

INGEGNERIA DEL SOFTWARE

UML

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it

web: <http://www.morpheusweb.it>

PERCHÉ UML?	4
PUNTI DI FORZA	4
MODELLO	5
DIAGRAMMI UML	5
VISTE STATICHE	5
VISTE DINAMICHE	5
USE CASE DIAGRAM	6
COSA NON SI PUÒ FARE	14
DOCUMENTAZIONE	14
SCENARI	15
INTERACTION DIAGRAM	16
SEQUENCE DIAGRAM	16
TIPI DI MESSAGGI	16
ETICHETTE	18
CREAZIONE E DISTRUZIONE	19
DESCRIZIONE DEL TEMPO	19
BARRE DI ESECUZIONE	20
MULTITHREADING	21
COLLABORATION DIAGRAM	22
MULTIOGGETTO	23
OGGETTI COMPOSTI	23
OGGETTI ATTIVI	23
MODEL CONTROL VIEW	24
ACTOR	25
CLASS DIAGRAM	26
CLASSE	26
ATTRIBUTI	26
FUNZIONI	27
RELAZIONI TRA CLASSI	27
ASSOCIAZIONE	27
AGGREGAZIONI	30
EREDITARIETA'	30
INTERFACCE	30
PACKAGE	31
OBJECT DIAGRAM	32
STATECHARTS DIAGRAM	33
ELEMENTI GRAFICI	33
OPERAZIONI	34
RAGGRUPPARE GLI STATI	35
SOTTOSTATI NASCOSTI	37
DECOMPOSIZIONE OR	38
DECOMPOSIZIONE AND	38
HISTORY	39
FORK/JOIN	40
EVENTI	40
ACTIVITY DIAGRAM	41
ELEMENTI GRAFICI	41

SWIMLANE	42
OGGETTI GENERATI	43
IMPLEMENTATION DIAGRAM.....	44
COMPONENT DIAGRAM	44
RESIDE.....	44
IMPLEMENT	45
DEPLOYMENT DIAGRAM.....	46
SUBSYSTEM	47
MODELLI.....	48
MODELLAZIONE.....	49
DESIGN PATTERN.....	50
CREAZIONALI: FACTORY.....	51
CREAZIONALI: SINGLETON.....	53
STRUTTURALI: ADAPTER	54
STRUTTURALI: COMPOSITE.....	55
STRUTTURALI: DECORATOR	56

PERCHÉ UML?

- E' il linguaggio visuale standard per definire, progettare, realizzare e documentare i sistemi software
Può essere usato in quasi tutti i domini applicativi
- riunisce molte proposte esistenti (Booch, Rumbaugh e Jacobson)
- è uno standard proposto da OMG
- copre l'intero processo di produzione tramite 7 tipologie di diagrammi
- è sponsorizzato dalle maggiori industrie produttrici di software
- riunisce aspetti dell'ingegneria del software, delle basi di dati e della progettazione di sistemi
- è indipendente da qualsiasi linguaggio di programmazione (che potrà essere scelto al momento più opportuno)
- è utilizzabile in domini applicativi diversi e per progetti di diverse dimensioni
- è estendibile per modellare meglio le diverse realtà (come processi real-time o web, la notazione però non è più standard)

PUNTI DI FORZA

- è sufficientemente espressivo ma può essere esteso
- è sufficientemente preciso ma non troppo difficile
- non richiede particolari conoscenze teoriche
 - è usabile con i più diffusi linguaggi di programmazione ad oggetti
 - non impone alcun processo di sviluppo predefinito
 - non impone l'acquisto di strumenti di sviluppo particolari

MODELLO

I modelli:

- rappresentano il linguaggio dei progettisti
- rappresentano il sistema da costruire o costruito
- sono un veicolo di comunicazione
- descrivono in modo “visuale” il sistema da costruire
- sono uno strumento per gestire la complessità (siamo in grado di perdere alcuni dettagli non necessari in un particolare contesto ed astrarre ciò che ci interessa; con un modello astratto è più chiaro capire cosa l’applicazione deve fare).
- consentono di analizzare caratteristiche particolari del sistema

DIAGRAMMI UML

VISTE STATICHE

Descrivono la struttura statica dell’applicazione

- Use Case Diagrams (per la parte di requirements; COSA l’applicazione deve fare)
- Class Diagrams (design)
- Object Diagrams (design)
- Component Diagrams (implementazione)
- Deployment Diagrams (implementazione)

VISTE DINAMICHE

Descrivono come l’applicazione può evolvere (cosa può fare e cosa no)

- Sequence Diagrams (requirements e design)
- Collaboration Diagrams (requirements e design)
- Statechart Diagrams (design ed implementazione)

- Activity Diagrams (design ed implementazione)

USE CASE DIAGRAM

Fa vedere cosa l'applicazione può fare; è uno dei diagrammi più di alto livello.

Il primo passo di "qualsiasi" processo di sviluppo è la definizione dei requisiti. Occorre avere le idee chiare su cosa l'applicazione deve fare.

Gli USE CASE:

- definiscono il comportamento del sistema (cosa deve fare e non come)
 - Le funzionalità (processi) principali
 - Come il sistema agisce e reagisce (ambiente \Leftrightarrow utenti)
- descrivono
 - Il sistema
 - L'ambiente
 - Le relazioni fra sistema e ambiente
 - ogni diagramma può avere diversi livelli di dettaglio

Definisco **cosa** il sistema offre (servizi per l'utente finale...)

Non definisco **come** il sistema è realizzato (implementazione, dettagli interni del sistema, architettura, topologia dell'applicazione...)

ELEMENTI GRAFICI



Actor

ACTOR: è qualcuno (utente) o qualcosa (sistemi esterni - hardware) che:

- Controlla le funzionalità
- Fornisce input o riceve output dal sistema

Non fa parte del sistema da sviluppare

Rappresenta una classe di utenti e non un singolo utente

ESEMPIO: APPLICAZIONE PER FILE

Si vuole realizzare una semplice applicazione di condivisione di file musicali (alla Napster).

Per poter utilizzare l'applicazione, gli utenti devono autenticarsi fornendo una login ed una password. A questo punto possono effettuare delle ricerche specificando, attraverso vari criteri, quali file stanno cercando. Una volta trovato il file desiderato, l'utente può decidere di scaricarlo. A sua volta l'utente deve essere in grado di scegliere quali file vuole mettere in condivisione con gli altri utenti. E' inoltre necessario prevedere alcune funzionalità destinate solo all'amministratore dell'applicazione, come il blocco di un utente e la terminare l'applicazione.

Gli utenti dell'applicazione utilizzeranno Internet per comunicare fra loro e per scambiarsi i file.

ACTOR

Gli actor identificano i ruoli degli utenti

- Più utenti con lo stesso ruolo
- Più ruoli per il medesimo utente

Attraverso gli actor definisco cosa deve fare il sistema

- Definizione lista attori
- Identificazione obiettivi
- Interazioni col sistema
- Funzionalità (use case) richieste
- Scomposizione gerarchica



User

Utente dell'applicazione



Administrator

Amministratore dell'applicazione (non fa sharing)

La RETE non è un attore dell'applicazione in quanto non è visibile all'esterno e non offre/utilizza alcuna funzionalità dell'applicazione, è solo un dettaglio implementativo.

A volte potrebbe essere definita come un actor, ad esempio se descrivo un web server, si può vedere la rete come l'actor che utilizza il web server.

USE CASE

Identificati gli actor, vediamo quali sono le funzionalità, cosa l'applicazione deve fare.

Gli use case possono essere ricavati dalle interviste con gli utenti. Identificano:

- Gli obiettivi: ciò che il sistema dovrebbe fare secondo gli utenti
- Le interazioni: cosa vorrebbero (potrebbero) fare i diversi utenti e come dovrebbe interagire il sistema con gli altri già presenti

Gli use case di alto livello sono volutamente generici

- I dettagli vanno aggiunti raffinando le funzionalità del sistema
- Bisogna esplorare le diverse possibilità per non introdurre prematuramente scelte progettuali

FUNZIONALITA' FONDAMENTALI

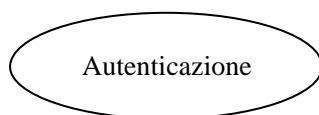
Il processo di definizione degli use case è iterativo

- Si inizia identificando il comportamento più semplici
- Si descrivono i comportamenti alternativi e più complessi

Quando smettere?

- Un buon livello di dettaglio facilita tutte le attività successive
- Troppi dettagli: complicherebbero inutilmente la descrizione, introdurrebbero prematuramente scelte progettuali, e precluderebbero la visione d'insieme

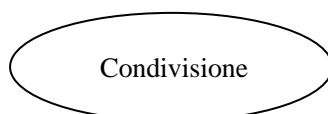
USE CASE BASE



Voglio sapere chi condivide



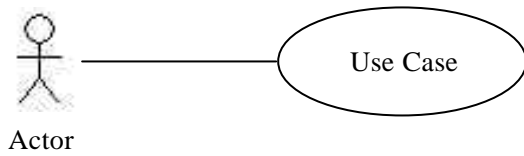
File disponibili, traffico effettivo...



L'utente deve condividere ed essere in grado di scaricare

Occorre capire la differenza tra user ed administrator (cosa possono fare)

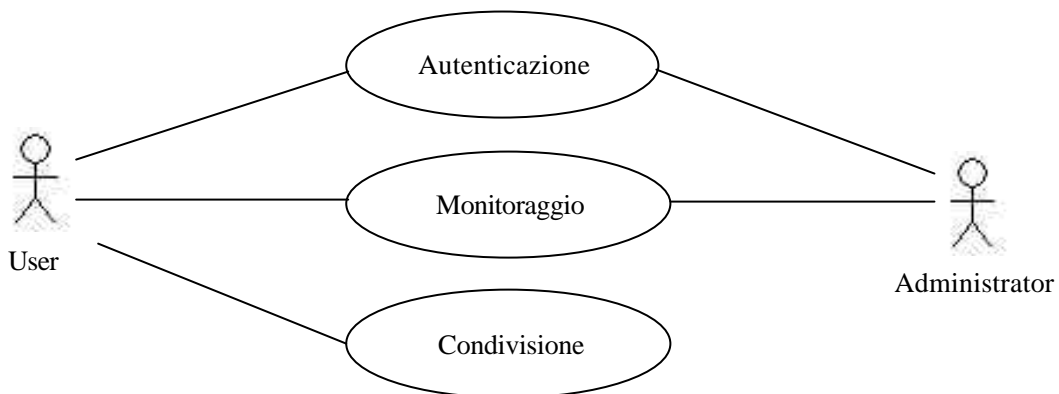
ASSOCIAZIONI



Identificano relazioni semplici tra attori e casi.

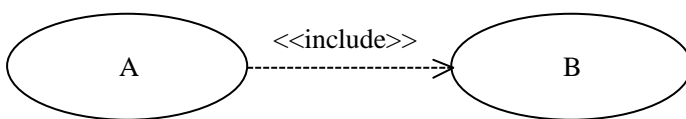
Associano ad ogni attore le funzionalità che gli vengono offerte dall'applicazione.

Nel nostro esempio:



L'amministratore non utilizza l'applicazione, non fa sharing, ma fa solo monitoraggio.

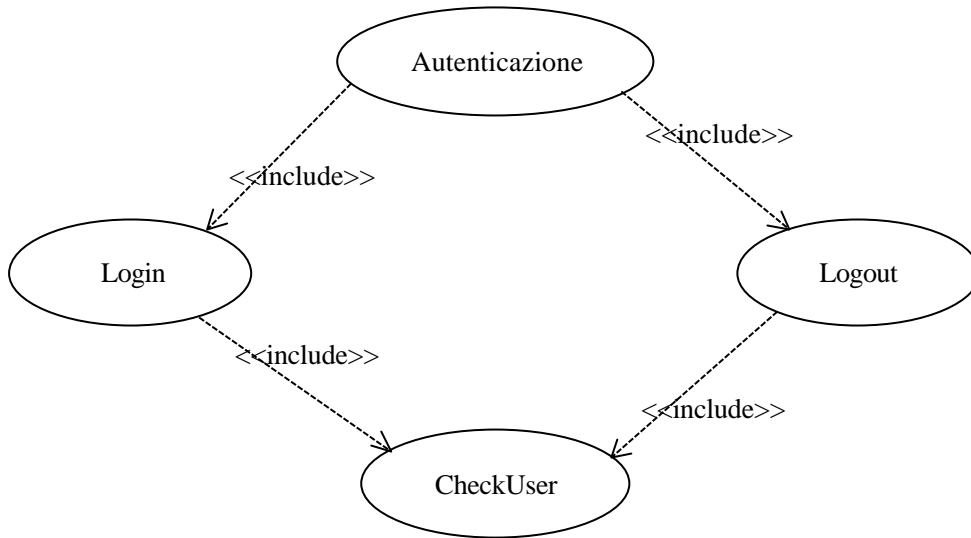
INCLUSIONE



<<include>> è uno "stereotipo", aggiunge una descrizione in più.

