Appunti di UML

Settembre 2020

Indice

1 Disclaimer

Questi appunti non sono da ritenersi né completi né corretti, comunque ci ho speso un bel po' di tempo nel farli, quindi ve li condivido nella speranza che qualcuno li usi come base per futuri ampliamenti e correzioni. Se vi siete trovati a usare appunti disponibili gratuitamente degli studenti degli anni precedenti, fate la vostra parte :)

2 Introduzione a UML

UML (Unified Modeling Language, "linguaggio di modellizzazione unificato") è un linguaggio di modellazione e di specifica basato sul paradigma orientato agli oggetti.

Nel contesto dell'ingegneria del software, viene usato soprattutto per descrivere il dominio applicativo di un sistema software e/o il comportamento e la struttura del sistema stesso. Il modello è strutturato secondo un insieme di viste che rappresentano diversi aspetti del sistema modellato (funzionamento, struttura, comportamento e così via), sia a scopo di **analisi** che di **progetto**, mantenendo la tracciabilità dei concetti impiegati nelle diverse viste. Oltre che per la modellazione di sistemi software, UML viene non di rado impiegato per descrivere domini di altri tipi, come sistemi hardware, strutture organizzative aziendali, processi di business.

Caratteristiche principali:

- Supporta lintero ciclo di vita del software: dal documento di Analisi dei Requisiti al documento di Piano di Qualifica.
- È indipendente dai linguaggi di sviluppo e programmazione
- Sintassi facile da imparare, ma semantica molto ricca [1]

Perché usare UML:

Parlare di un progetto usando solo il linguaggio naturale porta ad ambiguità perché è troppo astratto, d'altro canto discutere direttamente del codice può far perdere di vista ciò che si vuole modellare, perché i linguaggi di programmazione sono troppo concreti. UML si pone nel mezzo: non lascia spazio ad ambiguità, ma allo stesso tempo garantisce una solida visione d'insieme.

[1] Sintassi: regole attraverso le quali gli elementi di un linguaggio sono assemblati in espressioni. Semantica: regole attraverso le quali alle espressioni sintattiche viene attribuito un significato

Approcci a UML

- Come abbozzo (sketch): il più utilizzato
  - Forward engineering: il sistema (o parti di esso) è descritto con diagrammi UML prima della stesura del codice (aziende grandi)
  - Reverse engineering: Il diagramma UML è costruito a partire dal codice (aziende piccole)
- Come progetto: più ingegneristico
  - Documento di definizione di prodotto (DP): descrive formalmente il sistema per modelli con elevato grado di dettaglio non lasciando quindi nessuna decisione o interpretazione al programmatore
  - Definizione delle interfacce tra sottosistemi: i programmatori progettano e sviluppano le componenti dei sistemi in autonomia

Che cosa è quindi UML?

UML è un modello costituito da una collezione organizzata di diagrammi correlati, costruiti componendo elementi grafici (con significato formalmente definito), elementi testuali formali, ed elementi di testo libero.

Tipi più usati di diagrammi

- Diagrammi di struttura
  - Diagramma delle classi
  - Diagramma degli oggetti
  - Diagramma dei pacchetti
- Diagrammi di comportamento
  - Diagramma delle attività
  - Diagramma dei casi d'uso
  - Diagramma di flusso
  - Diagramma di sequenza

  Bisogna notare che non tutti i diagrammi sono usati regolarmente: il 20% dei digrammi viene usato l'80% delle volte (vedi Principio di Pareto)

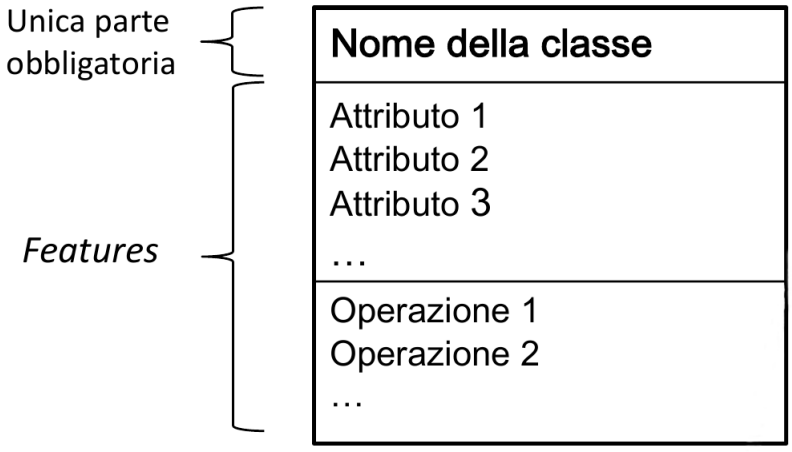
2.1 Molteplicità

La molteplicità è la definizione della cardinalità di qualche collezione di elementi tramite un intervallo di interi non negativi che specifica il numero permissibile di istanze dell'oggetto descritto.

Multiplicity	Option	Cardinality
0..0	**0**	Collection must be empty
0..1		No instances or one instance
1..1	**1**	Exactly one instance
0..*	*****	Zero or more instances
1..*		At least one instance
5..5	**5**	Exactly 5 instances
m..n		At least m but no more than n instances

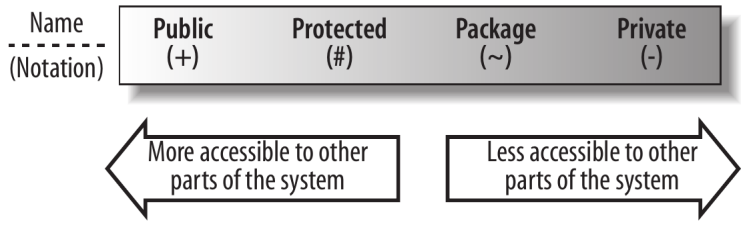
3 Diagramma delle classi

In termini generali, il diagramma delle classi consente di descrivere tipi di entità e le loro eventuali relazioni.



Ogni classe è rappresentata da un rettangolo che è suddiviso in tre compartimenti: **nome** (obbligatorio) e **attributi** e **operazioni** (opzionali).

I modificatori servono per specificare la visibilità di attributi e operazioni e sono i seguenti:



3.1 Attributi

Gli attributi possono essere rappresentati in un diagramma di classe sia posizionandoli nell'apposita sezione nel rettangolo della classe (attributi in linea) o tramite associazione con un'altra classe. Scegliere uno dei due modi dipende dal focus che si vuole dare al diagramma. Gli attributi in linea occupano poco spazio, quelli per associazione mostrano con chiarezza le relazioni tra le classi, ma occupando molto spazio possono essere d'intralcio. In generale si preferisce usare gli attributi in linea per classi semplici o tipi primitivi e quelli per associazione per le classi più complesse.

Gli attributi in linea sono riportati secondo questo formato:

```
Visibilità nome : tipo [molteplicità] = default {proprietà aggiuntive}
```

La proprietà aggiuntiva più usata è *readOnly* che ad esempio in Java si traduce con l'attributo *final*.
Se un attributo ha molteplicità maggiore di uno ciò significa che rappresenta una collezione di oggetti. (p.es. un array).
Inoltre un attributo può avere come proprietà aggiuntive ordered (array o vettori) o unordered (insiemi).

3.2 Operazioni

Le operazioni sono riportate secondo questo formato:

```
Visibilità nome (lista-parametri) : tipo-ritorno {proprietà aggiuntive}
```

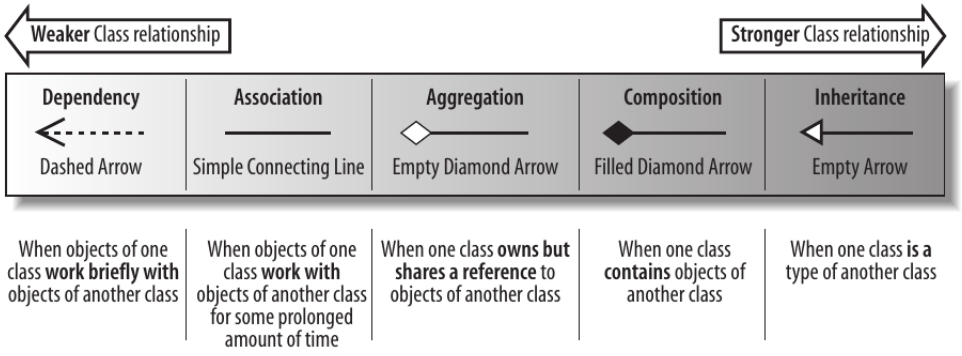
La lista parametri delle operazioni ha questo formato:

```
Lista-parametri := direzione nome : tipo = default
```

Dove direzione può essere in, out, inout (default in).

3.3 Relazioni tra classi

Si ha una relazione tra due elementi di un diagramma se la modifica alla definizione del primo (fornitore) può cambiare la definizione del secondo (cliente).



