

Martin Fowler - Introduzione a UML

L'unified model language consiste in un insieme di notazioni che aiutano la descrizione della progettazione software, in particolare tutti quei software che si basano sulla programmazione ad oggetti.

Secondo Fowler ci sono 3 modi diversi con i quali sfruttare UML:

1. Come sketch
2. Come blueprint
3. Come linguaggio di programmazione

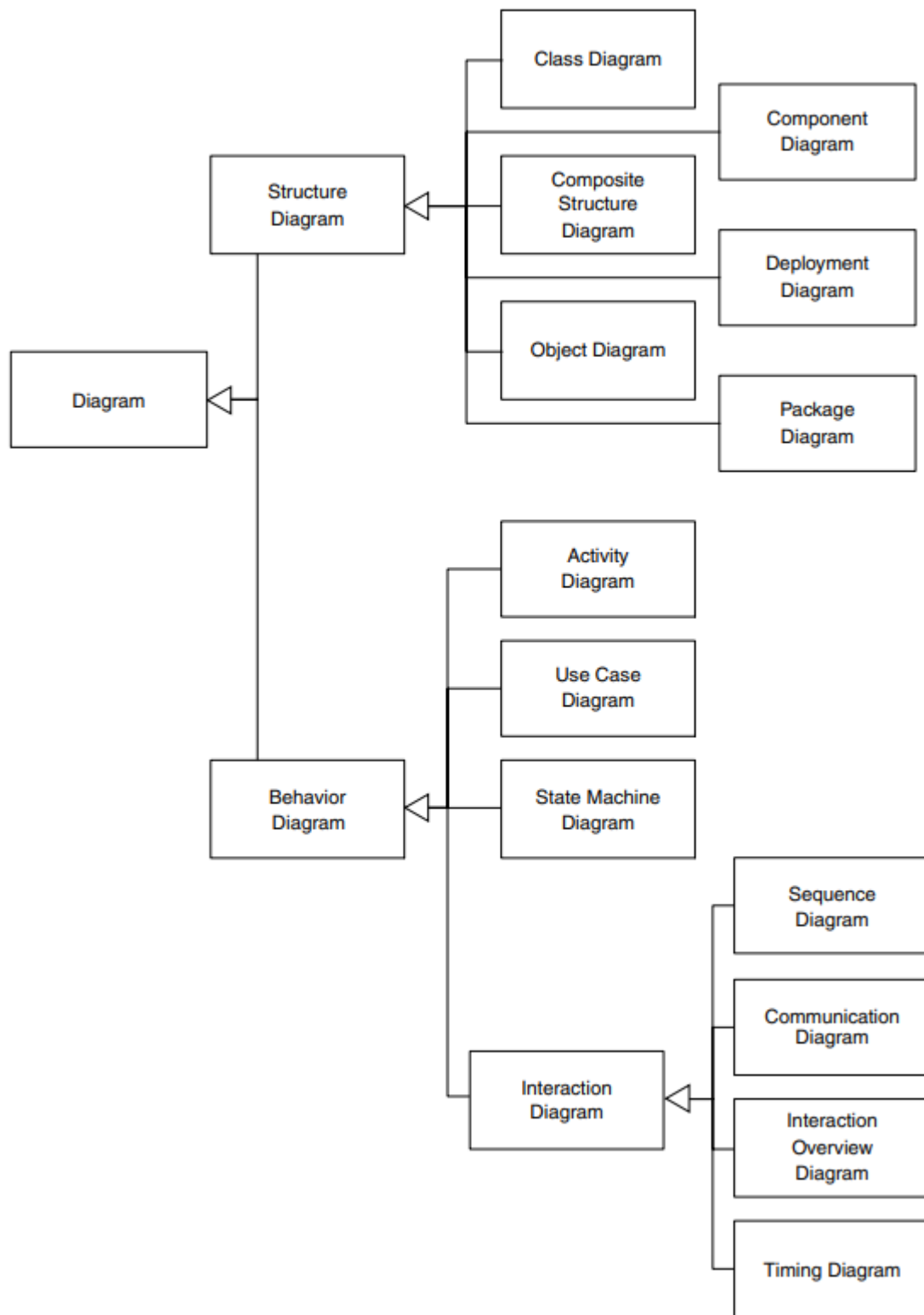
Quello più ampiamente utilizzato è il primo, in quanto UML usato come sketch aiuta gli sviluppatori a comunicare determinati aspetti del software.

UML è ampiamente utilizzato quando si parla di MODEL-DRIVEN ARCHITECTURE in quanto ciò che offre UML sono proprio dei modelli, nello specifico ne esistono diversi che servono gli scopi più disparati, vediamoli illustrati in questa tabella.

Diagram	Book Chapters	Purpose	Lineage
Activity	11	Procedural and parallel behavior	In UML 1
Class	3, 5	Class, features, and relationships	In UML 1
Communication	12	Interaction between objects; emphasis on links	UML 1 collaboration diagram
Component	14	Structure and connections of components	In UML 1
Composite structure	13	Runtime decomposition of a class	New to UML 2
Deployment	8	Deployment of artifacts to nodes	In UML 1
Interaction overview	16	Mix of sequence and activity diagram	New to UML 2
Object	6	Example configurations of instances	Unofficially in UML 1
Package	7	Compile-time hierarchic structure	Unofficially in UML 1
Sequence	4	Interaction between objects; emphasis on sequence	In UML 1
State machine	10	How events change an object over its life	In UML 1
Timing	17	Interaction between objects; emphasis on timing	New to UML 2
Use case	9	How users interact with a system	In UML 1

Come si può intuire ci sono diagrammi più utilizzati di altri, e diagrammi che in qualche modo si assomigliano, in ogni caso molte delle notazioni di un diagramma possono essere traslate in un altro, in quanto l'uso dei diagrammi non è rigido dal punto di vista teorico.

Per visualizzare le prime somiglianze/divergenze tra i vari diagrammi l'immagine a seguire ne mostra una prima classificazione in diagrammi di struttura e diagrammi comportamentali.



UML e analisi dei requisiti

L'analisi dei requisiti è uno dei primi passi nella realizzazione del software e serve a far capire allo sviluppatore cosa i clienti potrebbero volere dal software, UML può dare una mano nell'organizzare questo tipo di lavoro in vari modi:

- Use Case Diagram per descrivere come il cliente potrà interagire con il sistema.
- Class diagram, aiutano a costruire un vocabolario rigoroso per il dominio del sistema.
- Activity diagram, che descrive il workflow organizzativo, ovvero descrive come utenti e sistema interagiscono tra di loro.
- State diagram, che può essere utile se si vuole rappresentare una o più parti del software durante il loro ciclo di vita

UML e design software

Anche nella costruzione del design, UML risulta molto utile in vari modi:

- Class diagram per mostrare la prospettiva software, ovvero come le classi interagiscono tra di loro
- Sequence diagram che aiutano a decostruire i principali scenari
- Package diagram per visualizzare il software a larga scala
- State diagram se le classi del software hanno una storia particolarmente complessa
- Deployment diagram per far vedere il layout fisico del software

UML e documentazione

Una volta costruito il software, UML risulta utile anche per la documentazione, anche se Fowler suggerisce di non costruire diagrammi molto dettagliati per la documentazione, a tale scopo preferisce una documentazione basata sul codice, ad esempio tramite JavaDoc

Class Diagrams

La maggior parte degli sviluppatori pensa ai Class Diagrams quando UML viene menzionato, questo a ragione del fatto che questi diagrammi rappresentano la maggior parte dell'utilizzo che si fa di UML.

Un Class Diagram descrive i tipi di oggetti e vari tipi di relazioni statiche che intercorrono tra questi.

Proprietà

Le proprietà nei Class Diagrams corrispondono a quelli che generalmente sono i campi di una classe.

Con proprietà si intende un termine generico per rappresentare due concetti distinti: attributi e associazioni

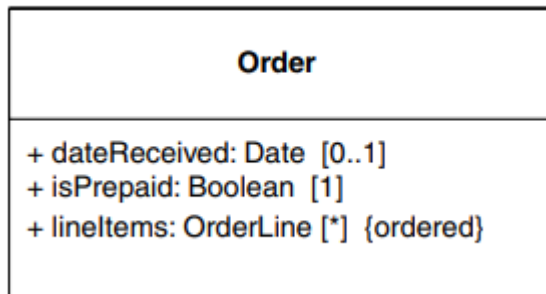
Attributi

L'attributo descrive una particolare proprietà su una singola linea, la forma di un attributo è del tipo:

```
`visibility name: type multiplicity = default {property-string}
```

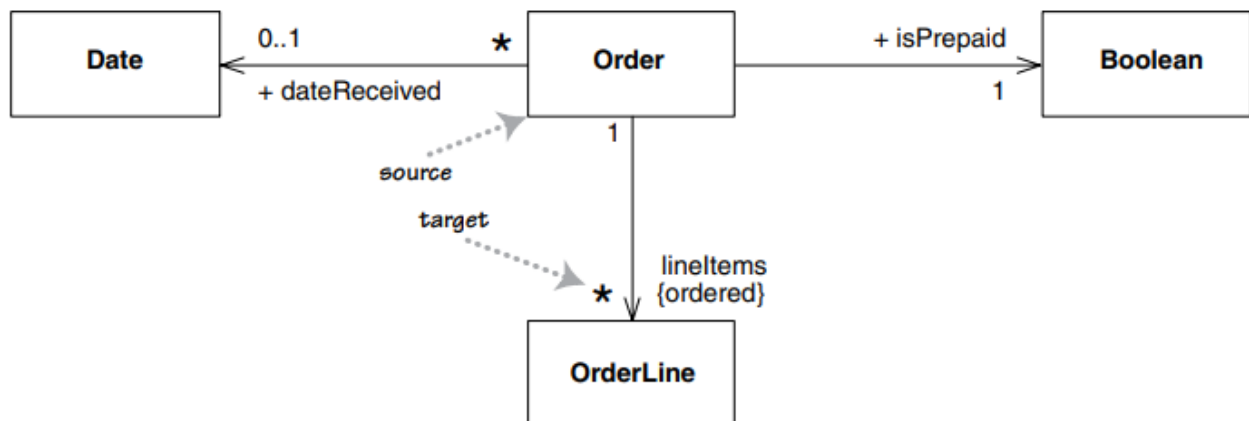
Un esempio potrebbe quindi essere:

```
- name: String [1] = "Untitled" {readOnly}
```



Associazioni

Un altro modo per descrivere la proprietà di un oggetto è tramite le associazioni; un associazione è un linea orientata che unisce due classi distinte, un associazione è descritta dalla sua molteplicità e da un nome che identifica l'azione svolta.