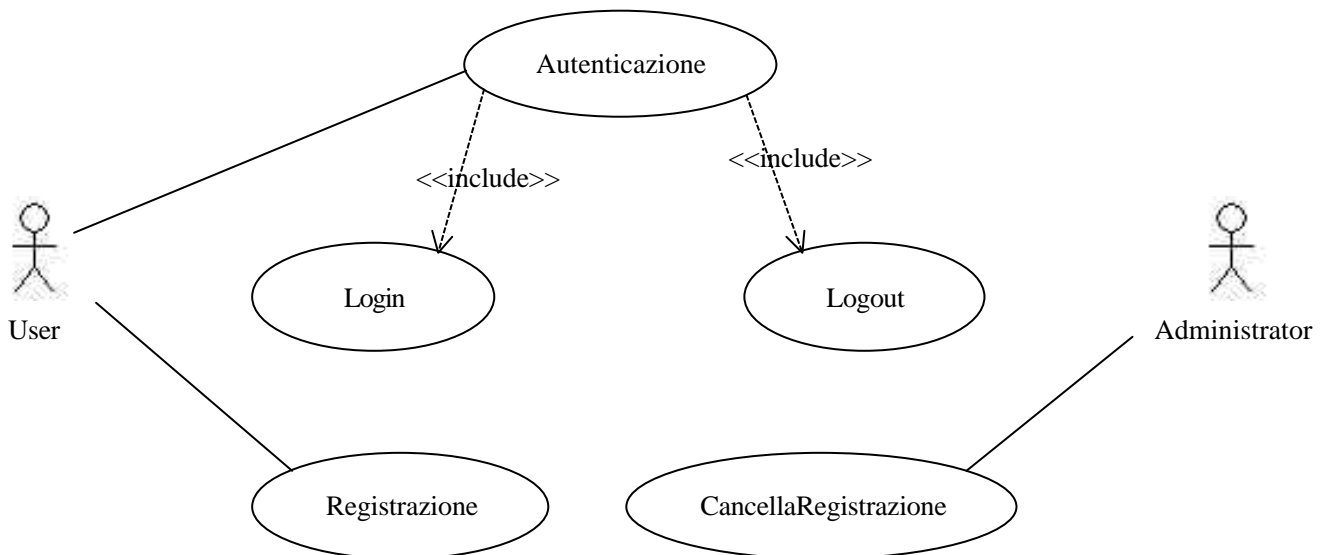
Vuol dire che una funzionalità ne include altre: quello che fa A, lo fa anche B

Nel nostro esempio:



CheckUser è inclusa nella fase di Login e Logout. Non è una funzionalità fornita all'utente ma quando l'utente si logga, ne viene controllata l'identità.

AUTENTICAZIONE



Include fattorizza attività ricorrenti e condivise

Analogie con

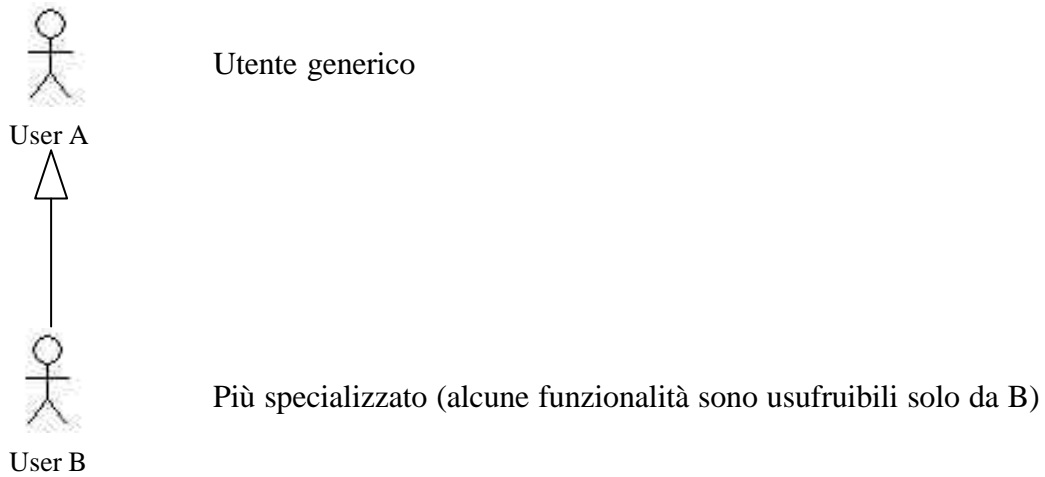
- Chiamata a sottoprocedura in un linguaggio di programmazione
- Scomposizione gerarchica

Altri esempi

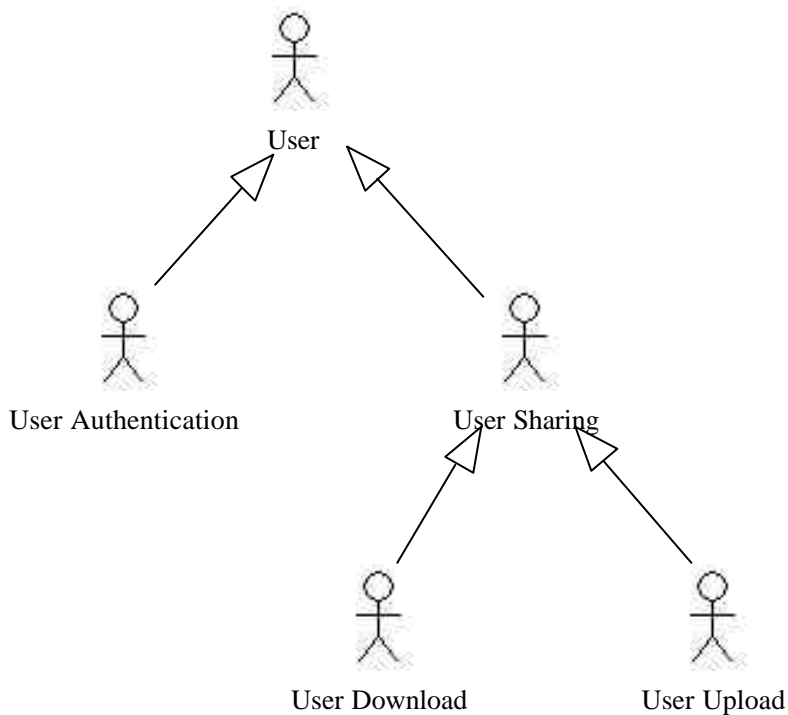
- La richiesta del numero di CC per ogni operazione bancaria
- Verifica PIN carta bancomat
- La consultazione del catalogo di una biblioteca
- Le funzioni matematiche più complesse (sin, cos, log, ecc.)

GENERALIZZAZIONE

Serve per specificare meglio alcune cose. (Diamo funzionalità diverse allo stesso attore)



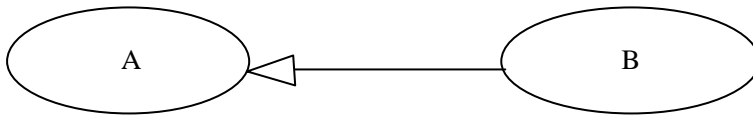
Nel nostro esempio:



Un utente può essere loggato oppure no e fa cose diverse nei due casi.

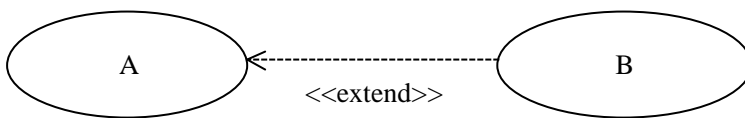
Un utente che fa sharing, può fare upload o download e nei due casi le funzionalità sono diverse.

La generalizzazione si può applicare anche agli use case.



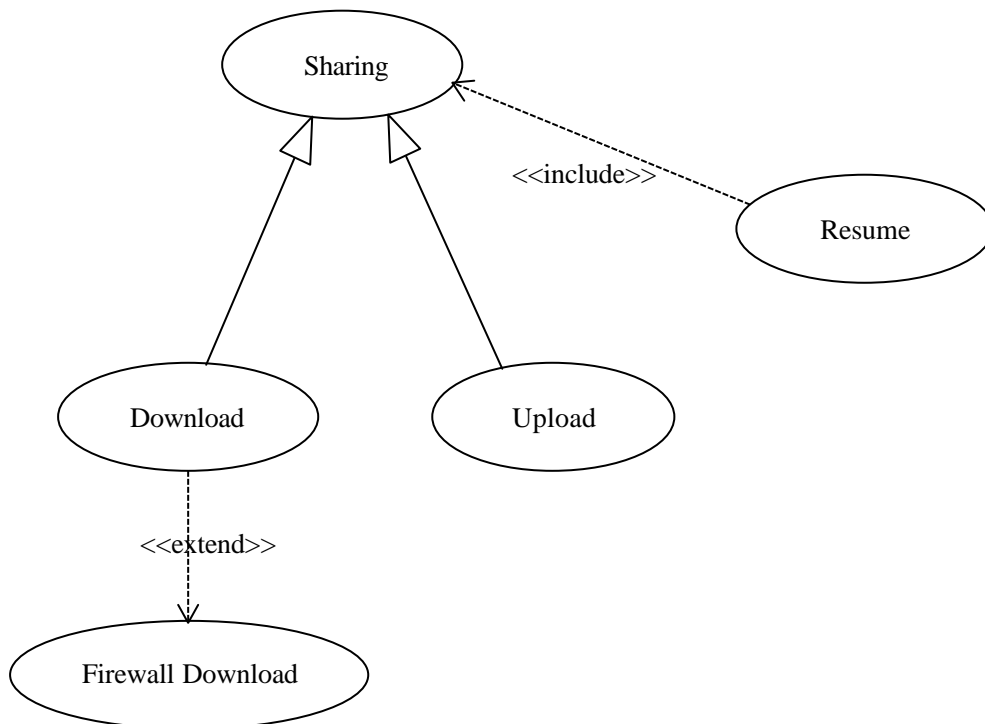
B offre le stesse funzionalità di A, ma ne aggiunge altre (arricchisce A)

EXTEND



Entrambi gli use case sono più o meno allo stesso livello, ma offrono funzionalità diverse (non si può dire che uno è più generico di un altro).

Nel nostro esempio:



Posso fare download anche attraverso un firewall (sono abbastanza simili).

Sharing include la funzionalità di resume (non scritta nelle specifiche ma è necessaria per la rete)

COSA NON SI PUÒ FARE

- Non ha senso collegare due actor con un association
- Se due utenti parlano fra loro, questo non fa parte dell'applicazione
- Non ha senso collegare due use case con un association
(La comunicazione all'interno dell'applicazione è un "dettaglio implementativo" e non va descritto nei requirement)

DOCUMENTAZIONE

Per ogni use case:

- Titolo: File Sharing
- Autori: Pippo
- Descrizione: Gli utenti possono autenticarsi e condividere i file
- Attori: User e Administrator
- Scenari:.....

SCENARI

Vogliamo definire come un'applicazione deve comportarsi.

- Uno scenario è una "esecuzione" particolare dello use case
- Rappresenta il comportamento (le azioni e gli eventi) del sistema nel caso particolare considerato
- Gli scenari definiscono requisiti di più basso livello rispetto agli use case
- Gli scenari sono solitamente definiti in *linguaggio naturale*
- UML propone una notazione particolare
- Ogni use case dovrebbe essere corredato da un insieme di scenari
 - **Scenari principali** (più possibile)
 - Tutto funziona correttamente
 - **Scenari secondari** (pochi e significativi)
 - Eccezioni (eventuali problemi o malfunzionamenti)
- Quanti scenari si devono definire?
 - Servono tanti scenari quanti sono quelli necessari per capire il corretto funzionamento del sistema e le eccezioni che si ritengono significative durante l'analisi

Gli scenari possono anche essere usati per:

- *Convalida* del sistema

Gli use case possono essere utilizzati per ricavare i dati di test con cui convalidare il sistema

Ogni use case rappresenta una funzionalità che andrebbe verificata

- *Gestione* del progetto

Gli use case propongono una "nuova" unità di misura

Gli use case potrebbero essere utili per: organizzare il progetto o stimare la complessità (sorta di function point)

INTERACTION DIAGRAM

Descrivono il comportamento dinamico di un gruppo di oggetti che “interagiscono” per risolvere un problema

Sono utilizzati per formalizzare gli scenari in termini

- Entità
- Messaggi scambiati


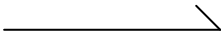
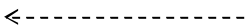

UML propone due diversi tipi di Interaction Diagram

- Sequence Diagram
- Collaboration Diagram

SEQUENCE DIAGRAM

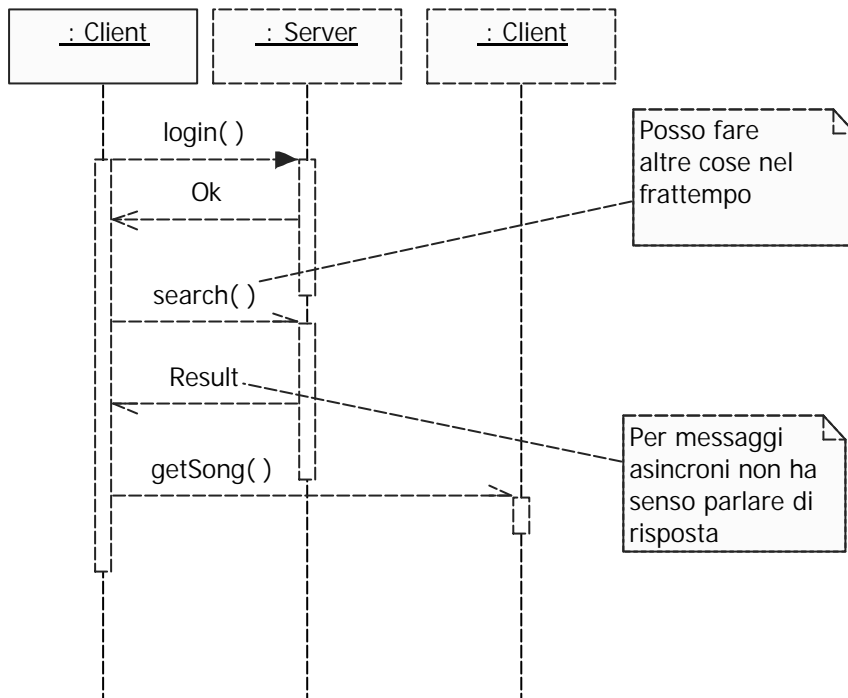
Evidenziano la sequenza temporale delle azioni

TIPI DI MESSAGGI

- **sincroni** (resto bloccato finché non ho una risposta)

- **asincroni** (posso fare altro mentre aspetto la risposta)

- **di risposta**

- **flussi di controllo**


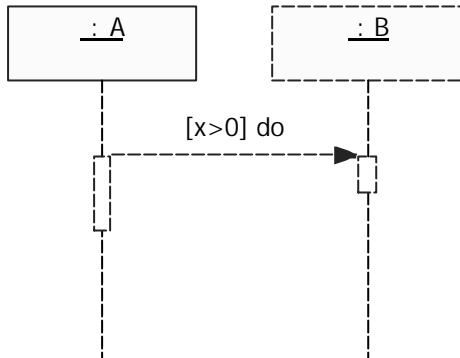
La risposta è generalmente legata ad un messaggio sincrono.

