

## NB per evidenziare segui le slide

### Lez 8 (26/10)

Si approfondiranno le fasi rilevanti nel ciclo di sviluppo software.

**Le fasi più importanti sono quelle *Definition-Oriented***, capire cosa fa il software, quindi **definizione dei requisiti** e **analisi di quei requisiti** (per produrre la specifica del software).

Nella seconda parte del corso si approfondisce la fase ***Production-Oriented*** (progettazione) e vedremo come trasformare il cosa in come.

### Requisiti Software

Le attività legate alla definizione e analisi dei requisiti sono così tanto importanti che fanno parte di un processo chiamato **Requirements Engineering**.

**Un requisito software rappresenta in generale una descrizione dei servizi che un sistema software deve fornire insieme ai vincoli da rispettare sia durante lo sviluppo che durante la manutenzione.**

Per lo standard IEEE il requisito software è una definizione articolata in 3 punti:

**A)** **è una condizione/capacità originata dalla necessità di un utente per risolvere un problema o arrivare a un obiettivo.**

**B)** **è una condizione/capacità che deve essere posseduta da un sistema per soddisfare un contratto, uno standard, una specifica o qualsiasi altro documento imposto.** (in questo caso non nasce dall'utente ma dalla necessità di soddisfare un contratto, normativa, legge etc...)

**C)** **una rappresentazione a livello di documento di una condizione/capacità in accordo con quanto visto nei punti A) e B)**

Perché definire un requisito è così complesso? **Perché un requisito software può esistere a diversi livelli di astrazione.**

Come già anticipato i requisiti vengono generati tramite un processo di ingegneria dei requisiti, ed essi possono cambiare livello d'astrazione man mano che il progetto evolve.

Se una compagnia/cliente avvia un contratto per un software di grandi dimensioni deve definire le sue **necessità in modo astratto** (ossia ad alto livello di astrazione) tale per cui non esista una soluzione preconfezionata (viene detto semplicemente ad alto livello ciò che si vuole). Infatti questi requisiti devono essere descritti in modo che diversi contractor possano partecipare al bando di gara per ottenere il contratto e realizzare il software. Ogni contractor farà la propria proposta dicendo come secondo lui dovrà essere realizzato il software per realizzare quelle necessità e la compagnia decide a quale assegnare il contratto. È allora che cambia il livello di astrazione: il contractor dovrà scrivere una **system definition** per il cliente molto più dettagliata in modo che questo capisca cosa il software dovrà fare.

**Entrambi i documenti sono definibili come documenti di requisiti.**

Ogni requisito in realtà deve essere valutato a due dimensioni: **livello di astrazione** e **tipologia**.

Riguardo il **livello di astrazione** ne esistono due tipi:

- **Requisiti Utente**: tipicamente scritti *in linguaggio naturale con eventuale aggiunta di diagrammi per descrivere i servizi che il sistema dovrà fornire e i vincoli operativi. Devono essere comprensibili a tutti in quanto sono scritti per e con il cliente.*
- **Requisiti di Sistema**: sono *specificati mediante la stesura di un documento ben strutturato che descrive in modo dettagliato i servizi che il sistema software deve fornire (è possibile utilizzare linguaggi specifici, noi useremo un linguaggio di modellazione).*

**Il documento contenente sia i requisiti utente che quelli di sistema è quello che costituisce il contratto tra cliente e fornitore.**

**NB** esiste anche in alcuni casi un altro tipo di requisito, il **Software Specification**, *ancora più dettagliato dei requisiti di sistema e spesso utilizzato per software critici (linguaggi formali per prevenire errori).*

*Se si dice Requirements Definition, allora si fa riferimento ai Requisiti Utente. Per i Requisiti di Sistema invece si usa il termine Specification.*

Ora un breve richiamo di definizioni di termini:

- **Cliente** == persona o organizzazione che paga per la fornitura di un prodotto software
- **Fornitore** == persona o organizzazione che produce software per il cliente
- **Utente Finale** == persona che interagisce direttamente con il prodotto software, non corrisponde necessariamente con il cliente

## Esempi di requisiti

- **Requisito utente**

1. Il sistema software deve fornire un mezzo per rappresentare e visualizzare file esterni generati da altri tool

- **Requisito di sistema**

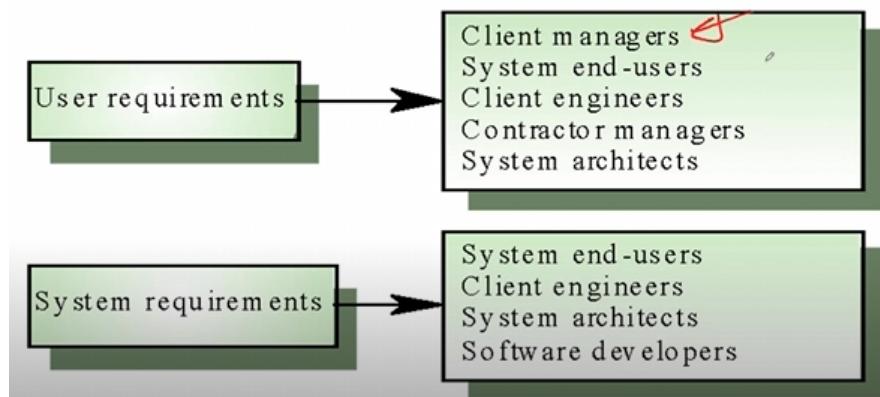
- 1.1 L'utente deve avere la possibilità di definire il tipo dei file esterni
- 1.2 Ad ogni tipo di file esterno deve essere associato il tool che lo ha generato
- 1.3 Ogni tipo di file esterno deve essere rappresentato mediante una specifica icona sullo schermo
- 1.4 L'utente deve avere la possibilità di definire l'icona che rappresenta il tipo di file esterno
- 1.5 Quando l'utente seleziona un'icona che rappresenta un file esterno, deve poter essere eseguito il tool in grado di visualizzare il file

In questo esempio è chiaro il differente livello di astrazione

## Lez 9 (30/10)

Chi sono però gli effettivi lettori dei requisiti?

### Chi legge i requisiti?



Per i requisiti utente si hanno i **manager del cliente**, i **system end-users** (coloro che useranno il software), gli **eventuali ingegneri del cliente**, i **manager del contractor** (quindi di colui che poi sviluppa il software) e i **system architects**.

*Scompaiono* nella parte di *System Requirements* il **client manager** e il **contracor manager**, *compaiono* invece i **software developers** (con developer si intende anche i progettisti oltre che gli sviluppatori di codice).

Come abbiamo accennato l'ultima lezione però la classificazione dei requisiti avviene lungo due dimensioni: una basata sul **livello d'astrazione** (appena vista) e l'altro basato sul concetto di **categoria**.

Esistono due principali categorie di requisiti: **Funzionali** e **non Funzionali**.

I **Requisiti Funzionali** I **requisiti funzionali** descrivono **ciò che il sistema deve fare**, ovvero le **funzionalità** e i **comportamenti** che deve offrire in risposta a determinati input o in determinate condizioni..

**Es.1** Il sistema software deve fornire un appropriato visualizzatore per i documenti memorizzati

**Es.2** L'utente deve essere in grado di effettuare ricerche sia sull'intero insieme di basi di dati che su un loro sottoinsieme

**Es.3** Ad ogni nuovo ordine deve essere associato un identificatore unico (Order\_ID)

Si noti come questi tre esempi siano requisiti utente in quanto molto ad alto livello, descrivono precisamente cosa vuole l'utente in termini di sue necessità e basta una frase per descriverli.

I **Requisiti Non Funzionali** (anche detti extrafunzionali) invece, banalmente, sono definibili come tutti quei requisiti che non sono funzionali.

*Questi quindi coprono una gamma più ampia di requisiti, infatti sono tutti quei requisiti che descrivono le proprietà del sistema software in relazione a determinati servizi o funzioni e possono anche essere relativi al processo di sviluppo.*

- **Tra i più comuni** requisiti non funzionali vi sono quelli che fanno riferimento a **caratteristiche legate alla qualità del software**, quindi **caratteristiche di efficienza, affidabilità, safety** etc...

- Vi sono poi le **caratteristiche del processo di sviluppo**, principalmente legate alla possibilità che il cliente abbia fornito vincoli come standard del processo, uso di ambienti CASE, linguaggi di programmazione, metodi di sviluppo (ad es. i vincoli possono essere imposti se poi il cliente dovrà fare manutenzione e gestisce un reparto software che usa uno specifico modello/linguaggio).

- Infine si hanno le **caratteristiche esterne** (laddove ad esempio esista interoperabilità con altre organizzazioni, vincoli legislativi es. software di contabilità che devono conformarsi alle normative in essere)

**Es.1** Il tempo di risposta del sistema all'inserimento della password utente deve essere inferiore a 10 sec

**Es.2** I documenti di progetto (*deliverable*) devono essere conformi allo standard XYZ-ABC-12345

**Es.3** Il sistema software non deve rilasciare ai suoi operatori nessuna informazione personale relativa ai clienti, tranne nominativo e identificatore

Ancora una volta tre esempi a livello utente. Il secondo esempio è legato a vincoli normativi e di standard (questo si applica molto spesso nel dominio applicativo della difesa).

Il terzo requisito può ad esempio far riferimento ad un callcenter che usa un software per chiamare automaticamente la gente, si tratta sicuramente di un vincolo legislativo per i diritti di privacy.

Per completezza ora si cita una “terza” categoria di requisiti, che però in realtà si sovrappone, con le altre due: i **Requisiti di Dominio**.

*Infatti un requisito di dominio rappresenta un requisito funzionale o non funzionale che invece di derivare dalle necessità dell'utente deriva dal dominio applicativo*

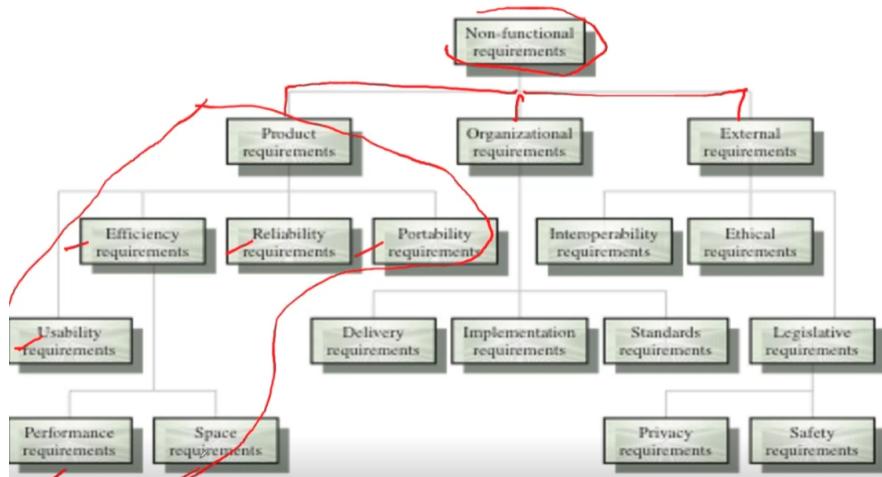
**Es.1** I documenti di rendiconto contabile, secondo la normativa XYZ.03, devono essere stampati alla ricezione e cancellati immediatamente

**Es.2** L'interfaccia utente per l'accesso al database magazzino deve essere conforme allo standard ZX.01

I requisiti non funzionali sono più difficili da trattare rispetto a quelli funzionali in quanto anzitutto coprono un insieme di possibilità più ampio rispetto a quelli funzionali, ma soprattutto perché per identificare le caratteristiche che il mio software dovrà assumere e quindi i relativi requisiti non funzionali devo far riferimento a un **modello di qualità**, che le descrive in modo preciso.

Qui di seguito una gerarchia dei requisiti non funzionali non del tutto esaustiva (ci ritorneremo nella seconda parte del corso) che fa intuire la complessità di questi requisiti.