Molteplicità

Le molteplicità nei Class Diagrams svolgono un ruolo simile a quello svolte negli schemi E-R nella costruzione di una base di dati. Sono:

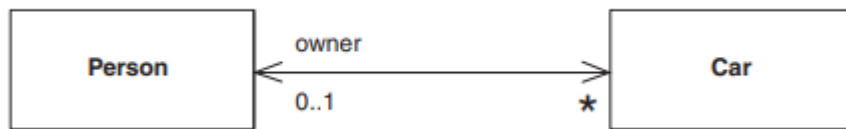
- 1
- 0..1
- • (Qualsiasi)

Inoltre la molteplicità può essere:

- Obbligatoria
- Opzionale
- A valore singolo (upper bound 1)
- A valore multiplo (upper bound maggiore di 1)

Associazioni bidirezionali

I Class Diagram permettono anche di rappresentare associazioni bidirezionali



La proprietà caratteristica delle associazioni bidirezionali è che dovremmo essere in grado di tornare al punto di partenza a prescindere dalla direzione che segui, citando l'esempio di Fowler:

For example, if i begin with a particular mg midget, find its owner, and then look at its owner's cars, that set should contain the midget that i started from.

Si può notare che questo tipo di associazione somiglia molto alle relazioni con il modello E-R.

Operazioni

Le operazioni sono le azioni che una classe è in grado di compiere, la sintassi completa per una generica operazione è:

```
visibility name (parameter-list) : return-type {property-string}
```

dove:

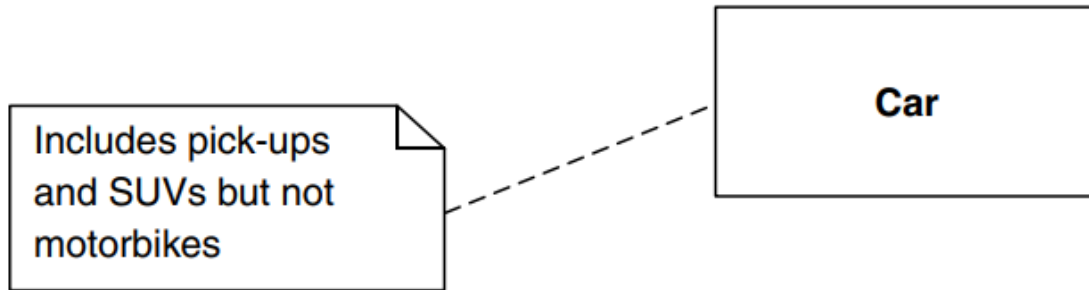
- This visibility marker is public (+) or private (-); others on page 83.
- The name is a string.
- The parameter-list is the list of parameters for the operation.
- The return-type is the type of the returned value, if there is one.
- The property-string indicates property values that apply to the given operation

Generalizzazioni

Da una prospettiva software quando si parla di generalizzazione si parla di ereditarietà: molto spesso quando vogliamo modellare il comportamento del cliente, vorremmo anche rappresentare il comportamento di clienti particolari, come ad esempio quelli fidati, in questo caso dobbiamo modellare il comportamento della superclasse, della sottoclasse, e la loro associazione, in modo tale che il cliente generico ed il cliente particolare siano sostituibili fra loro.

****Commenti**

Ogni linguaggio definisce un proprio standard per permettere a chi lo usa di poter commentare parti del codice, nel caso di UML utilizziamo le note:

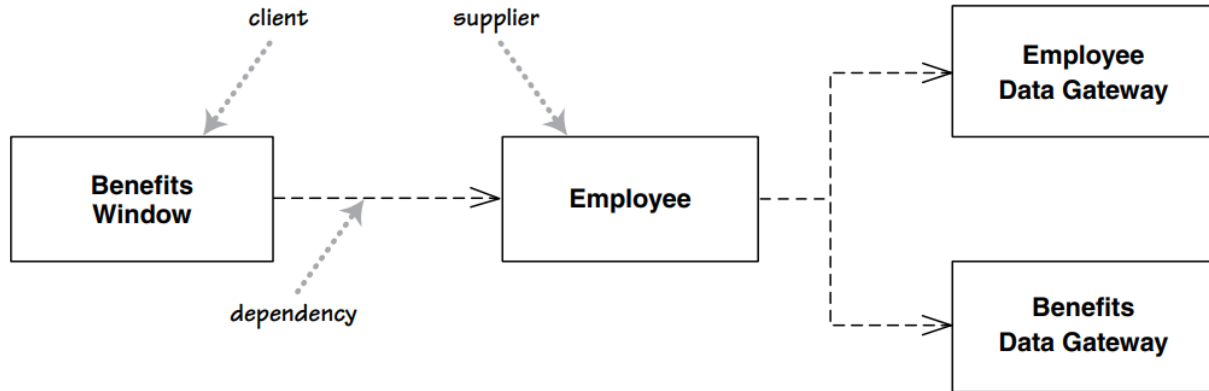


Dipendenze

Una dipendenza fra due oggetti indica che il cambiamento dell'una implica il cambiamento dell'altra.

Gestire le dipendenze è fondamentale in quanto al crescere del sistema, le dipendenze diventano sempre più complicate, e le conseguenze impattano a catena tutte le componenti del sistema.

UML offre un modo per rappresentare le dipendenze tra le varie classi attraverso l'uso di frecce tratteggiate unidirezionali:



Ci sono comunque molti modi di rappresentare dipendenze in UML, anche attraverso l'uso di keyword, Fowler ne mostra questa in quanto la sua preferita.

La regola principale sarebbe quella di minimizzare le dipendenze, in particolare quando quest'ultime attraversano ampie parti di sistema.

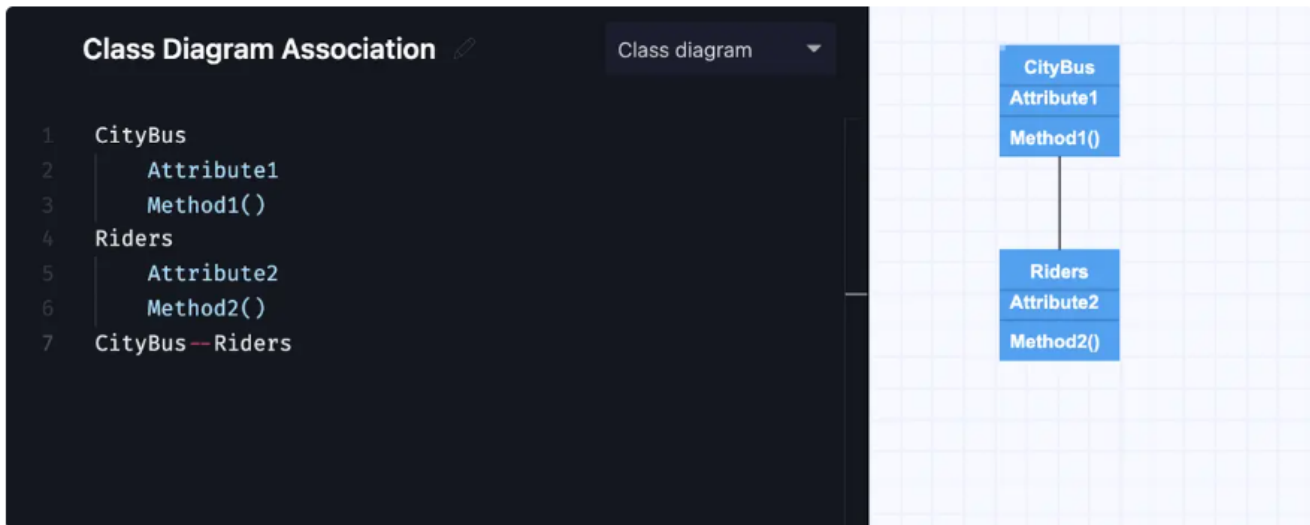
Di nuovo sulle varie associazioni nei Class Diagram

E' fondamentale saper distinguere i vari tipi di associazione nei Class Diagram, a tale proposito segue un approfondimento che schematizza le varie associazioni, prendendo spunto dall'articolo di gleek_ <https://www.gleek.io/blog/class-diagram-arrows>

Associazione semplice

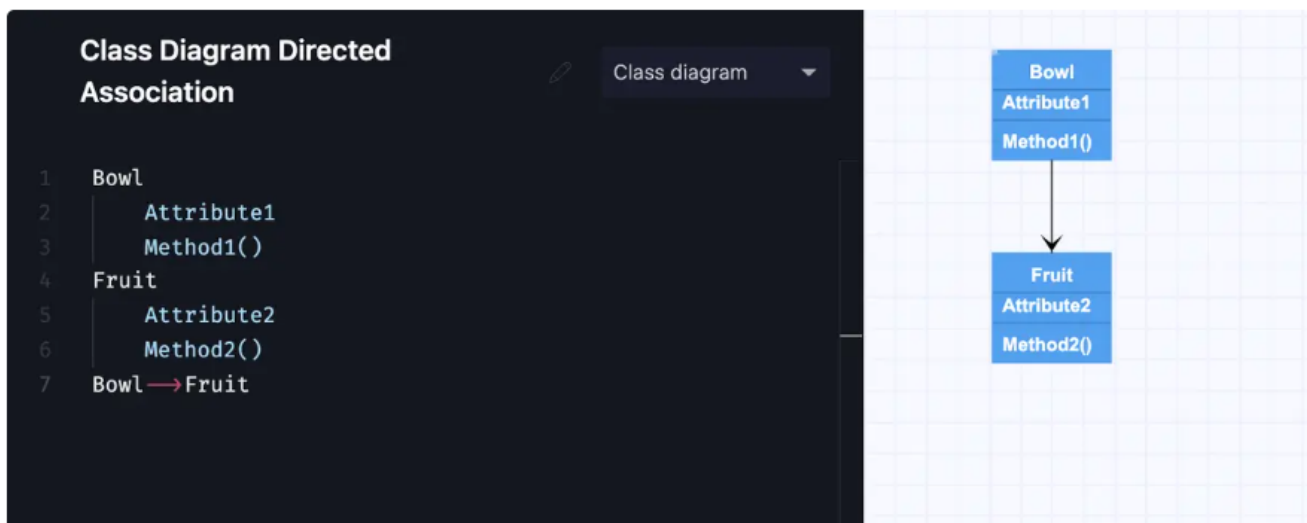
Association is the most basic of relationships. Association means any type of relationship or connection between classes. For example, we show a direct link between a city bus and its riders using an association line. We show a simple association with a straight line.

