademico o amministrativo
- Lo staff accademico insegna un numero arbitrario di moduli
- Lo staff accademico supervisiona diversi studenti, ciascuno dei quali segue un percorso formativo
- Alcuni rappresentanti dello staff amministrativo sono consiglieri ma non insegnano

Una buona classe di analisi?

Le qualità di una buona classe di analisi:

- il suo nome ne rispecchia l'intento
- è un'astrazione ben definita, che modella uno specifico elemento del dominio del problema
- corrisponde ad una caratteristica ben identificabile del dominio
- ha un insieme ridotto e definito di responsabilità
- ha la massima coesione interna
- ha la minima interdipendenza con altre classi

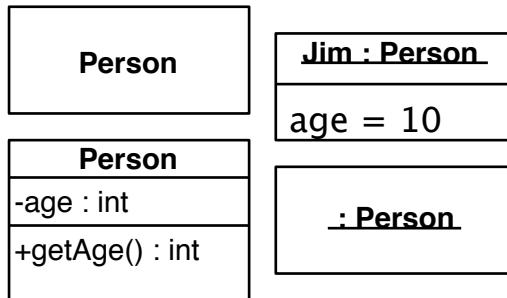
Qualche regola pratica

- 3-5 responsabilità per classe
- non isolata, collabora con un piccolo insieme di altre classi
- evitare il proliferare di classi troppo semplici
- evitare la concentrazione del modello in poche classi complesse
- evitare troppi livelli di ereditarietà
- tenere in mente i principi object-oriented

Classi di analisi vs. classi di progettazione

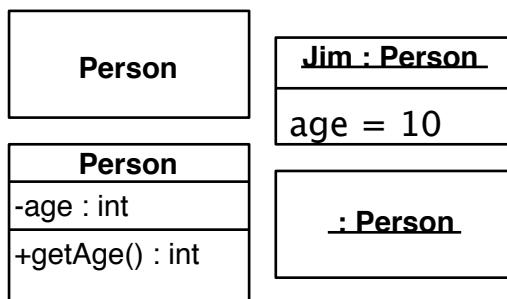
- Classi di analisi:
 - ▶ contengono tipicamente solo pochi attributi e operazioni (candidati)
 - ▶ a volte basta indicare solo il nome della classe
 - ▶ pochi ornamenti, specifica incompleta (omessi tipi, signature delle operazioni, etc.)
- Classi di progettazione:
 - ▶ modellano una classe del linguaggio di programmazione scelto
 - ▶ specifica completa (implementabile)

E in UML?



- Le classi possono avere fino a 3 slot:
 - ▶ uno per il nome (in UpperCamelCase) e l'eventuale stereotipo (slot obbligatorio)
 - ▶ uno per gli attributi (opzionale)
 - ▶ uno per le operazioni (opzionale)
- La stessa classe può apparire con diverse quantità di ornamenti in diagrammi diversi.

Classi e oggetti in UML



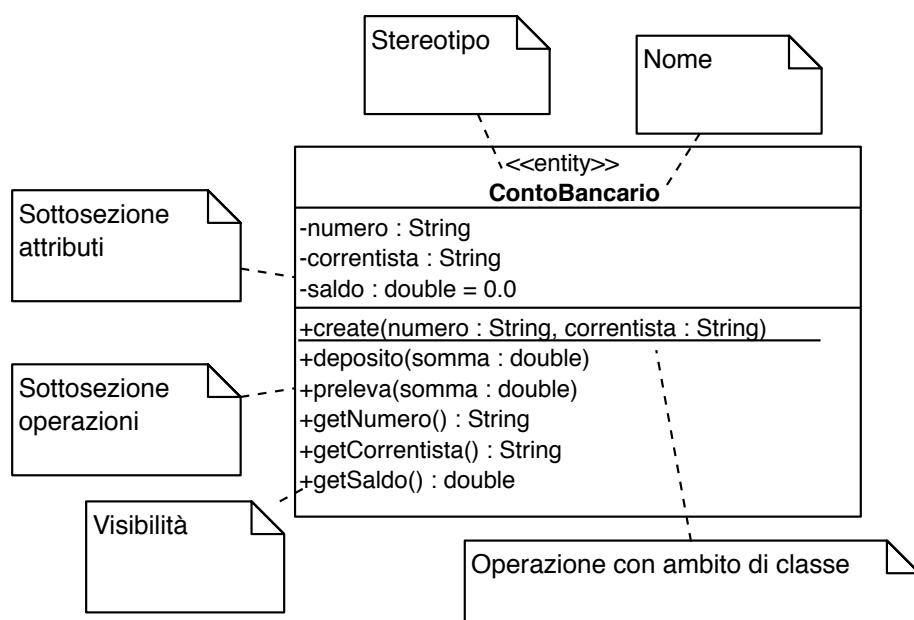
- Le istanze delle classi (oggetti) hanno una notazione molto simile
- Il titolo degli oggetti è sottolineato e del tipo 'nome : classe', con nome opzionale.
- Gli oggetti non hanno uno slot per le operazioni, possono definire valori per gli attributi.

Relazione tra classe e oggetto



Si tratta di una dipendenza con stereotipo «instanceOf» (o «istanzia» sul libro).

Classe: notazione



Nota: questo livello di dettaglio è tipico delle classi di progettazione, non quelle di analisi.

Attributi

visibilità nome molteplicità:tipo=valoreIniziale

- Solo il nome è obbligatorio
- Le **classi di analisi** solitamente contengono solo gli **attributi più importanti** (quelli che risultano evidenti dall'analisi del dominio); spesso specificano solo il nome di un attributo.
- Assegnare un valore iniziale in una classe di analisi può evidenziare i vincoli di un problema.
- Le **classi di progettazione** forniscono una **specifica completa (implementabile)** della classe e dei suoi attributi.

Tipi di visibilità

+ <i>public</i>	ogni elemento che può accedere alla classe può anche accedere a ogni suo membro con visibilità pubblica
- <i>private</i>	solo le operazioni della classe possono accedere ai membri con visibilità privata
# <i>protected</i>	solo le operazioni appartenenti alla classe o ai suoi discendenti possono accedere ai membri con visibilità protetta
~ <i>package</i>	ogni elemento nello stesso package della classe (o suo sottopackage annidato) può accedere ai membri della classe con visibilità package

Operazioni

visibilità nome (nomeParametro:tipoParametro, ...):
tipoRestituito

- Valgono le stesse considerazioni fatte a proposito degli attributi per quanto riguarda analisi e progettazione.
- In ogni classe non ci possono essere due operazioni con la stessa signature.
- Il nome di attributi ed operazioni è scritto in lowerCamelCase.

Attributi, operazioni e visibilità

Point
x : Integer
y : Integer

+ User
-id : long ~name : String #enabled : Boolean #emailAddress : String #password : String
+enable() : void +isValidPassword(password : String) : Boolean +isEnabled() : Boolean +changePassword(newPassword : String) : void

Dalle classi al codice (PHP)

```
<?php

// Short description of class Point
class Point{

    public $x;

    public $y;

    // Short description of method moveTo
    public function moveTo( Integer $newX, Integer $newY)
    {
        ...
    }

} // End of class Point
?>
```

Dalle classi al codice (Java)

```
public class User {

    private Long id;

    String name;

    protected Boolean enabled;

    protected String emailAddress;

    protected String password;

    ...

}
```

Dalle classi al codice (Java)

```
public class User {  
    ...  
  
    public void enable() {  
        ...  
    }  
  
    public Boolean isValidPassword(String password) {  
        return null;  
    }  
  
    public Boolean isEnabled() {  
        return null;  
    }  
  
    public void changePassword(String newPassword) {  
        ...  
    }  
}
```

Come si esprime in UML?

```
class Studente extends Persona {  
  
    private String name;  
    private String cognome;  
    protected SchedaImmatricolazione immatricolazione;  
  
    public String getName() {  
        ...  
    }  
  
    public void setName(String name) {  
        ...  
    }  
  
    public SchedaImmatricolazione getImmatricolazione() {  
        ...  
    }  
}
```

Ambito

- Ambito di istanza:
 - ▶ gli oggetti hanno una propria copia degli attributi, quindi oggetti diversi possono avere diversi valori negli attributi
 - ▶ Le operazioni agiscono su oggetti specifici
- Ambito di classe:
 - ▶ gli oggetti di una stessa classe condividono lo stesso valore per un attributo
 - ▶ le operazioni non operano solo su una particolare istanza della classe, ma alla classe stessa. Ad esempio: costruttori e distruttori di classe.
 - ▶ rappresentato sottolineando l'attributo/operazione
 - ▶ analogo alla parola chiave *static* in Java

Ambito e accessibilità

- Operazioni con ambito di istanza possono accedere sia ad altre operazioni o attributi con ambito di istanza sia con ambito di classe.
- Operazioni con ambito di classe possono accedere solo ad altre operazioni e attributi con ambito di classe (altrimenti non si saprebbe quale istanza scegliere).
- Valgono le usuali restrizioni di visibilità.