## Classificazione requisiti non funzionali



*Non esiste comunque una strategia migliore per l'identificazione dei requisiti non funzionali: si può prima decidere di scrivere i funzionali ma spesso si opera in modo congiunto.*

La cosa importante è però essere molto attenti a definire questi requisiti fin dall'inizio dal momento che questi tipicamente sono scritti in linguaggio naturale e quindi possono comportare problemi. *I principali problemi che si possono riscontrare:*

- **Ambiguità** == il requisito non deve poter essere interpretato in modi diversi.

es. specificare un tempo senza far riferimento al fuso orario

es. significato di "appropriato visualizzatore": dal punto di vista dell'utente magari un visualizzatore appropriato per il documento in questione (per word visualizzatore word, per excel excel etc..) mentre dal punto di vista dello sviluppatore magari implementare un generico visualizzatore di testo che mostri il contenuto del documento.

- **Incompletezza** == i requisiti non includono la descrizione di tutte le caratteristiche richieste. Questo punto non è semplice poiché spesso neanche il cliente sa definire con precisione quello che vuole, in questo senso uno strumento che può aiutare può essere il prototipo rapido.

- **Inconsistenza** == conflicti o contraddizioni nella descrizione delle caratteristiche del sistema tra i vari requisiti. Difficili da scovare soprattutto se si hanno tanti requisiti.

es. Req 1 ogni form di input deve contenere non più di 5 campi editabili dall'utente  
 Req 2 nella forma di input relativa all'inserimento di dati anagrafici deve introdurre nome, cognome, anno, nascita, cell, fax...

**Un ulteriore problema che però riguarda esclusivamente i requisiti non funzionali** è la **Verificabilità dei Requisiti**: il cliente potrebbe esprimere i requisiti non funzionali in modo generico, ad esempio chiedendo che il software sia facile da usare. Ma *dire una cosa del genere non è quantificabile ed è difficile da verificare in quanto per lo sviluppatore potrebbe essere facile ma per l'utente no.*

**In generale quindi i requisiti non funzionali devono essere espressi in modo quantitativo** affinché si possa dimostrare una volta terminato il software che quel requisito è stato rispettato.

## Esempi di misure per requisiti

Property	Measure
Speed	Processed transactions/second User/Event response time Screen refresh time
Size	K Bytes Number of RAM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Alcuni requisiti sono facili da misurare come efficienza (speed) e dimensioni, ma altri meno come l'**Usabilità** (facile da usare). Per misurarla ad esempio abbiamo il **training time** (inteso come il “tempo di addestramento” necessario per un utente affinché apprenda ad usare il software. Es. per un software per aerei un aggiornamento prevede un training time anche molto lungo affinché ci si assicuri che i piloti siano a piena conoscenza di quanto realizzato) oppure ancora **number of help frames** (i frame nel software che spiegano in dettaglio quando ci punto sopra cosa fa quella funzionalità).

**Reliability == affidabilità, Robustezza == capacità del prodotto di resistere a eventuali problemi**

Ulteriore requisito difficile da definire in termini di misure è la **Portabilità**: facilità nel portare un prodotto software in esecuzione su una piattaforma differente rispetto a quella nativa per cui è stata sviluppata. Una misura potrebbe essere la **percentuale di istruzioni dipendenti dalla piattaforma nativa** (come requisito dico di far sì che questa percentuale non superi un tot affinché in futuro possa sistemare il codice per spostarmi in un'altra piattaforma).

Come scriviamo i requisiti?

Come già anticipato **i requisiti utente**, essendo a livello di astrazione molto alto per essere compresi da tutti, sono tipicamente espressi in linguaggio naturale seguendo però alcune linee guida per evitare problemi, tra cui **evitare l'uso di termini tecnici, usare un formato standard per ogni requisito, evidenziare le parti fondamentali per ogni requisito, utilizzare il linguaggio naturale in modo consistente** (es. uso di “deve” per requisiti necessari e “dovrebbe” per quelli desiderabili)

## Esempio di requisito utente

**3.5.1 Adding nodes to a design** GAD

**3.5.1.1** The editor shall provide a facility for users to add nodes of a specified type to their design.

**3.5.1.2** The sequence of actions to add a node should be as follows:

1. The user should select the type of node to be added.
2. The user should move the cursor to the approximate node position in the diagram and indicate that the node symbol should be added at that point.
3. The user should then drag the node symbol to its final position.

**Rationale:** The user is the best person to decide where to position a node on the diagram. This approach gives the user direct control over node type selection and positioning.

*Specification:* ECLIPSE/WS/Tools/DE/FS/Section 3.5.1

Qui un esempio di standard per definire requisito utente, numerazione rigorosa e roba importante in grassetto, poi man mano che scendo specifica di come si vorrebbe ad alto livello che si comporti il software in accordo con il requisito.

Poi **Rationale** che sarebbe il perché esiste quel requisito e **Specification** che è un puntatore al corrispondente requisito sistema (alla specifica di questo requisito utente)

Quando invece si arriva a scrivere i **requisiti di sistema la possibile scelta di linguaggio è molto più ampia** rispetto al naturale dei requisiti utente.

Qui è necessario essere molto più precisi in quanto saranno questi i requisiti usati come base per lo sviluppo. Diverse possibili notazioni:

## Requisiti di sistema (specifiche)

- Specifiche più dettagliate dei requisiti utente
- Sono usati come base per il progetto software
- Possono essere espressi facendo uso di notazioni differenti

Notation	Description
<i>Structured natural language</i>	This approach depends on defining standard forms or templates to express the requirements specification.
<i>Program description languages (PDL)</i>	This approach uses a language like a programming language (PDL, Program Description Language) but with more abstract features to specify the requirements by defining an operational model of the system.
<i>Graphical notations</i>	A graphical language, supplemented by text annotations is used to define the functional requirements for the system. The graphical language is used to define system models.
<i>Mathematical specifications</i>	These are notations based on mathematical concepts such as finite-state machines or sets. These unambiguous specifications reduce the arguments between customer and contractor about system functionality. However, most customers don't understand formal specifications and are reluctant to accept it as a system contract.

Si considera **informale** l'utilizzo di linguaggio naturale strutturato (comunque più specifico di quello utente) e **formale** quello di specifiche matematiche (non ambigue per definizione e verificabili in modo automatico, utilizzate per software critici in quanto molto costosi), e tra questi due estremi le **notazioni semi-formali**: **PDL** come *pseudocodice*, poi **utilizzo di notazioni grafiche** (che è ciò che faremo con il progetto).

Ora vedremo linguaggio naturale strutturato e PDL, mentre svilupperemo poi più in dettaglio le notazioni grafiche. Riguardo le mathematical specifications vedremo solo qualche esempio.

Anzitutto vediamo il passaggio dal requisito utente di prima al corrispettivo requisito di sistema in linguaggio naturale strutturato.

## Esempio di requisito di sistema

- basato su *form* in linguaggio naturale

ECLIPSE/Workstation/Tools/DE/FS/3.5.1	
<b>Function</b>	Add node
<b>Description</b>	Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.
<b>Inputs</b>	Node type, Node position, Design identifier.
<b>Source</b>	Node type and Node position are input by the user, Design identifier from the database.
<b>Outputs</b>	Design identifier.
<b>Destination</b>	The design database. The design is committed to the database on completion of the operation.
<b>Requires</b>	Design graph rooted at input design identifier.
<b>Pre-condition</b>	The design is open and displayed on the user's screen.
<b>Post-condition</b>	The design is unchanged apart from the addition of a node of the specified type at the given position.
<b>Side-effects</b>	None
<b>Definition:</b>	ECLIPSE/Workstation/Tools/DE/RD/3.5.1

Si tratta di definire nei dettagli cosa devo realizzare, intuitivamente la “dichiarazione” della funzione (so esclusivamente la firma: parametri di input e output). Infatti si legge nell'esempio solo la descrizione, gli input, output (design identifier sta a rappresentare il fatto che si aggiorna il progetto con il nuovo nodo), source come sorgente dei valori dei parametri in input, destination (database del progetto perché il progetto deve essere aggiornato in relazione all'aggiunta del nodo), requisiti (l'utente deve avere un progetto aperto, identificato da un certo design identifier), pre-condizione (condizione che deve essere vera affinché possa eseguire la funzione, in questo caso quando il progetto è aperto e mostrato sullo schermo), post-condizione (condizione che deve essere vera dopo aver eseguito la funzione, cambia solo l'aggiunta del nodo), effetti collaterali e Definition che punta al requisito utente corrispettivo. Starà allo sviluppatore realizzare la “definizione” della funzione (scrivere il corpo affinché tutto il puttanaio sopra sia rispettato).

*Nonostante questa specifica sia comunque vicina a quanto necessario per descrivere la funzione, può comunque comportare problemi di ambiguità e per questo spesso si preferisce far riferimento a linguaggi semi-formali come PDL.*

## Esempio di requisito di sistema (2)

- basato su **PDL** (Java-like)

```
class ATM {  
    // declarations here  
    public static void main (String args[]) throws InvalidCard {  
        try {  
            thisCard.read () ; // may throw InvalidCard exception  
            pin = KeyPad.readPin () ; attempts = 1 ;  
            while ( !thisCard.pin.equals (pin) & attempts < 4 )  
                { pin = KeyPad.readPin () ; attempts = attempts + 1 ;  
                }  
            if (!thisCard.pin.equals (pin))  
                throw new InvalidCard ("Bad PIN");  
            thisBalance = thisCard.getBalance () ;  
            do { Screen.prompt (" Please select a service ") ;  
                service = Screen.touchKey () ;  
                switch (service) {  
                    case Services.withdrawalWithReceipt:  
                        receiptRequired = true ;  
                }  
            }  
        }  
    }  
}
```

Si descrive il comportamento per l'implementazione del comportamento di un ATM. Viene descritto in java-like cosa succede quando metto la carta (quindi anzitutto potrebbe non leggerla come valida, poi provo il pin e ogni pin corrisponde a un tentativo per un massimo di 3, poi se scrivo bene il pin arrivo alla schermata per gestire il mio conto etc...).

### Lez 10 (02/11)

*Usando pseudocodice come visto sopra sicuramente evito ambiguità che potrebbero esservi nell'uso di linguaggio naturale strutturato.*

Tuttavia **PDL ha uno svantaggio**: ci troviamo ancora nella fase di specifica del software e quindi della descrizione di cosa deve fare il software e non come deve essere implementato (ciò sarà a carico dalla fase di progettazione in poi).

Il rischio che quindi si corre con PDL è che si pesti i piedi al lavoro dei progettisti, si deve usare lo pseudocodice non per dettagli algoritmici. Quindi più che un esempio visto sopra **di solito è conveniente usare PDL esclusivamente per la definizione di interfaccia, che si limitano a dire cosa fare invece di come** (equivalenza di dichiarazione di funzione piuttosto che definizione)

## Esempio di requisito di sistema (3)

- specifica di interfaccia basata su **PDL**

---

```
interface PrintServer {  
  
    // defines an abstract printer server  
    // requires: interface Printer, interface PrintDoc  
    // provides: initialize, print, displayPrintQueue, cancelPrintJob, switchPrinter  
  
    void initialize ( Printer p ) ;  
    void print ( Printer p, PrintDoc d ) ;  
    void displayPrintQueue ( Printer p ) ;  
    void cancelPrintJob ( Printer p, PrintDoc d ) ;  
    void switchPrinter (Printer p1, Printer p2, PrintDoc d) ;  
}  
//PrintServer
```

---

Quanto viene scritto sia per i requisiti utente (definizione) che per requisiti di sistema (specifiche) viene inserito nel documento guida di ogni progetto di sviluppo software: il **Documento di Analisi dei Requisiti** (o **Documento di Specifica**).

**Questo documento quindi descrive COSA il sistema deve fornire** (dominio del problema) **e non COME** il sistema deve essere sviluppato (dominio della soluzione, sarà compito di fasi successive).