In gleek.io we create this by typing two hyphens: —



Associazione diretta

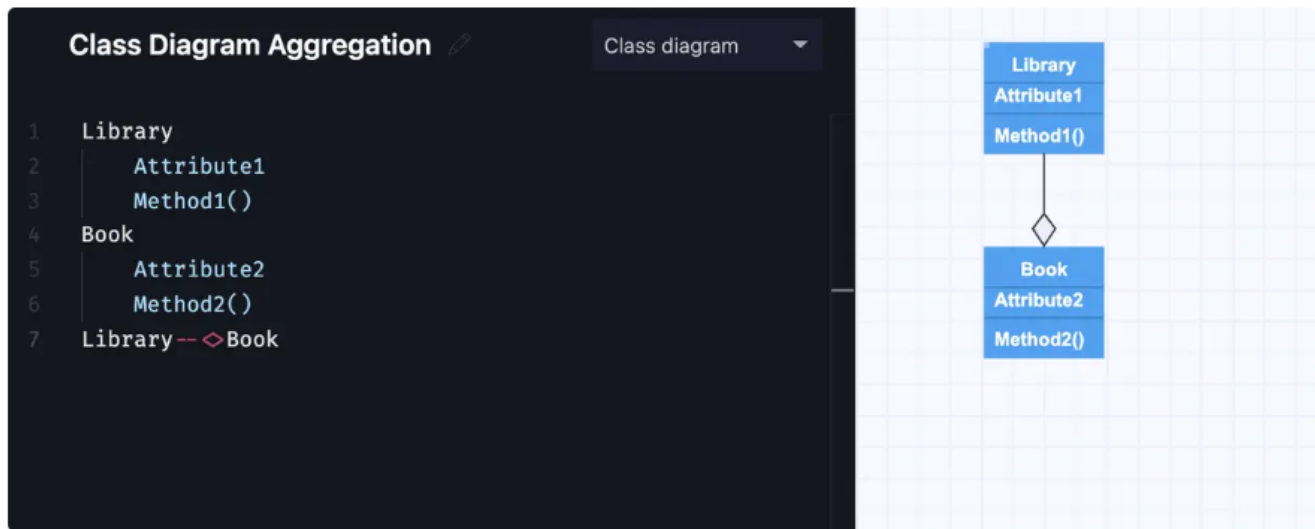
Directed association shows a strong relationship between classes. The classes must communicate. We represent a direct association with an arrow pointing to our object class. For example, a bowl might contain fruit. The bowl acts as a container class for the fruit class. In gleek.io we create this association with two hyphens and a greater-than symbol. The syntax looks like this: —>



Aggregazione

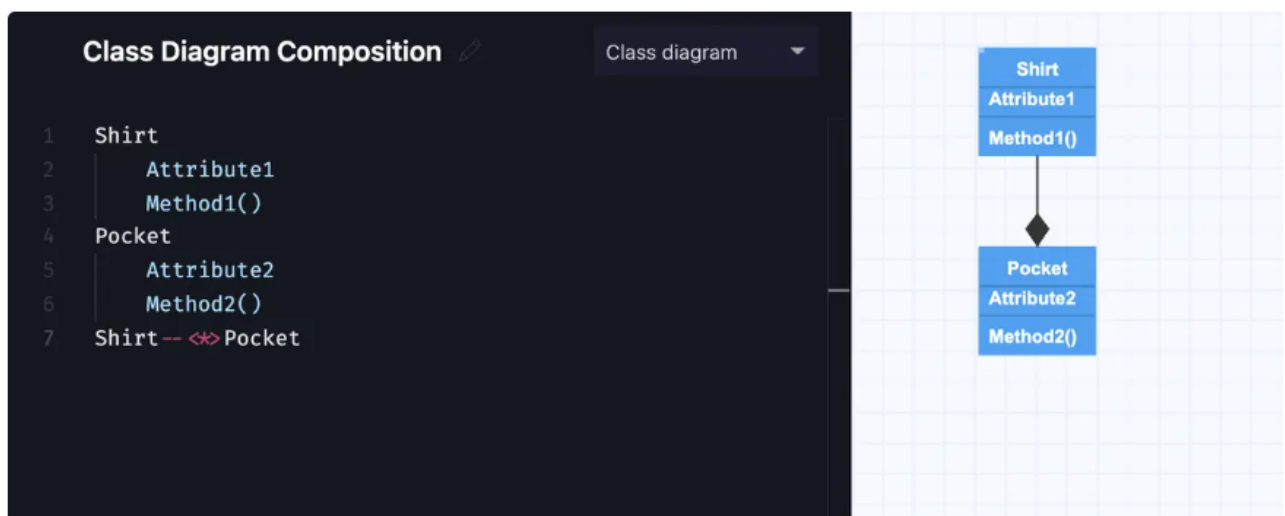
We use aggregation arrows when we want to convey that two classes are associated, but not as close as in direct association. The child class can exist independent of the parent element.

For example, a book still exists if somebody checks it out from the library. In gleek.io we create aggregation arrows by typing two hyphens followed by a lesser-than symbol followed by a greater-than symbol. The syntax looks like this: --<>



Composizione

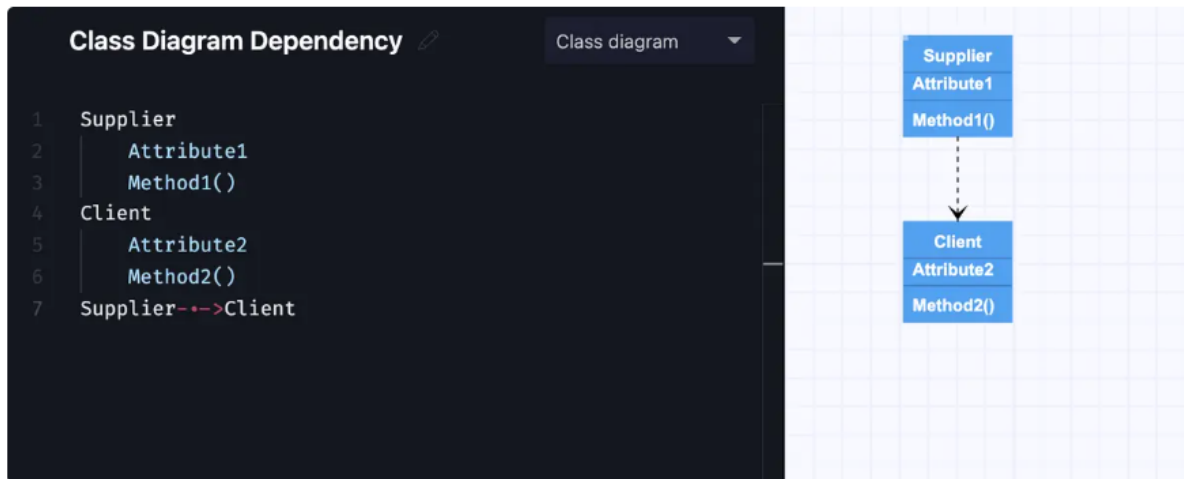
Composition arrows show up in UML class diagrams when we want to show a similar association to aggregation, with a key difference. Composition associations show relationships where the sub-object exists only as long as the container class exists. The classes have a common lifecycle. For example, a pocket on the front of a shirt cannot exist if we destroy the shirt. In gleek.io we create a composition arrow by typing two hyphens followed by a star inside a lesser-than and greater-than symbol. The syntax looks like this: --<*>



Dipendenza

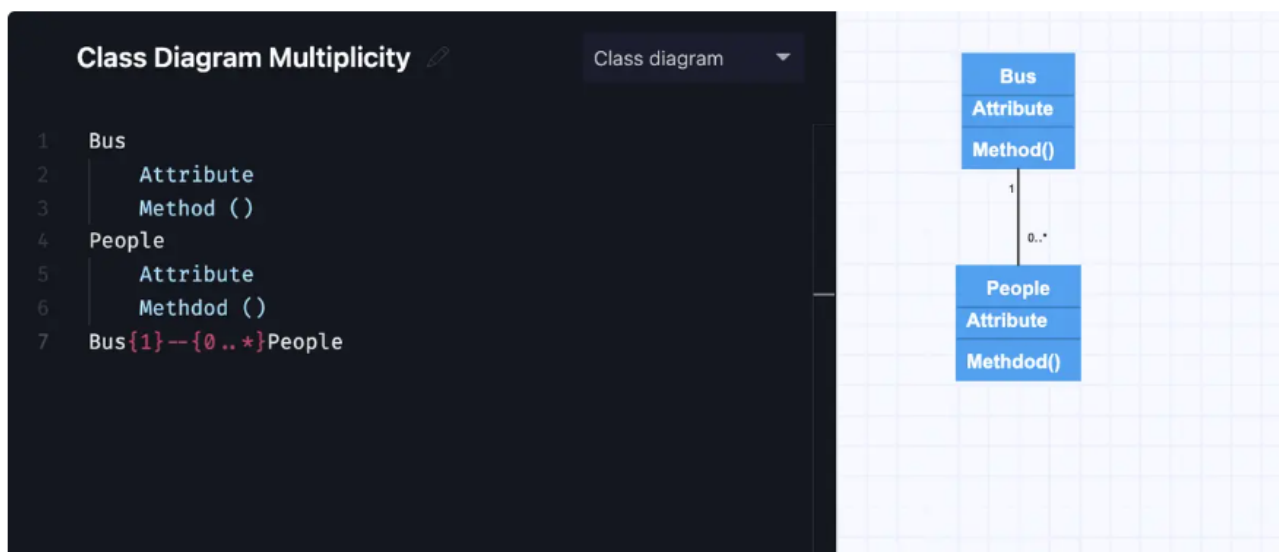
Dependency arrows show us where two elements depend on each other, but in a less strong relationship than a basic association. Changes to the parent class will also affect the child

class. Dependency shows a supplier-client type of relationship. In gleeek.io we create a dependency arrow with a hyphen, a period followed by another hyphen, and a greater-than symbol. Our syntax will look like this: `-.>`