Esempio:

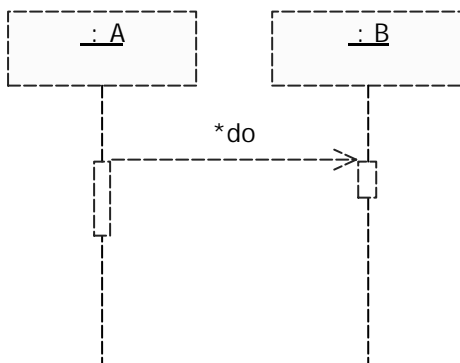


ETICHETTE

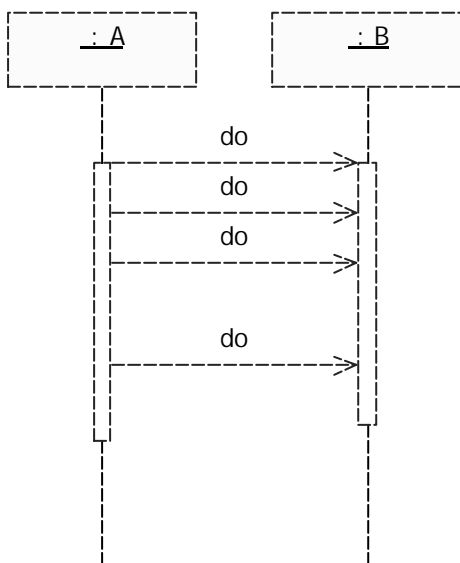
Possiamo avere delle **condizioni** (parentesi quadre)



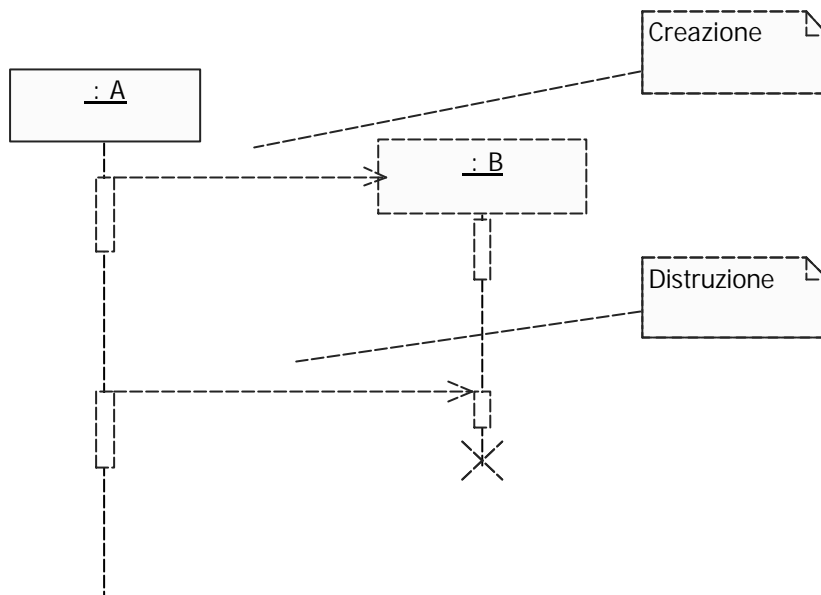
Oppure delle **iterazioni**



equivale a:



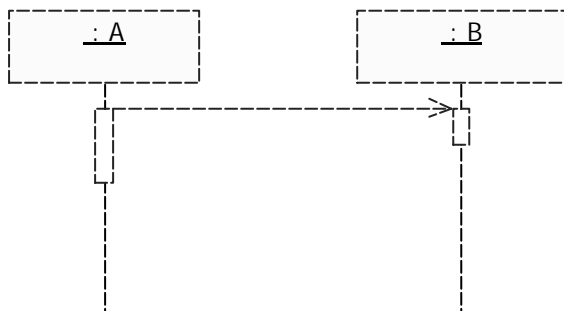
CREAZIONE E DISTRUZIONE



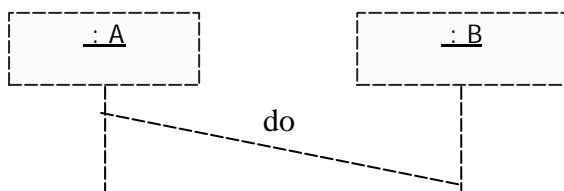
DESCRIZIONE DEL TEMPO

Il tempo scorre dall'alto verso il basso

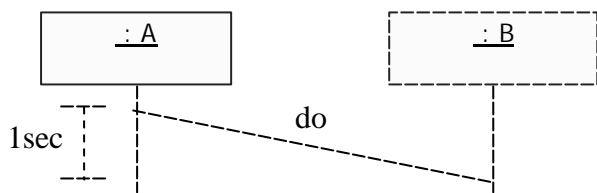
Per messaggi istantanei (tempo di trasmissione trascurabile)



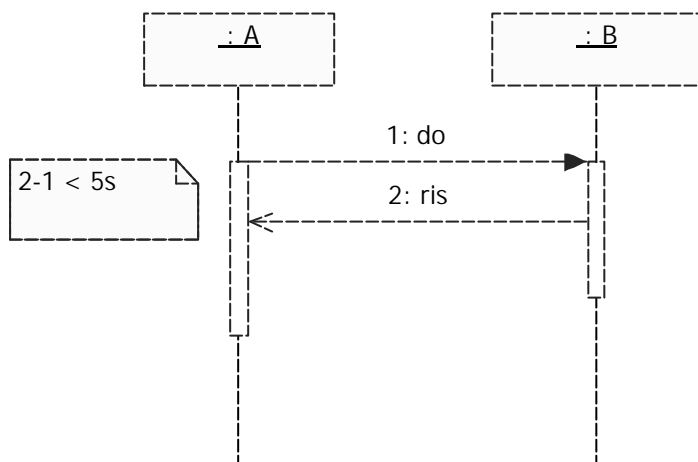
Se il tempo non è trascurabile:



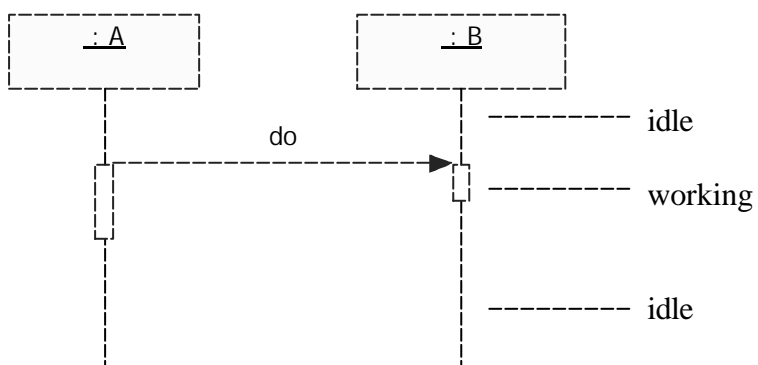
Se vogliamo descrivere quanto è il tempo:



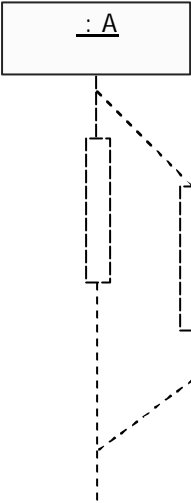
Oppure possiamo etichettare i messaggi



BARRE DI ESECUZIONE



MULTITHREADING



COLLABORATION DIAGRAM

Manca il concetto di tempo.

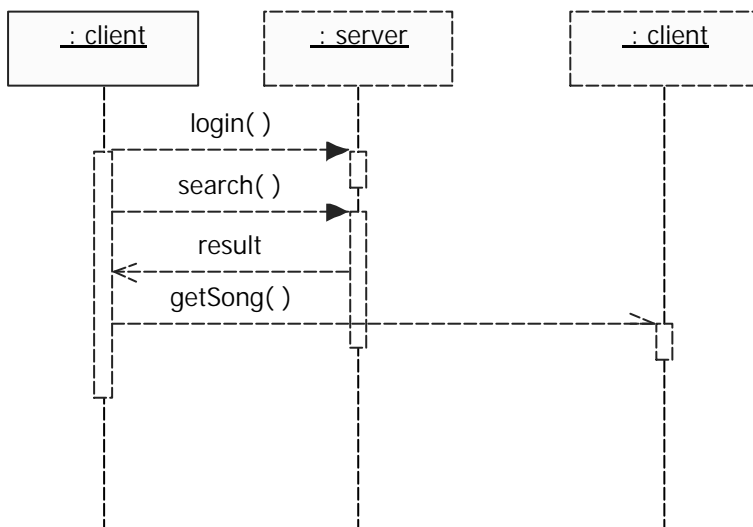
Permettono di descrivere la collaborazione tra entità del sistema.

Nei sequence diagram, la collaborazione è ricavata dai messaggi che le entità si scambiano, mentre nei collaboration diagram è espressa mediante dei **link**.

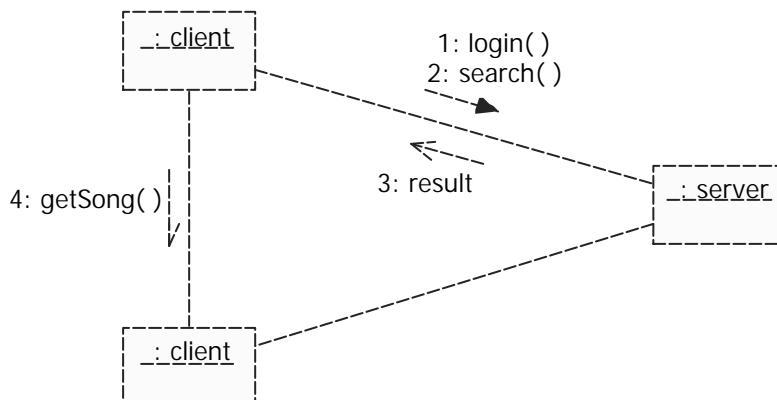
Abbiamo:

- Entità
- Messaggi
- Link

I messaggi possono essere scambiati solo se tra due entità ci sono dei link.



Equivalente a:



La numerazione può anche essere annidata:

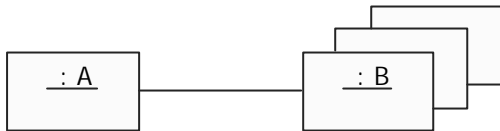
5.*

5.1.* etc...

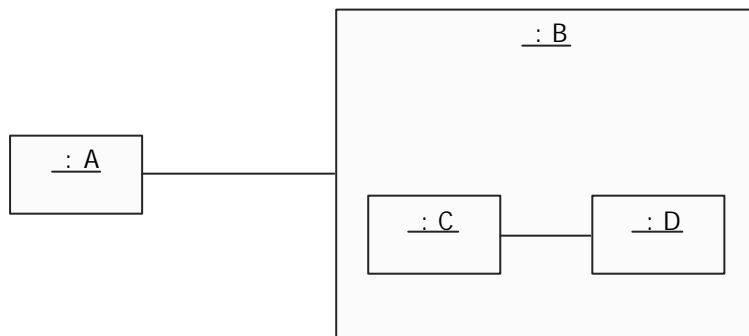
identificano la stessa macro operazione.

MULTIOGGETTO

Servono a descrivere che un'entità dialoga con un insieme di entità.



OGGETTI COMPOSTI



B è composto da C e D.

OGGETTI ATTIVI

Sono in grado di generare messaggi in maniera autonoma.



MODEL CONTROL VIEW

Dopo aver dato una descrizione generica.