*Questo è un documento che interviene continuamente sia durante lo sviluppo che successivamente per la manutenzione, tanto che l'assenza di questo documento abbiamo visto può comportare tecniche di reverse-engineering per ricavarlo a partire dal software.* Si hanno ruoli di tutti i tipi che contribuiscono alla scrittura del documento: **il Cliente** (ovvio), **i Manager** (tipicamente sprovvisti di competenze legate allo sviluppo software che usano il documento per capire quanto investire sullo sviluppo, se conviene o meno etc..), **System Engineers** (usano il documento per capire quanto deve essere sviluppato), **System Test Engineers** (usano il documento per generare il documento di **Plan Test**, insieme di test case dove ognuno di essi è una descrizione dal punto di vista procedurale di quanto si deve fare per procedere al testing di una certa funzione, tipicamente comprende funzione da considerare, input e output atteso. A partire dal Documento di Specifica quindi per ogni requisito vi sarà il corrispondente test case nel Plan Test ciò per verificare che ogni requisito sia soddisfatto), **System Maintenance Engineers** (si ricorda infatti che la manutenzione più comune non è quella correttiva ma quella perfettiva, che indirizza le modifiche ai requisiti)

Quando si modifica un requisito è necessario modificare il documento di specifica, per questo è importante mantenere i link di tracciabilità per capire quanto la modifica sia impattante sia lato definizione che lato specifica.

**Per scrivere il Documento di Specifica si fa uso di un formato standard** vista la sua importanza

## Struttura del documento di specifica

Basata sullo standard **IEEE 830-1998**

*(IEEE Recommended Practice for  
Software Requirements Specifications)* pag 1/2

### Preface

expected readership, version history, changes summary

### Introduction

purpose, brief description of the system, interaction with other systems, scope within the business context

### Glossary

definition of technical terms used in the document

### User requirements definition

functional and non-functional user requirements

### System architecture

high-level overview of the system components

### System requirements specification

functional and non-functional system requirements

## System models

description of the relationships between the system components and the system and its environment

## System evolution

assumptions on which the system is based and anticipated changes (hardware evolution, user needs changes, etc.)

## Appendices

specific information related to the application which is being developed (ex. HW and DB descriptions)

## Index

table of contents, alphabetic index, list of diagrams, etc.

Si passa ora alla parte di **Ingegneria dei Requisiti**.

Ci occuperemo del **processo di ingegneria dei requisiti**, questo varia *in base al dominio applicativo* (es. se software critico allora specifica matematica), *alle persone coinvolte e all'organizzazione che sviluppa*.

Noi useremo un approccio **Object Oriented**, ma è comunque possibile individuare un insieme di attività generiche comuni a tutti i processi:

- **Studio di Fattibilità**
- **Definizione e Analisi dei Requisiti**
- **Specifiche dei Requisiti**
- **Convalida Requisiti**
- **Gestione Requisiti**

Lo **Studio di Fattibilità** è la fase preliminare del processo di ingegneria dei requisiti ed è ciò che mi suggerisce se è opportuno procedere con lo sviluppo software.

Si basa su una descrizione sommaria del sistema software e delle necessità utente, ed è un'attività che deve essere svolta in tempi rapidi in quanto se l'esito è no devo passare a un'altra. Le informazioni necessarie per lo studio di fattibilità vengono raccolte tramite colloqui con Client Manager, gli Ingegneri del Software con esperienza nel dominio applicativo e Esperti delle tecnologie da utilizzare per capire se si può procedere e gli utenti finali del sistema.

Lo studio di fattibilità **produce come risultato un report** che stabilisce l'opportunità o meno di procedere allo sviluppo.

Nel caso in cui lo studio di fattibilità abbia dato esito positivo si parte con **la Definizione e Analisi di requisiti. Anzitutto il team di sviluppo si incontra con il cliente e gli utenti finali per identificare l'insieme di requisiti utente (definizione), dalla cui analisi si generano i requisiti di sistema (specifiche)**.

Tuttavia l'identificazione dei requisiti in realtà coinvolge personale che copre vari ruoli sia dentro che fuori l'organizzazione in quanto deve riguardare gli stakeholder. Si utilizza il termine stakeholder per identificare coloro che hanno interesse diretto o indiretto sui requisiti del sistema software da sviluppare e che quindi deve essere coinvolto in questa fase.

I **task principali di questa attività sono diversi**:

- **Comprensione del Dominio**: importante soprattutto dal punto di vista **dell'analista** **software**: egli deve studiare in tempi brevi il dominio applicativo (es. se il sistema software riguarderà un ufficio postale, allora l'analista ne deve comprendere il funzionamento)
- **Raccolta dei Requisiti** tramite interazione con gli stakeholder
- **Classificazione dei requisiti** (es. requisiti riguardanti gestione dei dati, riguardanti l'interfaccia utente etc..)
- **Risoluzione dei conflitti**: si devono identificare eventuali conflitti/contraddizioni tra requisiti
- **Assegnazione delle priorità ai requisiti** (importante soprattutto quando si utilizza un approccio incrementale per cui prima devo gestire quelli di maggior importanza)
- **Verifica dei requisiti per verificarne completezza e consistenza**

Esistono **diverse tecniche che aiutano l'analista a svolgere questi task**.

- **Tecniche di Identificazione dei Requisiti**: **Prototipazione** (già vista in passato, costruendo un prototipo posso capire meglio se l'utente vuole roba simile o no, in questo modo riduco i rischi di definire requisiti scorretti), **Casi d'uso** (basati su scenari, quando il cliente non sa nemmeno definire cosa vorrebbe gli si chiede di descrivere uno scenario in cui lo utilizzerebbe per capire i requisiti), **Etnografia** (quando non si riesce ad ottenere informazioni dall'utente l'analista osserva il lavoro di un'organizzazione per capire come definire i requisiti, es. ufficio postale l'analista studia a fondo il funzionamento per capire i requisiti).
- **Tecniche di analisi (e specifica) dei requisiti**: più che di tecniche qui si parla di metodi che possono essere semi-formali (basate su modelli del sistema e usate dai metodi di analisi strutturata o analisi orientata a oggetti) o formali (specifiche matematiche come macchine a stati finiti, Petri Net etc...)

Si procede poi con la **Convalida dei Requisiti**: è necessario trovare eventuali difetti per evitare costosi rework in fasi più avanzate del ciclo di vita.

I controlli includono quindi validità, consistenza, completezza, realizzabilità, verificabilità e ci sono delle *tecniche che supportano il lavoro dell'analista nella convalida dei requisiti*, tra cui:

- **Revisioni Informali** (es. come quella vista in microsoft per cui per ogni sviluppatore c'è un tester che gli guarda il codice, in questo caso però ci si riferisce ai requisiti)
- **Revisioni Formali**, che prevedono **tecniche di Walkthrough** (si coinvolgono tutti quelli che possono dare un contributo e si discute man mano che si legge il documento di eventuali problemi) e **Ispezioni** (simile a walkthrough solo molto più formale in quanto si identifica un team di ispezioni con ruoli precisi ed il processo di ispezione è definito precisamente anche in termini di tempistiche. A ogni membro viene chiesto di studiare il documento annotando osservazioni e dopo circa una settimana vi è la vera e propria riunione di confronto. **Molto costose ma anche grandi risultati in termini di scovare e correggere errori**).
- **Prototipazione**

- **Generazione di Test Case**, già descritto prima, per cui *tramite la generazione di test case capisco se ho scritto bene o male un requisito* (e se si ha problema nel generare test case è un problema legato ai requisiti e non chi lo genera tipicamente)
- **Analisi di consistenza automatizzata** (per requisiti formali)

L'ultima attività del processo generico di ingegneria dei requisiti è quello relativo alla **Gestione dei Requisiti**. Come abbiamo visto l'idea Waterfall per cui all'inizio faccio definizione e specifica di requisiti per poi non toccare più il documento risultante è irrealistica, si ritoccano i requisiti anche durante e dopo lo sviluppo.

Quindi *la Gestione dei Requisiti rappresenta il processo di identificazione e controllo delle modifiche subite da essi durante il ciclo di vita del sistema software*.

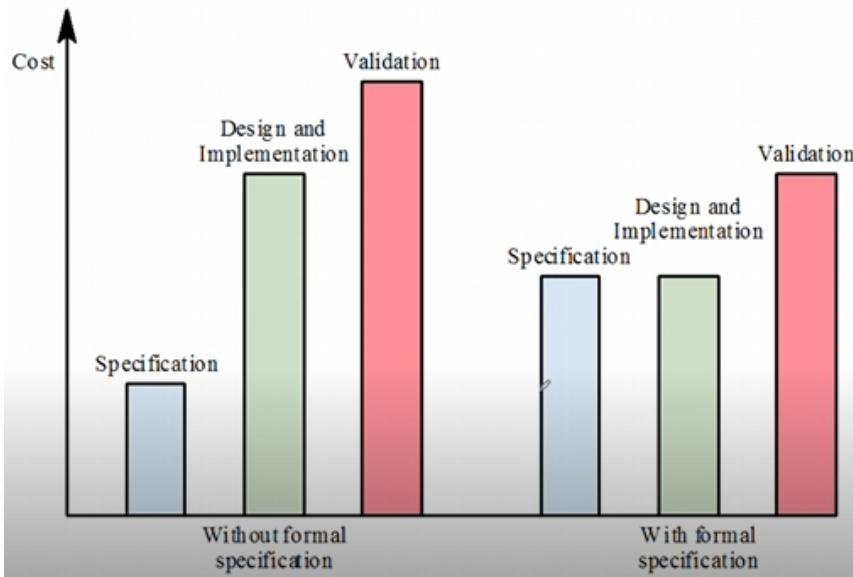
I requisiti di un sistema software possono essere classificati in questo senso in **Requisiti Stabili** se hanno probabilità minima di esser modificati nel tempo e **Volatili** se invece la probabilità che ciò succeda è elevata.

Riguardo i volatili ad esempio si hanno i *mutabili* (modifiche legate a cambiamenti dell'ambiente operativo), *emergenti* (modifiche causate da una migliore comprensione del sistema software dopo aver interagito con gli stakeholder), *consequenziali* (dovute all'introduzione di sistemi informatici nel flusso di lavoro es. se alle poste i bollettini a un certo punto vengono pagati digitalmente) e di *compatibilità* (legate a cambiamenti nei sistemi e nei processi aziendali)

Tali eventuali modifiche dei requisiti devono essere pianificate mediante:

- *identificazione univoca dei requisiti in questione*
- *gestione delle modifiche tramite analisi di costi, dell'impatto e della realizzazione*
- *politiche di tracciabilità per capire esattamente quali sono le relazioni tra requisiti e progetto del sistema software*
- *uso di tool CASE per supporto alle modifiche* (devo usare strumenti che mi supportino in questo lavoro) (lo strumento per eccellenza è IBM Doors non solo per questo ma in generale per la requirements engineering)

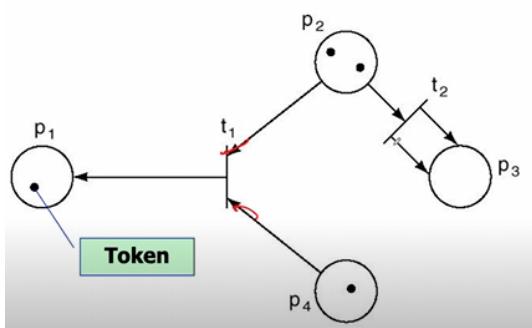
## Specifiche formali vs. informali



Abbiamo detto come normalmente l'uso di specifiche formali richieda effort maggiore, che però deve essere affiancato da una fase di verifica. Senza specifica formale la fase di specifica in sé è molto meno costosa della specifica senza specifica formale. Tuttavia la scelta di una specifica formale comporta tipicamente la riduzione dei costi per le fasi successive di progettazione, verifica e convalida grazie al supporto di strumenti automatizzati.

Vediamo ora qualche esempio per capire la forma di queste specifiche formali. Si tratta come detto di notazioni che hanno basi teoriche matematiche solide, ma spesso per facilitare il lavoro all'analista sono affiancate da una sintassi visuale più semplici da utilizzare.

**Petri Net:** sono state introdotte ancor prima della nascita della isw per specificare sistemi con elevato grado di concorrenza e problemi di temporizzazione e sincronizzazione come le telecomunicazioni.



La versione originale fornisce una sintassi visuale nella quale esistono tre costrutti fondamentali: **Place** (cerchio), **Transitions** (barre) e **Archi Orientati** (frecce, collegano sempre o posti a transizioni o transizioni a posti).

Nel caso del software **il place rappresenta una componente software, la transizione una possibile azione svolta dalla componente software e la freccia una relazione di input output tra places.**

**Questo sistema formale serve a capire come un certo sistema evolve nel tempo in fase di esecuzione.** Per rappresentare l'evoluzione del sistema in questo senso si utilizza un altro elemento, il **Token** (pallini neri inseriti all'interno dei places).