Molteplicità

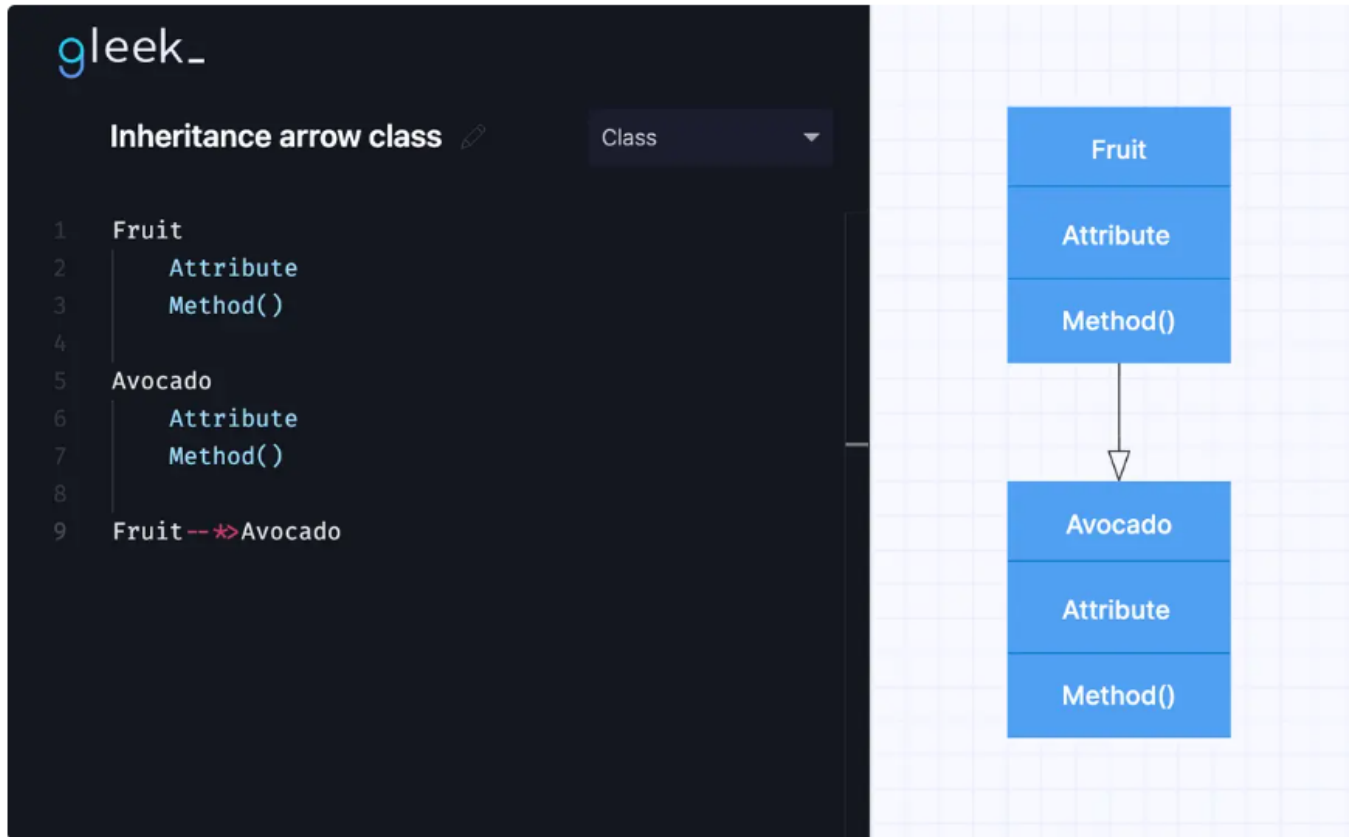
Multiplicity or cardinality arrows show a place in our UML diagram where a class might contain many (or none!) items. For example, a city bus might have any number of riders at a given time. People constantly get on and off as the bus moves through the streets. We show this in our diagram with the notation `0..*` meaning our class might contain zero to many objects. In gleeek.io we create multiplicity with numbers inside curly brackets with two hyphens in the middle. Our syntax looks like this: `**{1}-{0..*}`



Ereditarietà

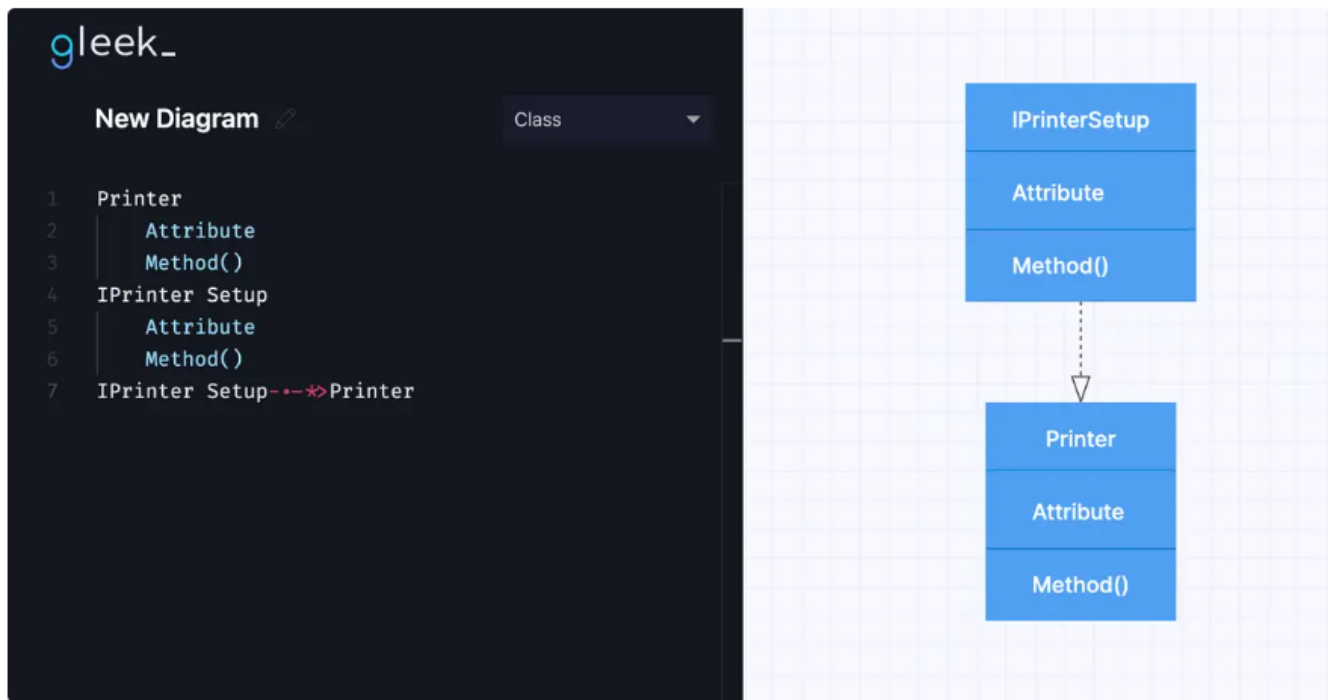
We use Inheritance arrows to show a child class inherits functionality from the parent class. For example, an avocado is a type of fruit. Fruit is the super-class. Avocado is the sub-class. The

avocado inherits its fruitiness from its fruit parent. To show inheritance in our UML class diagram in gleek.io, we type two hyphens followed by a star and a greater-than symbol. Our syntax will look like this: --*>



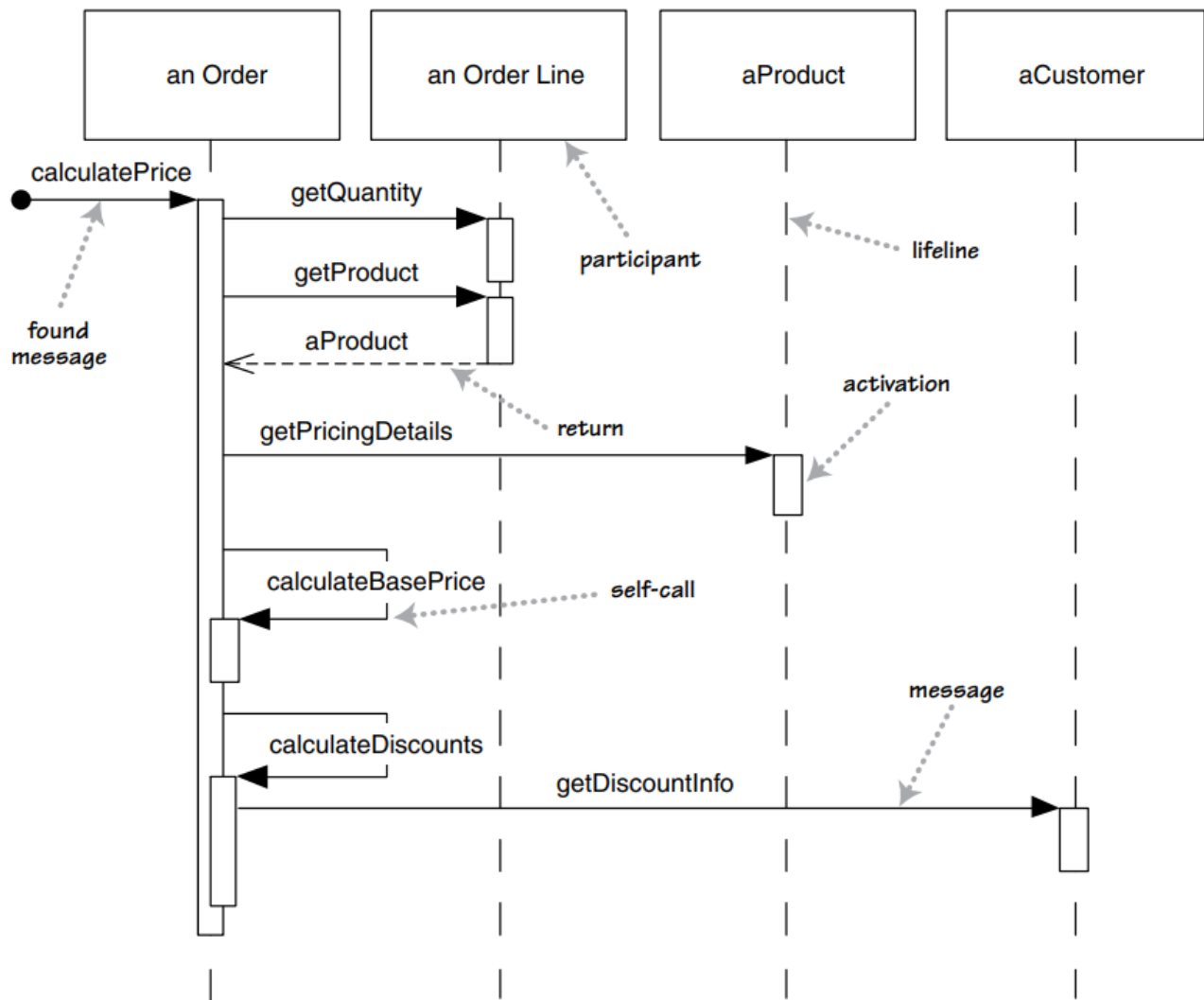
Implementazione

We use realization or implementation arrows to indicate a place where one class implements the function defined in another class. For example, the printer setup interface sets the printing preferences that are being implemented by the printer. The arrangement shows a realization association. To show the relation in gleek.io, we type a hyphen, a period followed by another hyphen, a star symbol, and a greater-than symbol. Our syntax will look like this: -. *>