- Le dipendenze vanno minimizzate! (loose coupling)
- Da inserire solo quando danno valore aggiunto
- Troppe dipendenze creano confusione nel diagramma
- Sopra la linea, le relazioni posso avere un'etichetta: meglio un nome e non un verbo
- Evitare le relazioni bidirezionali

3.3.1 Dipendenza (reference)

- Un metodo usa un oggetto di un'altra classe
- L'oggetto **non** è messo tra i campi dati
- Non c'è legame concettuale
- Linea tratteggiata, freccia aperta

```
public class EnrollmentService {
        public void enroll(Student s, Course c){}
}
```

3.3.2 Associazione (has-a)

- Una classe ha tra i campi dati un oggetto di un'altra classe, ciò permette di associare funzioni e proprietà della prima alla seconda
- **Non** ci sono relazioni di tipo "whole-part", cioè non ci sono relazioni di composizione o aggregazione
- Linea piena con freccia aperta

```
public class Order {
        private Customer customer
}
```

3.3.3 Aggregazione (has-a + whole-part)

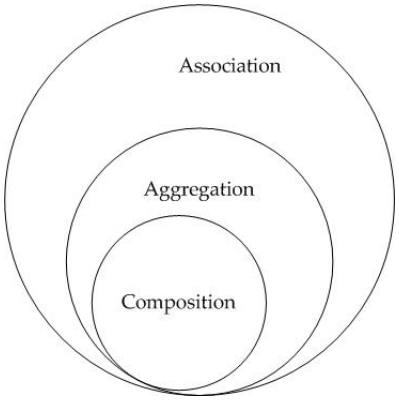
- Tipo speciale di associazione
- Relazione whole-part: una classe è parte di un'altra classe
- Un oggetto della classe figlia ha un ciclo di vita indipendente, cioè può essere creato prima dell'oggetto che lo conterrà e può vivere anche dopo la distruzione di un'oggetto della classe madre che lo conteneva.
- Un oggetto B può essere condiviso da più oggetti A
- Rombo vuoto, linea piena, freccia aperta

```
public class PlayList{
        private List<Song> songs;
}
```

3.3.4 Composizione (has-a + whole-part + ownership)

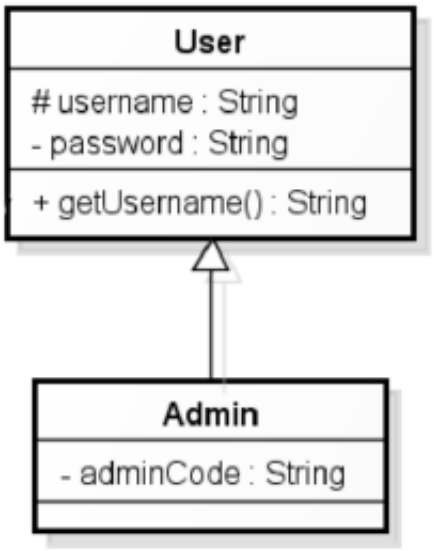
- Tipo speciale di aggregazione
- Relazione whole-part: una classe è parte di un'altra classe
- Un oggetto della classe B **non** ha un ciclo di vita indipendente, cioè può vivere solo dentro un oggetto della classe A, la cui distruzione comporta anche la distruzione dell'oggetto della classe B
- Un oggetto B **non** può essere condiviso da più oggetti A
- Rombo pieno, linea piena, freccia aperta

```
public class University{
        private Department math;
        public University() {
                math = new Department();
        }
}
```



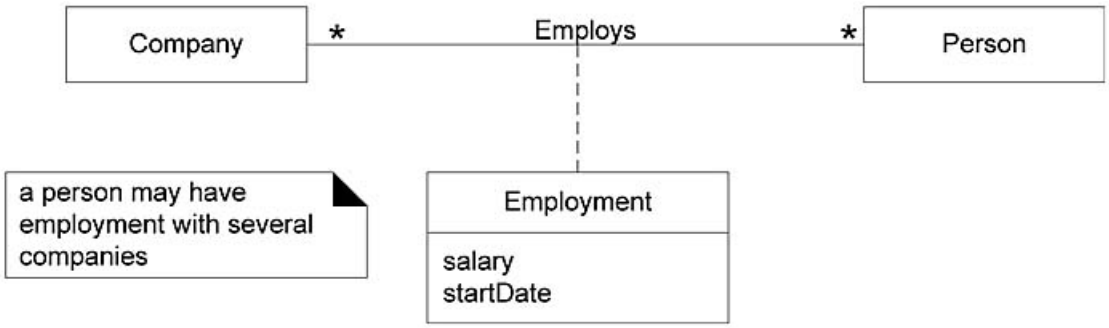
3.3.5 Generalizzazione (is-a)

- A generalizza B, se ogni oggetto di B è anche un oggetto di A
- Equivale all'ereditarietà dei linguaggi di programmazione
- Le proprietà della super-classe non si riportano nel diagramma della sotto-classe (a meno di override)
- Linea piena, freccia chiusa vuota



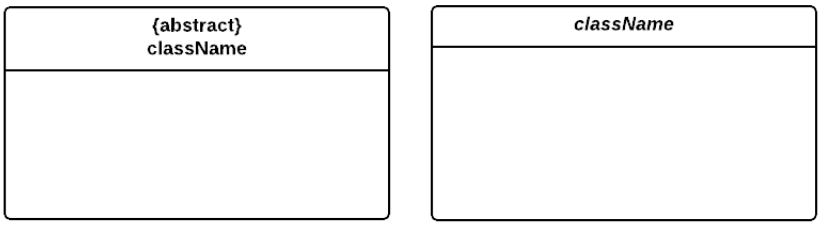
3.3.6 Classi di associazione

A volte per modellare un'associazione bisogna introdurre una nuova classe. Ne è esempio il salario di un dipendente di un'azienda.



Un rettangolo con un angolo ripiegato (come quello soprastante) è un commento. Esso può essere piazzato da solo o collegato a una parte del diagramma con una linea tratteggiata.

3.4 Classi astratte {abstract}

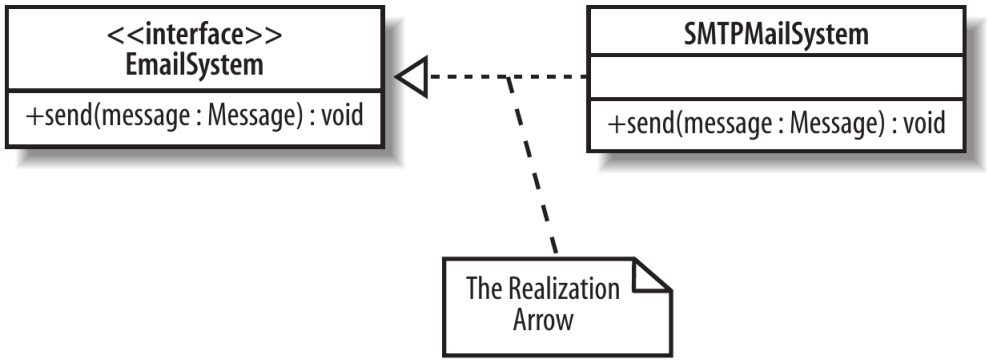


- Una classe astratta è una classe che non può essere istanziata
- Contiene almeno un'operazione astratta, cioè che non ha un'implementazione
- Può contenere anche operazioni implementate
- Il nome di una classe astratta è scritto in corsivo oppure è preceduto dalla dicitura "{abstract}"

3.5 Interfacce «interface»

- È una classe priva di implementazione
- Tutte le sue operazioni sono quindi astratte
- Una classe realizza uninterfaccia se ne implementa le operazioni

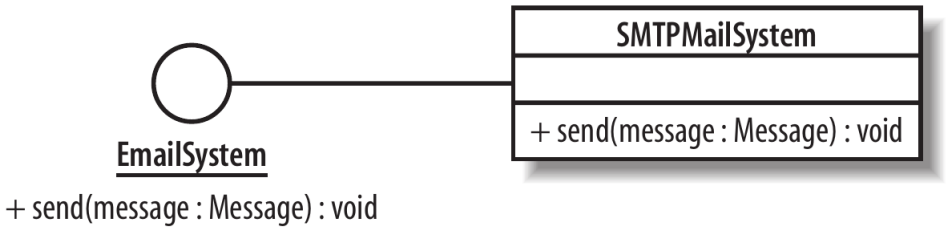
3.5.1 Stereotype notation, UML 1.x



Linea tratteggiata, freccia chiusa vuota

È da notare l'uso delle parentesi angolate per indicare una parola chiave di UML

3.5.2 Ball notation, UML 2.x

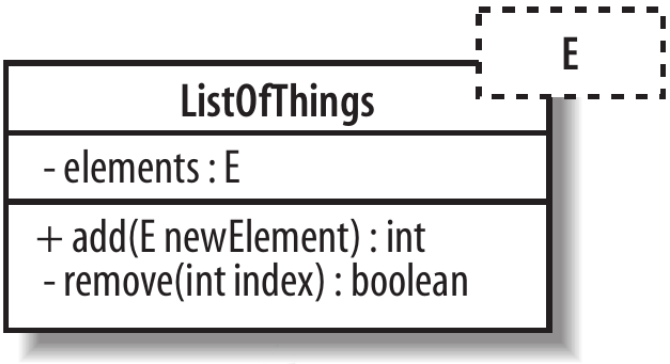


Linea continua, senza freccia

3.6 Static

Per indicare che un'operazione o un attributo sono statici nel diagramma vengono sottolineati.