L'operazione per cui inserisco token nei places è detta **Marking**. Poiché il sistema evolve esisterà un **Marking iniziale** (distribuzione iniziale di token), dei marking intermedi e uno finale.

Per definire l'evoluzione del sistema è necessario definire delle regole.

Anzitutto diremo che **una Transizione è Abilitata sse esiste almeno un token all'interno di ogni posto collegato in ingresso alla transizione**. Nel caso sopra sia t1 che t2 sono abilitate.

Ora l'idea è che a partire dallo stato iniziale il sistema potrebbe evolvere svolgendo la transizione t1 o la t2, l'idea è di svolgerle tutte per trovare eventuali inconsistenze.

Si definisce ora come **Firing** di una transizione come il prelievo da parte del place in ingresso e consegna al place in uscita di tanti token quanti sono gli archi uscenti.

Se ad es. è t1 a scattare allora si tolgono un token da p2 e uno da p4 e se ne mette uno a p1 (uno solo perché un solo arco in uscita).

Quindi stato iniziale  $S_0 = (1, 2, 0, 1)$  (dentro p1 1 token, in p2 2 etc..).

Quindi stato  $S_1 = (2, 1, 0, 0)$ . A questo punto t1 non è più abilitata perché in p4 non ci sono più token ma resta abilitata t2 perché 1 token in p2. A questo punto scatta t2 e quindi p2 avrà 0 token e p3 2 token poiché 2 archi in uscita quindi

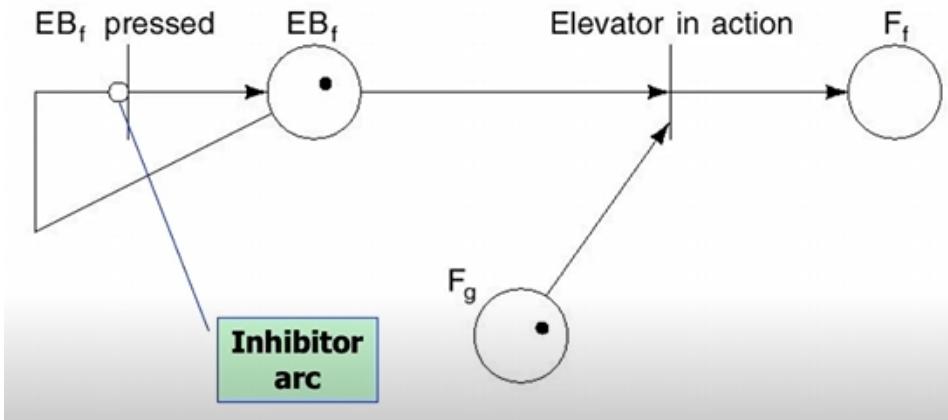
$S_2 = (2, 0, 2, 0)$  e a questo punto dato che non ci sono più transizioni abilitate  $S_2$  è stato finale.

Facendo invece partire prima t2 avrò  $S_1 = (1, 1, 2, 1)$ , ancora una volta mi accorgo di avere un bivio poiché sia t1 che t2 sono abilitate, scelgo t1  $S_2 = (2, 0, 2, 0)$  ed è finale. Se facevo scattare t2 invece  $S_2 = (1, 0, 4, 1)$  anch'esso finale.

L'obiettivo del modello in definitiva è verificare che in tutte le possibili esecuzioni il sistema funzioni correttamente (nel caso del software che vediamo la prossima lezione ad esempio che non si arrivi a deadlock o che non vi siano stati indesiderati).

## **Lez 11 (6/11)**

Dopo aver capito le regole generali della rete di Petri vediamole applicate al caso specifico di un software. *Ipotizziamo ad esempio di voler analizzare il comportamento di un software che regola la centralina di controllo di un ascensore* (considerabile sistema critico in quanto un danno potrebbe comportare danni a persone).



Viene esplicitato via rete di Petri un requisito in particolare: se l'ascensore si trova al piano terra e un utente lo chiama dal primo piano deve andare al primo piano.

Si necessitano in questo caso 3 posti:  $F_g$  == ascensore al ground floor,  $F_f$  == ascensore al first floor e  $Ebf$  == elevator button first floor (si chiama l'ascensore al primo piano).

Il token al posto  $F_g$  specifica che l'ascensore si trova al piano terra, mentre il token in  $Ebf$  specifica che il pulsante è stato premuto. Per rappresentare il fatto che l'ascensore si muove uso la transizione elevator in action: tale transizione è abilitata come detto nelle regole se esiste almeno un token sugli archi in ingresso.

Nel nostro caso vediamo che è abilitata, quindi si arriva al nuovo stato per cui non vi è alcun token in  $Ebf$  o  $F_g$  ma un token in  $F_f$ .

*Ci serve una transizione che specifichi la pressione del pulsante al primo piano  $Ebf$ , a questo serve  $Ebf\,pressed$  dove è presente anche un pallino piano con una barra davanti, detto **arco inibitore**. Il significato di questo simbolo è che quella transizione funziona esattamente al contrario rispetto a una transizione tradizionale: la transizione è abilitata sse non vi è alcun token in ingresso.*

Quando quindi  $Ebf$  non ha alcun token per rappresentare il fatto che qualcuno preme il pulsante la self transition inserisce il token in  $Ebf$ , e se ha un token abilitato e qualcuno ripreme il pulsante non succede nulla.

**In generale ci si accorge come usando una specifica formale anche un requisito semplice come quello visto richiede parecchio lavoro, molto costoso utilizzarle**

(notazione molto più ampia del linguaggio naturale). *Solitamente limitata alla parte di controllo del software (non ha senso usarla ad esempio per l'interfaccia).*

Ulteriore costo sta nel fatto che oltre che costruire il modello questo deve anche essere validato (devo verificare che i comportamenti corrispondano con il mio software quindi convalida molto importante). Il principale vantaggio come già visto è che esistono tool automatici che verificano il tutto.

A fronte di alcune limitazioni della Petri Net, nel corso del tempo sono stati sviluppati diversi "dialetti" di questa specifica formale.

**Una limitazione lampante ad esempio è che ogni transizione rappresenta un'azione che è istantanea, per cui non vi è un tempo associato alla relativa esecuzione.**

Tale limitazione è stata superata da un dialetto chiamato **GSPN** (generalised

*stochastic petri net*) dove **ad ogni transizione è associato un tempo che permette di convalidare anche i tempi prestazionali del sistema** (basarsi anche sul tempo di esecuzione di specifiche funzioni del software).

Un altro ancora è **CPN colored pn**, si sono introdotti i colori per i token che **descrivono comportamenti diversi del sistema per diverse classi di utenti** (in base a chi lo utilizza).

**Un'altra caratteristica della pn è che il concetto di stato è rappresentato indirettamente** (per conoscerlo devo vedere la distribuzione di token nei place).

La notazione che ora vediamo invece prevede un altro approccio: **Specific Formale con Macchine a Stati Finiti FSM**.

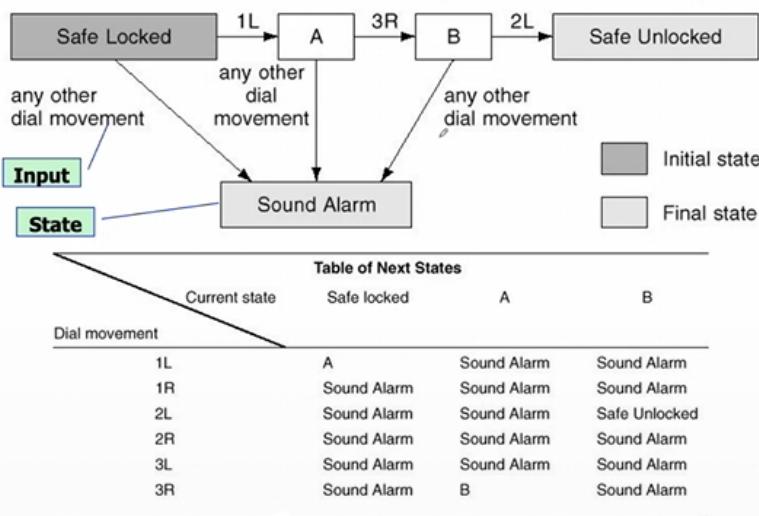
**L'obiettivo è come per pn rappresentare la dinamica (evoluzione) di un sistema attraverso un insieme di stati che il sistema attraversa.**

*La differenza principale è che la primitiva di base permette di rappresentare direttamente lo stato del sistema.*

**La primitiva di base è il rettangolo che appunto rappresenta lo stato, l'arco orientato rappresenta un evento che porta il passaggio a uno stato successivo.**

**Stati iniziali** con grigio più scuro, **finali** più chiaro.

### Specifiche formali con Finite State Machine (FSM): esempio



L'esempio qui presente rappresenta il comportamento di una cassaforte.

Si parte dalla cassaforte chiusa, che per essere aperta prevede il movimento di una "manopola". Stati intermedi per rappresentare il passaggio dopo ogni movimento. (es. 1L == un movimento a sx). Qualsiasi movimento sbagliato porta allo stato Sound Alarm.

**Un problema in questo caso come in pn è che rappresentando un sistema molto complesso posso avere un numero di stati enorme.**

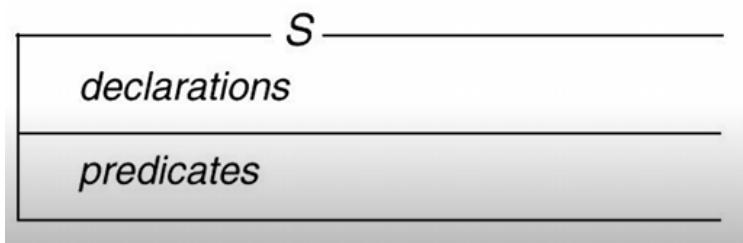
Ciò che si fa normalmente è associare a ogni fsm una **Table of Next States** con colonne stato e righe azione.

Sia *FSM* che *PN* non sono notazioni create per la specifica di software, come terzo esempio vediamo una notazione formale creata appositamente, il **linguaggio Z**.

La primitiva di base è il **concetto di schema**, Z consiste in un set di schemi.

Sia con *PN* che con *FSM* i due concetti fondamentali erano **stato e azione** per descrivere l'evoluzione del sistema, in questo caso lo schema basta a rappresentare entrambe.

Ogni schema Z ha il seguente formato:



Nome, variabili, predici che agiscono sulle variabili.

Vediamo un esempio sia per rappresentare lo stato che l'azione.

## Linguaggio Z esempio di specifica di stato

### Button\_State

floor\_buttons, elevator\_buttons : **P** Button  
buttons : **P** Button  
pushed : **P** Button

floor\_buttons  $\cap$  elevator\_buttons =  $\emptyset$

floor\_buttons  $\cup$  elevator\_buttons = buttons

### Abstract Initial State

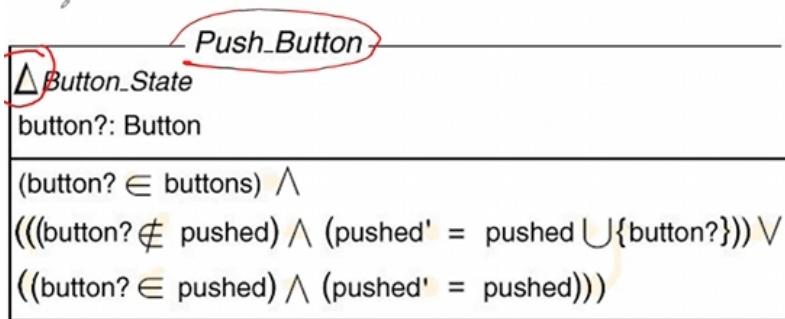
Button\_init := [Button\_State' | pushed' =  $\emptyset$ ]

Si specifica lo schema del bottone di un ascensore. Si distinguono i pulsanti dentro l'ascensore e sui piani per chiamarlo. Si definiscono le variabili con il nome e per definire il tipo si usa la notazione : **P** Button dove in questo caso Button è l'insieme di tutti i pulsanti e **P** denota l'insieme potenza (insieme di tutti i possibili sottinsiemi). Di questo tipo si definiscono i pulsanti dentro e fuori ascensore, i pulsanti e i pulsanti premuti.