Esempio di ambito di istanza/classe (1)

- Definiamo una classe *Somma* responsabile di sommare due interi
- Usiamo attributi e metodi con ambito di classe per contare quante somme sono state eseguite.

+ Somma
<u>+numSommeEseguite : Integer</u>
-a : Integer
-b : Integer
<u>+getSomma() : Integer</u>
<u>+loadIntegers(a : Integer,b : Integer)</u>
<u>-EseguitaSomma()</u>

Esempio di ambito di istanza/classe (2)

```
public class Somma {  
  
    public static Integer numSommeEseguite = 0;  
    private Integer a;  
    private Integer b;  
  
    public void loadIntegers(int a, int b) {  
        this.a = a; this.b = b;  
    }  
  
    public Integer getSomma() {  
        EseguitaSomma();  
        return a + b;  
    }  
  
    private static void EseguitaSomma() {  
        numSommeEseguite = numSommeEseguite + 1;  
    }  
}
```

Esempio di ambito di istanza/classe (3)

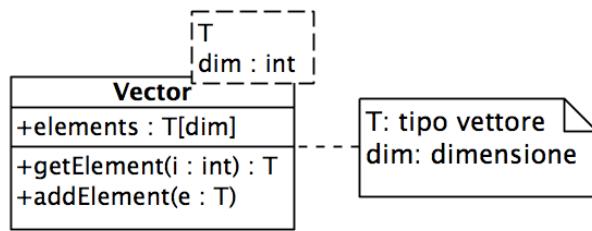
```
Somma s = new Somma();  
  
s.loadIntegers(1, 1);  
s.getSomma(); // da stampare o  
// salvare in una variable!  
  
s.loadIntegers(8, 8);  
s.getSomma();  
  
Somma s2 = new Somma();  
  
s2.loadIntegers(5, 5);  
s2.getSomma();  
  
System.out.println("Hai eseguito " +  
                    Somma.numSommeEseguite +  
                    " somme");
```

Quante somme sono state eseguite?

Template di classe

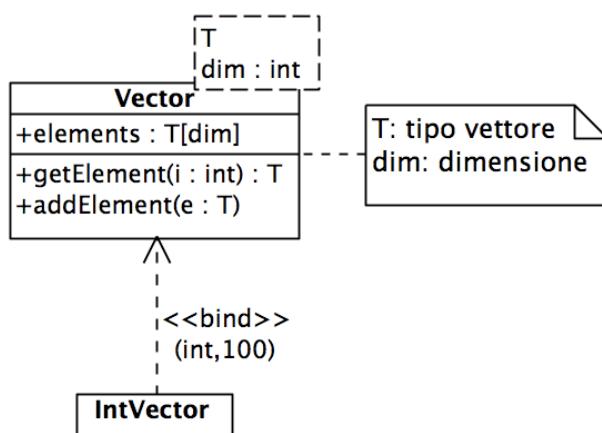
- Un potente strumento di modellazione che non è limitato alle classi.
- Permette di parametrizzare i tipi che compaiono nella specifica di una classe.
- Perché definire come classi diverse un vettore di integer, un vettore di float, un vettore di string, etc.?
- Conviene molto di più definire un 'vettore di X', e sostituire X con quello che serve di volta in volta!

Notazione dei template



- I parametri del template sono indicati in un rettangolo tratteggiato.
- Chiaramente servono primitive per istanziare classi basate sui template, sostituendo valori ai parametri.
- Questa operazione si chiama *binding*.

Binding esplicito



Si usa lo stereotipo «bind», seguito dai parametri di binding.

Binding implicito

```
Vector<int,100>
```

Si usa il nome della classe seguito dai parametri di binding.

- Pro: più compatto.
- Contro: la classe non ha un suo nome proprio (`IntVector` nel binding esplicito).

Relazioni tra classi

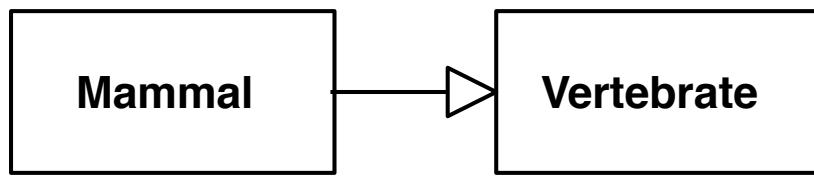
Ci sono due relazioni statiche tra classi particolarmente importanti in UML:

- Generalizzazione
- Associazione

Vi sono poi altre due relazioni che possono legare le classi anche ad altri tipi di elementi:

- Dipendenza
- Realizzazione

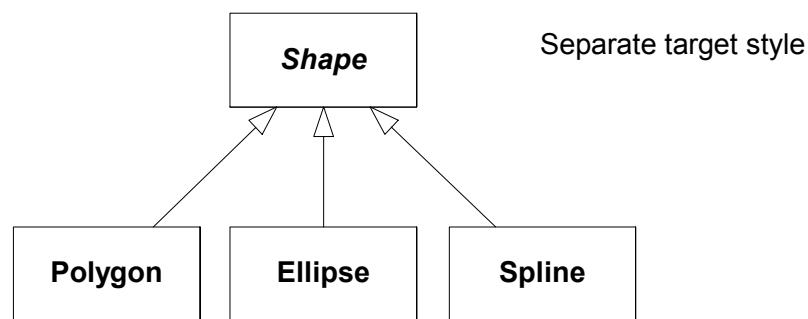
Generalizzazione



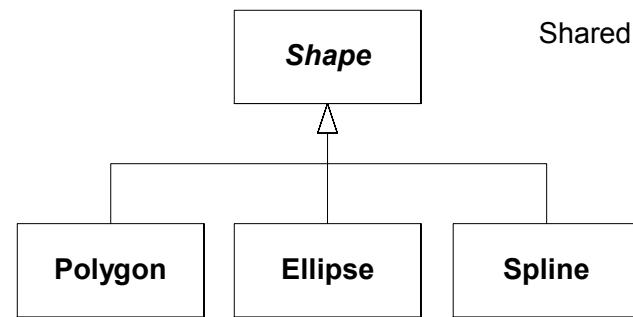
- Relazione tassonomica tra un elemento più generale e uno che lo specifica.
- La freccia parte dall'elemento specifico e punta verso quello più generale.
- Si tratta dell'ereditarietà in UML.
- Tra tutte le relazioni, questa è la più forte e vincolante.

Generalizzazione: stili grafici

Separate target style



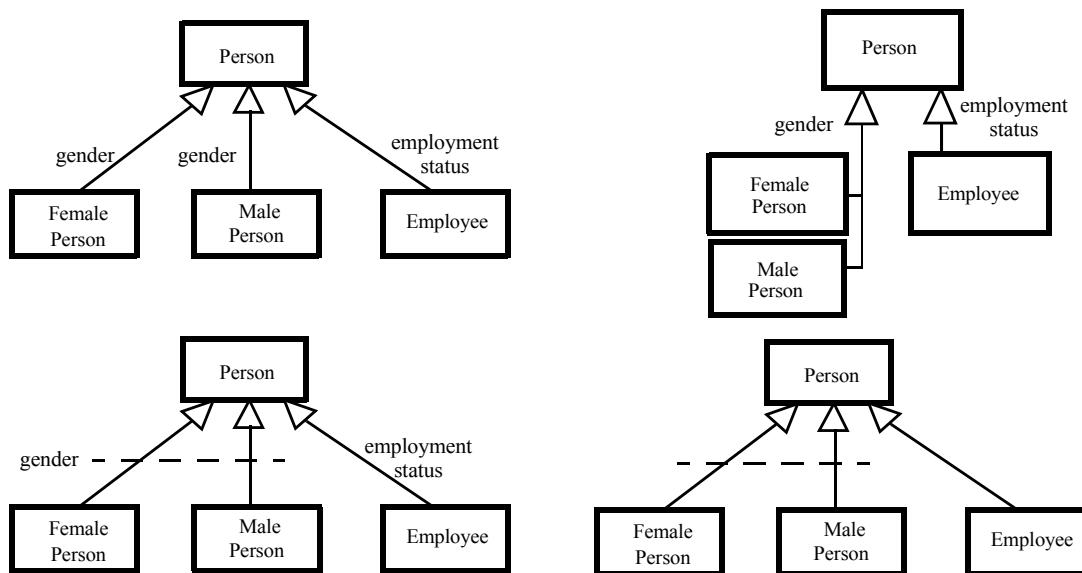
Shared target style



Generalizzazione: da UML al codice (Java)

```
public class Shape {  
    ...  
}  
  
public class Polygon extends Shape {  
    ...  
}  
  
public class Ellipse extends Shape {  
    ...  
}  
  
public class Spline extends Shape {  
    ...  
}
```

Insiemi di generalizzazione

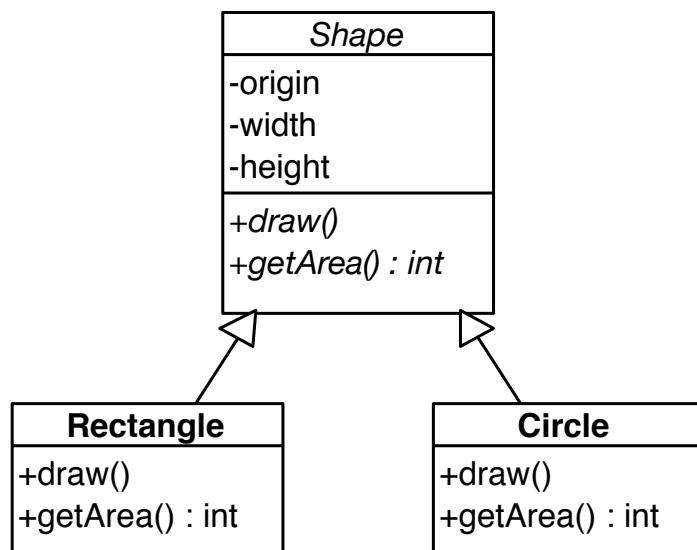


Permettono di partizionare lo stesso elemento generale in modi diversi.