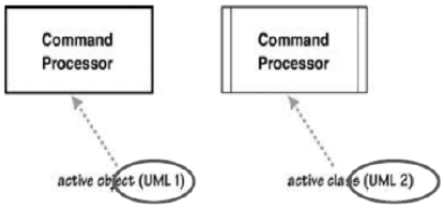
3.7 Classi parametriche



- "E" è detto segnaposto
- "E" è dentro un rettangolo tratteggiato
- In C++ template, in Java generics

3.8 Classi attive

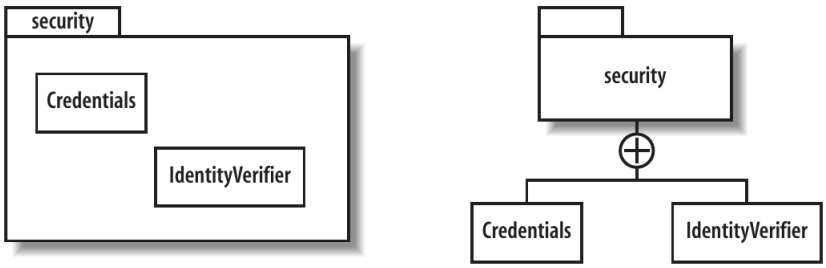
Eseguono e controllano il proprio thread



4 Diagramma dei package

4.1 Cos'è un package

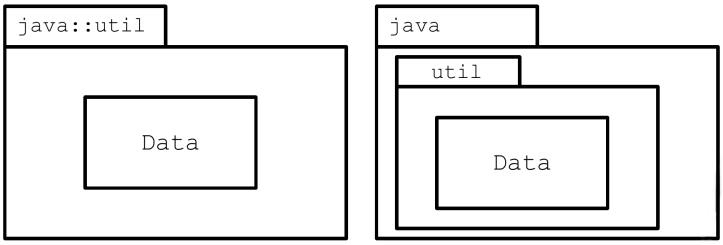
- Un package è un raggruppamento di un numero arbitrario di elementi UML in una unità di livello più alto
- Praticamente si raggruppano solo classi
- Il simbolo di un package è un cartella con una etichetta
- Ogni classe appartiene ad un solo package
- Gli elementi del package possono avere visibilità pubblica (+) o privata (-)
- L'interfaccia di un package è l'insieme dei tipi pubblici che contiene
- Un package può contenere un altro package



Un package può essere rappresento in due modi equivalenti, però nella maggior parte dei casi è preferita la rappresentazione di sinistra, dove il nome

del package è messo nell'etichetta e nella sezione principale sono posizionate le classi appartenenti a quel package. La rappresentazione di destra è considerata pesante ed è quindi usata solo in caso di mancanza di spazio.

- Il package individua un namespace
- Ogni elemento deve avere un nome distinto all'interno dello spazio di nomi

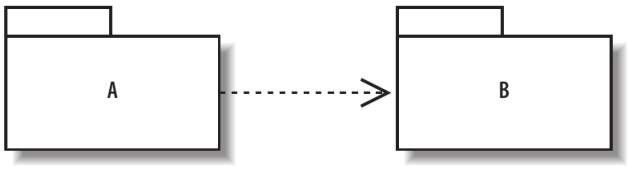


La rappresentazione del package "util" di sinistra e di destra è equivalente

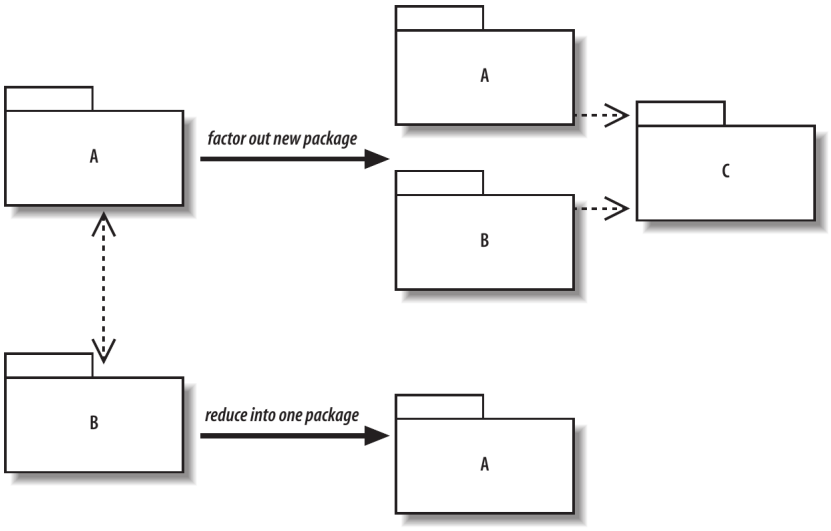
- La creazione di un package può avvenire per raggruppare le classi che condividono la stessa causa di cambiamento (Common Closure Principle) oppure per raggruppare quelle classi che dovrebbero sempre essere riusate insieme (Common Reuse Principle).

4.2 Package e Dipedenze

Il diagramma dei package documenta le dipendenze tra le classi: qualsiasi tipo di dipendenza.



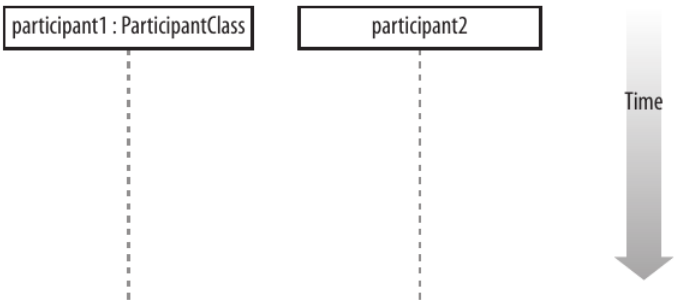
- Tutte le dipendenze dovrebbero seguire la stessa direzione
- Le relazioni di dipendenza non sono transitive: dati A,B,C tali che A dipende da B e B dipende da C, se modifico A non necessariamente devo modificare C
- Evitare le dipendenze circolari



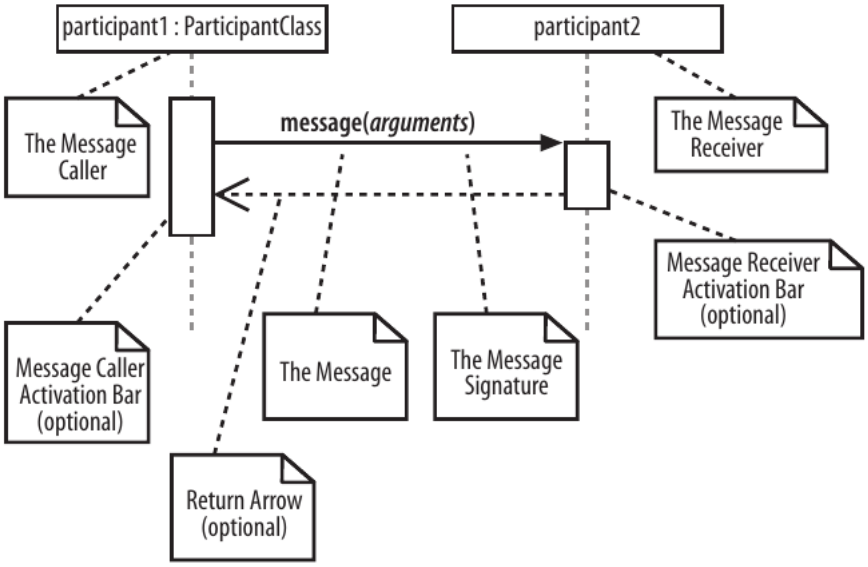
A sinistra viene mostrato che A e B sono dipendenti l'un l'altro. Sono fornite due soluzioni: una in alto a destra e una in basso a destra.

5 Diagramma di sequenza

Un diagramma di sequenza descrive la collaborazione di un gruppo di oggetti che devono implementare collettivamente un comportamento. In altre parole lo scopo di un diagramma di sequenza è descrivere l'ordine delle interazioni tra le parti di un sistema, che vengono dette partecipanti.