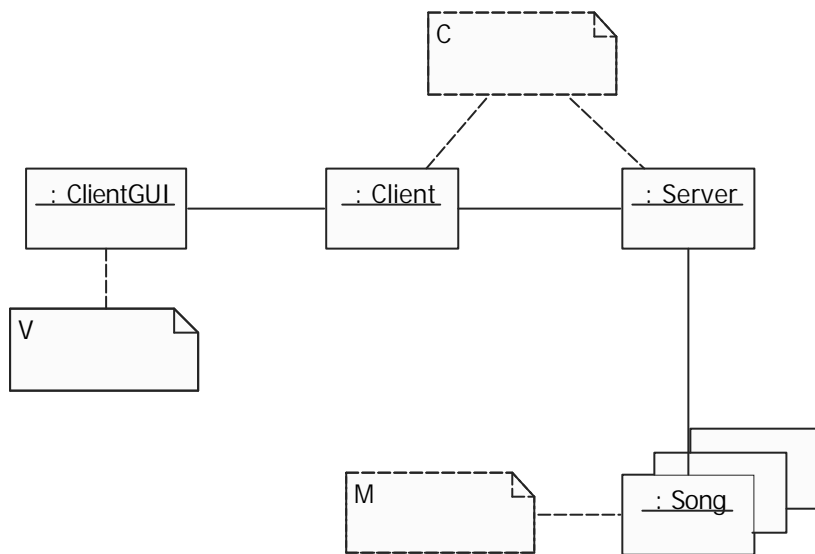
Model → Dati del sistema

Control → Come i dati vengono manipolati

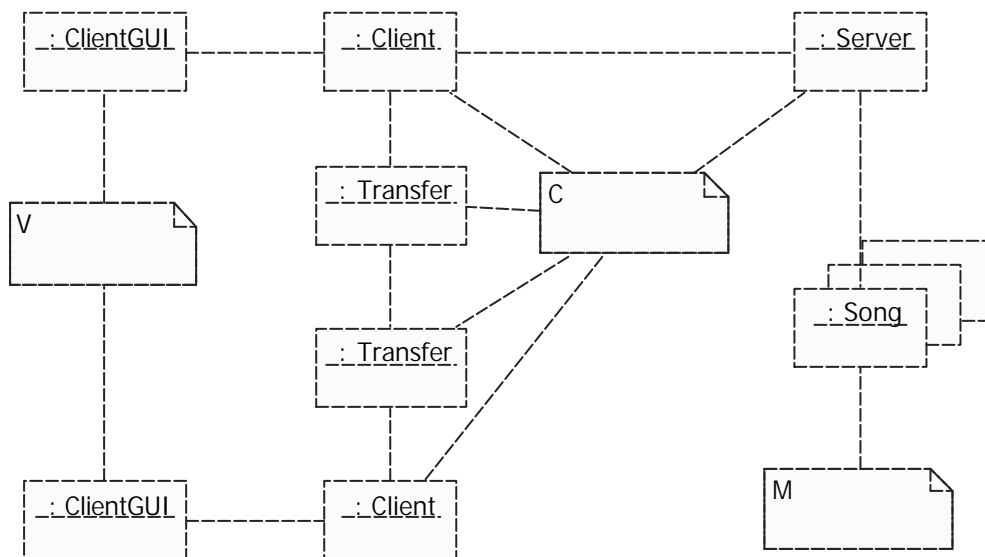
View → Come i dati vengono rappresentati

Esempio

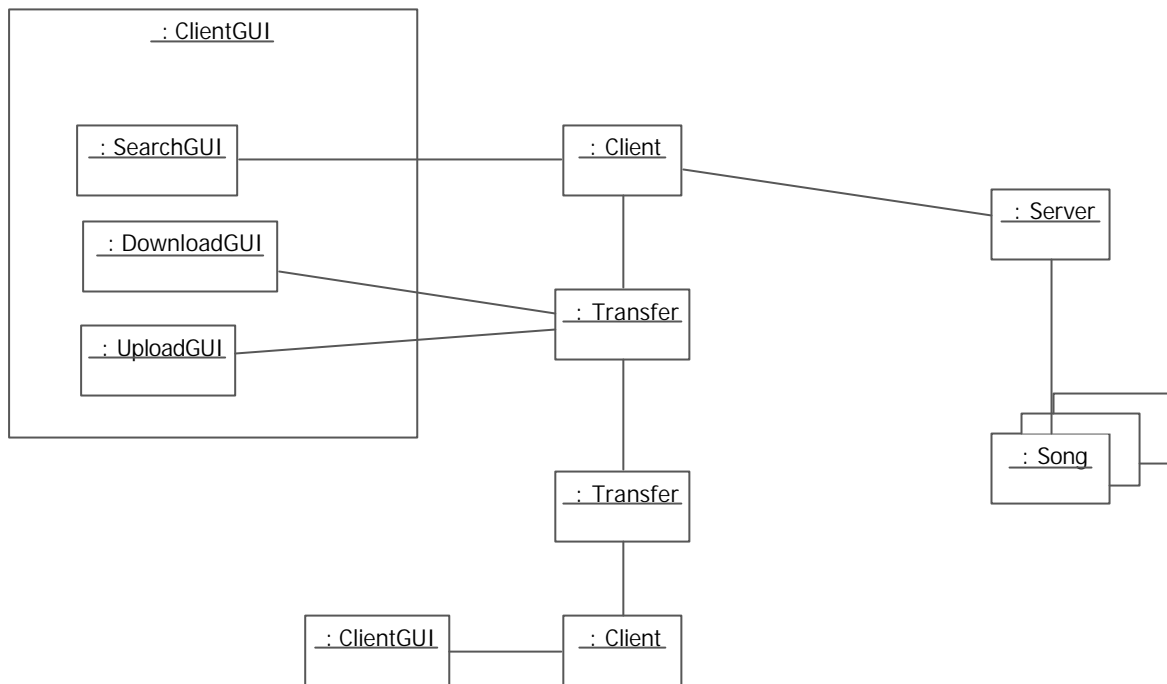
Se vogliamo definire la ricerca:



Per il trasferimento:

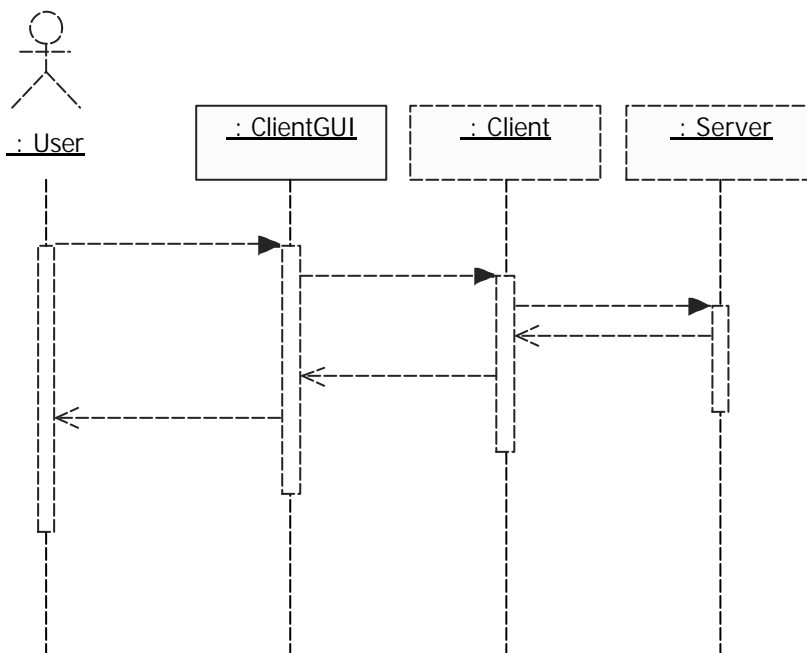


Potremmo definire in maniera più precisa la ClientGUI:



ACTOR

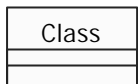
La stessa cosa possiamo farla nei sequence diagram in cui può essere inoltre utile introdurre l'utente (**actor**) che interagisce con l'applicazione anche se non ne fa parte.



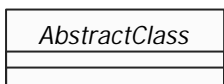
CLASS DIAGRAM

Danno una visione statica del sistema

CLASSE



Classe astratta: nome in corsivo

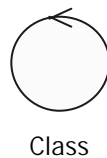
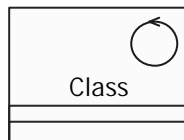
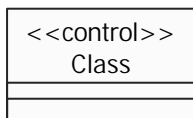


Stereotipi:

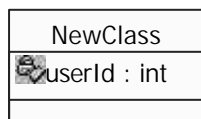
Permettono di definire particolari qualità della classe.

Possono essere:

- testuali <<nome_stereotipo>>
- icone



ATTRIBUTI

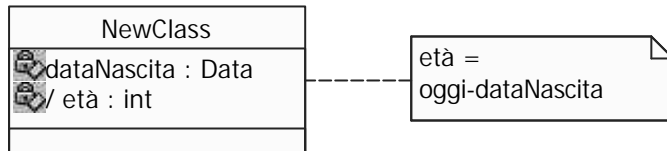


Possono essere:

- pubblici +
- privati -
- protetti #

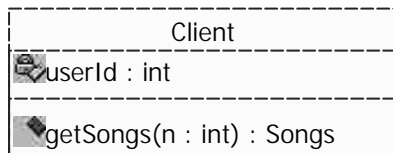
- astratti *corsivo*
- final {final}, static {static} (non UML standard)

Attributi derivati: un attributo è derivato se il suo valore deriva da altri attributi. Si indica con / e si mette un commento.



FUNZIONI

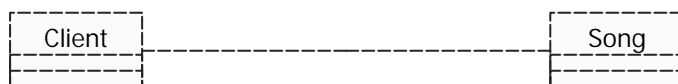
Valgono gli stessi discorsi visti per gli attributi.



RELAZIONI TRA CLASSI

ASSOCIAZIONE

Esprime una comunicazione tra due classi

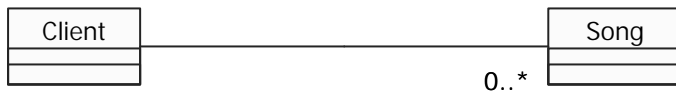


Cardinalità:

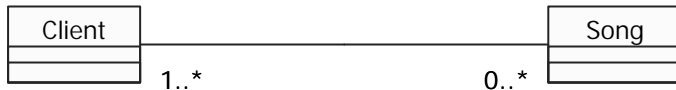
Un'associazione può essere arricchita con le cardinalità.

- 0..1
- 1
- 0..*
- 1..*
- n..m

Posso dire che il client può condividere da 0 ad un numero illimitato di canzoni

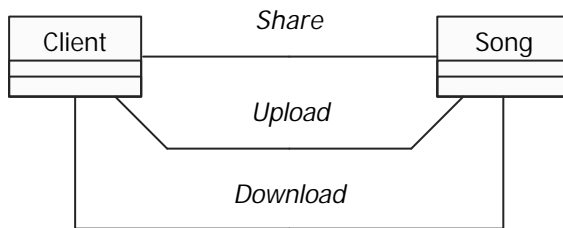


e anche che una canzone deve essere condivisa da almeno un client



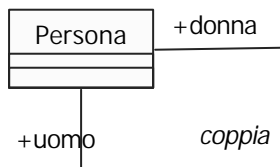
Nome:

E' possibile dare un nome ad una relazione indicando anche il verso di lettura (opzionale)



Ruoli:

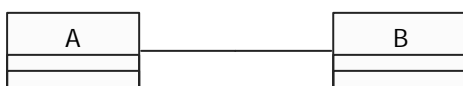
Con i ruoli posso specificare meglio il ruolo di una classe in un'associazione:



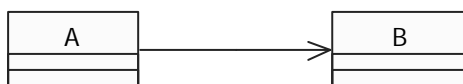
Verso di navigazione:

Possiamo avere:

- comunicazione in entrambi i sensi

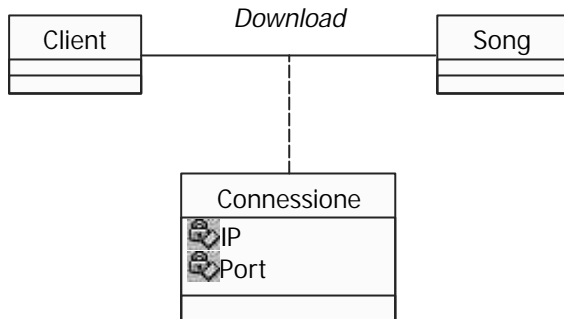


- comunicazione a senso unico



Associare una classe:

Quando vogliamo aggiungere dei parametri in più (non si usa mai perché la maggior parte dei linguaggi di programmazione non lo permette).

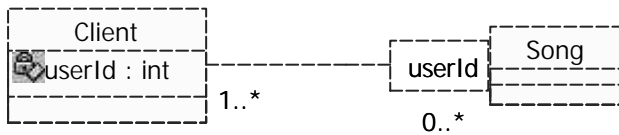


Qualificatori:

Sono utili per definire la cardinalità di un'associazione.



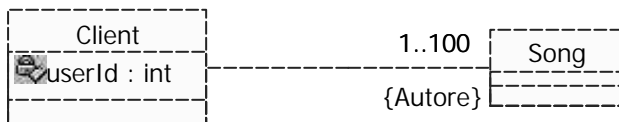
permette ad un client di condividere più volte la stessa canzone.



userId è il nostro qualificatore, considero un client con quella particolare userId

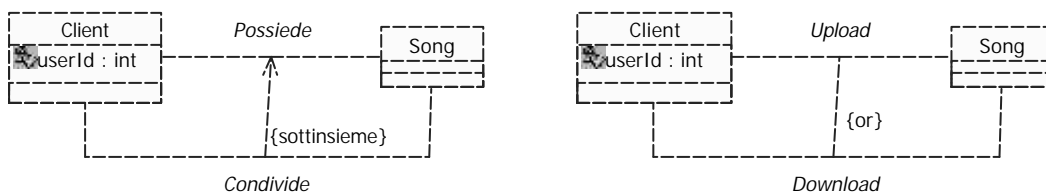
Vincoli:

Permettono di descrivere l'ordinamento con cui posso accedere alle classi

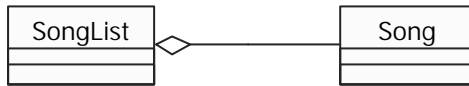


le classi sono ordinate secondo l'autore.

Se descrivo vincoli relativi a più associazioni:



AGGREGAZIONI

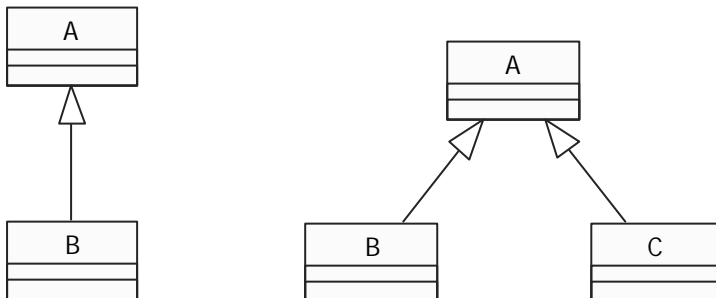


Una lista di canzoni è formata da canzoni

COMPOSIZIONI

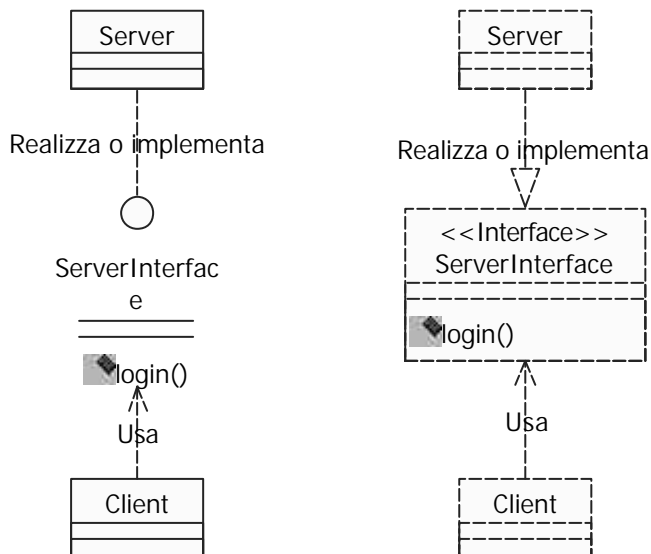
Il rombo è pieno, la differenza è che una canzone esiste anche se non esiste una lista, con la composizione no.