Si definisce lo stato iniziale come  $\text{Button\_init} := \{\text{Button\_State}' \mid \text{pushed}' = \emptyset\}$  (nello stato iniziale l'insieme dei pulsanti dell'ascensore ha stato tale per cui non è premuto, l'insieme pushed è vuoto).

Nella parte dei predici specifico come deve comportarsi lo stato del bottone: l'intersezione delle variabili floor\_buttons e elevator\_buttons è vuoto (sono pulsanti distinti) mentre la loro unione sono tutti i pulsanti.

Vediamo invece l'utilizzo dello schema per un'azione: premere il bottone.



Delta rappresenta gli stati su cui agisce l'azione (in questo caso lo schema definito prima). Nella parte declarations, trattandosi di un'operazione (funzione), definisco i parametri della funzione (Button). Il fatto che sia parametro di input è evidenziato da ? (output sarebbe stato !). Nella zona predici definisco invece come agisce quest'operazione sulle variabili dello stato che prende in considerazione (button\_state).

La prima riga mi dice che il parametro di input deve appartenere all'insieme dei buttons, messo in AND con ciò che specifica che se il bottone è spento si accende, se è già acceso invece non si fa nulla.

(se il bottone non appartiene a pushed allora (“implicito” nell’and) il nuovo valore dell’insieme di pushed (nuovo valore perché ') è uguale al vecchio valore + il bottone che ho appena premuto, oppure se il bottone appartiene a pushed allora il nuovo insieme è uguale al precedente).

Vediamo ora le specifiche **semi-formali** (*livello intermedio tra linguaggio naturale e specifiche formali*).

Come visto tipicamente per specificare il software ciò che si fa è costruire un modello del sistema, ossia una sua rappresentazione astratta che facilita la comprensione del suo funzionamento.

*Ciò che infatti faremo è studiare una notazione visuale che ci permetterà di costruire tale modello, tramite il quale sarà possibile definire le specifiche.*

*Per descrivere completamente il sistema è necessario costruire diversi modelli, che lo rappresentino da vari punti di vista (informazioni, funzioni e comportamento dinamico (evoluzione))*

*Esistono due principali metodi di specifica semiformale (per effettuare la specifica dei requisiti);*

- **metodi di analisi strutturata (o procedurale)**
- **metodi di analisi orientata agli oggetti**

*Per costruire questi modelli del sistema si necessita di un **linguaggio di modellazione**. Ne esistono di diversi tipi, ognuno dei quali definisce un modello specifico.*

## Tipi di modelli del sistema

Per descrivere la specifica semi-formale di un sistema software si usano 3 tipi di modelli:

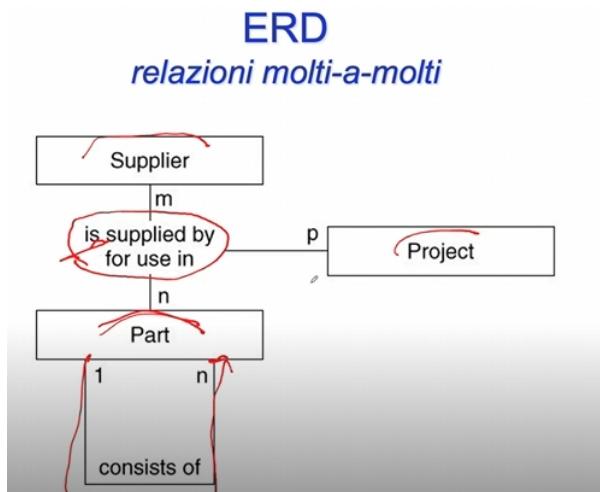
- ① **modello dei dati**: rappresenta gli aspetti statici e strutturali relativi ai dati (*data requirements*)
  - *ERD (not UML)*
  - *class diagram (UML)*
- ② **modello comportamentale**: rappresenta gli aspetti funzionali del sistema (*functional requirements*)
  - *data flow diagram (not UML)*
  - *use case diagram (UML)*
  - *activity diagram (UML)*
  - *interaction diagram (UML)*
- ③ **modello dinamico**: rappresenta gli aspetti di "controllo" e di come le funzioni del modello comportamentale modifichino i dati introdotti nel modello dei dati
  - *state diagram (UML)*

**UML** == *linguaggio standard di modellazione che utilizzeremo a supporto del nostro metodo di specifica object oriented.*

I not UML sono linguaggi di modellazione che venivano usati prima dell'arrivo dei metodi oo, quando ancora si usavano solo metodi di analisi strutturata.

Si tratta di **ERD** e **Data Flow Diagram**.

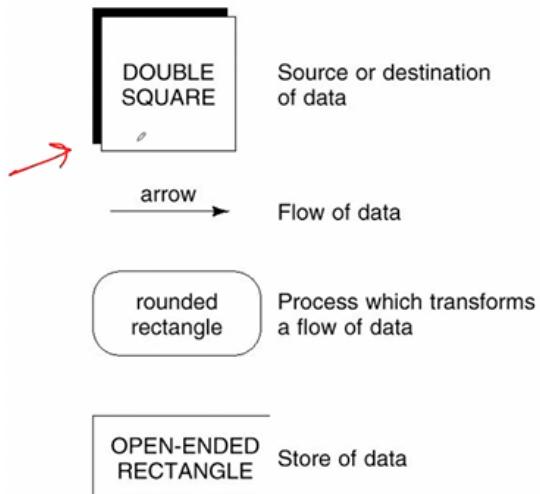
**ERD** diagramma entità relazione per costruire **modello dei dati** (ovvio).



(In pratica tra i semiformali o PDL o diagrammi. Se diagrammi per definire correttamente i requisiti di sistema conviene utilizzare un modello astratto che rappresenti il software da realizzare sotto tutti i punti di vista (dati, comportamentale, dinamico), e per fare sto modello si usano sti diagrammi che sono fatti a loro volta con sti linguaggi semi formali che possono essere OO (UML) o strutturati (ERD o Data Flow Diagram).

Il Data Flow Diagram invece **serve a definire modelli comportamentali**.

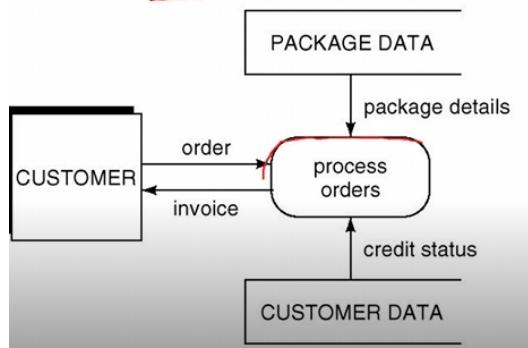
## Data Flow Diagram (DFD)



Utilizzo questi quattro costrutti per rappresentare le varie funzioni del software e come elaborano i dati, come partano da una sorgente e arrivino ad una destinazione, come vengano depositati o provengano da un archivio. Questa notazione è molto efficace dal momento che può essere usata a diversi livelli di astrazione.

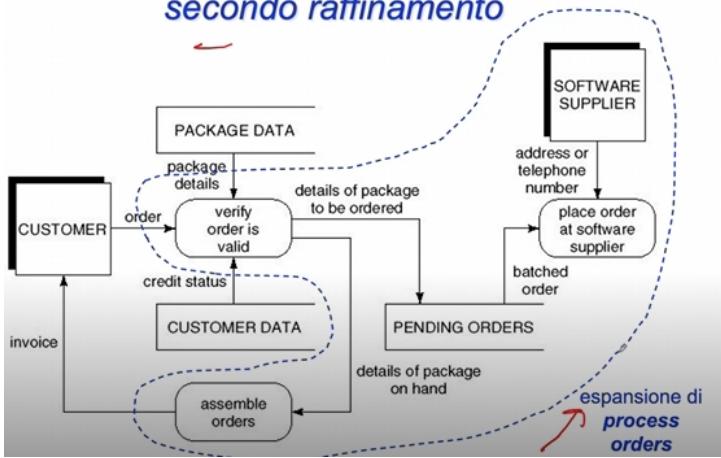
### Esempio di DFD

#### primo raffinamento



In questo esempio si usa DFD ad un livello di astrazione molto alto: l'intero software viene modellato facendo uso di un singolo processo. La sorgente e destinazione dati è il cliente, due archivi; uno legato ai dati del cliente e uno dei package venduti. Qui sotto un secondo raffinamento che più a basso livello mi approfondisce il processo *process orders*

## Esempio di DFD secondo raffinamento



L'ordine va assemblato insieme alla fattura ed inviato al cliente, è aggiunto all'archivio di ordini in attesa, gli ordini che necessitano di materiale che non è disponibile vengono elaborati dal processo di place order at software supplier, che interagisce con l'archivio software supplier.

Il diagramma non deve essere letto in ordine temporale, ma solo in termini di come i dati viaggiano all'interno del software.

Ovviamente si può scendere sempre più di livello, e normalmente si costruisce una gerarchia di DFD dove ad ogni livello dettaglio maggiore.

### Lez 12 (9/11)

Uno dei metodi di **analisi strutturata** più usati **prima dell'avvento degli object oriented** era lo **SSA Structured System Analysis**. Esso è costituito da **9 step** e basato sul concetto di **step-wise refinement**, l'obiettivo è fornire passo passo una guida al fine di completare l'attività di specifica.

Il **primo step** del metodo **suggerisce all'analista di partire producendo il Data Flow Diagram**. In particolare si dovrebbe utilizzare il documento dei requisiti utente o il prototipo (all'epoca si utilizzava principalmente waterfall e prototipo rapido) per identificare i flussi di dati, le sorgenti e destinazioni di dati e i processi (funzioni) che trasformano i dati.

Inoltre si vuole che si proceda per raffinamenti successivi aggiungendo dei dati ai data flow esistenti (gerarchia livelli astrazione).

Lo **step 2** **prevede di decidere quali sezioni devono essere automatizzate e come**. Si parte dall'analisi costi-benefici per decidere quali sezioni DFD devono essere automatizzate, e si decide come devono esserlo: **Batch** o **Online**.

**Online Processing** == i dati sono analizzati *real time*, *online* (es. quando inserisco i dati nel form e il software si accorge se manca qualcosa)

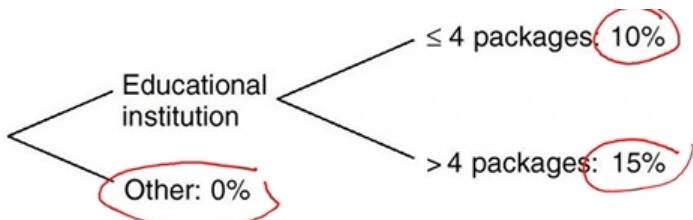
**Batch Processing** == operazioni che avvengono per lotti, ad es. nel caso degli ordini invece di prenderlo online e farlo partire immediatamente posso decidere di accumularne durante la giornata ed effettuare le operazioni insieme a fine giornata.

Nell'esempio di DFD con secondo raffinamento di prima il processo "verify order is valid" è sicuramente online, mentre "place order at software supplier" batch (infatti archivio i dettagli ordine nel pending orders per poi magari successivamente chiamare i fornitori "software supplier" per inviare gli ordini).

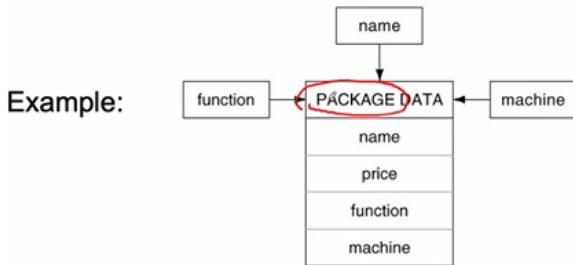
I successivi tre passi sono relativi rispettivi a un raffinamento di data flow, processi e data stores.