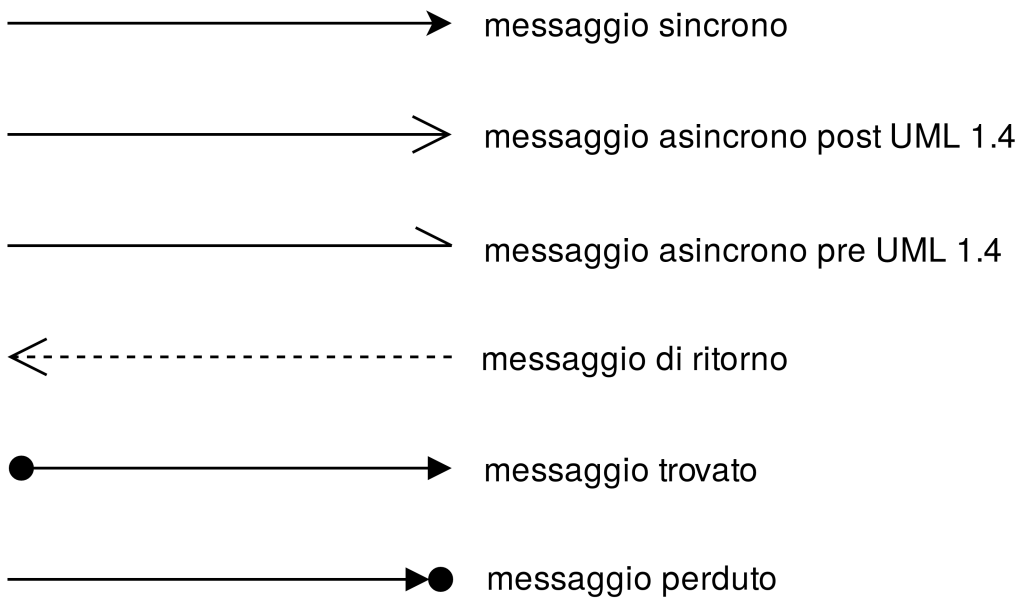


Le linee tratteggiate sono chiamate linee della vita e dichiarano che un partecipante esiste a quel punto della sequenza



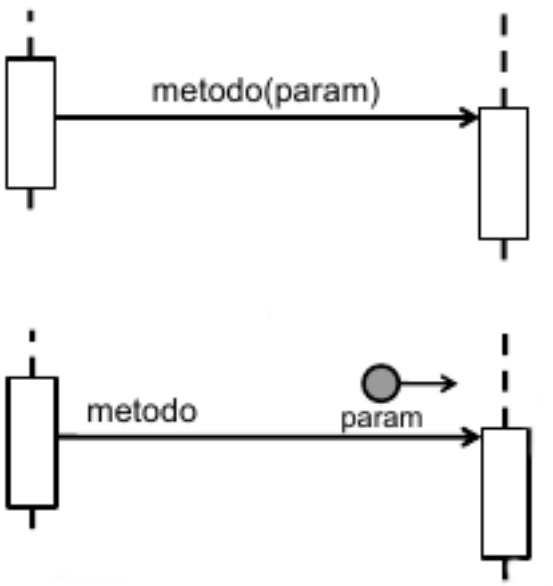
- Un'interazione avviene quando un partecipante decide di mandare un messaggio a un altro partecipante.
- Un messaggio induce il ricevente a fare qualcosa, a quel punto il ricevente è attivo, questo viene descritto opzionalmente tramite le barre di attivazione.
- Le barre di attivazione terminano quando un partecipante non è più attivo.

5.1 Messaggi



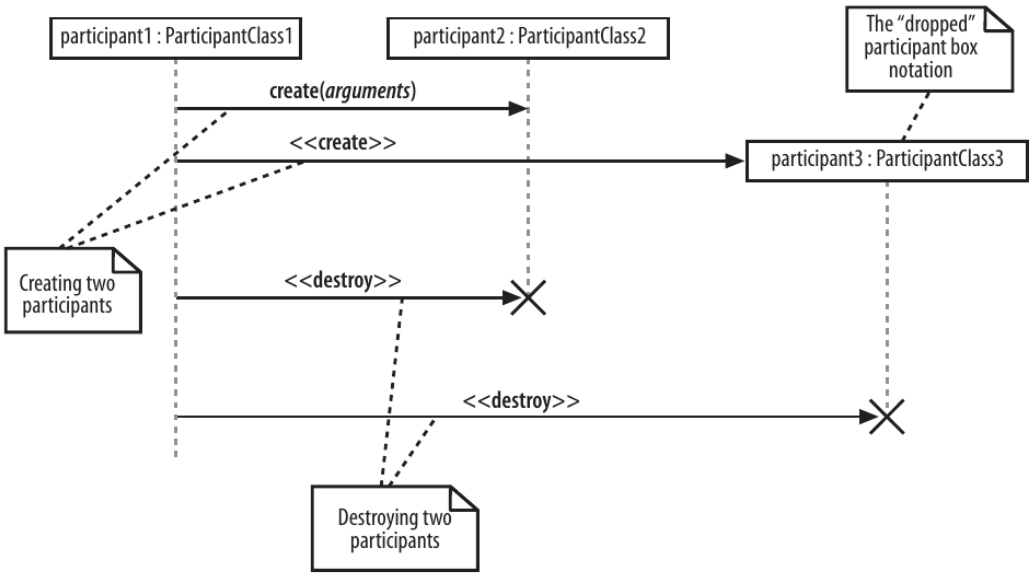
Se un chiamante manda un messaggio sincrono deve aspettare la risposta, se invece manda un messaggio asincrono può continuare a lavorare senza aspettare la risposta. I messaggi asincroni in genere migliorano la responsività ma sono più difficili da debuggare.

5.2 Passaggio parametri



Il passaggio dei parametri può essere descritto con il metodo classico che prevede di inserire tra parentesi tonde la lista dei parametri, oppure con il metodo dei girini dei dati (data tadpoles).

5.3 Creazione e distruzione partecipanti

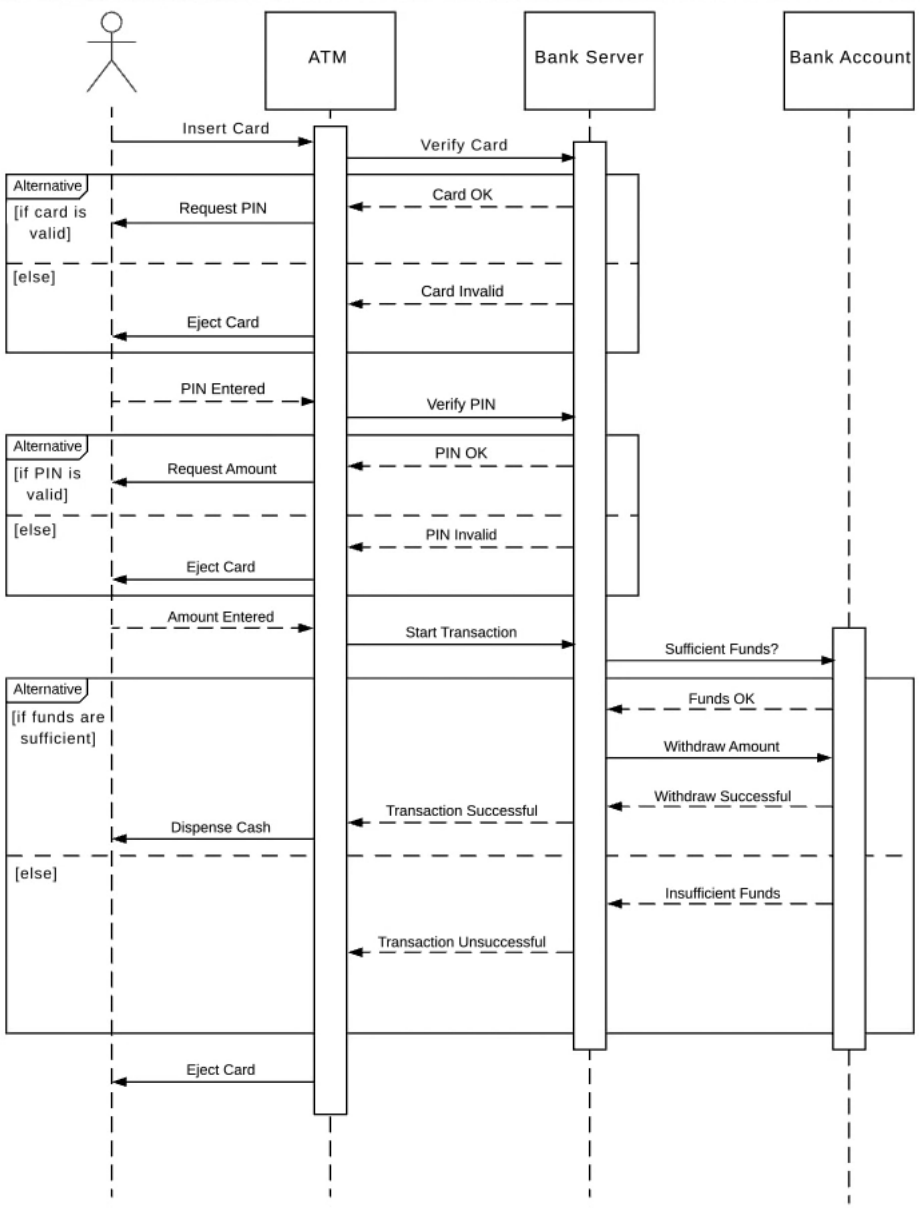


Un partecipante viene creato tramite un messaggio che può essere rappresentato con il metodo *create(...)* o la parola chiave *«create»*. La creazione di un partecipante si può evidenziare posizionando il rettangolo del partecipante da creare in posizione ribassata al livello che corrisponde al momento della sua creazione(il tempo scorre verso il basso).