Classi astratte

- In UML, un classificatore può essere dichiarato *astratto*: significa che non è istanziabile.
- I classificatori astratti si riconoscono per il nome in corsivo.
- Le classi possono definire operazioni astratte (in corsivo), di cui non è data la specifica.
- Una classe con almeno un'operazione astratta è automaticamente una classe astratta.
- Una classe astratta non è istanziabile... ma i suoi figli possono esserlo.

Figli di classi astratte



I figli forniscono il comportamento definito come astratto in Shape.

Polimorfismo

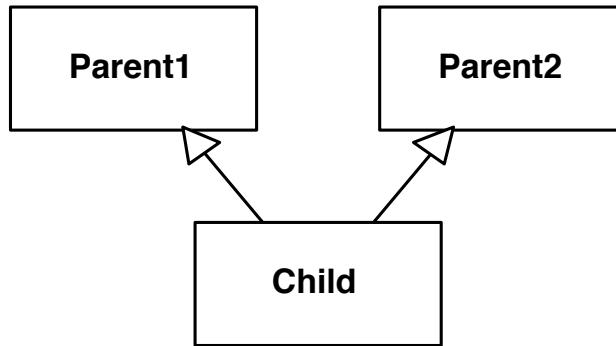
- I figli completano la specifica che il genitore ha volutamente lasciato incompleta.
- Figli diversi forniscono comportamenti diversi alla stessa chiamata di funzione.
- Comportamento diverso di operazioni con la stessa signature è una delle definizioni di *polimorfismo*, uno dei principi cardine del metodo object-oriented.
- Se i figli non ridefiniscono le operazioni astratte, devono rimanere anch'essi astratti, o il modello non è valido.

Esercizio su Polimorfismo

Disegnare il diagramma delle classi della seguente specifica:

- In un ristorante è possibile richiedere i piatti elencati nel menù.
- I piatti vengono tutti preparati ma ciascuno ha un modo diverso per essere cucinato.
- I piatti presenti nel menù sono:
 - ▶ Pasta condita con uno dei sughi disponibili
 - ▶ Sughi: bolognese, siciliana, matriciana
 - ▶ Secondi piatti: frittata, caprese, carne alla griglia
 - ▶ Contorni: verdure grigliate, impanate, fritte
 - ▶ Dolci: mascarpone, torta di pinoli, torta della nonna
- I piatti vengono preparati da un cuoco.

Ereditarietà multipla



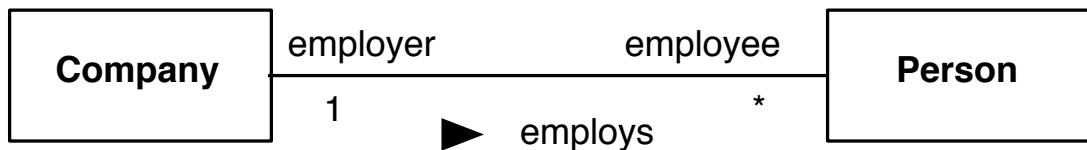
- UML supporta l'ereditarietà multipla: un classificatore può ereditare da un numero arbitrario di altri classificatori.
- A tempo di analisi questo permette di modellare efficacemente situazioni del mondo reale.
- Tuttavia, classi con ereditarietà multipla dovranno essere risolte a tempo di progettazione se il linguaggio scelto non la supporta.

Associazione



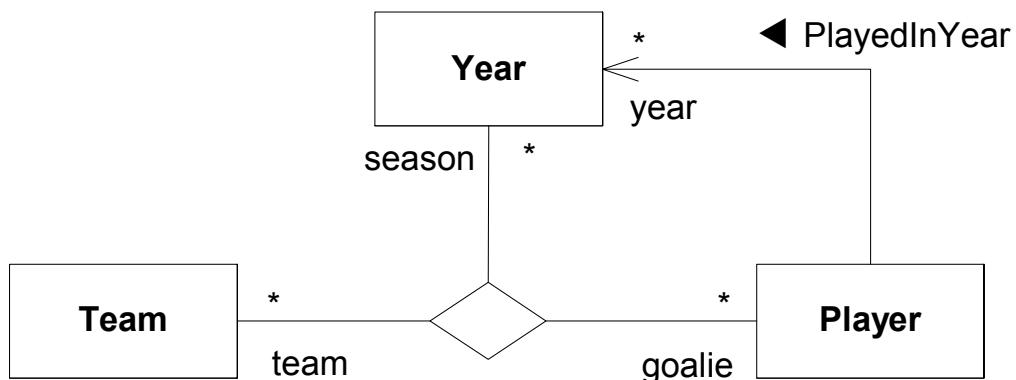
- Si tratta del tipo di relazione più generico: indica solo l'esistenza di collegamenti (link) tra le istanze delle classi.
- Rappresenta l'abilità di un'istanza di mandare messaggi a un'altra istanza.
- Formalmente, definisce l'esistenza di tuple tra istanze tipate (se o_1 è associato a o_2 , la coppia ordinata (o_1, o_2) fa parte della relazione).
- Può coinvolgere più di due classi e la stessa classe più di una volta.
- Tra le relazioni è anche la più flessibile e la meno vincolante.

Associazione: alcuni ornamenti



- Nome: opzionale.
- Triangolo direzionale: opzionale. Specifica la direzione in cui leggere l'associazione (aumenta la leggibilità).
- Ruoli: opzionali a ciascun estremo.
- Molteplicità: opzionale a ciascun estremo.

Associazioni n-arie



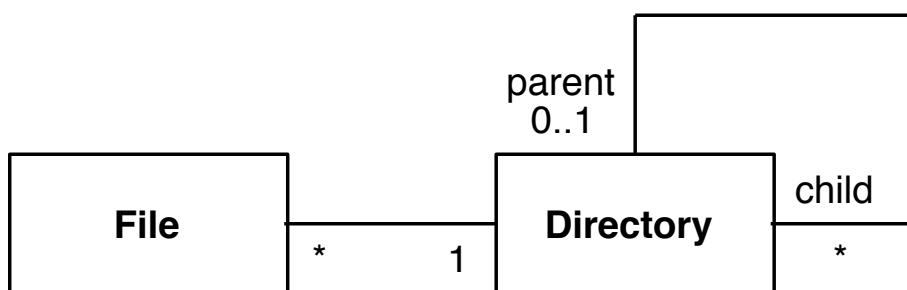
- Si usa un rombo per congiungere i vari estremi; in questo caso non abbiamo coppie ma triple di elementi.
- Il triangolo direzionale si può usare solo nelle relazioni binarie.
- Molti ritengono che le relazioni non binarie siano superflue.

Molteplicità



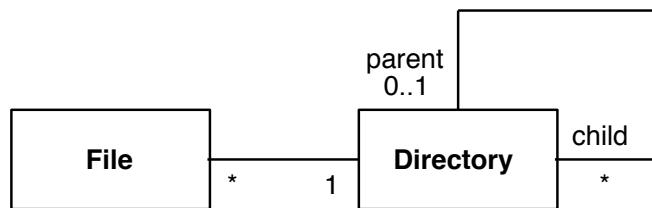
- In una relazione n-aria, indica quante istanze della classe in quell'estremo possono partecipare alla relazione dopo aver fissato gli altri n-1 estremi.
- Può essere un numero o un intervallo min..max, con * che indica l'infinito.
- 1..3,7 significa 'da 1 a 3 oppure 7'.
- Molteplicità frequenti sono:
 - ▶ 1
 - ▶ 0..1
 - ▶ 1..*
 - ▶ *

Associazioni riflessive

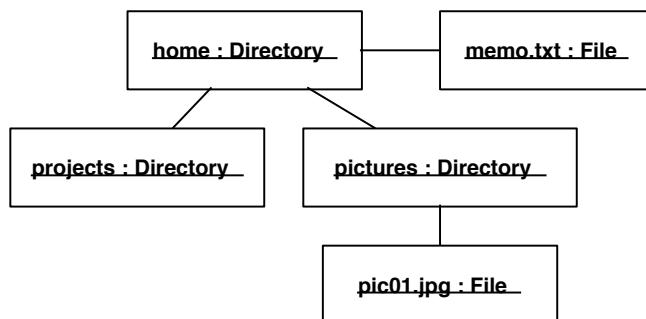


- Un'associazione riflessiva coinvolge la stessa classe più di una volta.
- Esempio: gerarchia di file system.
- Se si sostituisce la molteplicità 0..1 di parent con *, si generano grafi invece di alberi.