Lo **step 3** prevede quindi di **determinare i dettagli del data flow**, es. nel DFD di prima ho un arco "order" da cliente a processo, devo raffinarlo per capirne la vera e propria struttura ottenendo quindi dettagli come order\_identification, customer\_details, package\_details. Si procede poi ad approfondire ulteriormente questi dati per vedere se sono dati elementari o ancora di tipo strutturato. order\_identification ad esempio è un intero a 12 interi, costumer\_details è invece informazione strutturata composta da customer\_name, customer\_address etc... **Devo fare l'operazione fino a definire tutto come dato elementare per ogni data flow.**

Nello **step 4** mi occupo di **definire la logica processi**. Ad esempio posso utilizzare un albero decisionale per definire il comportamento di una funzione give\_educational\_discount:



Lo **step 5** si occupa dei **dettagli dei data store, si definisce il contenuto esatto di ogni archivio e come rappresentarlo** (secondo il formato specifico di un certo linguaggio di programmazione). Es. utilizzo di **DIAD data-immediate-access diagram**



Per memorizzare i dettagli di un package li organizzo per nome, prezzo, funzione e machine (sistema operativo). Il DIAD specifica quali sono i campi utilizzabili per effettuare ricerche dirette sui dati (in questo caso name, function e machine, per cui posso chiedere ad esempio quali sono tutti i software di un certo nome, oppure che girano su windows, o che sono elaboratori di testo, ma non in base al prezzo).

Dai successivi step capiamo come questo metodo sia figlio della sua epoca.

Lo **step 6** prevede di **definire le risorse fisiche**, oggi si sfrutta un **DBMS** per l'archiviazione di dati ma all'epoca c'era la possibilità di sfruttare il **file system** come strumento di archiviazione dei dati. In tal caso ne andava specificato il nome, l'organizzazione in termini di accesso (sequenziale, indicizzato...), il mezzo di

memorizzazione, i record il tutto arrivando a livello di dettaglio “field level” (es. il nome lo specifico come una stringa di 30 caratteri...)

**step 7: si determina la specifica dell'Input/Output.**

*Si devono specificare le forme di input (le maschere di inserimento dei dati) a livello di componenti e layout, ugualmente per gli screen di output e l'output da stampare. Queste informazioni erano necessarie perché all'epoca non erano disponibili interfacce utente di tipo grafico, ma solo testuale e con spazi molto limitati.*

**step 8: determinare il dimensionamento.** *Si stima il volume dell'input (es. numero di ordini attesi dal cliente giornalmente), frequenza di ogni report da stampare e la sua scadenza, dimensione e numero di record da passare dalla CPU alla memoria di massa (oggi impensabile) e dimensione di ogni file (es. file anagrafica clienti).*

**step 9: determinare i requisiti hardware,** *si definiscono quindi i requisiti della memoria di massa, per il backup (all'epoca dischi magnetici o nastro), le caratteristiche dei terminali utente e dispositivi di output, adeguatezza dell'hardware esistente per il software ed eventuali costi per hardware aggiuntivo.*

Chiaramente molte di queste scelte sono state abbandonate con l'evoluzione tecnologica, ma grazie a questi step l'analista software procedeva per **raffinamenti successivi** a completare la specifica.

**Il documento ottenuto doveva esser approvato dal cliente per poi passare alla fase di progettazione.**

Iniziamo ora la parte sul metodo di analisi dei requisiti Object Oriented.

**OOA == Object Oriented Analysis.**

*Questa è ancora una fase di definizione come già detto, si deve concentrare su COSA deve fare il prodotto senza occuparsi del COME.*

**OOA deve fornire una rappresentazione corretta, completa e consistente.**

**Completa** significa che oltre che considerare tutti i requisiti, deve anche specificare il software da tutti i punti di vista tramite **modello dei dati, modello comportamentale e modello dinamico**.

Un metodo OOA non definisce solo le tecniche, ma anche l'insieme di procedure, strumenti e linguaggi per avere un approccio sistematico alla gestione e allo sviluppo della fase di OOA.

**L'input del metodo OOA è l'insieme dei requisiti utente, e restituisce in output l'insieme dei modelli del sistema che definiscono la specifica del software.**

*OOA fa principalmente uso di notazioni visuali (diagrammi), ma essendo semiformale queste notazioni sono affiancate da metodi più tradizionali basati su linguaggio naturale.*

**Lo sviluppo dei modelli di OOA non è sequenziale ma iterativo** per cui si definiscono *man mano raffinamenti* successivi, inoltre **il lavoro sui modelli avviene in parallelo** dal momento che un modello può offrire informazioni utili agli altri e viceversa.

*Come per i metodi strutturali, esistono diversi metodi OOA.*

## Alcuni metodi di OOA (e OOD)

- **Catalysis:** metodo OO particolarmente indicato per lo sviluppo di sistemi software a componenti distribuiti.
- **Objectory:** metodo ideato da I. Jacobson che fonda lo sviluppo di prodotti software ad oggetti sull'individuazione dei casi d'uso utente (*use case driven*).
- **Shlaer/Mellor:** metodo OO particolarmente indicato per lo sviluppo di sistemi software *real-time*.
- **OMT (Object Modeling Technique):** metodo sviluppato da J. Rumbaugh basato su tecniche di modellazione del software iterative. Pone in particolare risalto la *fase di OOA*.
- **Booch:** metodo basato su tecniche di modellazione del software iterative. Pone in particolare risalto la *fase di OOD*.
- **Fusion:** metodo sviluppato dalla HP a metà degli anni novanta. Rappresenta il primo tentativo di standardizzazione per lo sviluppo di software orientato agli oggetti. Si basa sulla fusione dei metodi OMT e Booch.

Il metodo che studieremo incorpora alcune delle caratteristiche dei metodi elencati qua sopra. (NB *OOD == Object Oriented Design per la fase di progettazione, per ora ci concentriamo su OOA*).

I metodi da tenere in considerazione per l'introduzione di **UML Objectory** (che ha come principale caratteristica quella di basarsi su un particolare diagramma fornito dal linguaggio di modellazione UML, **il diagramma dei casi d'uso** (scenari)),

**OMT Object Modeling Technique** (tecniche di modellazione iterative, focalizzato principalmente sulla parte di analisi OOA) e **Booch** (simile a OMT ma focalizzato sulla parte di design OOD).

Gli elementi su cui si basano questi metodi (classi, oggetti...) sono sempre gli stessi ma ognuno di essi aveva il proprio linguaggio di modellazione.

È stato quindi proposto un linguaggio standard **UML** (Unified Modeling Language) per la descrizione dei sistemi software la cui origine è molto legata ai tre metodi citati sopra in quanto realizzato dai tre rispettivi autori.

**UML** si compone di 9 formalismi di base (diagrammi) e di un insieme di estensioni (“dialetti” come già visto per PetriNet, ma a differenza di PN è stato introdotto in UML un meccanismo standard di estensioni per evitare che fossero svincolate e che si alterasse la struttura del modello standard)

**UML** non è un metodo né un processo di OOA in quanto non dice come specificare il software (non definisce delle regole), è solo una notazione/linguaggio che non vincola la scelta del metodo.

Si è tentato di creare anche un metodo standard OOA ma fallimentare.

Vediamo ora i 9 formalismi di base UML.

1- **Use Case Diagram** == evidenziano la modalità (caso d'uso) con cui gli utenti (attori) utilizzano il sistema. Molto flessibile in quanto utilizzabile per definizione requisiti utente, specifica e anche progettazione (vari livelli di astrazione)

2- **Class Diagram** == diagramma strutturale che permette di descrivere le classi (insieme di informazioni che il sistema deve gestire)

3- **State Diagram** == simil diagramma a stati finiti, permette di descrivere lo stato di una certa entità

4- **Activity Diagram** == rappresentano modelli di flusso di attività (work-flow) che devono essere eseguite, utilizzate per il **modello comportamentale**

5- **Sequence Diagram** e 6- **Collaboration Diagram** == sono definiti come diagrammi di interazione. Un programma object oriented è un insieme di oggetti creati dalle relative classi che interagiscono tra loro attraverso l'invocazione di metodi.

Queste interazioni tra oggetti possono essere descritte usando i diagrammi di interazioni. Entrambi i diagrammi hanno lo stesso potere espressivo ma hanno costrutti diversi dal punto di vista visuale (i sequence più adatti per la parte di analisi orientata a oggetti, i collaboration più adatti per OOD. La loro semantica è identica quindi è possibile passare dal primo al secondo in modo semi-automatico)

7- **Object Diagram** == legato al class diagram, permette di rappresentare gli oggetti e le relazioni tra essi nell'ambito di uno specifico caso d'uso

8- **Component Diagram** e 9- **Deployment Diagram** == diagrammi di implementazione che si soffermano più sulla parte di progettazione (il primo evidenzia le dipendenze esistenti tra componenti software dove componente software == eseguibile mentre il secondo rappresenta la configurazione della piattaforma hardware di esecuzione)

**Come detto i metodi OOA sono iterativi, e il punto di partenza cambia in base al modello.**

*Nel nostro caso partiamo dal Modello dei Dati*, che abbiamo detto soffermarsi sulle informazioni che devono esser gestite dal sistema da un punto di vista statico e strutturale. Per il **modello di dati si utilizza il Class Diagram che permette di definire classi, attributi, operazioni e associazioni tra classi**.

Modello dei dati fondamentale perché nell'approccio a oggetti un sistema software è costituito da un insieme di oggetti che “collaborano”.

**Come già detto il modello dei dati è costruito in modo iterativo e incrementale.**

Durante la prima iterazione, conviene concentrarsi solo sulle cosiddette **entity classes**, ossia *le classi rilevanti per il dominio applicativo su cui lavorerà il software*.

Nel class diagram infatti dovrò inserire *non solo le classi che descrivono informazioni, ma anche quelle che si occupano di gestire la logica di esecuzione del software (control classes) e che rappresentano l'interfaccia utente (boundary classes)*, ma su queste informazioni *ci si concentrerà successivamente usando le informazioni del modello comportamentale*.

Inoltre per ognuna delle entity classes inizialmente ci si concentra sugli attributi e sulle associazioni, le operazioni saranno suggerite ancora una volta dal modello comportamentale.

**Lez 12** (13/11)

**Per costruire come detto il Class Diagram vediamo alcuni approcci utili per l'identificazione delle classi.**

- **Approccio Noun Phrase**: *una frase nominale è una frase dove il sostantivo ha prevalenza sulla parte verbale* (i requisiti utente sono espressi con frasi assertive, di questo tipo, “il prodotto software deve fornire...”).

**Nel primo step quindi si prendono tutte le frasi nominali dei requisiti utente, e ogni sostantivo diviene una classe candidata** (meccanico).

Nel secondo step invece entra in gioco l'esperienza dell'analista, **per ognuna delle classi candidate bisogna capire se si tratta di classe Rilevante** (evidenzia caratteristiche di entity classes e quindi può farne parte), **Irrilevante** o **Fuzzy** (non si hanno sufficienti informazioni per stabilire se una classe è rilevante o irrilevante, si analizzerà successivamente)

- **Approccio Common Class Patterns**: **basato sulla teoria di classificazione**. Invece di guardare il documento dei requisiti utente, con la sola conoscenza del dominio applicativo, **si cerca di identificare un certo insieme di classi a partire da gruppi predefiniti**: **Concept** (per esempio in un software per prenotazione biglietti aerei reservation), **Events** (es. arrival), **Organization** (es. AirCompany), **People** (es. passenger), **Places** (es. TravelOffice).

*Non si tratta di un approccio sistematico come il noun phrase ma può rappresentare un'utile guida, non si concentra sul documento dei requisiti utente e può causare problemi d'interpretazione dei nomi delle classi* (es. arrival arrivo in pista o al terminale..)

- **Approccio Use Case Driven**: i requisiti utente piuttosto che esser espressi come

**frase nominale sono definiti facendo uso di Use Case Diagram** (dove caso d'uso == una sorta di scenario di funzionamento del software). Il diagramma di caso d'uso permette di specificare solo il nome del caso d'uso, *per descrivere cosa succede in quello scenario di funzionamento viene fornita per ogni Use Case Diagram anche una descrizione testuale*.

Questo approccio è simile all'approccio noun-phrase in quanto considera l'insieme di use case come insieme dei requisiti utente. Si assume inoltre che l'insieme degli use case sia completo e corretto.

- Approccio **CRC Class Responsibility Collaboration**: è basato su riunioni in cui si fa uso di apposite card.