Se la distruzione è causata da un messaggio si usa la parola chiave *«destroy»* e una croce di cancellazione, se invece è automatica si usa solo la croce.

5.4 Frammento (frame) di interazione

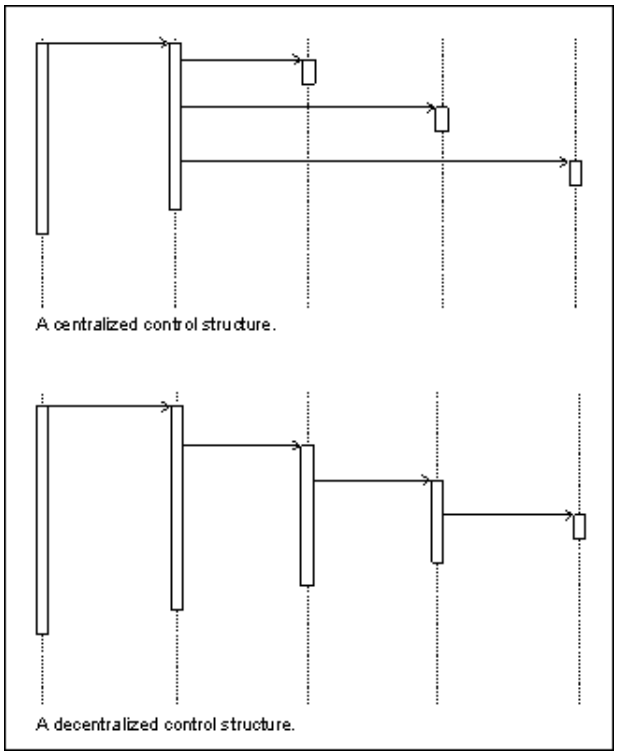
Modellare la logica di controllo (cicli e condizioni) non è cosa enfatizza meglio l'utilità dei diagrammi di sequenza, poiché sarebbe meglio usare un diagramma delle attività o il codice. Detto ciò è comunque possibile farlo dentro un diagramma di sequenza tramite i frammenti di interazione.



Poiché è (arbitrariamente) sotto inteso, faccio notare che *Eject Card* porta alla terminazione del programma. Quindi le operazioni sotto di esso non sono esaminate se *Eject Card* viene invocato. Le alternative per essere precise dovrebbero essere state innestate, ma questo avrebbe creato un diagramma poco leggibile.

Operator	Meaning
alt	Alternative multiple fragments; only the one whose condition is true will execute (Figure 4.4).
opt	Optional; the fragment executes only if the supplied condition is true. Equivalent to an alt with only one trace (Figure 4.4).
par	Parallel; each fragment is run in parallel.
loop	Loop; the fragment may execute multiple times, and the guard indicates the basis of iteration (Figure 4.4).
region	Critical region; the fragment can have only one thread executing it at once.
neg	Negative; the fragment shows an invalid interaction.
ref	Reference; refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram; used to surround an entire sequence diagram, if you wish.

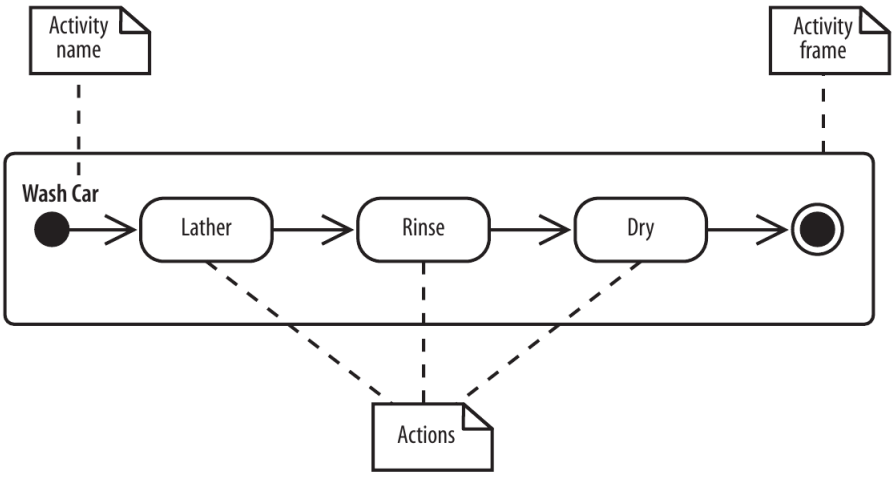
5.5 Controllo elaborazione



- Controllo centralizzato: un unico partecipante governa l'elaborazione, gli altri semplicemente forniscono i dati necessari

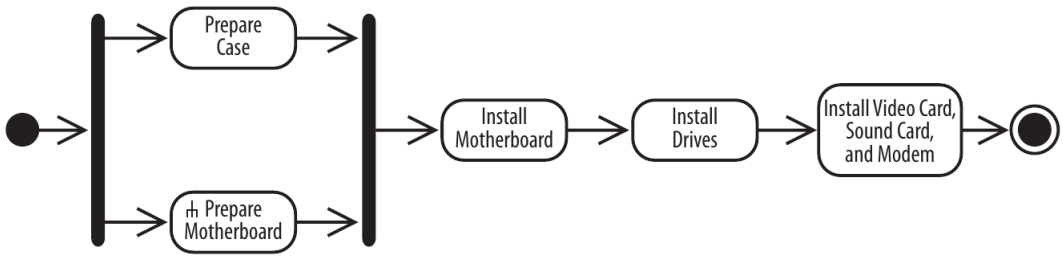
- Controllo distribuito: i compiti vengono suddivisi tra partecipanti

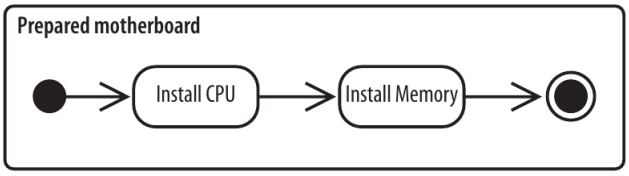
6 Diagramma di attività



- I diagrammi di attività descrivono la logica procedurale, i processi di business e il flusso del lavoro
- Un'attività è il processo che viene modellato, un'azione è un passo di un'attività

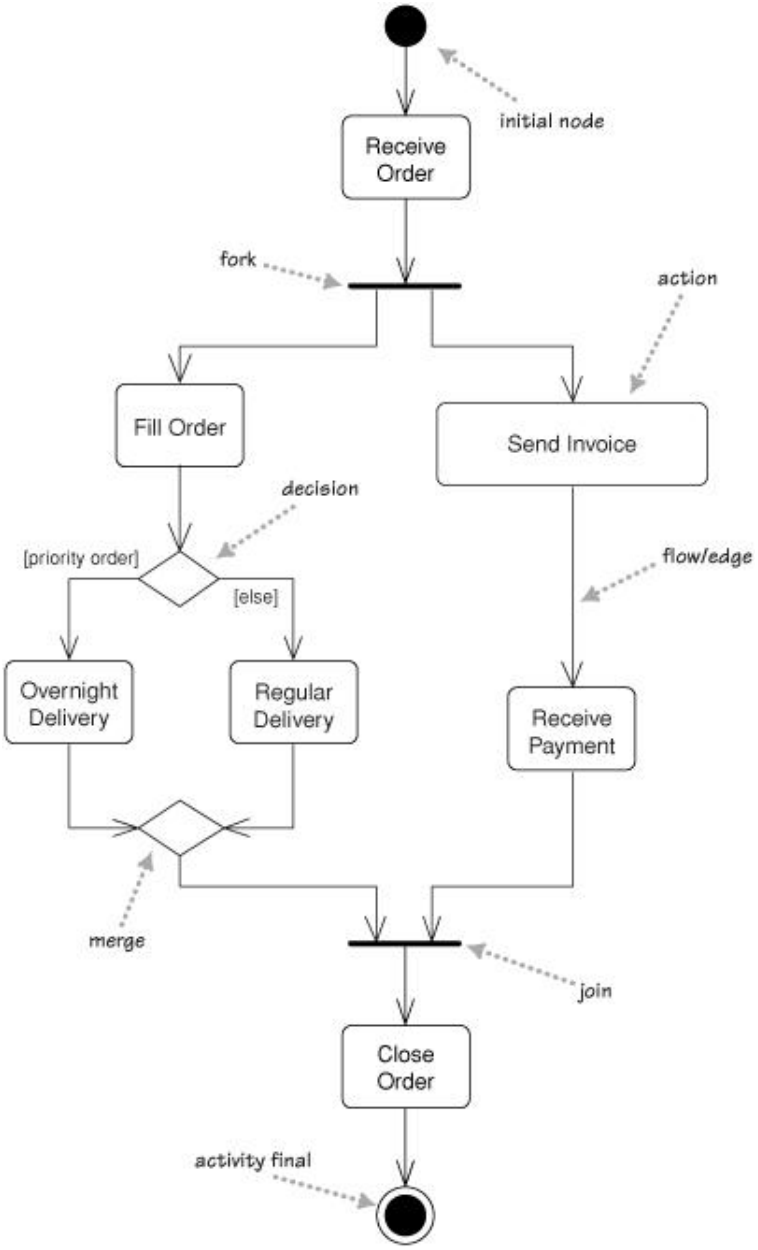
6.1 Chiamata ad un'altra attività





Un'attività può richiamarne un'altra con il simbolo del tridente.