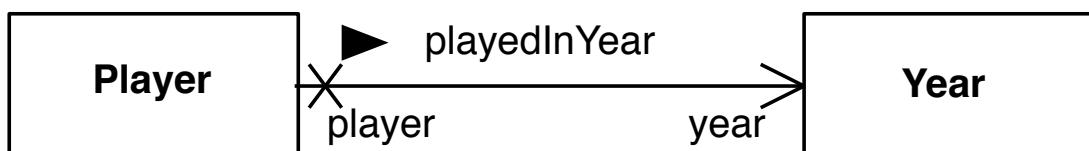
Associazioni riflessive e istanze



Una possibile realtà che soddisfa l'associazione...

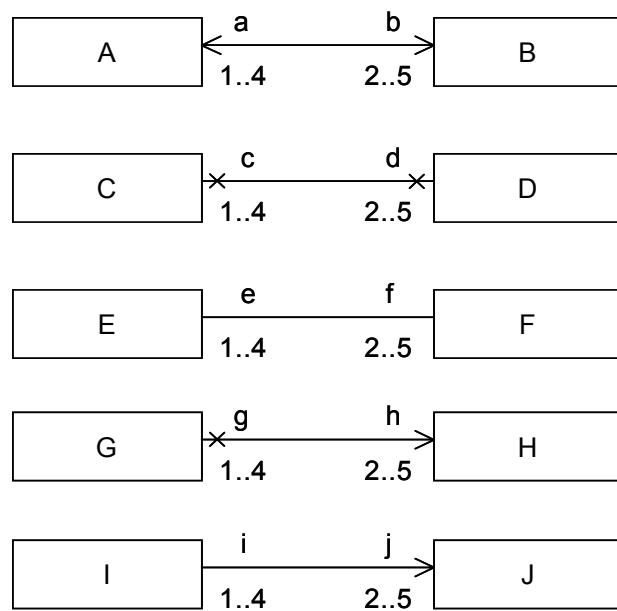


Navigabilità (1)



- Specifica se gli altri estremi dell'associazione **possono sapere** a quali istanze sono associati.
- La freccia indica navigabilità.
- La croce indica assenza di navigabilità.
- La mancanza di entrambe significa navigabilità non specificata (tipico della fase di analisi).
- Un oggetto di tipo Player sa in quali anni ha giocato, un oggetto di tipo Year non sa quali giocatori giocarono quell'anno.

Navigabilità (2)

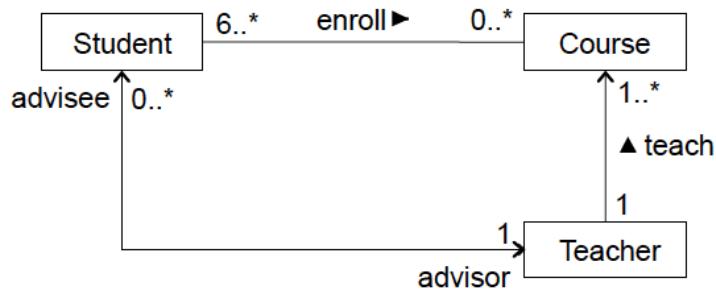


Alcuni esempi di navigabilità.

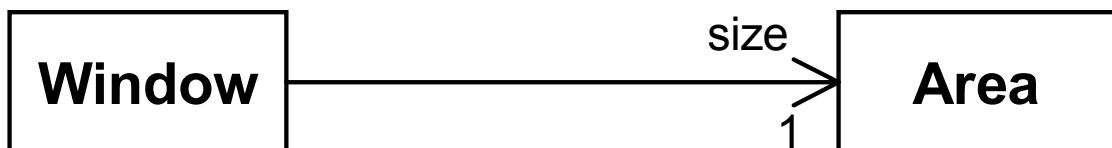
Notazione pratica per la navigabilità

- In pratica, si tende a non specificare la navigabilità di ogni estremo di ogni relazione.
- Un metodo diffuso è il seguente:
 - ▶ non si usano le croci
 - ▶ l'assenza di frecce indica navigabilità in entrambe le direzioni
 - ▶ una freccia indica navigabilità in quella direzione e assenza di navigabilità nell'altra
- Doppia navigabilità risulta indistinguibile da navigabilità non specificata, ma non è un problema in pratica

Un esempio completo

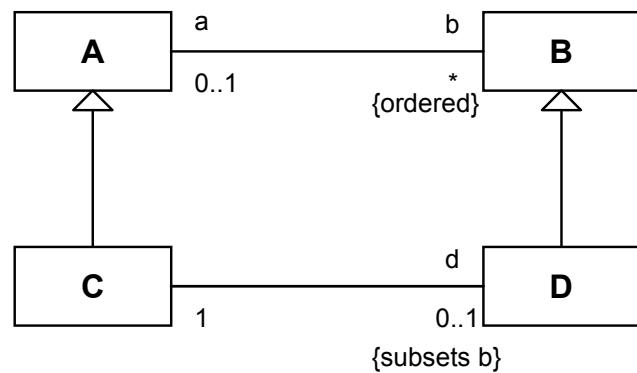


Attributi come associazioni



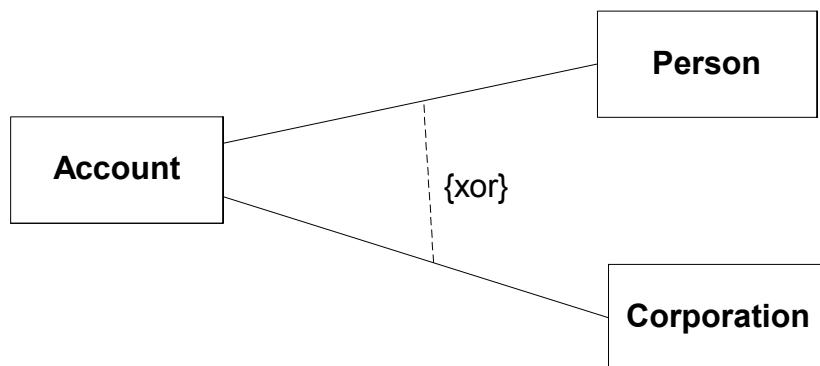
- UML permette di rappresentare un attributo di una classe con la notazione tipica di un'associazione.
- Il nome dell'attributo compare come ruolo insieme alla sua molteplicità.
- Non ci sono ornamenti dalla parte della classe che contiene l'attributo.

Vincoli di associazione



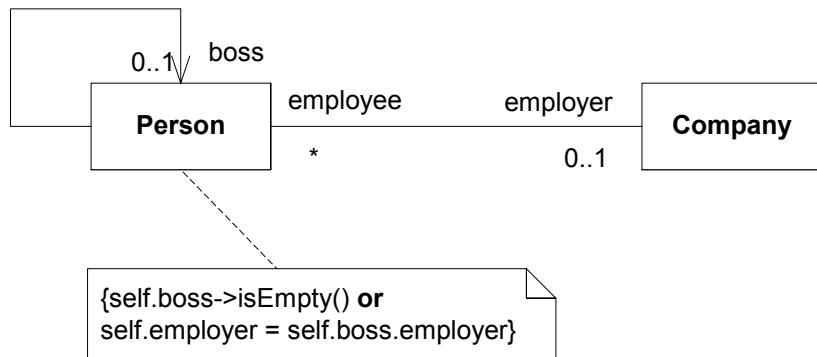
- Possono comparire tra parentesi graffe vicino all'estremo di una relazione.
- Specificano informazioni aggiuntive sull'insieme di istanze che partecipano alla relazione.
- Utili per catturare vincoli del problema durante l'analisi.

Il vincolo xor



- Un particolare vincolo che esprime una scelta mutualmente esclusiva.
- Un conto è associato a una persona fisica oppure a una persona giuridica.