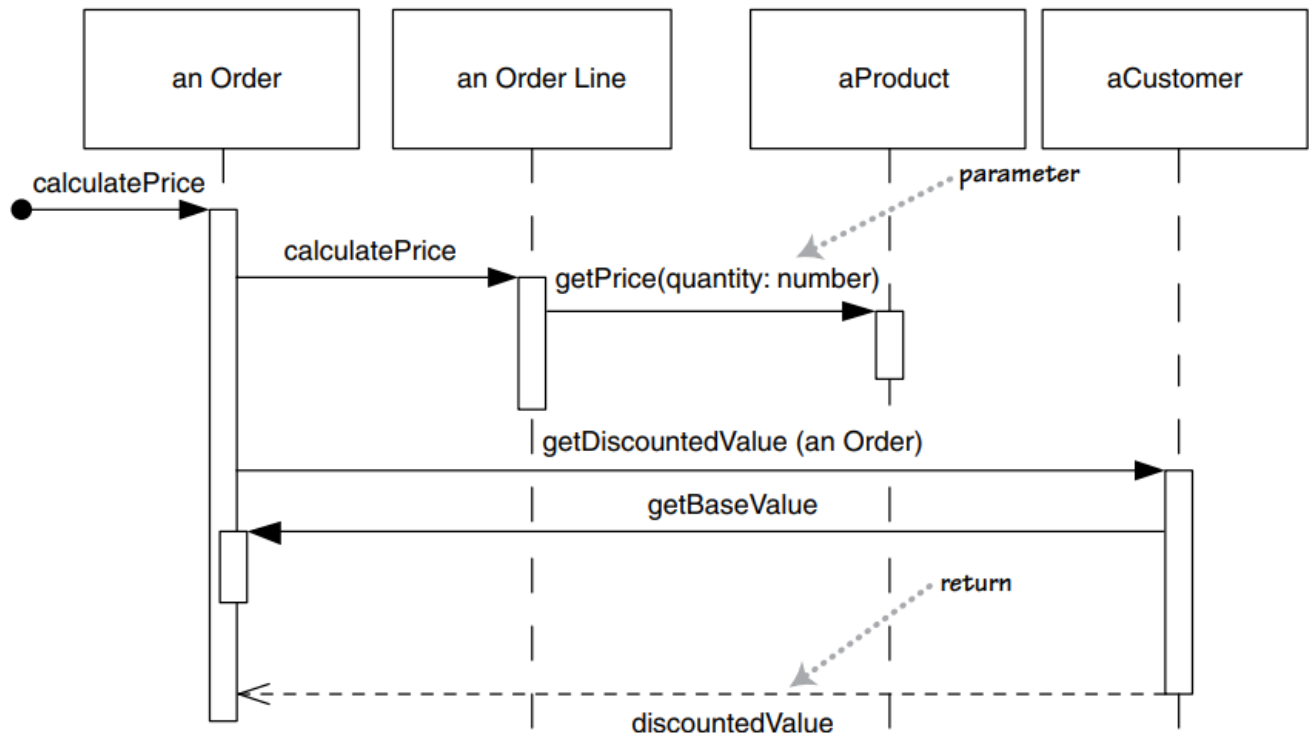
Sequence Diagrams

In genere, i diagrammi di interazione mostrano come gruppi di oggetti collaborano in determinati aspetti comportamentali, nel caso dei Sequence Diagrams ciò che modelliamo è il comportamento di un singolo scenario. Vediamo un semplice esempio.



In questo caso modelliamo delle iterazioni che servono a calcolare il prezzo del singolo prodotto, e per ognuno calcolare uno sconto in base al proprio prezzo in maniera centralizzata. Anche se dal grafico non è possibile evincerlo, per eseguire questo programma i metodi `getQuantity`, `getProduct`, `getPricingDetails`, e `calculateBasePrice` devono essere eseguiti in ordine per ogni prodotto, mentre il metodo `calculateDiscounts` va invocato solo una volta.

Possiamo modellare la stessa applicazione nella sua versione distribuita:

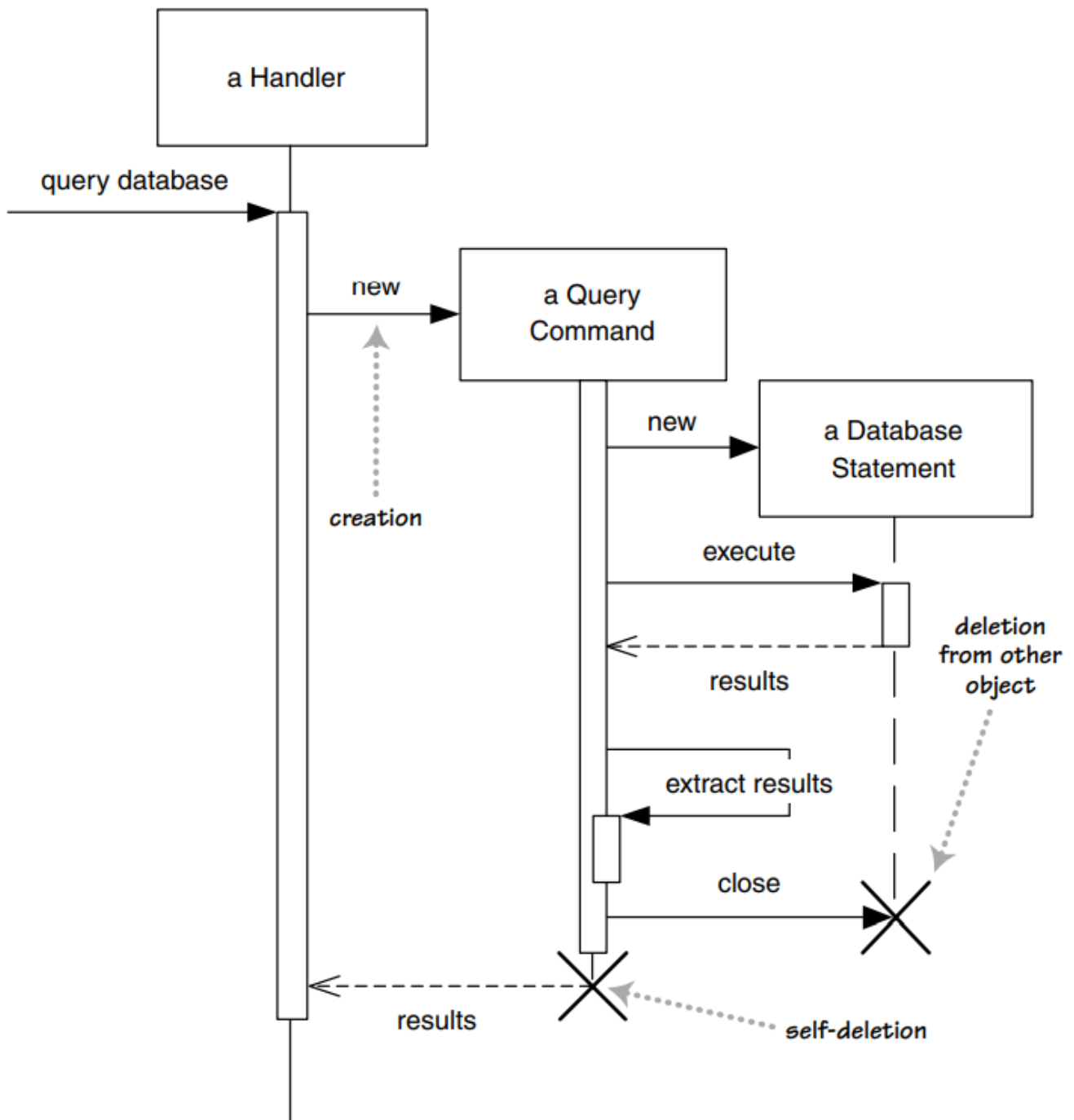


Le classi che partecipano al diagramma prendono informalmente il nome di partecipanti.