CLASS
Elevator Controller
RESPONSIBILITY
1. Turn on elevator button 2. Turn off elevator button 3. Turn on fl oor button 4. Turn off fl oor button 5. Open elevator doors 6. Close elevator doors 7. Move elevator one fl oor up 8. Move elevator one fl oor down
COLLABORATION
1. Class Elevator Button 2. Class Floor Button 3. Class Elevator

*Ogni carta rappresenta un'apposita classe ed è divisa in tre parti: sopra nome, al centro responsabilità (cosa fa) e sotto le classi con cui si relaziona.*

La classe sopra è il controller dell'ascensore che si occupa di molte funzioni, che per essere soddisfatte necessita collaborazione con altre tre classi. *La collaborazione poi ovviamente non si realizza al livello di classe ma a livello di oggetti.*

**Questo approccio è utile per la Verifica di Classi** (capiere se classi identificate usando altri approcci sono effettivamente necessarie o anche se me ne manca qualcuna) e per **Identificazione di attributi e operazioni di ciascuna classe** (infatti responsabilità come metodi futuri da implementare e collaborazioni come dichiarazione di attributi ambo le classi) -> **in generale questo approccio è più utile quando le classi sono già state identificate**

- **Approccio Mixed**: non rappresenta un nuovo approccio ma una fusione dei precedenti. Un possibile scenario: **1) l'insieme delle classi è identificato in base all'esperienza dell'analista facendosi guidare dall'approccio common class patterns** **2) altre classi possono essere aggiunte usando l'approccio noun phrase e lo use case driven** (se gli use case diagram sono disponibili) **3) infine l'approccio CRC verifica**

Vediamo ora alcune linee guida per l'identificazione di entity classes:

- 1) **essa deve avere uno specifico obiettivo, statement of purpose**
- 2) A partire da essa si deve prevedere di poter istanziare più oggetti. **Le cosiddette singleton classes (per cui si prevede una sola istanza) non sono di norma classificabili come entity classes.**
- 3) **per ogni classe si prevede un insieme di attributi (non un solo attributo)**
- 4) **distinguere tra elementi che possono esser modellati come classi o attributi** (qui importante l'esperienza dell'analista)
- 5) **si prevede per ogni classe un insieme di operazioni** (anche se inizialmente trascurate, derivabili successivamente dallo statement of purpose della classe)

Vediamo ora degli esempi, casi di studio, per approfondire questa roba.

Vedremo per software di **gestione iscrizioni università** (dominio che conosciamo), **video noleggio** (gestionale, tipicamente frontend per relazione tra cliente e commesso), **contact management** (applicativi che supportano attività di backend) e **telemarketing** (per la gestione di concorrenza).

## A. University Enrolment

### Problem statement

- The **university** offers
  - Undergraduate and postgraduate degrees
  - To full-time and part-time students
- The **university structure**
  - Divisions containing departments
  - Single division administers each degree
  - Degree may include courses from other divisions
- **University enrolment system**
  - Individually tailored programs of study
  - Prerequisite courses
  - Compulsory courses
  - Restrictions
    - Timetable clashes
    - Maximum class sizes, etc.

Riguardo la struttura si fa riferimento quando ancora non c'era la macroaera ma era diviso in facoltà per didattica e dipartimento per ricerca. Division == facoltà contiene i dipartimenti, ogni corso di studio è associato a una singola facoltà e un corso di studio potrebbe includere materie di altre facoltà.

Il software serve alla gestione dei programmi di studio per ogni studente con corsi propedeutici, obbligatori e in termini di vincoli come conflitti di orari e dimensione massima delle classi.

È poi richiesto (vedi sotto) che il sistema supporti pre-iscrizioni e iscrizioni, inviando mail con istruzioni per iscriversi e i voti presi nell'ultimo semestre se ci si iscrive dal secondo anno in poi.

Al momento dell'iscrizione il sistema deve accettare il piano di studi e convalidarla (alcuni aspetti saranno convalidati automaticamente, altri no)

- The system is required to
  - Assist in pre-enrolment activities
  - Handle the enrolment procedures
- **Pre-enrolment activities** 
  - Mail-outs of
    - Last semester's examination grades to students
    - Enrolment instructions
- **During enrolment**
  - Accept students' proposed programs of study
  - Validate for prerequisites, timetable clashes, class sizes, special approvals, etc.
- Resolutions to some of the problems may require consultation with academic advisers or academics in charge of course offerings

Nel Video Store ogni cassetta e disco e ogni cliente è identificato da un codice a barre. I clienti possono prenotare video da prendere in una data specifica e il software deve aiutare l'impiegato a rispondere alle domande del cliente (es. se ha un film). Software di tipo gestionale (rapporto con clienti frontend)

## B. Video Store Problem statement

- The **video store**
  - Rentals of video tapes and disks to customers
  - All video tapes and disks bar-coded
  - Customer membership also be bar-coded.
- Existing customers can place reservations on videos to be collected at specific date
- Answering customer enquiries, including enquiries about movies that the video store does not stock (but may order on request)

Contact Management è una componente importante per sistemi di gestione aziendale chiamati ERP (enterprise resource planning) per la gestione e automatizzazione di attività di backend di un'organizzazione.

Componenti tipici: package che gestisce contabilità, produzione...

Il Contact Management CMS è package che gestisce le risorse umane.

## C. Contact Management Problem statement

- The market research company with established customer base of organizations that buy market analysis reports
- The company is constantly on the search for new customers
- **Contact management** system
  - Prospective customers
  - Actual customers
  - Past customers
- The new contact management system to be developed internally and be available to all employees in the company, but with varying levels of access
  - Employees of Customer Services Department will take the ownership of the system
- The system to permit flexible scheduling and re-scheduling of contact-related activities so that the employees can successfully collaborate to win new customers and foster existing relationships

Il CMS serve all'azienda, che si occupa di ricerche di mercato, per gestire i rapporti con i clienti ossia con tutte le organizzazioni che comprano i prodotti (ricerche di mercato) dell'azienda.

Tramite il software l'azienda può interagire con clienti passati, presenti e potenziali. Il sistema deve essere disponibile a tutti gli impiegati ma con diversi livelli di accesso. Il sistema gestisce lo scheduling di attività legate al contatto clienti.

## **D. Telemarketing**

### Problem statement

- The **charitable society** sells lottery tickets to raise funds
  - **Campaigns** to support currently important charitable causes
  - Past contributors (**supporters**) targeted through telemarketing and/or direct mail-outs
- Rewards (special bonus campaigns)
  - For bulk buying
  - For attracting new contributors
- The society does not randomly target potential supporters by using telephone directories or similar means

Si ha infine il caso di telemarketing. In questo caso si guarda un'azienda che vuole vendere biglietti della lotteria per beneficenza.

Questi sistemi di telemarketing si basano sulla capacità del database di gestire la coda delle chiamate. Per incentivare si chiamano i vecchi supporter, si danno dei bonus se si comprano tanti biglietti o se si diventa nuovi supporter.

La compagnia non utilizza le pagine gialle per chiamare (random) ma solo potenziali contributori.

#### • **Telemarketing application**

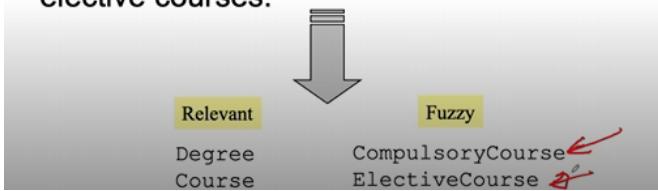
- To support up to fifty telemarketers working simultaneously
- To schedule the phone calls according to pre-specified priorities and other known constraints
- To dial up the scheduled phone calls
- To re-schedule unsuccessful connections
- To arrange other telephone callbacks to supporters
- To records the conversation outcomes, including ticket orders and any changes to supporter records

Vediamo ora la prima applicazione per l'University Enrolment (*A.1 == prima iterazione*). Consideriamo il seguente requisito e identifichiamo usando noun phrase le classi candidate:

- Ogni corso di studi ha un certo numero di corsi obbligatori e un certo numero di corsi opzionali.

I sostantivi di questa frase si ha corso di studio, numero e corsi. Corso di studio e Corso Rilevanti, infatti il sistema ne dovrà mantenere informazione. Quindi sicuramente faranno parte del class diagram. Number d'altro canto è irrilevante, concetto generico.

- Each university degree has a number of compulsory courses and a number of elective courses.



Gli altri due concetti sono corsobbligatorio e corsopzionale, che sicuramente sono informazioni da mantenere nel nostro sistema. Tuttavia per memorizzare queste informazioni è davvero necessario introdurre classi a parte o mi basta usare un attributo tipo? Allora per adesso lascio in Fuzzy.

- More requirements:
  - Each course is at a given level and has a credit-point value
  - A course can be part of any number of degrees
  - Each degree specifies minimum total credit points value required for degree completion
  - Students may combine course offerings into programs of study suited to their individual needs and leading to the degree in which enrolled

Nella seconda si vuole sapere un certo corso in quali corsi di studio è incluso, questa è una **tipica associazione** (nel class diagram attributi, classi e associazioni tra classi). **Vedremo come le associazioni sono implementate come attributi.**

Ultimo requisito: studente rilevante, course offering (corso erogato in diverse istante durante l'anno a diversi canali) fuzzy perché potrebbe essere interessante avere informazioni sull'"offerta" di un certo corso, programs of study rilevante.

Quanto viene detto dall'analista è leggermente diverso:

Relevant classes	Fuzzy classes
Course	CompulsoryCourse
Degree	ElectiveCourse
Student	StudyProgram
CourseOffering	

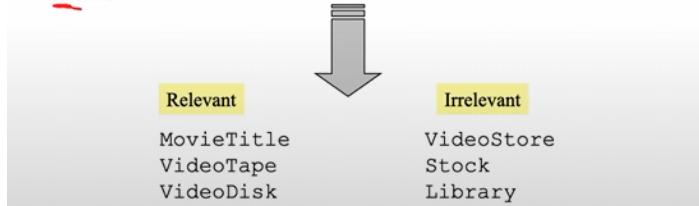
StudyProgram in fuzzy perché l'obiettivo è comunque minimizzare il numero di classi e probabilmente posso gestire l'informazione in un altro modo.

Passiamo ora a B.1, prima iterazione del caso di studio Video Store. Primo Requisito:

- Il video store mantiene in stock un'ampia libreria di film popolari. Un film può esser mantenuto come disco o cassetta.

Video store, stock, library, movies, tapes e disks sostantivi

- The video store keeps in stock an extensive library of current and popular movie titles. A particular movie may be held on video tapes or disks.



*Videostore irrilevante perché le linee guida dicevano che per identificare una classe è necessario identificare un certo insieme di istanze, e poiché il software è dedicato a un singolo negozietto e non a una catena non ha senso identificare la classe.*

### Lez 13 (16/11)

- More requirements: *Videostore*
- Video tapes are in either "Beta" or "VHS" format
- Video disks are in DVD format
- Each movie has a particular rental period (expressed in days), with a rental charge to that period
- The video store must be able to immediately answer any inquiries about a movie's stock availability and how many tapes and/or disks are available for rental
- The current condition of each tape and disk must be known and recorded

<b>Relevant classes</b>	<b>Fuzzy classes</b>
MovieTitle	RentalConditions
VideoMedium	
VideoTape	
VideoDisk (or DVDDisk)	
BetaTape	
VHSTape	

Betatape, VHSTape non come attributi ma come classi dal momento che evidentemente necessito di stabilirne diverse proprietà. Videomedium scelta dell'analista di creare una classe dove inserire tutte le proprietà comuni sia a dischi che a cassette. RentalConditions come fuzzy perché il prestito potrebbe esser gestito sia come classe che come attributi, ancora da decidere.

Passiamo alla prima iterazione Contact Management:

### Example C.1 – Contact Management

- Consider the following requirements for the Contact Management system and identify the candidate classes:
  - To "keep in touch" with current and prospective customer base
  - To store the names, phone numbers, postal and courier addresses, etc. of organizations and contact persons in these organizations
  - To schedule tasks and events for the employees with regard to relevant contact persons
  - Employees can schedule tasks and events for other employees or for themselves
  - A task is a group of events that take place to achieve a result (e.g. to solve customer's problem)
  - Typical types of events are: phone call, visit, sending a fax, arranging for training, etc.