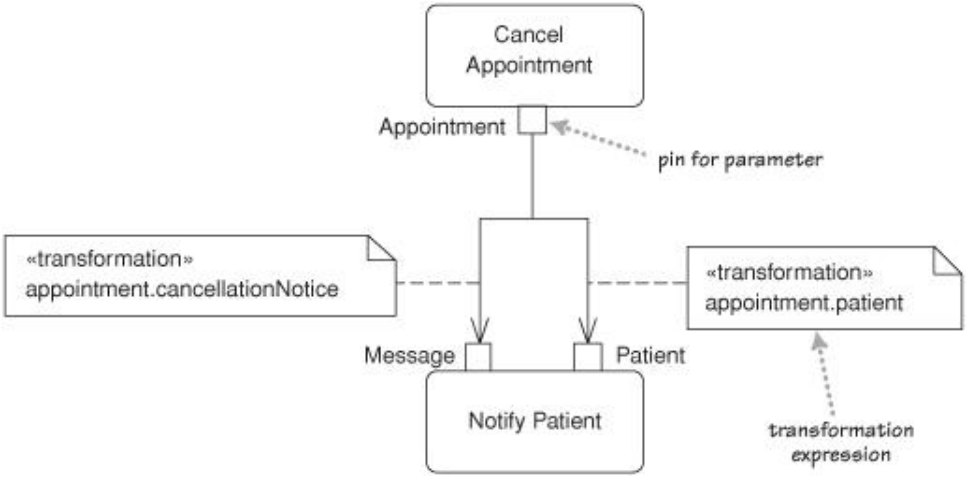
6.2 Fork e decisioni



- L'elaborazione parallela [2], è supportata tramite fork(inizio) e join(fine).
- Il fork ha un solo flusso entrante e molteplici uscenti. I flussi uscenti non hanno un ordine di esecuzione.
- Il join sincronizza i processi iniziati dal fork, in altre parole permette il proseguimento dell'esecuzione solo se tutti i flussi hanno terminato.
- L'elaborazione condizionale è supportata da decisioni(inizio) e merge(fine); permette di scegliere un solo percorso.
- Decisione: rombo con un flusso entrante e multipli uscenti, condizioni tra parentesi quadre.
- Merge: rombo con flussi multipli entranti e un solo flusso uscente.

6.3 Pin e trasformazioni

Le azioni possono avere parametri, così come i metodi. I parametri si possono mostrare opzionalmente con i *pin*. Se i parametri di output non corrispondono a quelli di input si dice che deve avvenire una trasformazione. Una trasformazione è un'espressione senza effetti collaterali, in pratica una query che fornisce al pin di input il tipo giusto.

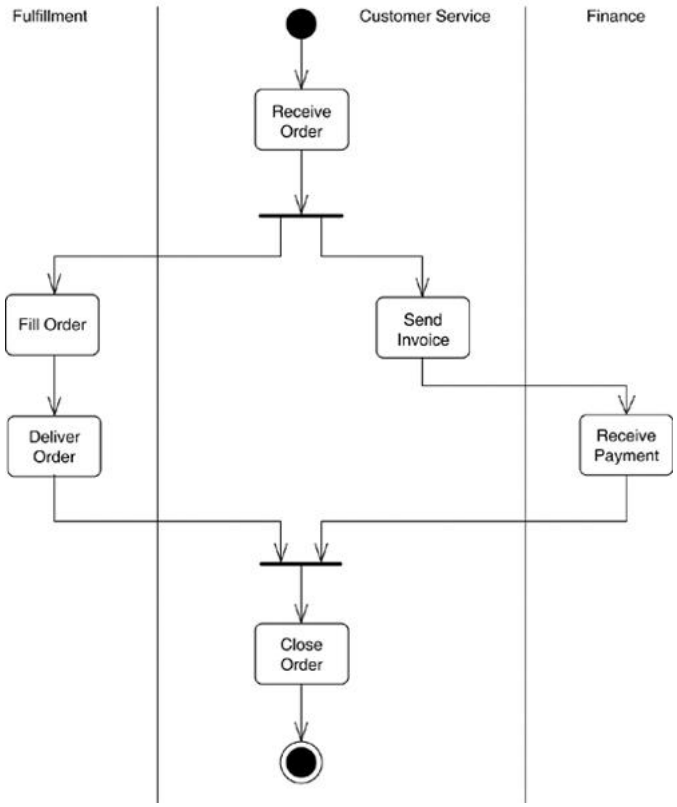


[2]A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously. The key concept and difference between these definitions is the phrase "in progress." This definition says that, in concurrent systems, multiple actions can be in progress (may not be executed) at the same time. Meanwhile, multiple actions are simultaneously executed in parallel systems.

Nello sviluppo di modelli di business si possono usare i pin per mostrare le risorse prodotte e consumate dalle azioni.

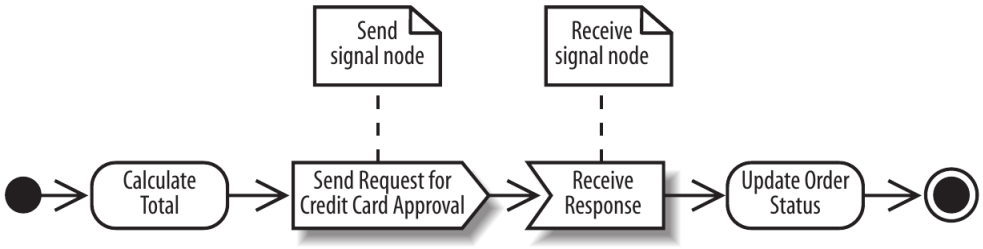
6.4 Partizioni (swimlanes)

Per mostrare chi fa cosa si può dividere il diagramma di attività in partizioni. Nell'esempio viene mostrato come le azioni per processare un ordine si possono separare in base al dipartimento che ne è responsabile.

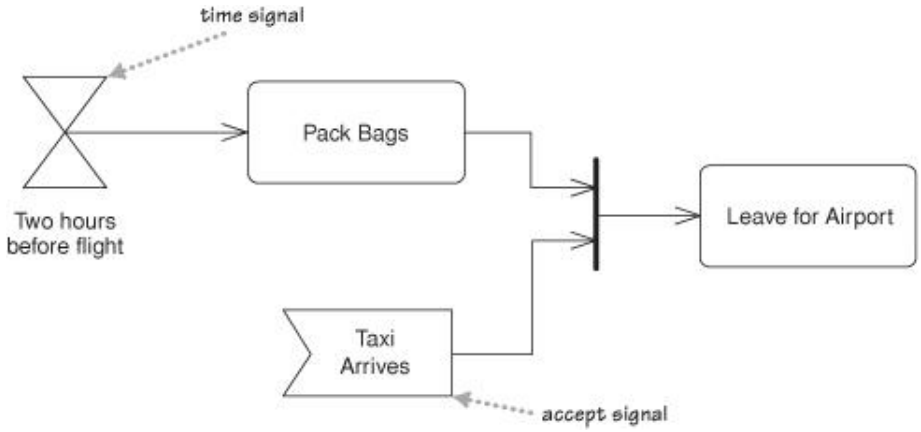


6.5 Segnali

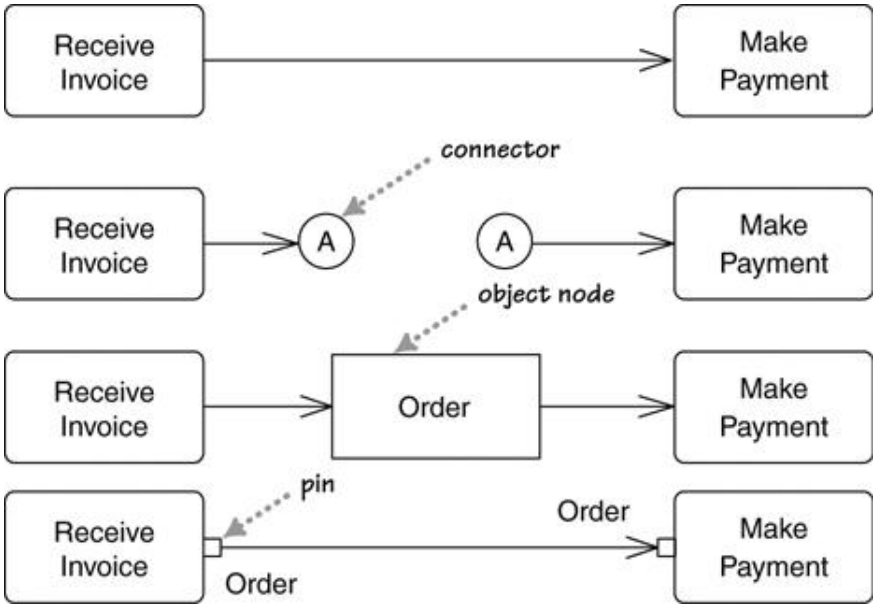
Nei diagrammi di attività i segnali rappresentano le interazioni con i partecipanti esterni. Un segnale di ricezione(receive) sveglia un'azione nel diagramma. Un segnale di invio (send) è un segnale inviato a un partecipante esterno,che probabilmente quando lo riceverà farà qualcosa, ma ciò non viene modellato.