Altri vincoli

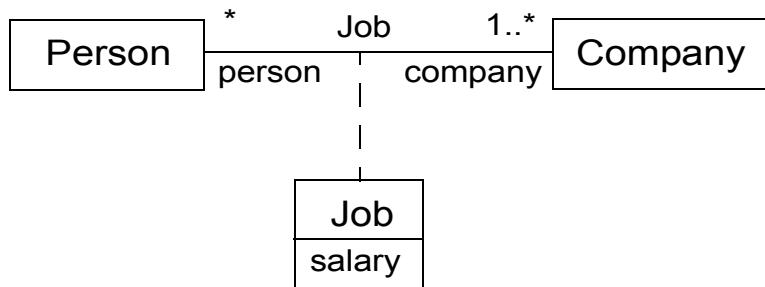


- Possono essere inclusi in note ed espressi in molti modi: OCL, linguaggi di programmazione, linguaggio naturale, ...
- Ogni impiegato, se ha un capo, lavora per la stessa compagnia del suo capo.

Classi di associazione (1)

- Spesso può capitare di avere un'associazione tra classi e la necessità di aggiungere attributi e comportamento che non sembrano adatti a nessuna delle due classi.
- Ad esempio, si considerino le classi Person e Company...
 - ▶ dove inserire l'attributo 'salary'?
 - ▶ non ha senso parlare di salario per un disoccupato
 - ▶ un salario non sembra un attributo adatto a definire la classe Company
 - ▶ questo attributo non si riferisce alla persona o alla compagnia, ma al rapporto che esiste tra una persona e una compagnia
- Si usano le **classi di associazione**.

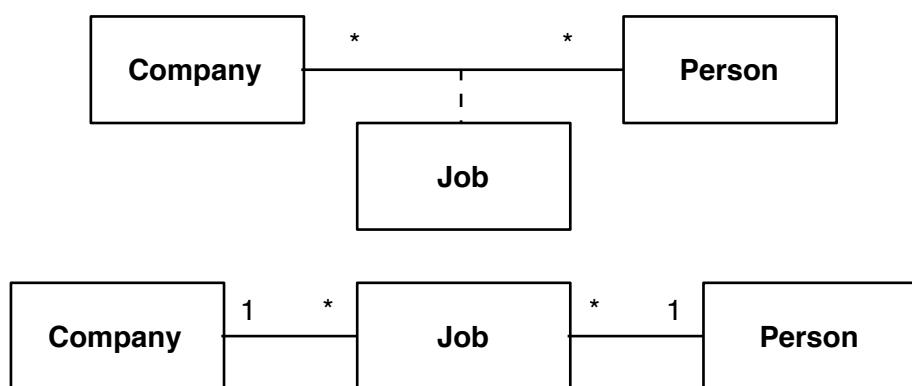
Classi di associazione (2)



- Si tratta di una classe a tutti gli effetti, chiamata come la relazione.
- Collegata alla relazione tramite una linea tratteggiata.
- Possiede tutte le proprietà delle classi e quelle delle associazioni.
- La classe di associazione è definita univocamente dai suoi estremi.

Reificare le classi di associazione

- Le classi di associazione sono un ottimo strumento di analisi... ma nella progettazione?
- Trasformarle in una forma implementabile significa *reificarle*.
- Si spezza l'associazione in due e si trasforma in una normale classe.



Aggregazione e composizione

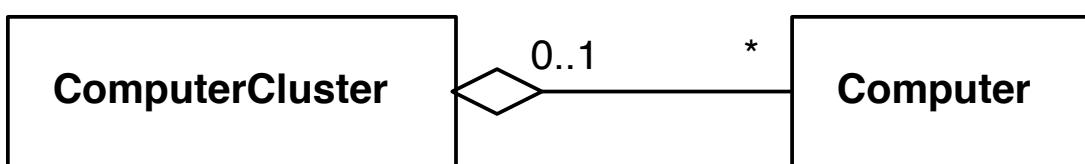
Si tratta di particolari forme di associazione che rappresentano la relazione *whole-part* (tutto-parte) tra un aggregato e le sue parti.

- **Aggregazione:** relazione poco forte (le parti esistono anche senza il tutto; es. i computer e il loro cluster).
- **Composizione:** relazione molto forte (le parti dipendono dal tutto e non possono esistere al di fuori di esso; es. le stanze e la casa).

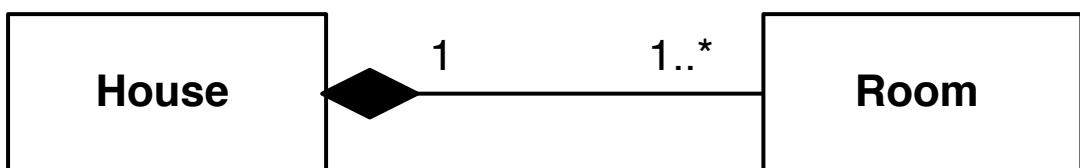
Non è sempre semplice capire quale delle due modella meglio una situazione.

Aggregazione e composizione: notazione

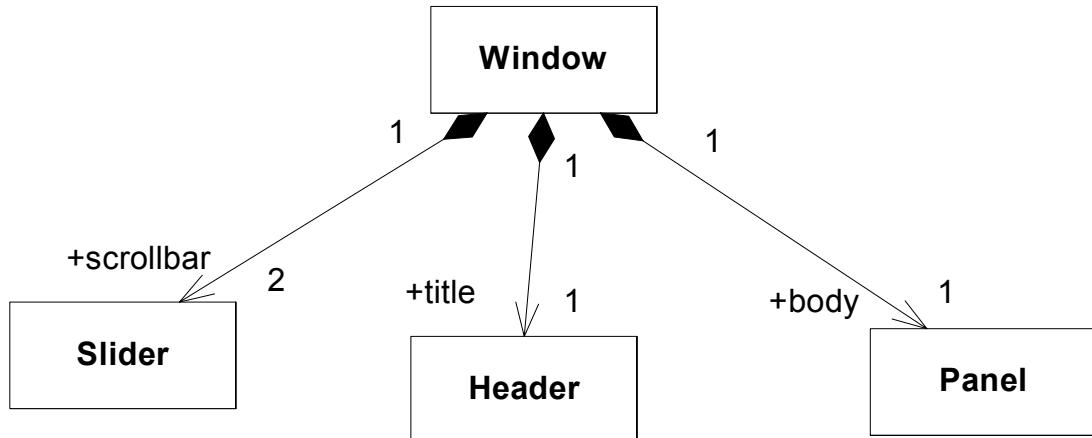
Aggregazione



Composizione



Ancora sulla notazione



Aggregazione e composizione possono essere combinate con le altre notazioni per le associazioni.

Aggregazione: semantica (1)

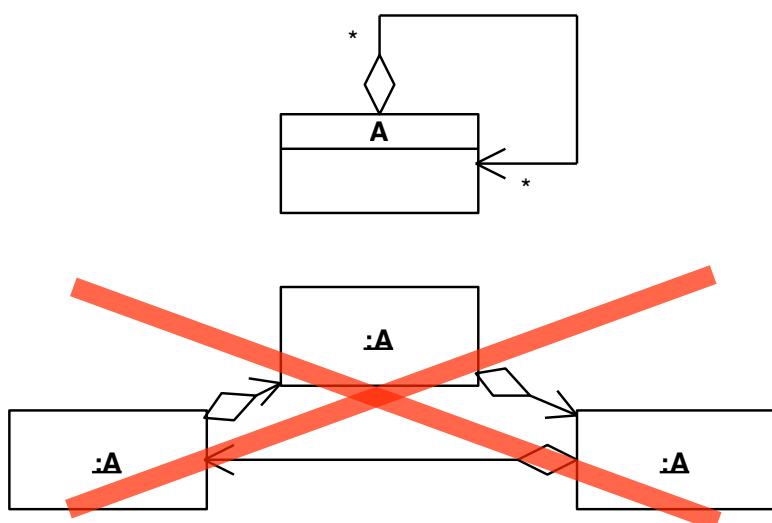
- L'aggregato può in alcuni casi esistere indipendentemente dalle parti, ma in altri casi no.
- Le parti possono esistere indipendentemente dall'aggregato.
- L'aggregato è in qualche modo incompleto se mancano alcune delle sue parti.
- È possibile che più aggregati condividano una stessa parte.
- L'aggregazione è transitiva.
- L'aggregazione è asimmetrica.

Aggregazione: semantica (2)

Il vincolo più importante di una aggregazione è che le **istanze** devono essere **acicliche**.

- In altre parole, la parte non deve essere in grado di contenere il tutto.
- Se il problema richiede di avere cicli, bisogna usare una normale associazione.