Creare ed eliminare partecipanti

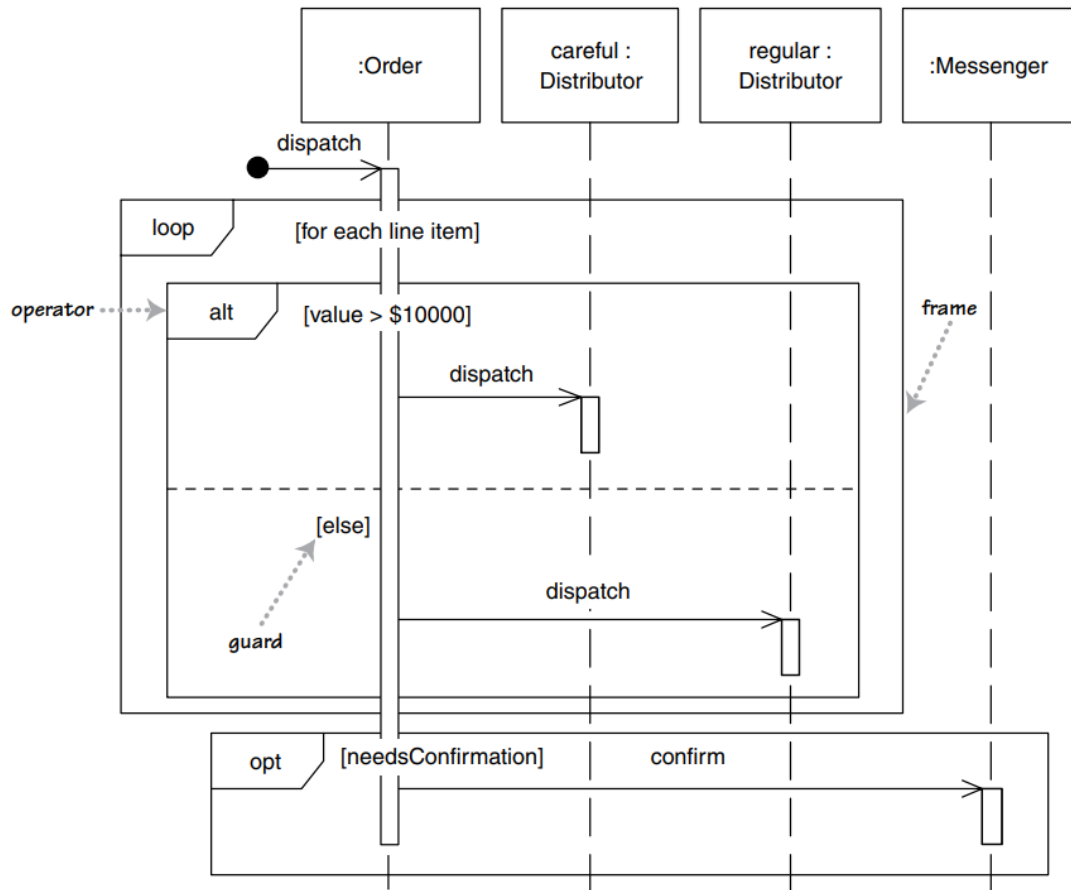
La notazione per creare un partecipante è semplicemente una freccia che punta al box della classe, che diventa subito operativa, si può anche rafforzare il concetto usando un messaggio, a tal proposito Fowler utilizza il termine 'new'. Mentre per eliminare il messaggio si usa una X maiuscola.

Vediamo un esempio:



Loop e condizioni

I Sequence Diagram non sono molto efficienti nell'esprimere situazioni di loop, infatti Fowler preferisce in questi casi l'uso di Activity Diagram o del codice stesso, in ogni caso le notazioni di loop e condizione si rappresentano attraverso l'implementazione di Frames, vediamo come:



Quando usare i sequence diagrams

Quando dobbiamo comprendere (e far comprendere) il comportamento di vari oggetti in uno scenario ben preciso, l'uso di Sequence Diagrams è fortemente consigliato.

Object Diagrams

Un Object Diagram è uno snapshot degli oggetti, generalmente viene usato per integrare informazioni che non possono essere intercettate utilizzando solo Class Diagram

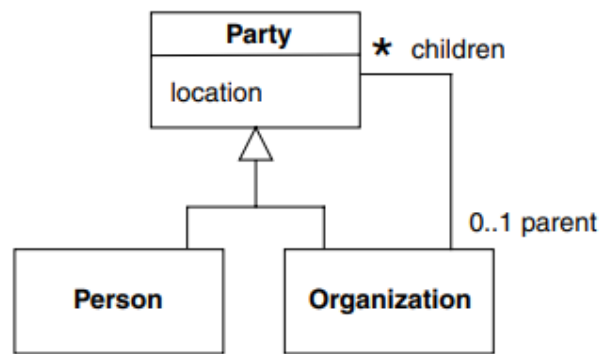


Figure 6.1 *Class diagram of Party composition structure*

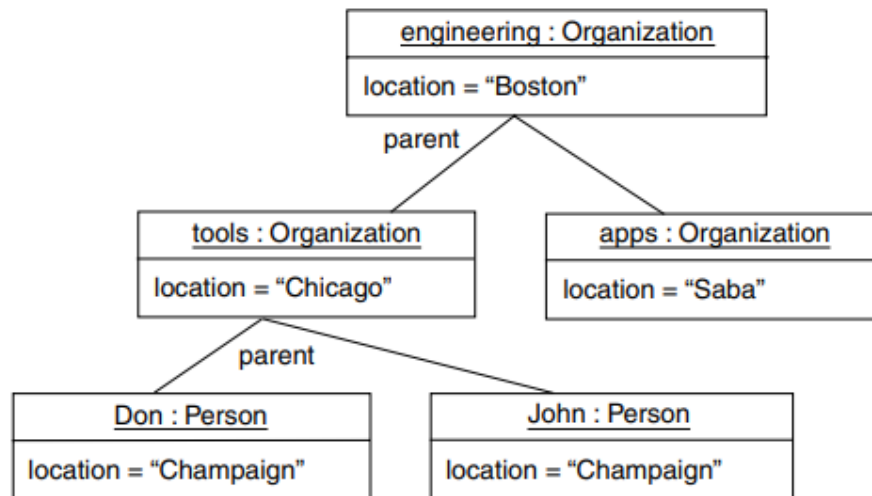
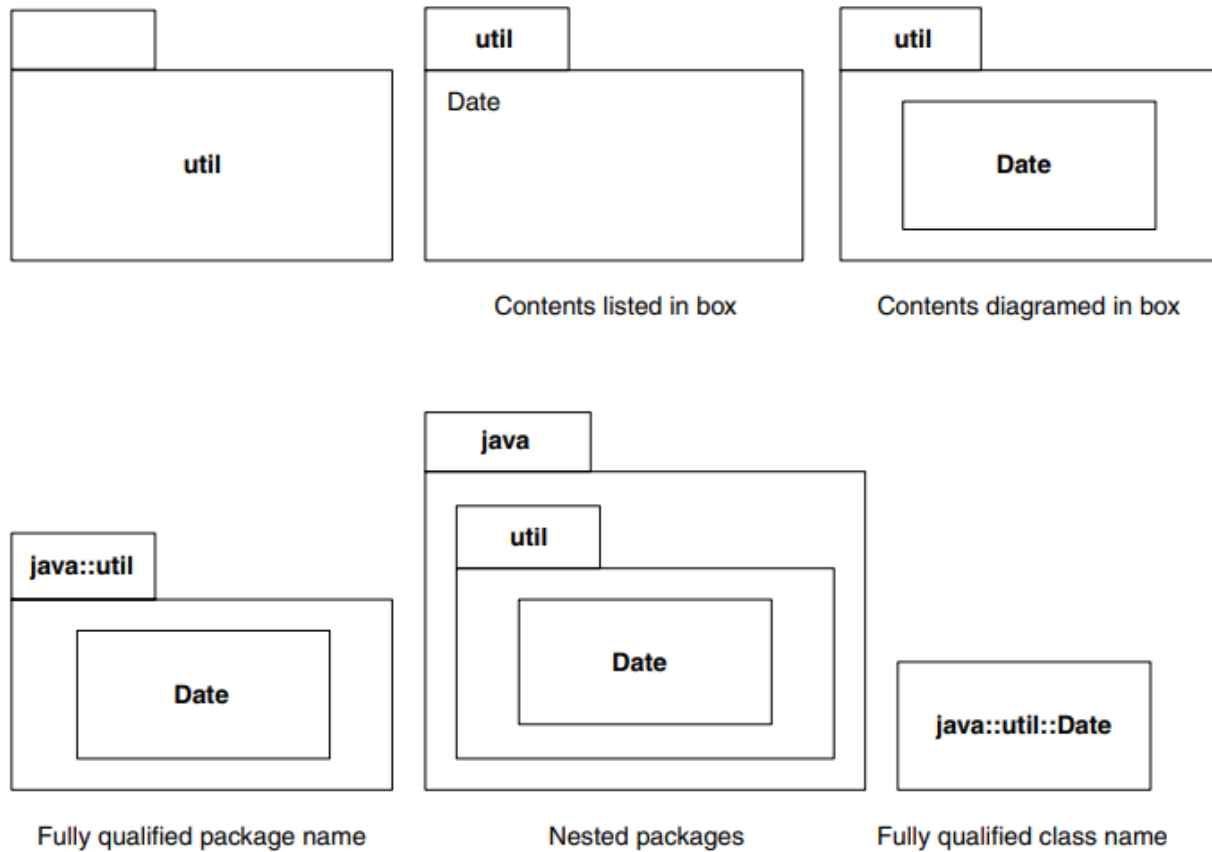


Figure 6.2 *Object diagram showing example instances of Party*

Package Diagram

Organizzare in classi è certamente il modo più popolare di organizzare il proprio lavoro tramite il design in UML, ma non è l'unico. Quando si ritiene utile vedere come può essere strutturato il sistema a larga scala, i Package Diagrams risultano un ottimo strumento di design.