EREDITARIETA'



INTERFACCE

Sono particolari classi non istanziabili che hanno solo metodi. Possiamo avere due relazioni: **usa** e **implementa**.

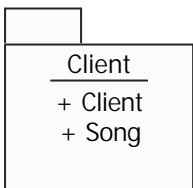


PACKAGE

Hanno lo stesso significato che in Java, servono a raggruppare delle classi con funzionalità simili.

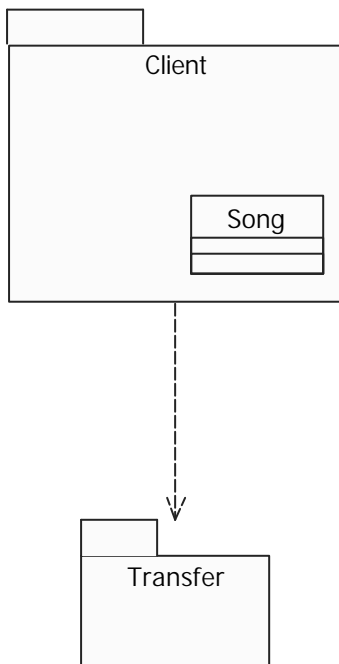


All'interno posso specificare le classi del package



Associazioni:

Sono **import** e **access**



Le classi del Client possono essere accedute da quelle di transfer.

Con **import** perdo il namespace, con **access** mantengo il namespace

Nota: namespace

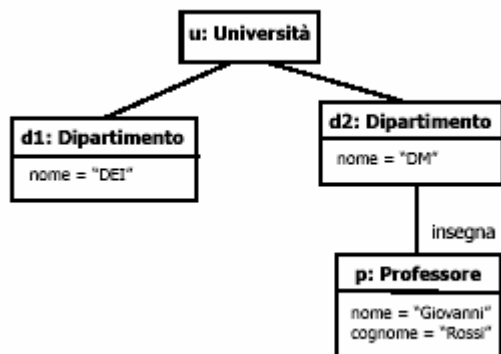
In Java con `import java.lang.*` importo tutte le classi che appartengono al package e posso fare riferimento a `String`, se non lo faccio devo scrivere `java.lang.String`. Con `access` devo specificare tutte le volte il namespace.

OBJECT DIAGRAM

Anche gli object diagram danno una visione statica del sistema.

Un object diagram, fissato un istante, dice quali sono gli oggetti creati dall'applicazione e quali sono i collegamenti.

Servono a valutare la complessità dell'applicazione o degli oggetti che voglio descrivere.



Abbiamo **solo** link tra oggetti. Sono usati solo in fase di test, valutando gli oggetti mentre l'applicazione è in esecuzione.

STATECHARTS DIAGRAM

Fanno vedere il comportamento di un attività o di un oggetto.

Sono in grado di dire come un oggetto reagisce quando riceve un segnale oppure in base ad un evento mediante **azioni** e **transizioni**.

ELEMENTI GRAFICI

Stato iniziale: ●

Stato finale: ●

Stato: 

Transizione: 

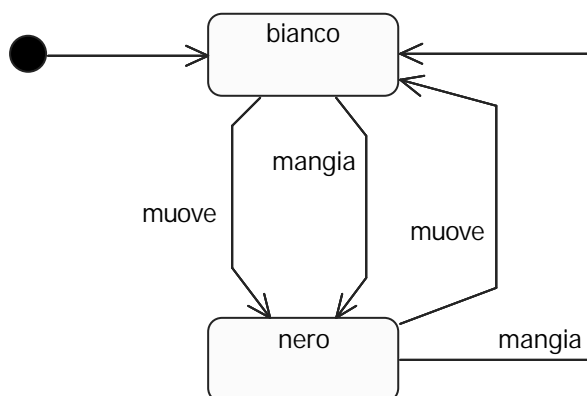
c = condizioni

a = azioni

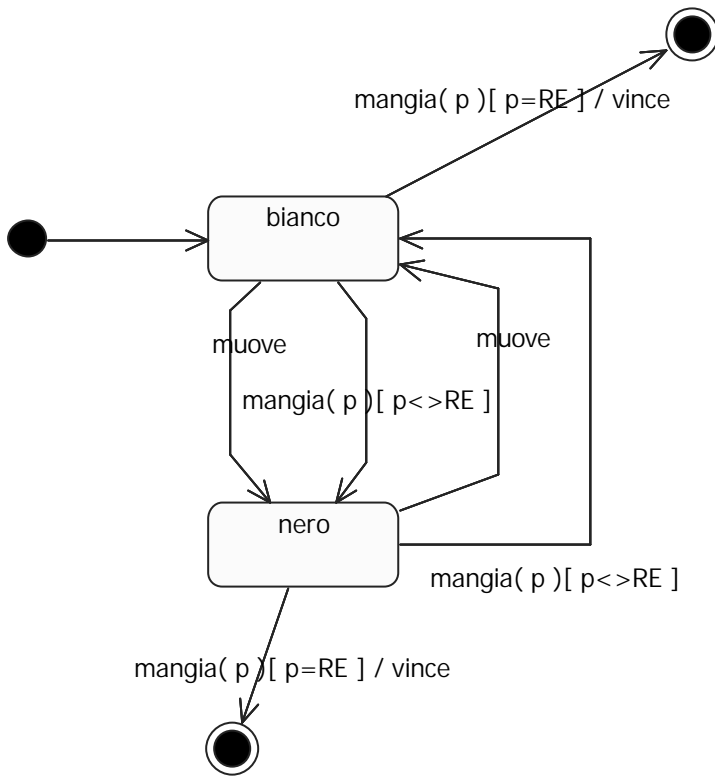
Se le condizioni sono vere, la transizione scatta e vengono eseguite le azioni

ESEMPIO: SCACCHIERA

Inizialmente deve muovere il bianco che può muovere oppure mangiare un pezzo, poi tocca al nero.



Il gioco si ferma quando uno dei due mangia il re all'altro. Mettiamo un parametro p a *mangia*.



OPERAZIONI

Abbiamo **azioni** e **attività**.

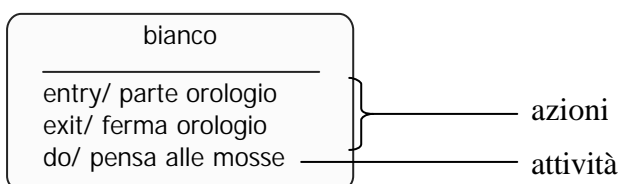
AZIONE:

- durata istantanea
- associata a transizioni – entrata/uscita da uno stato

ATTIVITA':

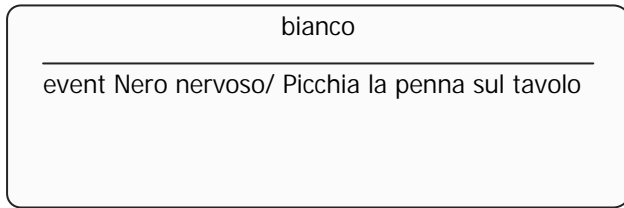
- durata prolungata
- associata agli stati

ESEMPIO:



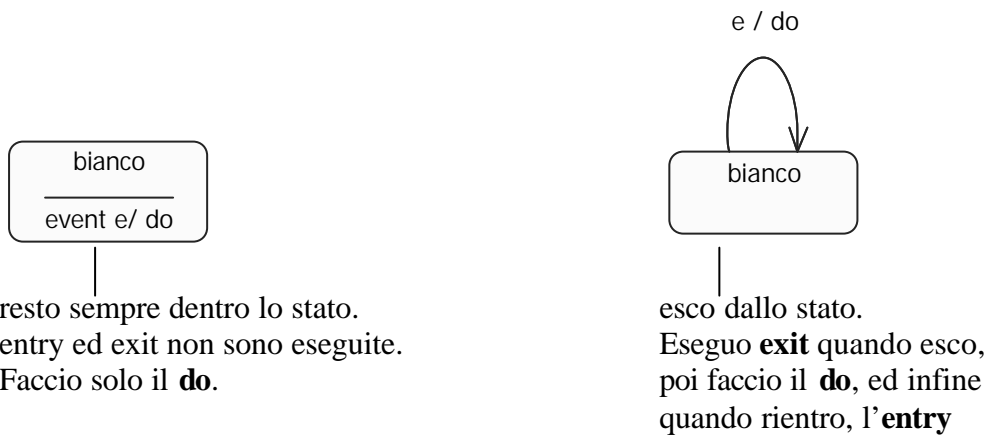
Possiamo associare un'attività ad un evento:

ESEMPIO:

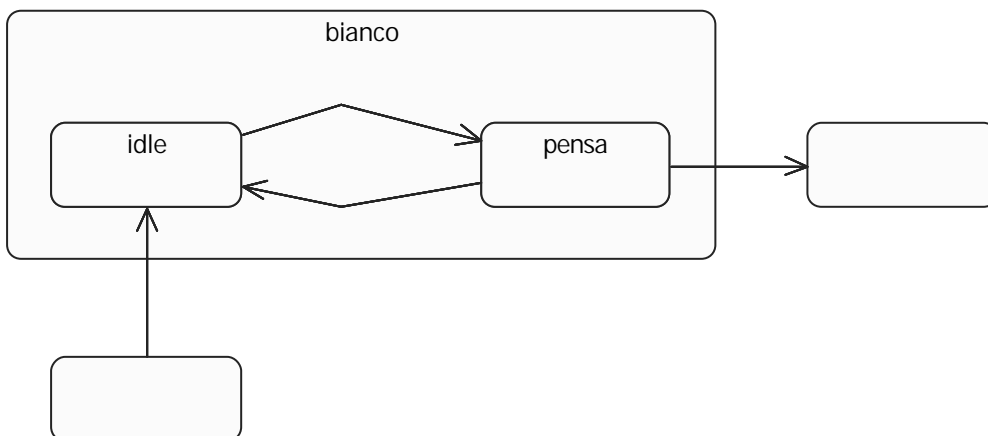


NOTA:

I simboli seguenti potrebbero sembrare equivalenti, ma c'è una differenza:



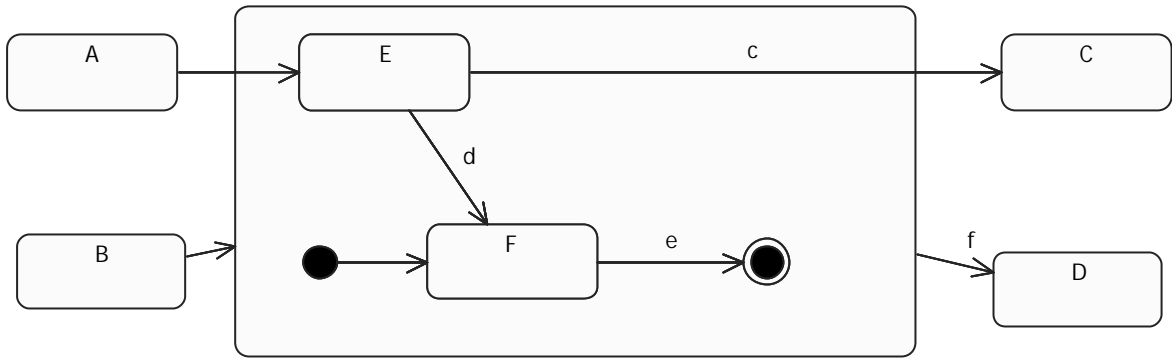
RAGGRUPPARE GLI STATI



Posso avere transizioni dentro gli stati, oppure che entrano/escono.

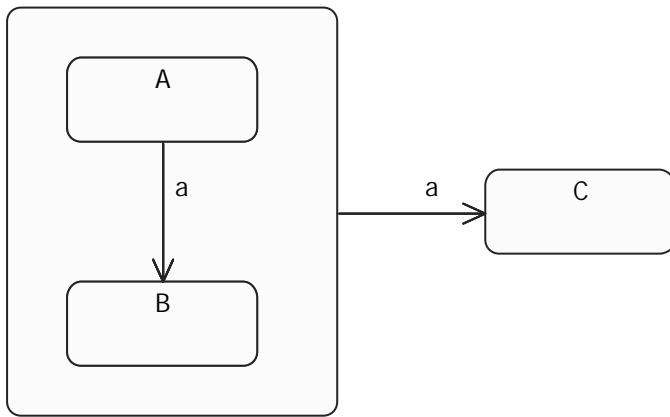
Nell'esempio:

Quando entro in idle: prima scatta entry di bianco e poi quello di idle,
Quando esco da pensa: prima scatta l'exit di pensa e poi quello di bianco.