Esempio di ciclo illegale



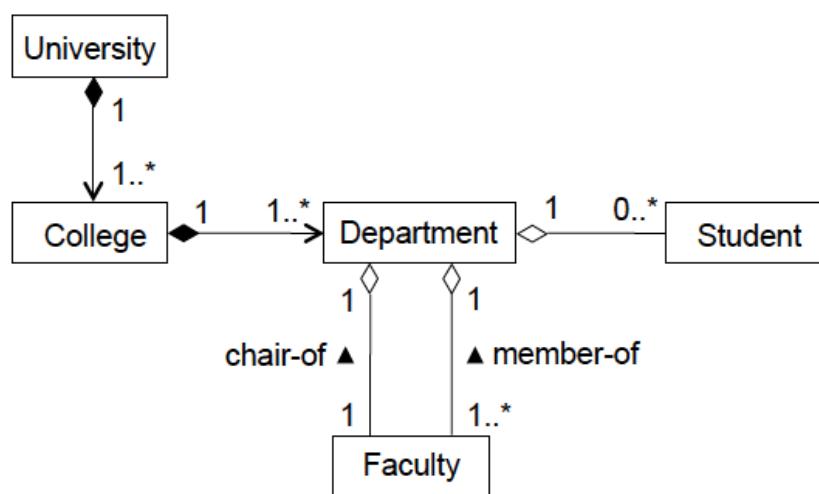
Data un'aggregazione riflessiva (es. struttura dati albero), ecco un diagramma degli oggetti non valido.

Composizione: semantica (1)

- Ogni parte può appartenere ad un solo composito per volta.
- Il composito è l'unico responsabile di tutte le sue parti: questo vuol dire che è responsabile della loro creazione e distruzione.
- Il composito può rilasciare una sua parte, a patto che un altro oggetto si prenda la relativa responsabilità.
- Se il composito viene distrutto, deve distruggere tutte le sue parti o cederne la responsabilità a qualche altro oggetto.

In altre parole, la composizione è come l'aggregazione, ma **la parte non può esistere separata dal tutto**.

Un esempio completo



Associazioni e linguaggi di programmazione

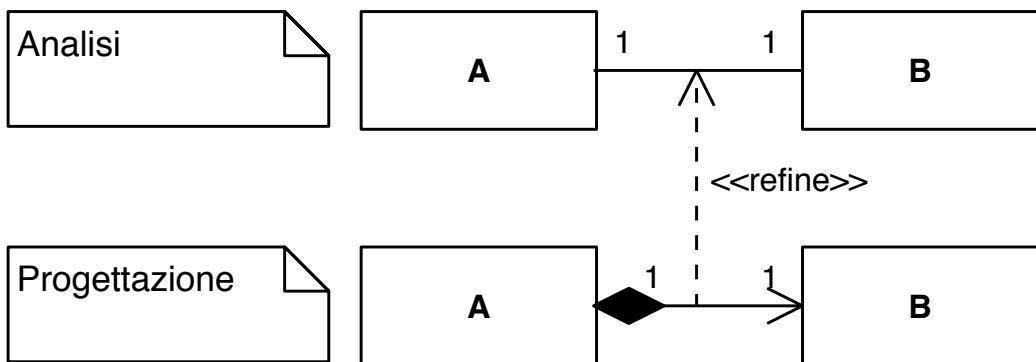
- Cosa diventano le associazioni quando sono implementate in un linguaggio di programmazione?
- Dipende dalle caratteristiche del linguaggio e dal tipo e molteplicità dell'associazione.
 - ▶ puntatori
 - ▶ array
 - ▶ references
 - ▶ oggetti
- Alcune associazioni vanno trasformate in una forma implementabile (reificate) durante la progettazione.

Associazioni durante la progettazione

- La navigabilità viene generalmente aggiunta in fase di progettazione (le associazioni di analisi tendono ad essere bidirezionali).
- La navigabilità in un solo senso è vantaggiosa in vista dell'implementazione.
- Aggregazione e composizione sono spesso aggiunte in fase di progettazione.

Reificare: 1 a 1

A è il tutto, B la parte, fortemente legati: una possibilità.

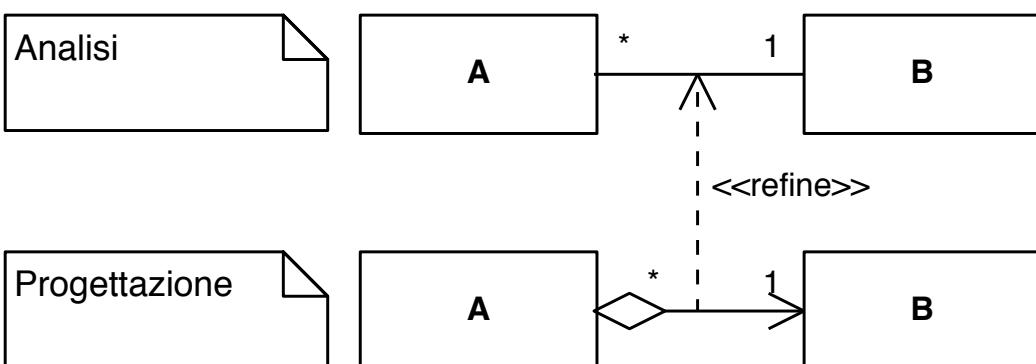


Implementata come:

- B è un campo di A (in un linguaggio come C++)
- A ha un puntatore/reference a B (es. Java)
- B non ha nulla di A

Reificare: molti a 1

A è il tutto, B la parte che può appartenere a molti A: una possibilità.

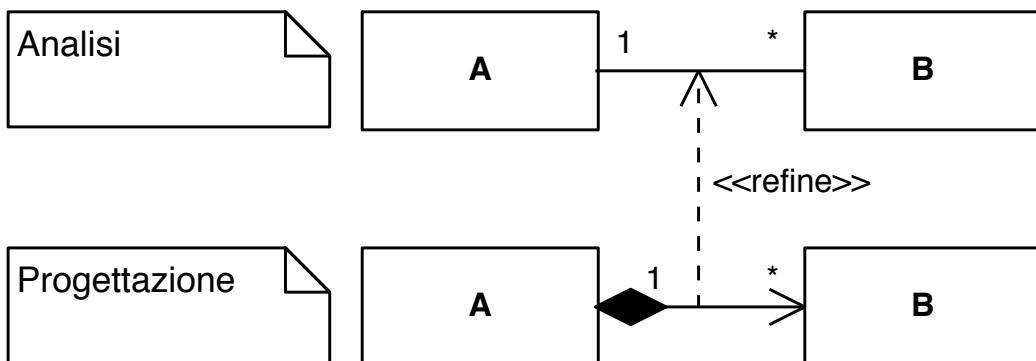


Implementata come:

- A ha un puntatore/reference a B (non può possedere B in maniera esclusiva)
- B non ha nulla di A

Reificare: molti a 1

A è il tutto con molte parti B, che possiede in maniera esclusiva: una possibilità.

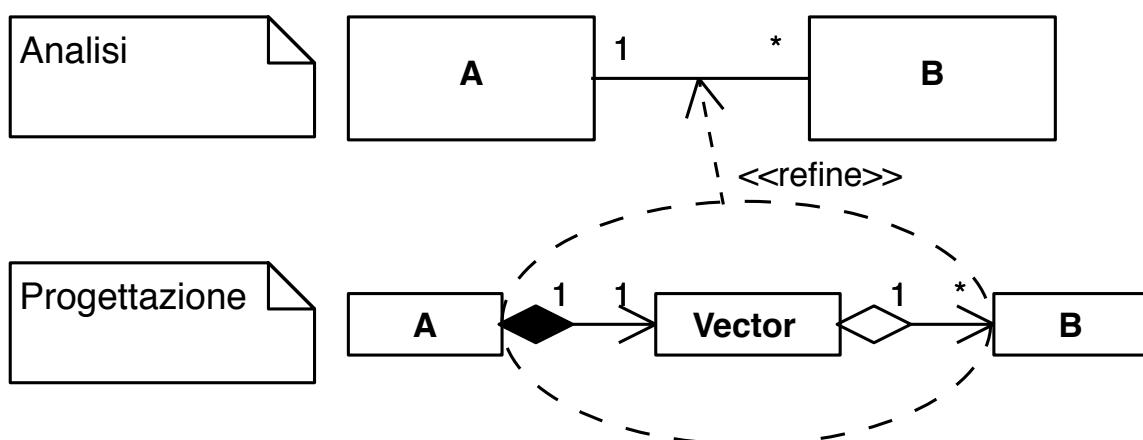


Implementata come:

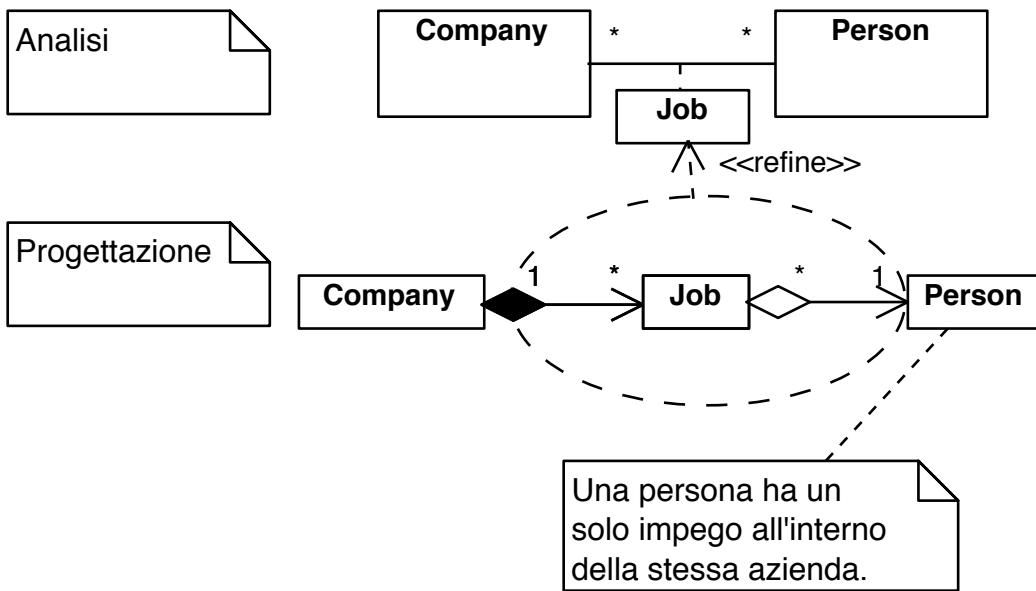
- A ha un array o lista di B fornita come primitiva dal linguaggio
- A usa una struttura di supporto (come Vector in Java)
- B non ha nulla di A

Molti a 1 con classe di supporto

L'esempio precedente usando una classe Vector.