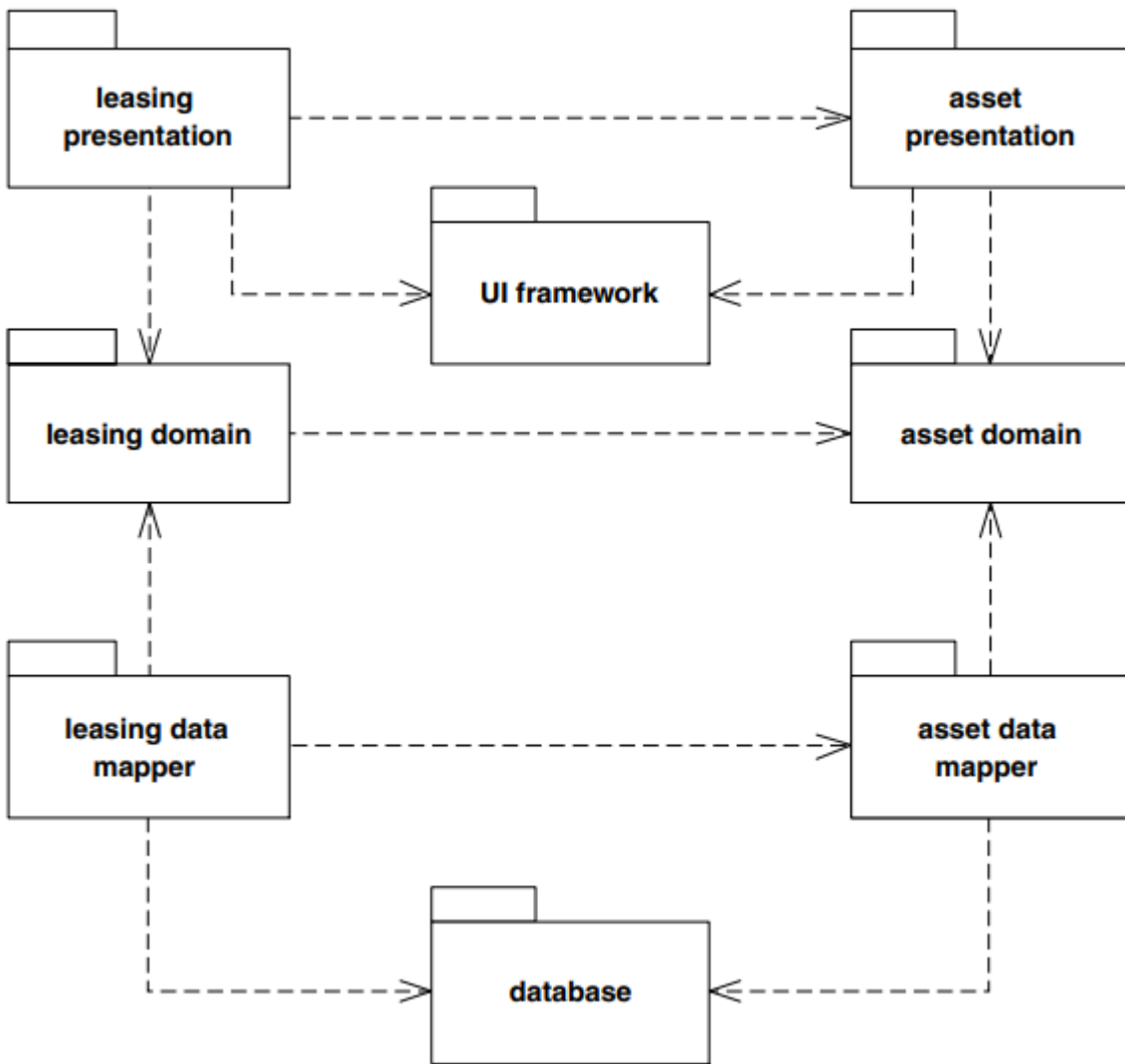
La convenzione in UML è quella di rappresentare i vari package come cartelle, ma esistono diverse notazioni per visualizzare la medesima informazione, vediamo quali:



Questi diagrammi offrono pure una buona rappresentazione di come i vari package interagiscono tra loro, ovvero ne catturano le dipendenze.

Come regola generale di pratica utilità , un buon diagramma dovrebbe non avere cicli, in modo tale da poter visualizzare meglio il workflow del sistema

Vediamo un esempio di come più pacchetti potrebbero interagire tra loro:



Deployment Diagrams

I Deployments Diagrams sono molto utili per rappresentare il layout fisico del sistema. Questi diagrammi sono composti dai nodi e dai percorsi di comunicazione, un nodo è un host di qualche particolare software e possono essere di due tipi:

- Device, ovvero la parte hardware che ospita l'applicazione
- Execution environment, ovvero del software che fa da host ad altro software

Ogni nodo contiene un artefatto che corrisponde solitamente a dei file, questi file sono eseguibili, data files, HTML, documenti ecc..

Si dice che un artefatto è deployed ad un nodo.

I percorsi di comunicazione (Communication paths) indicano invece come i vari nodi comunicano fra loro.

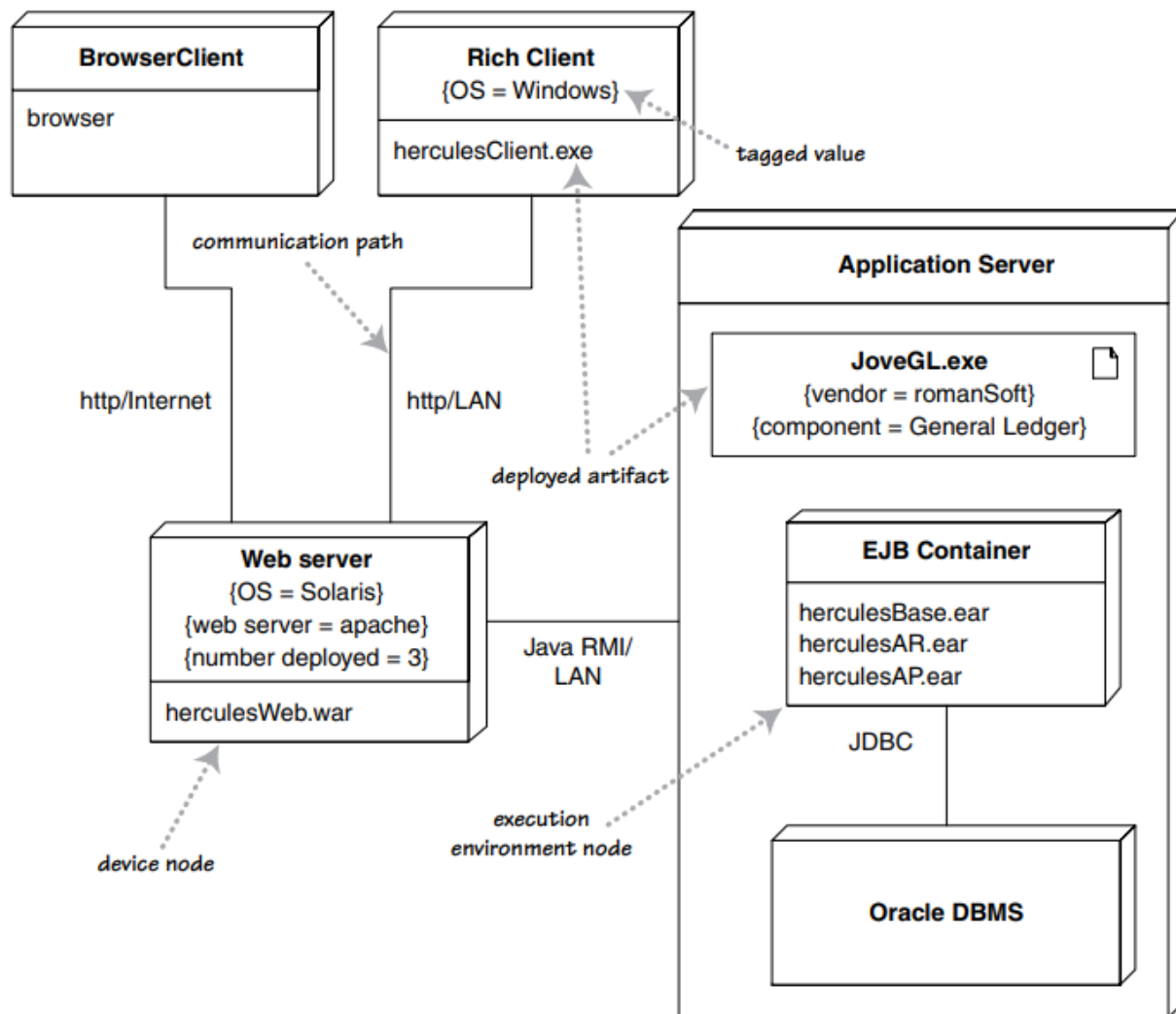


Figure 8.1 *Example deployment diagram*

Use cases


Gli Use Cases si utilizzano per comprendere i requisiti funzionali di un sistema. Questi diagrammi fanno uso di scenari, ovvero una sequenza di passaggi (steps) che descrivono l'interazione tra il potenziale cliente e il sistema. Supponiamo di avere uno web store online e di voler comprare un prodotto, uno scenario potrebbe dover descrivere i vari step:

The customer browses the catalog and adds desired items to the shopping basket. when the customer wishes to pay, the customer describes the shipping and credit card information and confirms the sale. the system checks the authorization on the credit card and confirms the sale both immediately and with a follow up e mail.

Questo potrebbe indicare lo scenario più comune, però ogni step dovrebbe tenere in considerazione i vari imprevisti e in generale tutto ciò che devia dallo scenario, come ad esempio la carta di credito che non autorizza il pagamento.

L'utente che interagisce nel sistema negli Use Cases si chiama attore, in questo caso un attore potrebbe essere il cliente finale, ma spesso si vorrebbe prevedere l'interazione dei product analyst, dei sales manager ecc...

Curiosità sulla scelta del nome attore:

 **Actor isn't really the right term; role would be much better. apparently, there was a mistranslation from swedish, and actor is the term the use case community uses**

Costruire un Use Case

Non c'è un modo predefinito di definire questi Use Case, ma il procedimento logico è quasi sempre simile:

1. Si parte con comprendere quale degli scenari si vuole rappresentare e si dichiara come main scenario
2. Si scrivono i vari step dello scenario principale
3. si osservano gli altri scenari e si cerca un modo di estenderli allo scenario principale, uno scenario scelto come estensioni possono essere sia successi, sia fallimenti, sia casi limite.