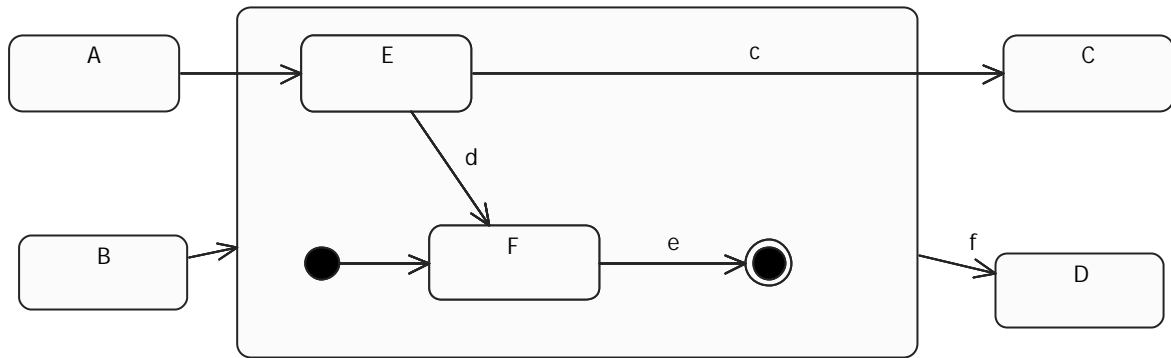
Se sono in B ed entro mi trovo in F

Se sono in F e scatta e resto nello stato composto (per uscire deve capitare f)



Se sono in A e scatta a la priorità più alta è quella dello stato composto e quindi vado in C

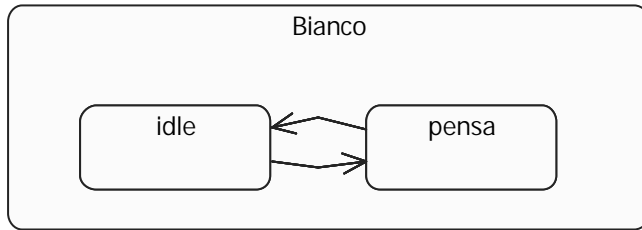
SOTTOSTATI NASCOSTI



Posso scriverlo come:



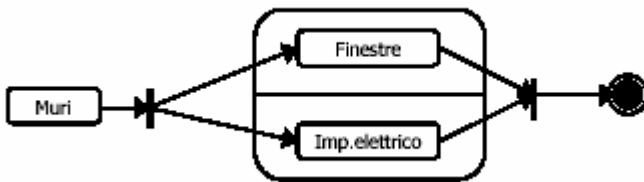
DECOMPOSIZIONE OR



Ho un solo stato attivo nello stato composto.

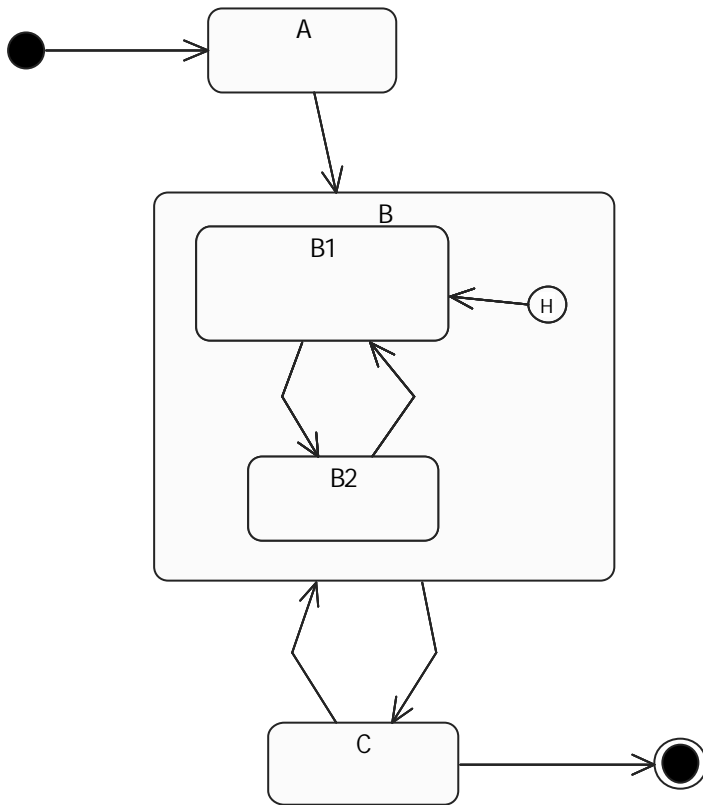
O è idle oppure pensa.

DECOMPOSIZIONE AND



HISTORY

Permette di salvare l'ultimo stato che ho visitato nel mio stato composto.

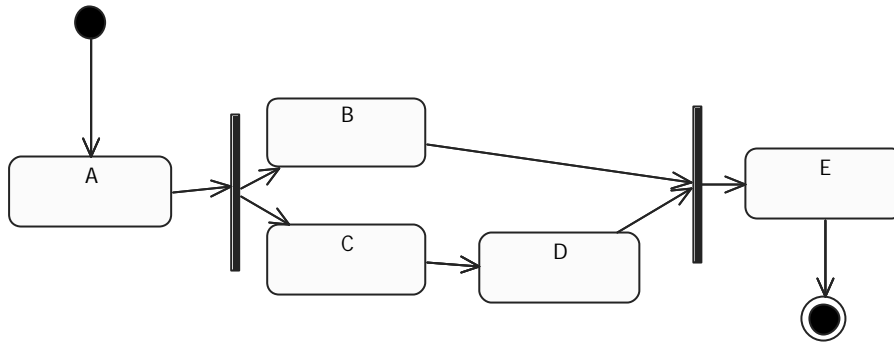


H indica:

- lo stato iniziale se è la prima volta che entriamo in B
- l'ultimo stato che ho visitato se ero già entrato in B

FORK/JOIN

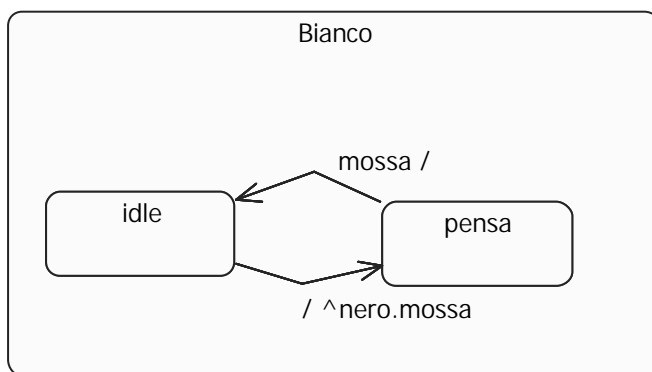
Possiamo descrivere il parallelismo non solo con la decomposizione AND ma anche con FORK/JOIN.



Con FORK genero due flussi paralleli, con JOIN devo aspettare che siano scattate le due transizioni da B e D per passare ad E.

EVENTI

Per il parallelismo abbiamo bisogno del concetto di **sincronizzazione** ottenibile attraverso gli **eventi**



- ^ simbolo dell'evento
- nero** oggetto a cui invio l'evento
- .
- mossa** evento generato

ACTIVITY DIAGRAM

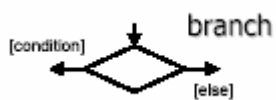
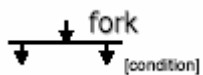
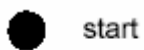
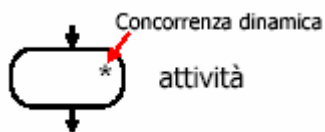
Descriviamo il comportamento attraverso un insieme di **attività**

Definiamo le **azioni** che compongono un' **attività** della nostra applicazione.

Gli activity diagram sono utili per descrivere:

- sequenze di azioni
- concorrenza tra azioni
- come la concorrenza è distribuita
- nel business plan, i passi per arrivare ad un goal

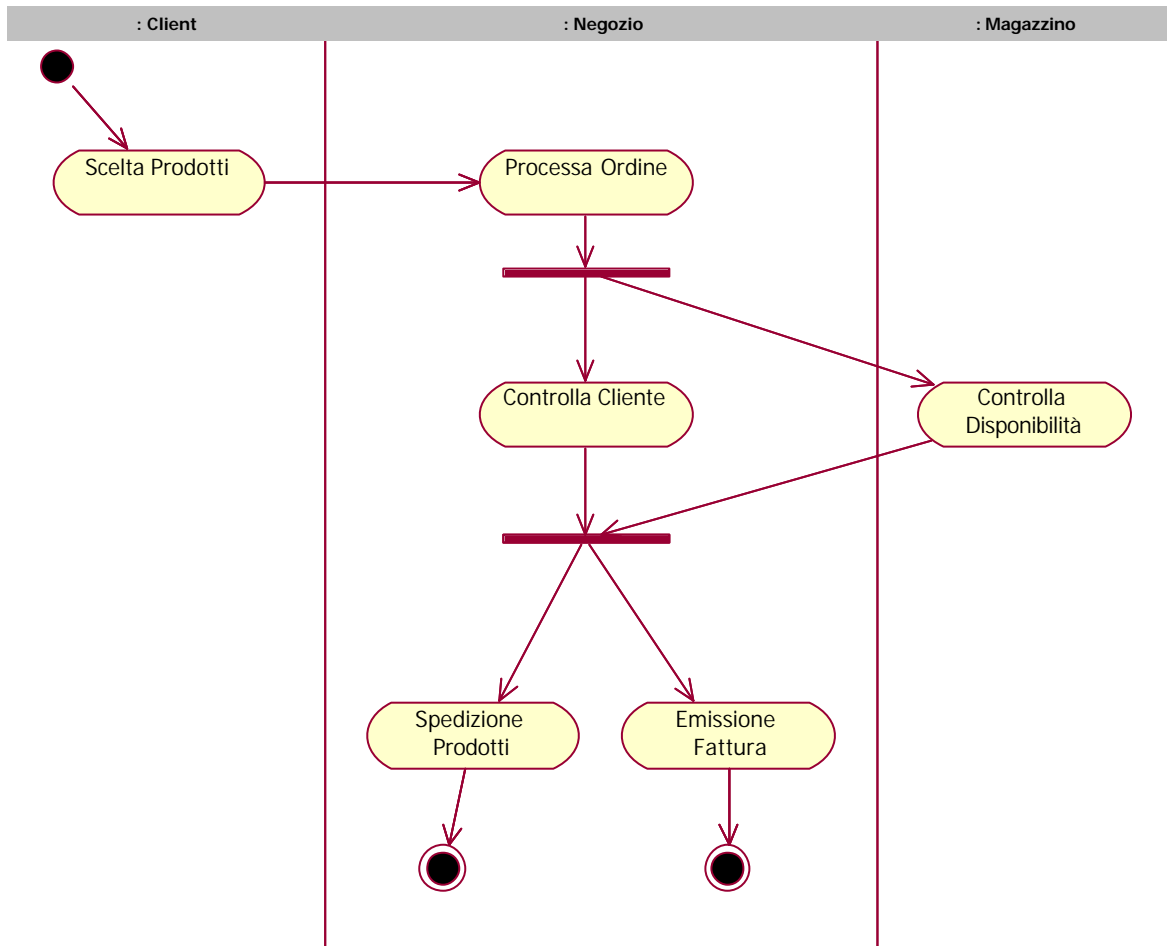
ELEMENTI GRAFICI



SWIMLANE

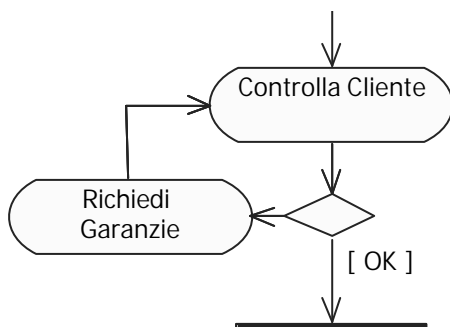
Per dividere le attività in base agli host.

ESEMPIO



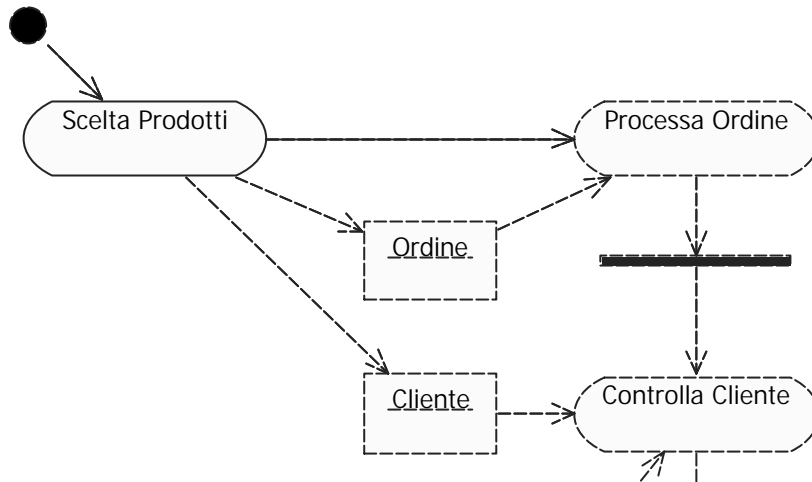
All'inizio il cliente sceglie il prodotto, il negozio processa l'ordine. Deve fare due cose: controllare la disponibilità e la storia passata del cliente per vedere se è affidabile. Poi viene emessa la fattura e vengono spediti i prodotti.

Ho descritto il sistema quando tutto va bene, ma potrebbero esserci errori, li gestisco con le if.



OGGETTI GENERATI

Possiamo dire quali sono gli oggetti creati durante le attività.



Scelta Prodotti genera due oggetti:

- Ordine: usato da Processa Ordine
- Cliente: usato da Controlla Cliente

IMPLEMENTATION DIAGRAM

Descrivono come le parti del sistema devono essere rappresentate e suddivise.

Ce ne sono due:

- **COMPONENT DIAGRAM**

Definiscono i componenti del sistema come delle black box che fanno parte del sistema ed offrono dei servizi.

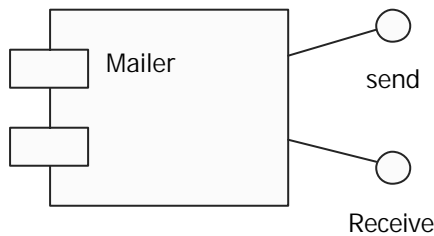
- **DEPLOYMENT DIAGRAM**

Definiscono come i componenti sono distribuiti (quali sono gli host dell'applicazione e cosa devo installare negli host)

COMPONENT DIAGRAM

Attraverso un componente posso organizzare il codice dell'applicazione in funzione di quello che deve fare.