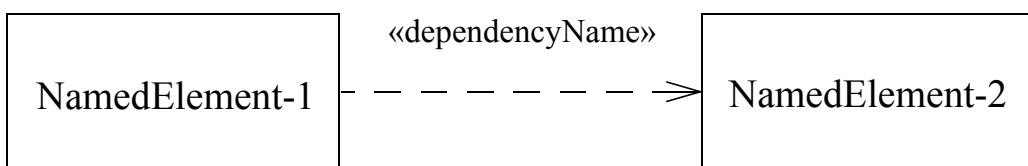


Molti a molti con classe di associazione



- Il vincolo, implicito nel primo modello, ora va aggiunto in una nota.

Dipendenza



- Una relazione semantica tra elementi (in particolare classi e loro operazioni).
- Due ruoli: **supplier** (fornitore) **client** (cliente); entrambi possono essere insiemi di elementi.
- La freccia va dal cliente verso il fornitore, lo stereotipo indica il tipo della dipendenza.
- Una dipendenza significa che il cliente richiede il fornitore per la propria specifica o implementazione.
- Il cliente dipende strutturalmente o semanticamente dal fornitore, e se la specifica del fornitore cambia, può cambiare anche quella del cliente.

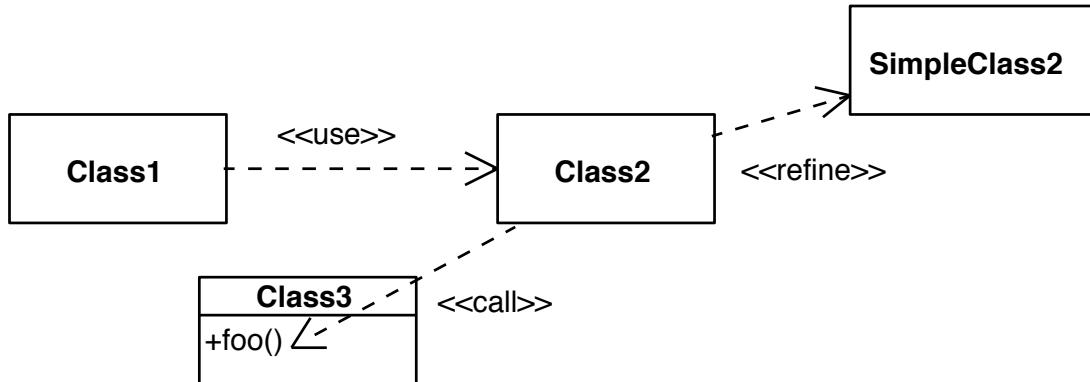
Tipi di dipendenza

- **Uso:** il cliente utilizza alcuni dei servizi resi disponibili dal fornitore per implementare il proprio comportamento.
- **Astrazione:** una relazione tra cliente e fornitore in cui il fornitore è più astratto del cliente.
- **Accesso:** il fornitore assegna al cliente un qualche tipo di permesso di accesso al proprio contenuto.
- **Binding:** il fornitore assegna al cliente dei parametri che saranno istanziati a valori specifici dal cliente.

Dipendenze fornite da UML (1)

- Uso:
 - ▶ «**use**»: il cliente richiede il fornitore per la sua implementazione completa
 - ▶ «**call**»: il cliente invoca il fornitore (che è un'operazione o la classe che la contiene)
 - ▶ «**responsibility**»: un obbligo o contratto che il cliente ha verso il fornitore
 - ▶ «**send**»: il cliente (solitamente un'operazione) manda un messaggio al fornitore
 - ▶ «**instantiate**»: il cliente può creare istanze del fornitore

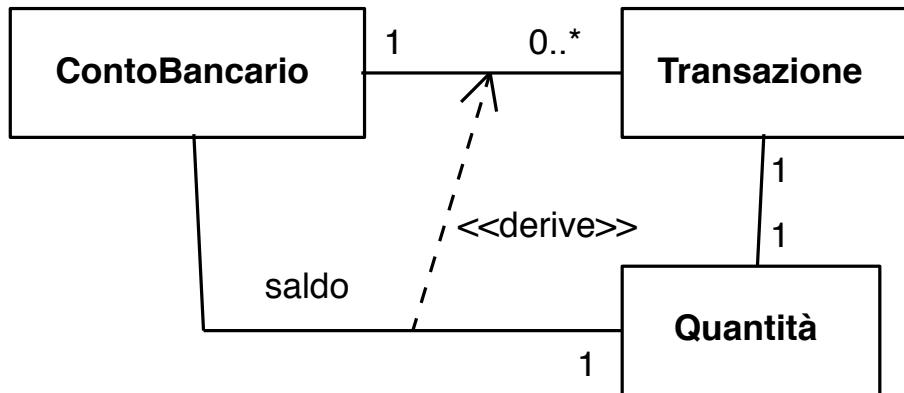
Dipendenza: esempi (1)



Dipendenze fornite da UML (2)

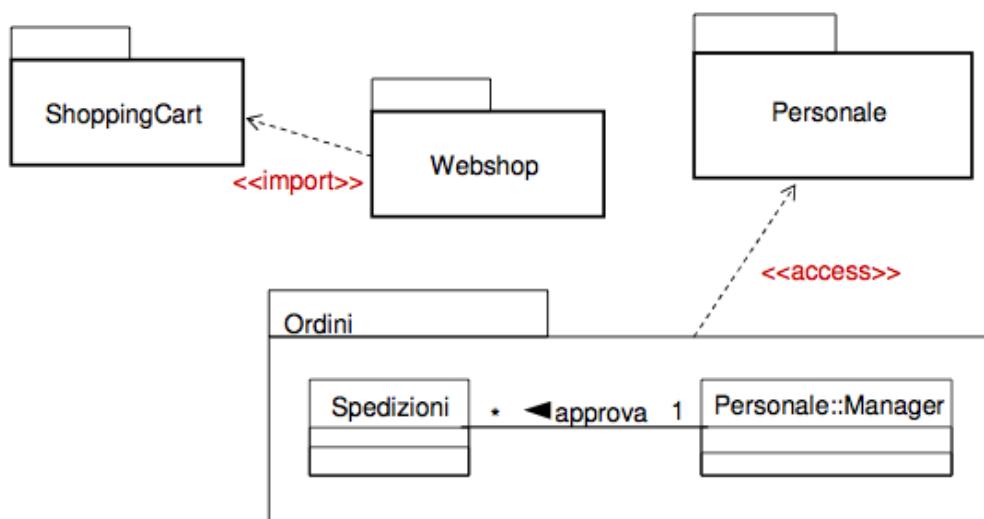
- Astrazione:
 - ▶ «**derive**»: il cliente può essere creato o calcolato (derivato) a partire dal fornitore
 - ▶ «**refine**»: indica raffinamenti successivi (es. il cliente è una classe di progettazione e il fornitore una di analisi)
 - ▶ «**trace**»: mostra lo stesso concetto in elementi diversi
- Accesso:
 - ▶ «**import**»: il cliente importa il fornitore (un package o un elemento in un package diverso)
 - ▶ «**access**»: come «import», ma gli elementi importati diventano privati (non ulteriormente importabili)
- Binding:
 - ▶ «**bind**»: il cliente fornisce i valori di binding di un template

Dipendenza: esempi (2)



Il saldo di un conto può essere derivato dall'elenco delle sue transazioni.