Buy a Product

Goal Level: Sea Level

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

Extensions:

3a: Customer is regular customer

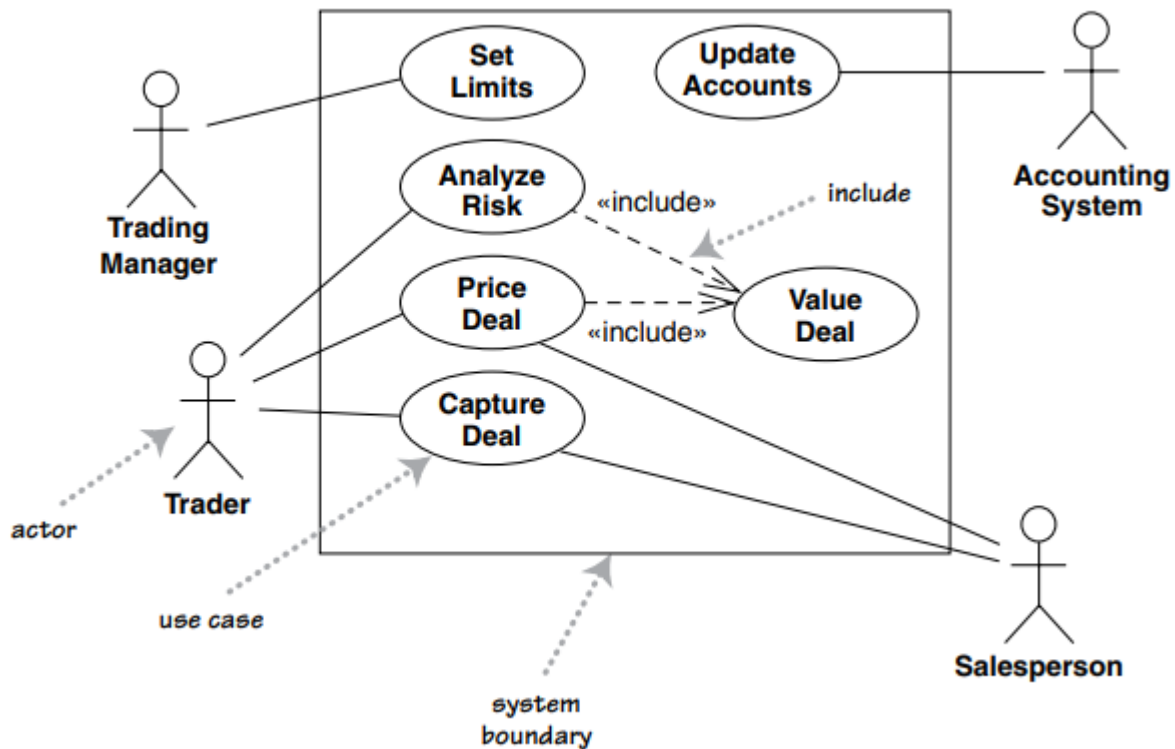
- .1: System displays current shipping, pricing, and billing information
- .2: Customer may accept or override these defaults, returns to MSS at step 6

6a: System fails to authorize credit purchase

- .1: Customer may reenter credit card information or may cancel

Ma quindi i diagrammi

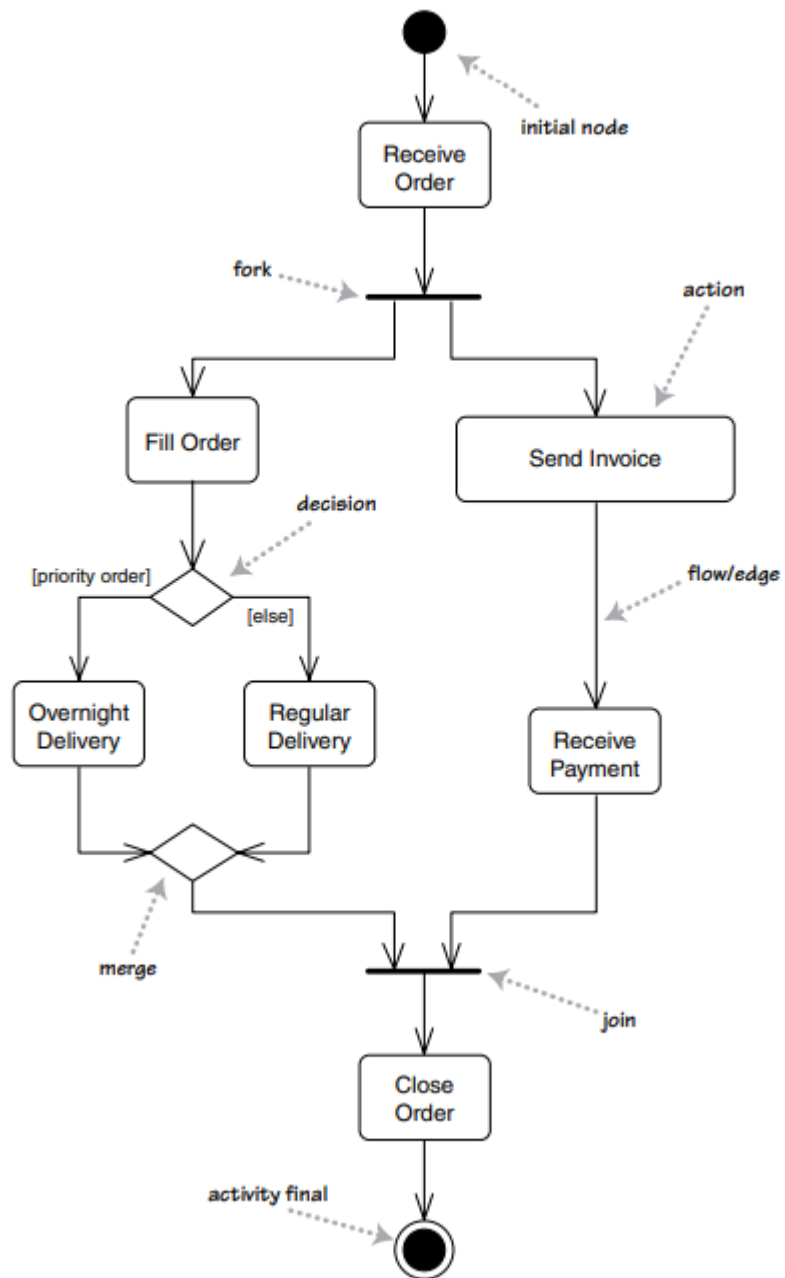
Oltre questa notazione, un pò insolita per UML, quest'ultimo fornisce anche dei diagrammi nonostante Fowler suggerisce di non mettere troppo 'effort' nei diagrammi e di concentrarci di più su una stesura di tipo testuale. In ogni caso vediamo di seguito come un diagramma UML può essere rappresentato.

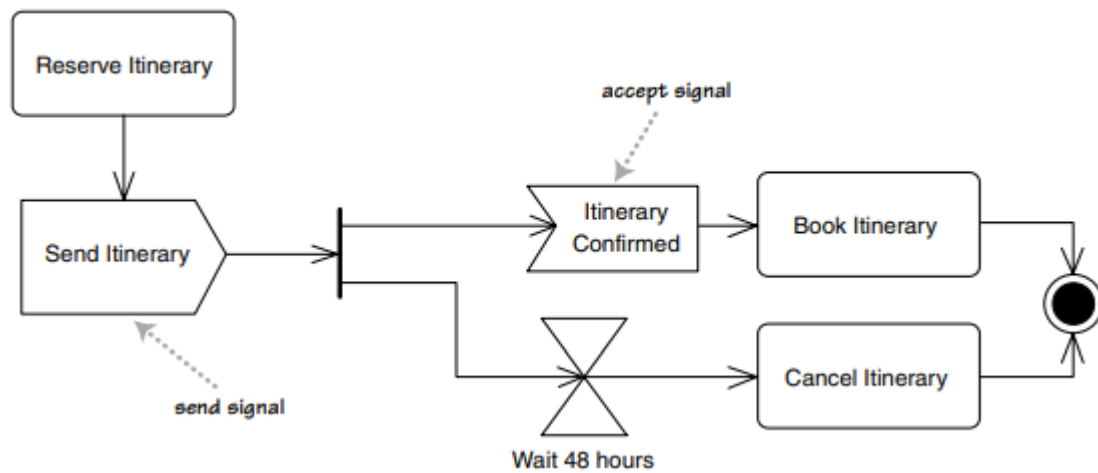


Activity Diagrams

Gli Activity Diagrams aiutano a descrivere i vari workflow, come si potrebbe intuire quindi questi diagrammi sono simili ai flowcharts. In effetti questi due diagrammi sono simili, con la differenza che gli Activity Diagrams sono in grado di descrivere il comportamento in parallelo.

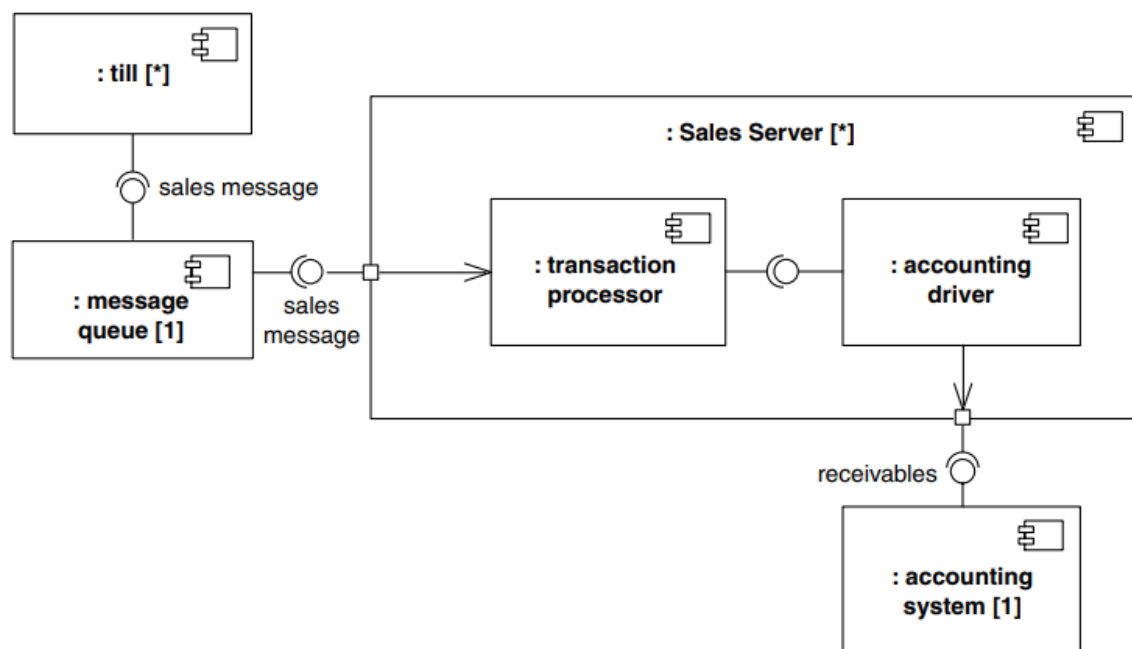
Vediamo due esempi auto esplicativi che mostrano le parti fondamentali:





Component Diagram

I component diagrams si usano per mostrare il funzionamento del sistema tramite le sue componenti e descrivono come essi si relazionano alle interfacce



Timing Diagrams

Questi diagrammi sono utili quando è necessario rappresentare il tempo come un dominio.