Un programma richiede a una compagnia di carte di credito di approvare una transazione



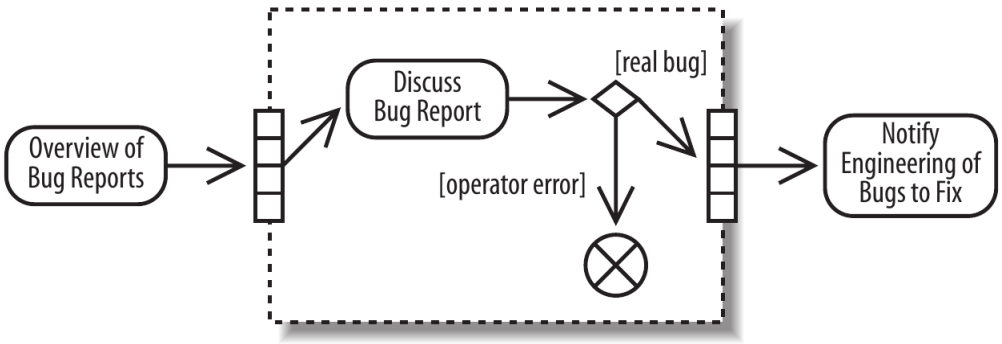
DA FARE!!!!!!!!!!!!!!!!!!!



Quattro modi di mostrare un flusso

6.6 Regioni di espansione

Una regione di espansione mostra che le azioni dentro di essa sono eseguite per ogni elemento di una collezione in ingresso. Viene disegnata come un rettangolo con i bordi tratteggiati con quattro quadrati allineati su due lati opposti. I quadrati rappresentano delle collezioni generiche, il fatto che siano quattro con comporta che la dimensione della collezione sia quattro.



Il nodo con una croce dentro rappresenta la fine del flusso (attenzione non dell'intera attività)

7 Casi d'uso e relativo diagramma

7.1 Casi d'uso

- I casi d'uso sono una tecnica per individuare i requisiti funzionali di un sistema.
- I casi d'uso descrivono le interazioni tra gli utenti e il sistema specificando in questo modo come il sistema è usato.
- **Scenario**: sequenza di passi che descrivono un'interazione tra un utente e un sistema
- Si può definire un caso d'uso come un insieme di scenari legati tra loro da un *fine* comune dell'utente.
- I casi d'uso rappresentano una visione esterna del sistema, quindi non bisogna aspettarsi alcuna correlazione tra i casi d'uso e le classi del sistema.
- Nei casi d'uso è convenzione chiamare gli utenti **attori** (che siano umani o meno)

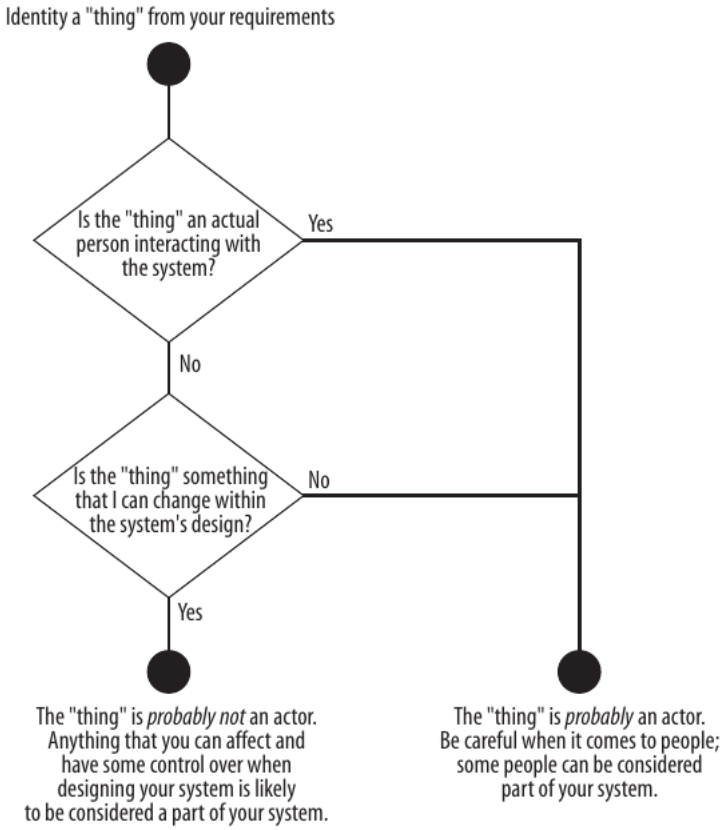


Figura 12: Come identificare gli attori

7.1.1 Contenuto dei casi d'uso

Il contenuto di un caso d'uso inizia con la scrittura del principale scenario di successo come una sequenza di passi numerati. Gli altri scenari vanno scritti come estensioni, descrivendoli in termini di variazioni dallo scenario di successo principale. Le estensioni possono essere dei successi(l'utente raggiunge il suo obbiettivo) o fallimenti. Detto questo bisogna dire che non è stato definito un modo standard per descrivere testualmente i casi d'uso, quindi circolano vari modelli (template) che si possono o meno seguire. (cercare su Internet "template use case")

Buy a Product

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

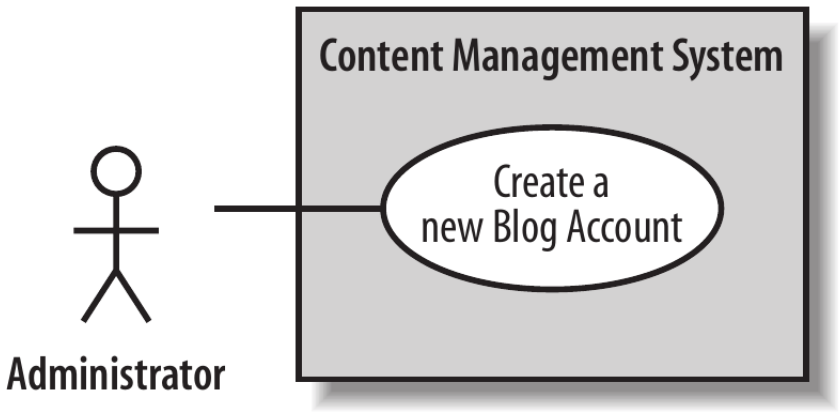
6a: System fails to authorize credit purchase
.1: Customer may reenter credit card information or may cancel

Figura 13: Descrizione testuale casi d'uso

7.2 Diagramma dei casi d'uso

Rappresentazione grafica dei casi d'uso.

- Nodi del grafo possono essere:
  - Attori
  - Use case
- Archi del grafo rappresentano:
  - Comunicazioni tra attori e use case
  - Legami tra use case che possono essere:
    * Relazione di estensione
    * Relazione di inclusione
    * Relazione di generalizzazione

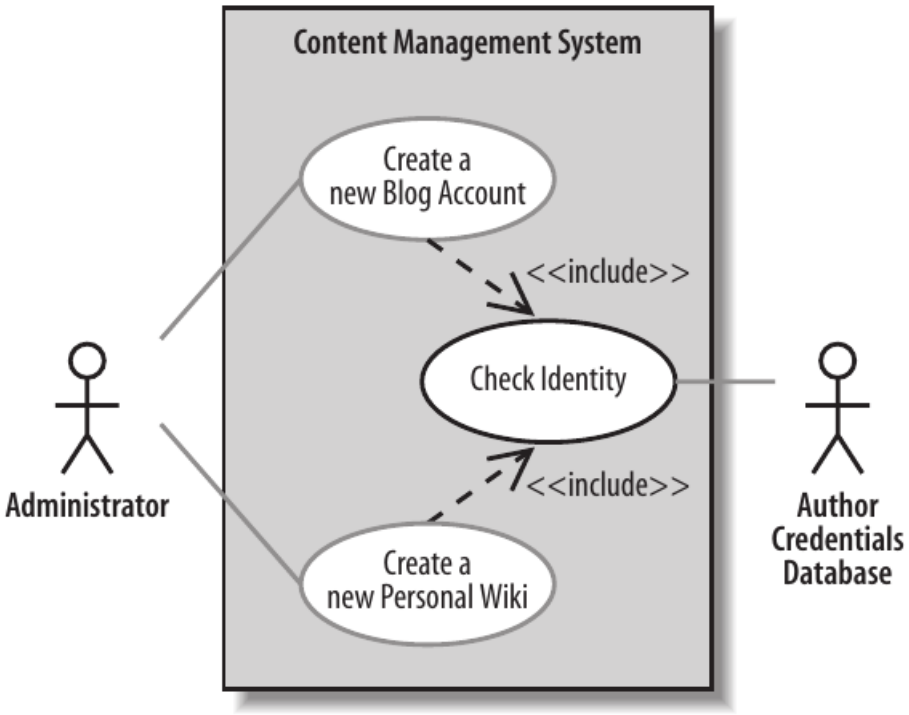


- Attori sono rappresentati da degli "stick man"
- I casi d'uso sono rappresentati da un ovale
- Una linea di comunicazione unisce attore e caso d'uso e indica che l'attore *partecipa* al caso d'uso
- I limiti del sistema sono rappresentati con un rettangolo. Gli attori naturalmente sono fuori di esso perché esterni al sistema