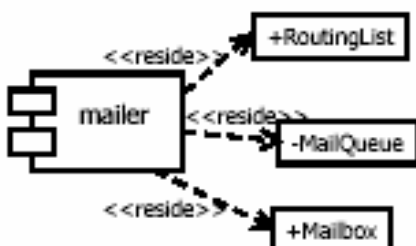
ESEMPIO: MAILER



Ho rappresentato anche le interfacce che il componente esporta.

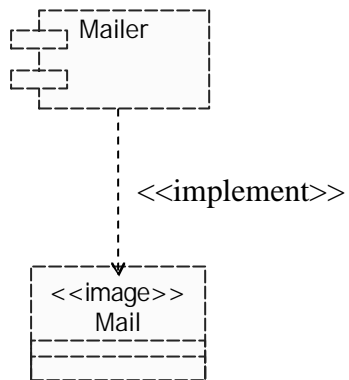
RESIDE

Vuol dire che la classe appartiene al componente



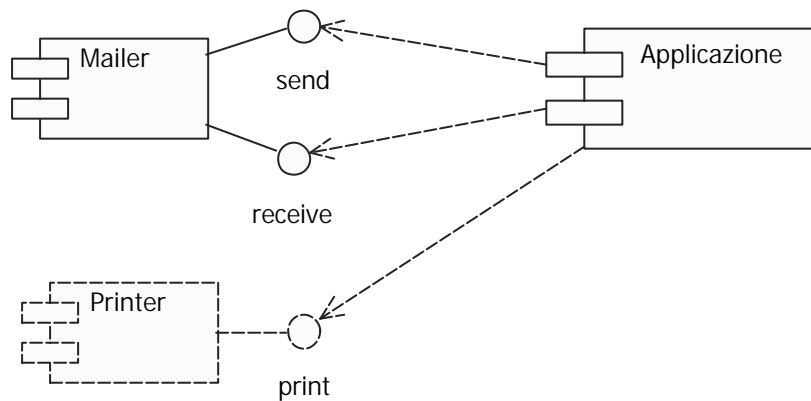
IMPLEMENT

Per includere ad esempio un'immagine. Dice che il componente utilizza il file.



Al di là delle classi, quando installo il programma devo inserire anche le immagini.

Possiamo anche descrivere che un componente utilizza l'interfaccia di un altro componente.



Possiamo anche disegnare le classi dentro il componente.

DEPLOYMENT DIAGRAM

I deployment diagram dicono:

- dove devo installare le cose (i nodi dell'applicazione)
- cosa c'è in ogni nodo

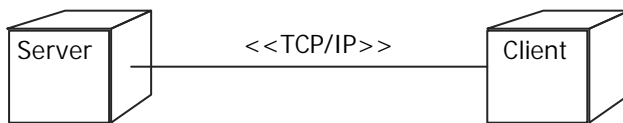
Un **nodo** può essere

- un host
- un processore di una macchina

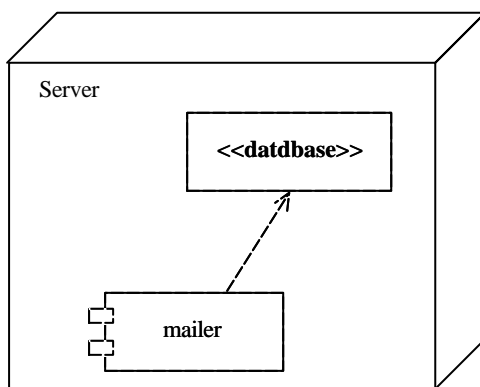
I nodi sono disegnati come cubi e la comunicazione tra due nodi è disegnata come una linea continua.



Posso specificare il protocollo di comunicazione

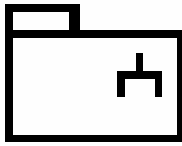


Supponiamo che nel server sia presente un database installato sullo stesso nodo:

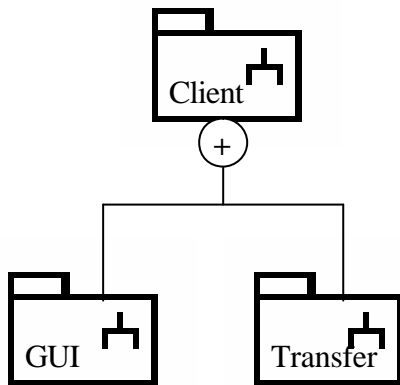


Per grandi sistemi è difficile descrivere il sistema completo. Si usano **subsystem** (descrivono parti del sistema) e **model** (descrivono il sistema completo considerando aspetti diversi come la sicurezza, la gestione dei dati etc...).

SUBSYSTEM



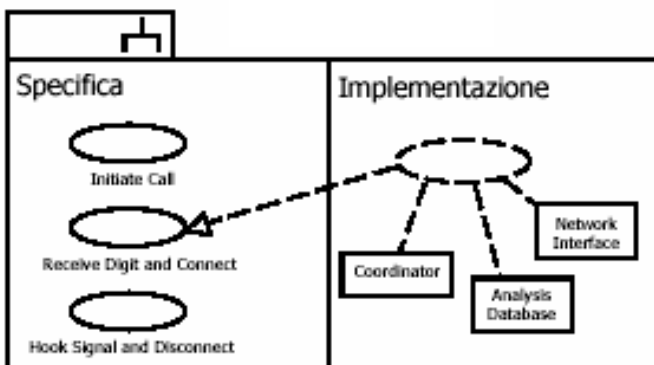
Un sottosistema può essere scomposto in tanti sottosistemi



Abbiamo una visione esterna del sistema (come i sottosistemi comunicano tra loro).

Oppure una visione interna (come un sottosistema è composto)

Un sottosistema può essere visto come un'applicazione vera e propria e per questo possiamo dire quali sono i requisiti e qual è l'implementazione.

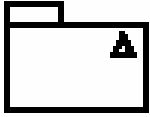


con -----> dico come la specifica è realizzata.

Per ogni sottosistema posso dire anche le interfacce che esporta.

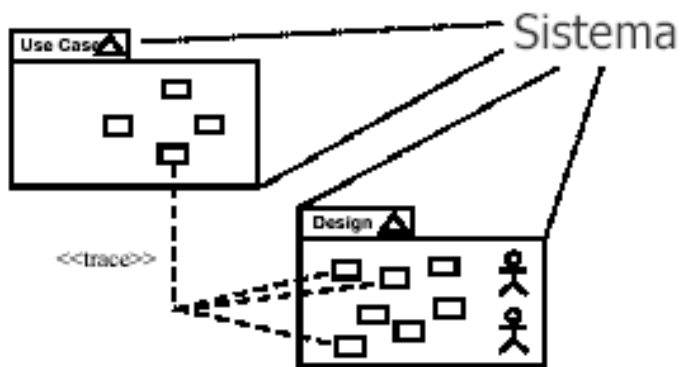
MODELLI

Servono a dare una visione globale del sistema.



E' una visione astratta, considera solo alcuni aspetti.

Possiamo ad esempio avere il modello della specifica, il modello del design...



Per dire che un requisito è soddisfatto da una classe si usa lo stereotipo <<trace>>

MODELLAZIONE

Dovrò sempre fornire:

- **Modello degli USE CASE**

Dove definisco i requisiti

- use case diagram

- **Modello di ANALISI**

Prima analisi del sistema (bozza dell'applicazione)

- sequence o collaboration diagram (alto livello)
- class diagram
- eventualmente statechart o activity diagram

- **Modello di DESIGN**

Progetto l'applicazione

- sequence o collaboration (dettagliati)
- class diagram
- statechart o activity diagram

- **Modello di DEPLOYMENT**

Più vicino all'implementazione

- component diagram
- deployment diagram

descivo il sistema in termini di componenti dicendo come dialogano e come sono distribuiti.

- **Modello di TEST**

Usato in fase di test

- object diagram
- use case diagram (per vedere se i requisiti sono stati soddisfatti)
- sequence o collaboration diagram

DESIGN PATTERN

Propongono soluzioni per problemi già noti

Servono a:

- risolvere i problemi più velocemente
- scrivere codice più leggibile
- riutilizzare il codice

Abbiamo tre categorie:

- creazionali (creazione di oggetti)
- strutturali (coordinazione di oggetti)
- comportamentali (flusso di dati tra gli oggetti)

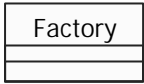
Un design pattern è diviso in tre parti:

- ambiente
- problema da risolvere
- soluzione

CREAZIONALI: FACTORY

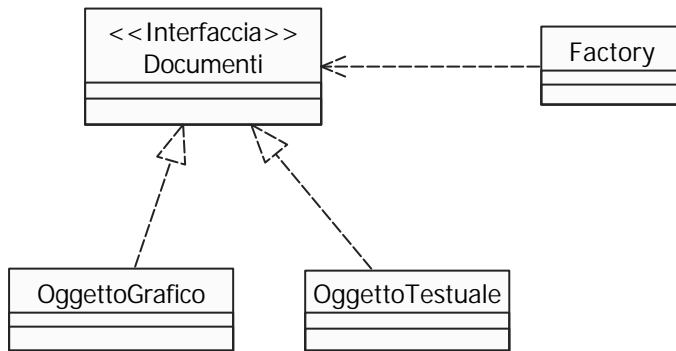
Usato quando vogliamo creare degli oggetti simili (dello stesso tipo che possono essere creati in maniera diversa o che ereditano da una stessa classe ma con comportamento diverso)

Abbiamo un oggetto Factory che creerà i nostri oggetti

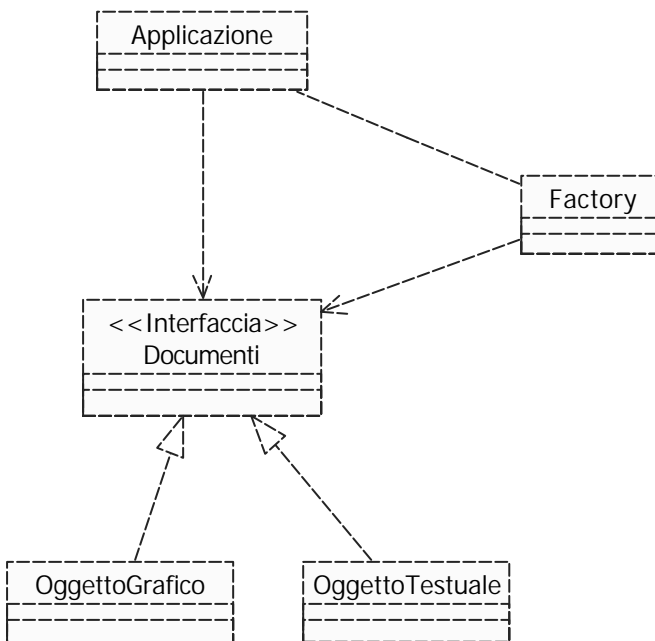


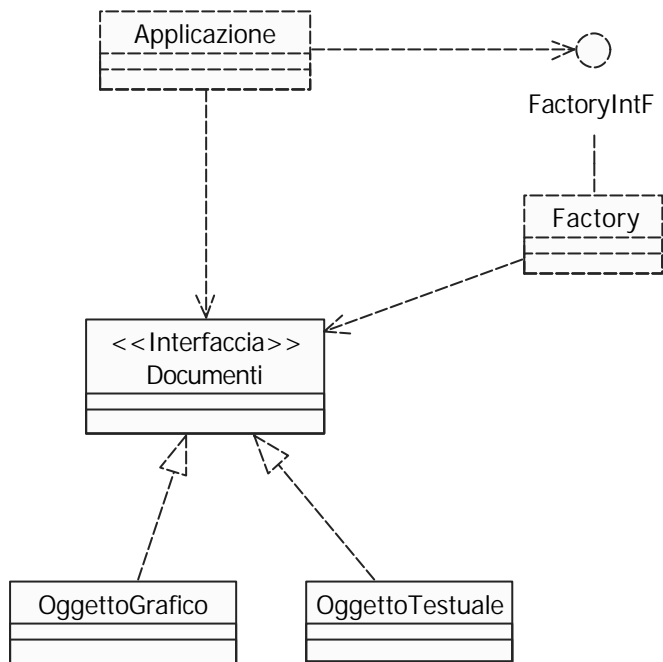
Che può essere usato in modo diverso

Factory creerà oggetti (ad esempio documenti che potranno essere creato in modi diversi)



L'applicazione userà solo l'interfaccia document





Riesco a disaccoppiare meglio (posso usarlo per la memorizzazione delle informazioni nel progetto)

CREAZIONALI: SINGLETON

Quando voglio avere solo un'istanza di un oggetto utilizzato da molte classi dell'applicazione.

Si usa una classe singleton con tutti i costruttori privati in modo che siano visibili solo all'interno della classe.

Quando una classe vuole fare riferimento all'oggetto devo dichiarare devo dichiarare un membro statico chiamato getInstance() che restituisce un riferimento all'oggetto

Esempio

Vogliamo realizzare un oggetto che distribuisce dei token



```
public class TokenManager {
    private static int TokenManager;

    private TokenManager(){
    }

    TokenManager static getInstance() {
        if (Token==null)
            Token = new TokenManager();
        return Token;
    }
}
```

Se esiste l'oggetto viene restituita la reference, altrimenti il token viene creato (solo una volta)

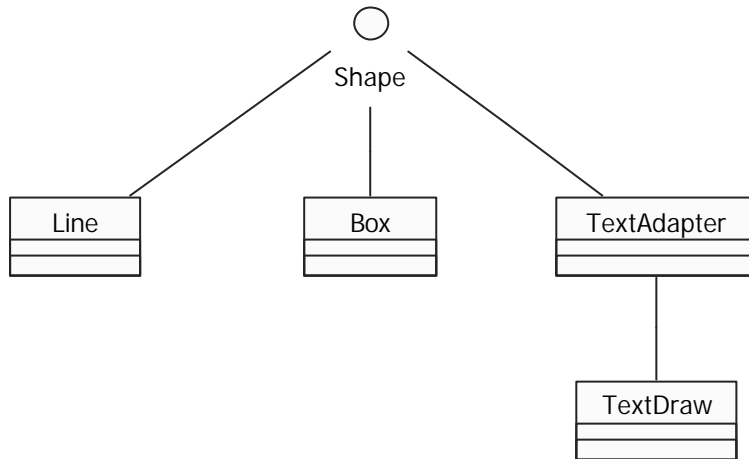
Ad esempio la classe run-time di Java funziona così

STRUTTURALI: ADAPTER

Abbiamo una classe che fa a caso nostro, ma con un'interfaccia diversa da quella che vorremmo.