Dipendenza: esempi (3)

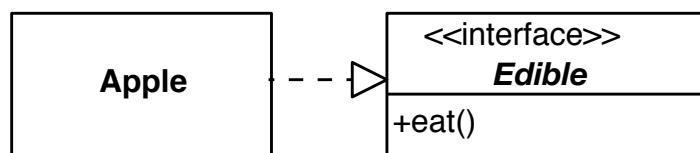


- Il package **WebShop** contiene una copia delle classi (classificatori) del package **ShoppingCart**
- I classificatori del package **Ordini** possono accedere ai classificatori (pubblici) del package **Personale**

Realizzazione

- Si tratta di una **relazione semantica** in cui il **fornitore rappresenta una specifica**, e il **cliente la implementa**.
- L'esempio canonico di realizzazione è quello in cui il fornitore è un'interfaccia, e il cliente è la classe che la implementa.
- A livello di analisi si può usare la realizzazione anche tra altri elementi del modello per indicare il loro legame **specifico/implementazione**.

Realizzazione di un'interfaccia



- La classe Apple si impegna a fornire il comportamento (operazioni) specificato dall'interfaccia Edible.
- Lo stereotipo «interface» è usato per indicare una realizzazione

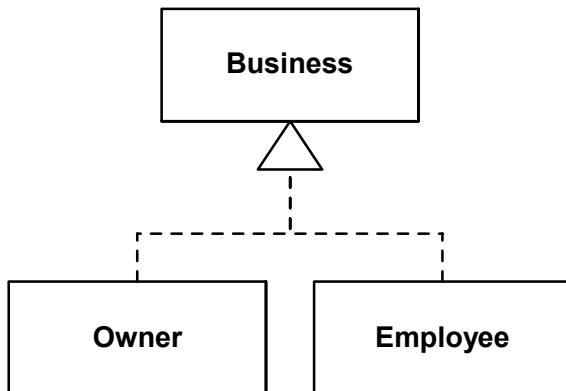
Da UML al codice (Java)

```
interface Edible {  
    public void eat();  
}  
  
public class Apple implements Edible {  
  
    @Override  
    public void eat() {  
    }  
  
}
```

Classi, sottoclassi e interfacce

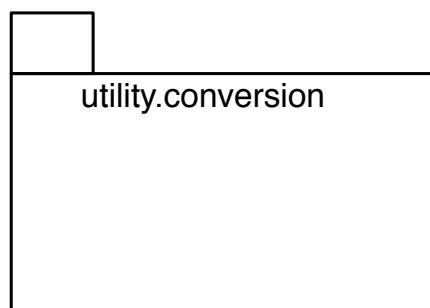
- Le interfacce sono simili alle classi (astratte) ma non hanno ‘implementazione’
- Una classe astratta può avere alcuni metodi astratti (implementati nelle sottoclassi) e altri non astratti. Può essere quindi parzialmente definita.
- Una classe può avere da zero a più istanze del suo tipo
- Un’interfaccia ha almeno una classe che la implementa

Altri tipi di realizzazione



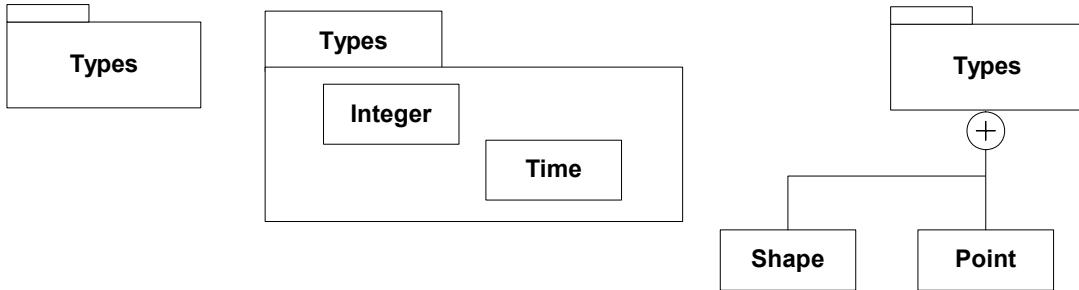
Un tipo più astratto di realizzazione: la classe `Business` è realizzata da una combinazione delle classi `Owner` ed `Employee`.

Package



- Entità di raggruppamento.
- Raggruppa elementi logicamente coesi tra loro.
- Fornisce un *namespace* all'interno del quale ogni elemento è definito univocamente dal suo nome.

Notazioni package

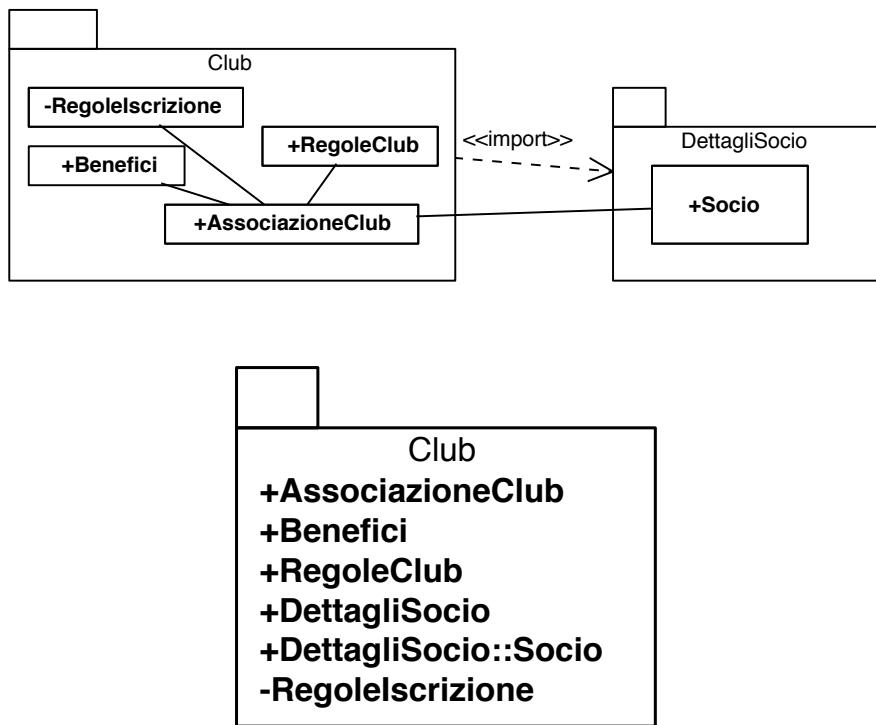


Modi diversi di rappresentare un package e il suo contenuto.

Visibilità nei package

- Ogni elemento di un package (ad es. una classe) ha una visibilità associata, che può essere *public* (+) o *private* (-).
- Quando un package viene importato o acceduto, gli elementi privati non risultano visibili dall'esterno.
- Di solito è opportuno limitare al massimo il numero di elementi pubblici, proprio come è opportuno limitare gli attributi e le operazioni pubbliche in una classe.

Esempio package



Package di analisi

- I package di analisi sono creati a partire dalle classi di analisi, per scomporre lo spazio logico del modello in sottospazi considerabili separatamente.
- Un package raggruppa classi molto coese tra loro (la maggior parte delle comunicazioni di queste classe avviene con altre classi dello stesso package).
- Se una classe deve accedere ad una classe in un package diverso, deve esistere una dipendenza «import» oppure «access».
- Sono proibiti i cicli nelle dipendenze tra package: se necessario, i package devono essere divisi per evitarli.

Conclusioni

- I diagrammi di struttura in UML modellano l'architettura logica o fisica di un sistema indipendentemente dal tempo.
- Le classi e le loro istanze, gli oggetti, sono i mattoni con i quali si costruisce un sistema.
- I diagrammi di struttura aiutano sia in fase di analisi che di progettazione, in quanto possono essere usati a diversi livelli di astrazione.

Esercizi

Esercizio Appartamento

- Rappresentare il seguente testo tramite un diagramma UML delle classi:
- Un appartamento è composto da una o più stanze, ciascuna delle quali ha una lunghezza e una larghezza, pubbliche e di tipo float. Un appartamento è posseduto da uno o più persone. Un palazzo è composto da uno o più appartamenti e possiede un'operazione privata senza parametri che ne restituisce il numero di piani.

Esercizio Treno

- Si rappresenti questo dominio con un diagramma UML delle classi:
- Si deve modellare un singolo viaggio in treno. Un treno ha 200 posti, ciascuno dei quali numerato e accessibile tramite una prenotazione che contiene la stazione di partenza e quella di arrivo (per semplicità, si assumano integer). Il viaggiatore prenota un qualunque numero di posti, e ovviamente più viaggiatori possono occupare lo stesso posto in tempi diversi durante il viaggio.

Esercizio Filosofi

- Si usino un diagramma delle classi e uno degli oggetti per rappresentare:
- Tutti i filosofi sono uomini e tutti gli uomini sono mortali. Tutti gli uomini hanno un nome. Ogni filosofo è discepolo di al massimo un altro filosofo, e un filosofo può avere un qualunque numero di discepoli. Inoltre, un filosofo può produrre un qualunque numero di opere, ciascuna delle quali ha un titolo. Socrate, Platone e Aristotele sono filosofi; Platone è discepolo di Socrate e Aristotele è discepolo di Platone. Platone ha scritto ‘La Repubblica’.