7.2.1 Relazione di inclusione («include»)

Se due o più use case hanno dei passi identici è meglio separarli e inserirli in un use case nuovo che viene incluso in quello di partenza. Si dice quindi che A include B se A riusa *tutti* i passi di B

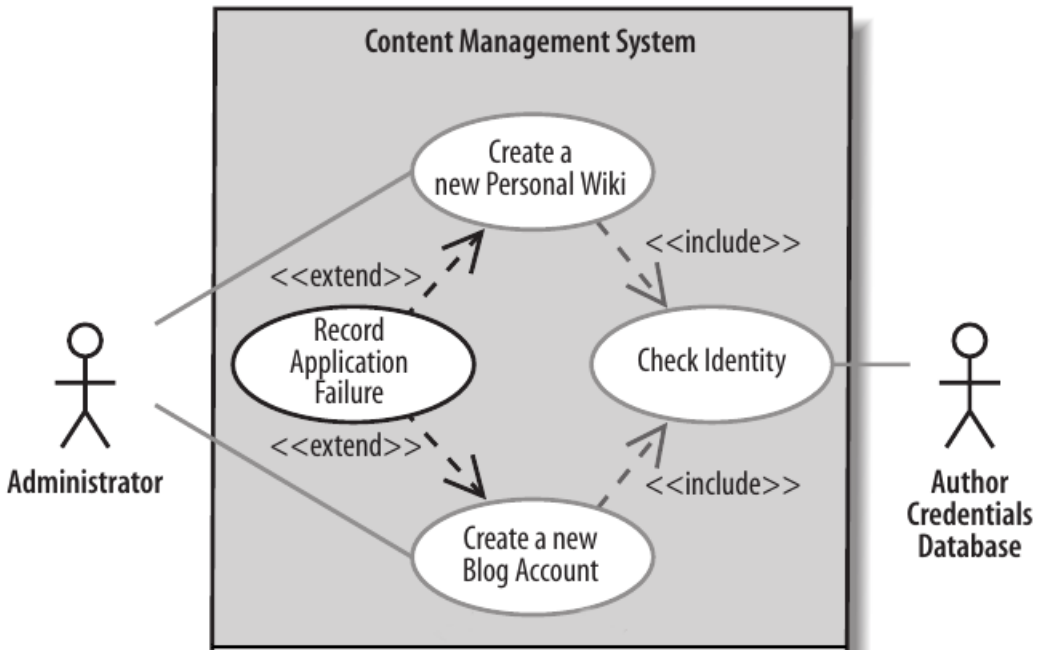
- Ogni istanza di A esegue B
- B è incondizionatamente incluso nell'esecuzione di A
- A non conosce i dettagli di B, ma solo i suoi risultati
- B non sa di essere stato incluso da A
- La responsabilità dell'esecuzione di B è completamente di A
- Utilizzo: in caso di funzionalità che si ripetono in più use case



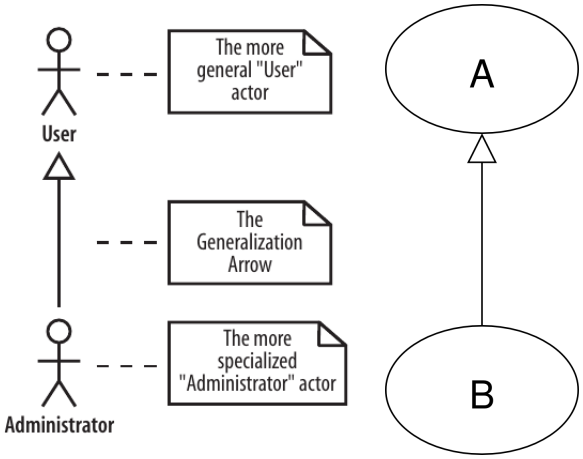
7.2.2 Relazione di estensione («extend»)

Intuitivamente verrebbe da pensare che «extend» abbia un significato simile a quello che ha in Java (o C++), cioè di ereditarietà in OOP, ma sfortunatamente non è così. In questo contesto se A estende B significa che A *potrebbe* completamente riusare B. Quindi è simile a «include» ma con la differenza che il riuso è opzionale.

- Ogni istanza di A esegue B in modo condizionato
- L'esecuzione di B interrompe A
- La responsabilità dei casi di estensione è di chi estende B
- Utilizzo: s vogliono descrivere variazioni dalla funzionalità standard



7.2.3 Relazione di generalizzazione

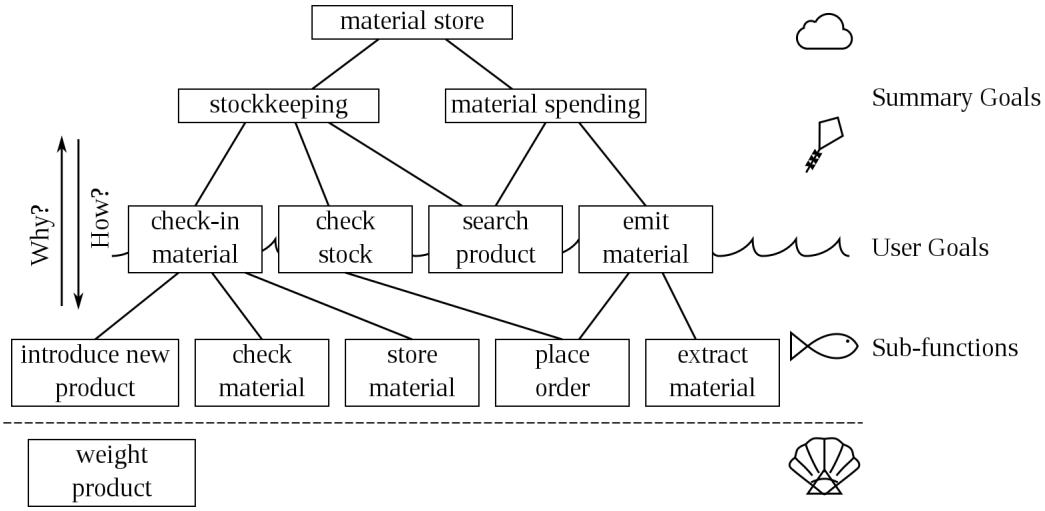


Attori: A è generalizzazione di B se B condivide almeno le funzionalità di A.

Use case: I casi d'uso figli possono aggiungere funzionalità rispetto ai padri, o modificarne il comportamento. Tutte le funzionalità non ridefinite nel figlio si mantengono in questo come definite nel padre.

7.3 Livello di dettaglio (goal level)

Goal level	Icon
Very Hight Summary	Cloud
Summary	Flyng Kite
User Goal	Waves at Sea
Subfunction	Fish
Too Low	Seabed Clam-Shell



Lista (non completa) di link in cui ho trovato informazioni utli

[1] https://stackoverflow.com/questions/885937/what-is-the-difference-between-association-aggregation-and-composition , quarta risposta (51 upvotes)

[2] https://it.wikipedia.org/wiki/Unified_Modeling_Language

[3] https://it.wikipedia.org/wiki/Principio_di_Pareto

[4] https://tallyfy.com/uml-diagram/

[5] https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/

[6] https://www.ibm.com/support/knowledgecenter/SS8PJ7_8.5.1/com.ibm.xtools.sequence.doc/topics/cmsg_v.html

[7] https://www.uml-diagrams.org/sequence-diagrams-combined-fragment.html

[8] https://www.ibm.com/support/knowledgecenter/SS8PJ7_8.5.1/com.ibm.xtools.sequence.doc/topics/cmsg_v.html

[9] https://www.youtube.com/watch?v=pCK6prSq8aw

[10] https://takuti.me/note/parallel-vs-concurrent/

[11] https://en.wikipedia.org/wiki/Use_case

[12] https://www.uml-diagrams.org/sequence-diagrams-combined-fragment.html

[13] https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/diagrammi-sequenza-uml/

[14] https://www.youtube.com/watch?v=pCK6prSq8aw