Definiamo una nuova classe che adatta l'interfaccia della vecchia classe in modo da poterla utilizzare.

Esempio:

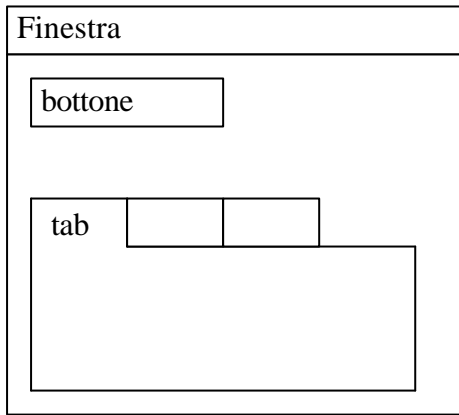


TextAdapter implementa shape ed è in comunicazione con TextDraw

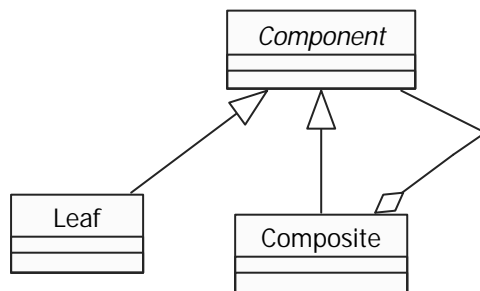
STRUTTURALI: COMPOSITE

Un oggetto può contenere altri oggetti simili a lui.

Chiamo un metodo su un oggetto, viene chiamato in cascata su tutti gli oggetti contenuti.

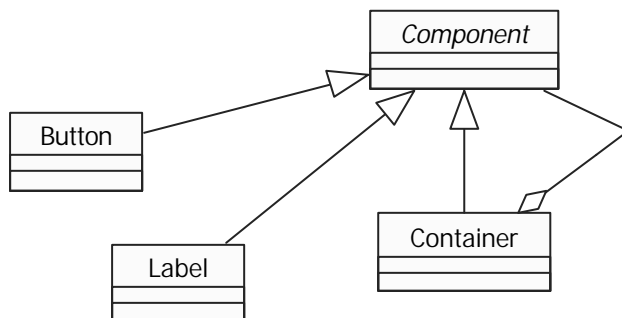


Soluzione: definire un componente astratto.



Esempio:

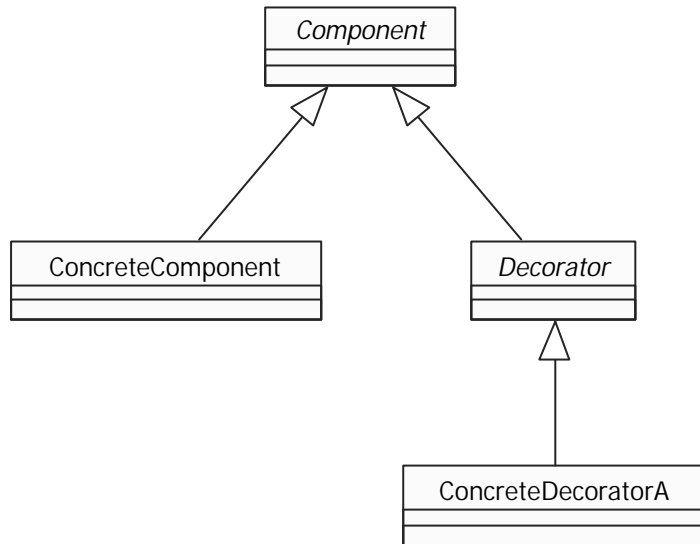
AWT funziona così.



Quando chiamo il metodo `paint()` su container verrà richiamato in cascata sugli altri

STRUTTURALI: DECORATOR

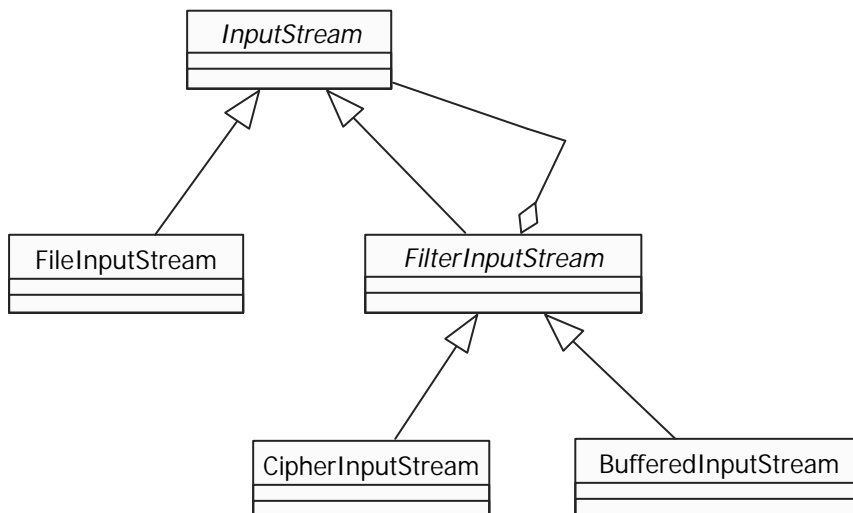
Vogliamo ridefinire le funzionalità di una classe in maniera dinamica



Esempio:

In Java la gestione dei dati in ingresso.

Esiste una classe *InputStream* che prende dati da rete o da file.



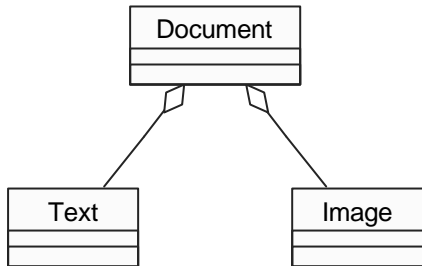
Posso fare:

```
t = new FileInputStream();
t1 = new CipherInputStream(t);
t2 = new BufferedInputStream(t1);
```

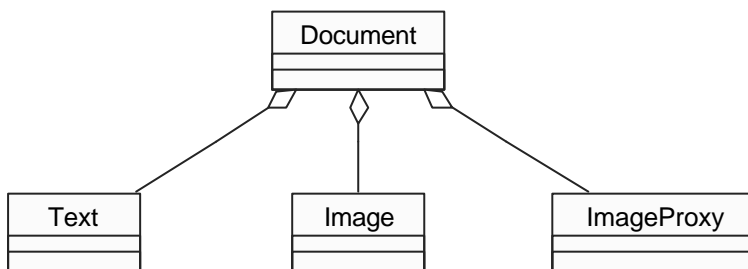
PROXY

Supponiamo di avere un documento con immagini e del testo. Il testo si carica rapidamente, mentre la immagini no.

All'inizio creiamo solo le parti testuali e quando serve anche le immagini.



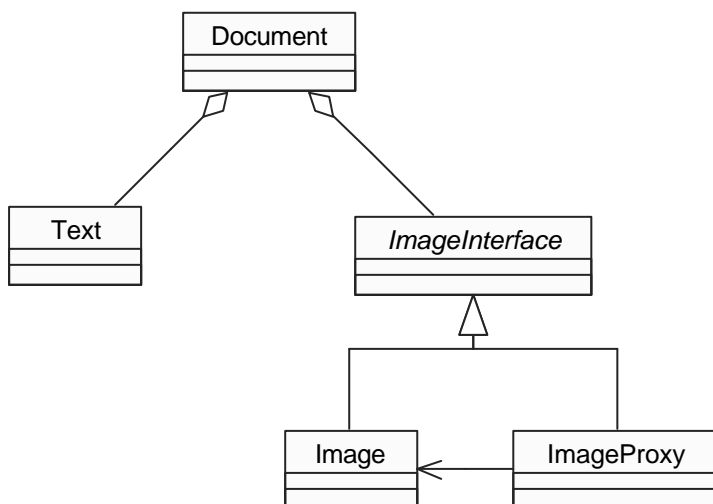
Così siamo obbligati a crearli immediatamente



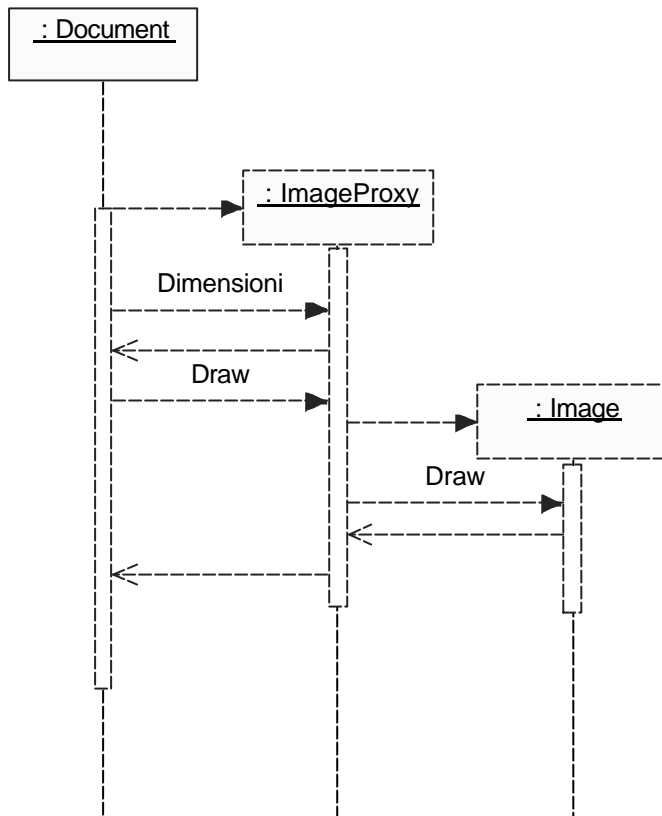
ImageProxy è una figura con meno dettagli (ad esempio solo le dimensioni) e rappresenta le immagini non caricate

Image rappresenta le immagini caricate.

Però **Document** deve differenziare l'accesso:



Vediamo come funziona:

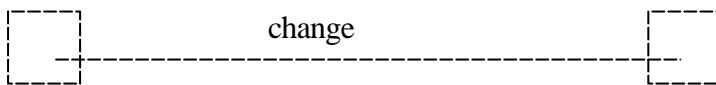
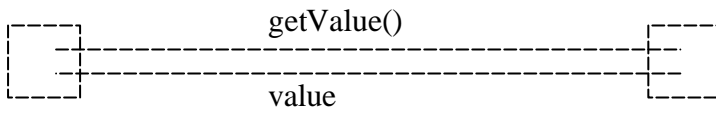


All'inizio abbiamo il documento con delle ImageProxy, quando servono, vengono create le immagini.

OBSERVER

Problema: Il sistema è composto da tanti sensori e tanti controllori.

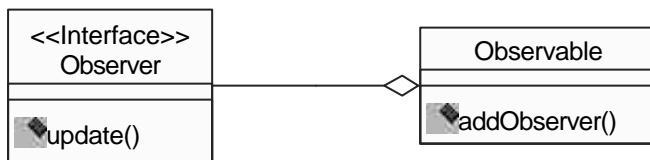
Posso avere:



Sensori

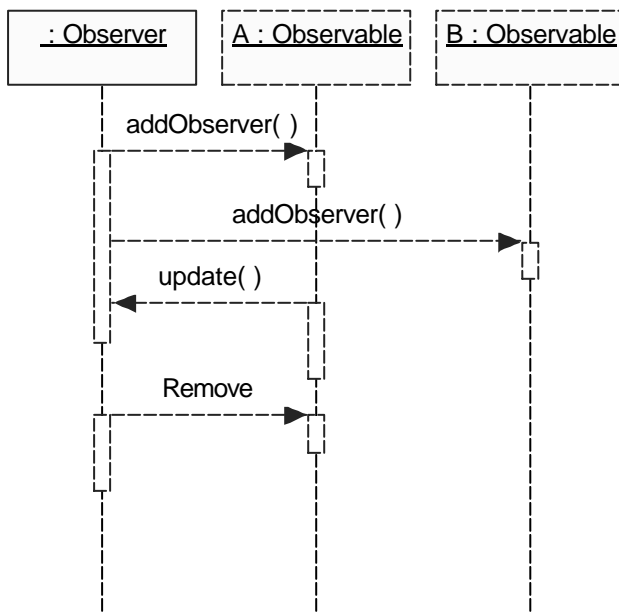
Controllori

Il controllore può richiedere un valore o può essere il sensore a fornirglielo. Vogliamo rendere la cosa dinamica:



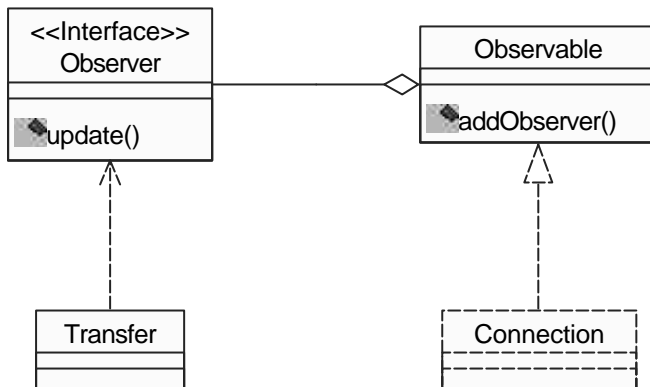
Sono già presenti in Java.

Posso aggiungere o togliere dinamicamente i sensori e fare l'update.



Esempio: file sharing

Voglio monitorare il download mediante una barra.



Connection notifica a Transfer ogni certo numero di K che riceve