Relevant classes	Fuzzy classes
Organization	CurrentOrg
Contact	ProspectiveOrg
Employee	PostalAddress
Task	CourierAddress
Event	

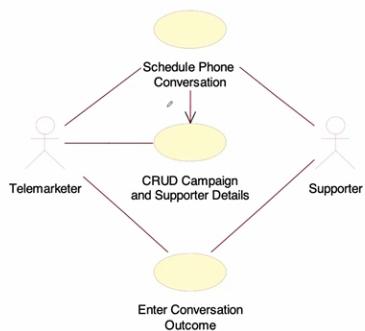
L'idea è di specificare il concetto di cliente (generico) usando le classi più specifiche *Organization* e *Contact*. Similmente a quanto visto con l'uni per corsi obbligatori e non, qua si parla di clienti potenziali o attuali che potrei implementare come classi o con attributi -> fuzzy.

L'ultimo requisito sembrava evidente dover essere implementato tramite l'uso di attributi, ma perché allora *postaladdress* e *courieraddress* in fuzzy? Potrei volerle rendere classi in base all'uso del software: attributi se me la cavo a specificare l'indirizzo come semplice stringa, altrimenti potrebbe essermi utile implementarla con tanti campi diversi.

Vediamo adesso la prima iterazione per il Telemarketing dove invece sfruttiamo l'approccio **Use Case Driven**:

### Example D.1 – Telemarketing

*Business use case diagram*



Si parla di **Business use case diagram** dal momento che il *livello di astrazione è molto alto*. Come già detto in precedenza l'use case diagram mi permette di specificare i possibili scenari di utilizzo del software. **Le ellissi rappresentano la primitiva di caso d'uso**, poi si ha **l'omino che rappresenta la primitiva attore**, ossia colui (o qualcosa, non è detto che sia un essere umano) con cui il software interagisce.

In questo caso si hanno due attori: telemarketer (che chiama) e supporter (colui che eventualmente compra biglietti della lotteria).

In particolare in questo diagramma il telemarketer rappresenta *l'utente di questo diagramma in quanto come vedremo è colui che attiva i 3 casi d'uso*, mentre il *supporter è un attore che può essere coinvolto durante la loro esecuzione*.

Come avevamo specificato inizialmente i telemarketer effettuano le chiamate e alla fine di esse devono registrare i risultati della chiamata -> i due casi d'uso principali sono quello in alto e in basso. Il primo si attiva quando deve fare la chiamata al supporter, il secondo a fine chiamata.

Si ha poi un caso d'uso nel mezzo: CRUD Dettagli Campagna e Supporter.

**CRUD == Create, Read, Update, Delete** (*le 4 operazioni base che si possono fare con i dati*), questo caso d'uso mi serve quando devo compiere una di queste operazioni per i dati relativi sia la campagna che il supporter (es. se il supporter acquista 3 biglietti devo segnare indirizzo, numero etc... e devo aggiornare il fatto che 3 biglietti sono stati venduti).

I casi d'uso e gli attori sono connessi da delle linee che rappresentano il coinvolgimento dell'attore o l'attivazione da parte dell'attore di quel caso d'uso.

Quando scenderemo a livello d'astrazione specifica vedremo come dovremo introdurre le frecce e non sarà sufficiente l'uso generico di linee.

Appare poi il collegamento tra i primi due casi d'uso (**NB dovrebbe essere freccia tratteggiata**) e *indica la relazione generica tra due elementi UML*.

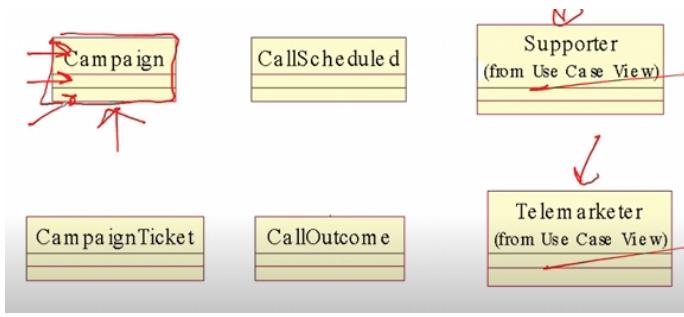
*In questo caso rappresenta il fatto che mentre faccio una chiamata può nascere l'esigenza di aggiornare i dati (CRUD).*

Come avevamo anticipato non è effettivamente scritto nel diagramma cosa deve fare il software una volta attivato il caso d'uso. Per questa ragione affianco al diagramma per ogni caso d'uso specificato va inclusa una nota testuale in linguaggio naturale che lo specifichi.

**Ora, dal punto di vista della ricerca di entity classes, tutti gli attori diventano automaticamente delle classi entity.** Oltre a questo devo far riferimento alla notazione testuale:

- Consider the following textual description for the Telemarketing system's use cases and identify the candidate classes:
  - The telemarketer requests the system that the phone call to a supporter be scheduled and dialed up
  - Upon successful connection, the telemarketer offers lottery tickets to the supporter. During a conversation, the telemarketer may need to access and modify both campaign and supporter details (CRUD, create – read – update – delete)
  - Finally, the telemarketer enters the conversation outcome, i.e. the successful or unsuccessful results of the telemarketing action

Si procede come nel noun phrase evidenziando i sostantivi e ragionandoci sopra.



Per la prima volta vediamo la soluzione presentata con il **formalismo del class diagram** (nome, attributi, operazioni, gli ultimi due per ora vuoti).

CallScheduled == Phone Call

Prima di passare alla seconda iterazione vediamo delle **linee guida fornite dal metodo per la specifica delle classi**.

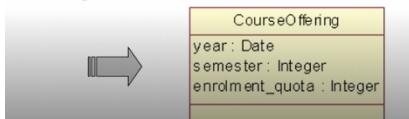
Ad ogni classe deve essere associato un Nome significativo, che abbia una lunghezza massima e che adotti una convenzione standard (di modo che già solo dalla sintassi del nome capisca se è classe o attributo, es. come abbiamo visto parole multiple congiunte, nome singolare e ciascuna iniziale maiuscola come CallOutcome).

*Riguardo gli attributi invece devono ovviamente anch'essi adottare una convenzione standard* (nel nostro caso minuscolo, snake case tipo street\_name).

*Riguardo le operazioni queste conviene aggiungerle solo a partire dalla presenza di un modello comportamentale.*

#### Example A2 - University Enrolment

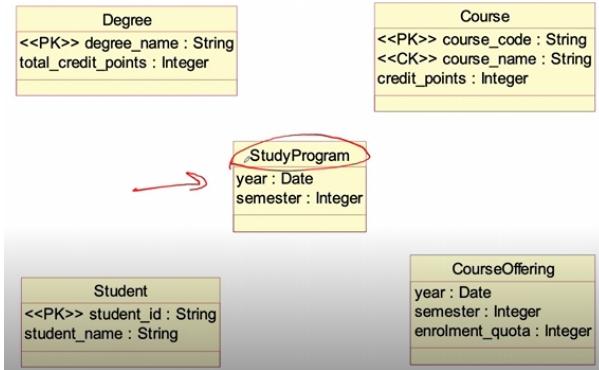
- Refer to Example A.1
- Consider the following additional requirements from the Requirements Document:
  - A student's choice of courses may be restricted by timetable clashes and by limitations on the number of students who can be enrolled in the current course offering.



Si passa alla seconda iterazione dell'università. Poiché oltre che iterativo anche incrementale, mi riferisco alla prima iterazione per andare avanti.

Per questo requisito mi è sufficiente introdurre gli attributi per CourseOffering già definito in precedenza dove enrolment\_quota == numero max di studenti

- More requirements:
  - A student's proposed program of study is entered in the on-line enrolment system
  - The system checks the program's consistency and reports any problems
  - The problems need to be resolved with the help of an academic adviser
  - The final program of study is subject to academic approval by the delegate of the Head of Division and it is then forwarded to the Registrar



Vi è, visti i nuovi requisiti, la necessità di introdurre **StudyProgram** come classe (prima era rimasta fuzzy) e degli attributi per ogni classe. *Questi attributi non fanno riferimento solo ai requisiti appena letti, ma anche a quelli della prima iterazione* (es. per ogni corso era stato specificato di dover conoscere i crediti -> `credit_points` in **Course**). **CumpolsoryCourse** e **ElectiveCourse** stavano in Fuzzy ma non riappaiono -> conveniva toglierli.

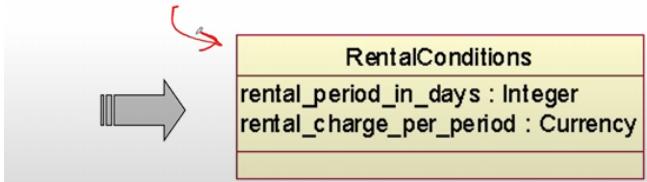
Ci si accorge inoltre che come prefisso di alcuni attributi vi **sono <<PK>> Primary Key** e **<<CK>> Candidate Key** dove i simboli <>> sono chiamati **Stereotipi in UML**.

**Una caratteristica importante in UML abbiamo detto essere la possibilità di estensione standard creando un Profilo** (es. estendo UML nel settore finanza con il profilo Finanza dove creo primitive UML specifiche per quel dominio).

Un profilo rappresenta null'altro che un insieme di stereotipi.

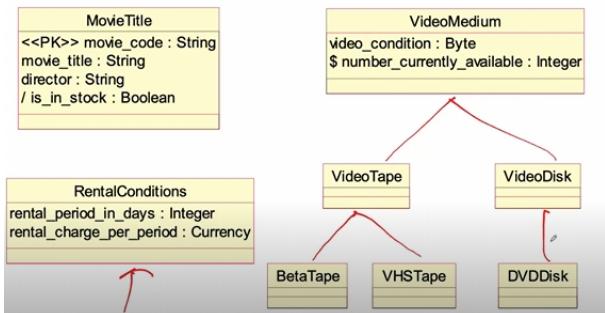
**In questo caso specifico si sta dicendo di voler introdurre un Profilo DBMS** (orientato quindi alla traduzione delle classi in tabelle) in cui si annotano gli attributi della classe con caratteristiche tipiche della progettazione database.

- Refer to Example B.1
- The additional requirements are:
  - The rental charge differs depending on video medium: tape or disk (but it is the same for the two categories of tapes: Beta and VHS).



- More requirements:
  - The system should accommodate future video storage formats in addition to VHS tapes, Beta tapes and **DVD disks**
  - The employees frequently use a **movie code**, instead of **movie title**, to identify the movie
  - The same movie title may have more than one release by different directors

Chiaramente movie code e title sono attributi della classe movie, e anche director.



Capiamo già da come sono posizionate le classi che esiste una sorta di gerarchia che verrà introdotta e sarà chiara più avanti. *Inoltre il requisito per cui si potrebbero dover introdurre nuove tecnologie per cassette e dvd è soddisfatto introducendo videodisk e videotape come “generalizzazioni”.*

Is\_in\_stock deriva da un requisito della prima iterazione: il software deve essere utilizzato per rispondere alla query del cliente se un certo film è disponibile o meno. / **is\_in\_stock ha il simbolo /** che è standard UML e rappresenta **derived attribute**: **il valore dell'attributo non è inserito dall'utente ma derivato runtime** (è il sistema in grado di capire se c'è il film a disposizione o meno).

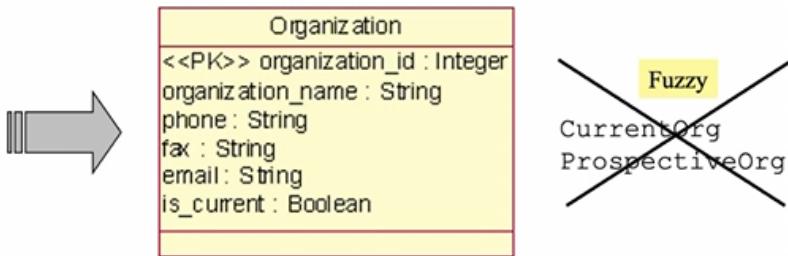
L'altra informazione di cui avevamo bisogno era il numero di videomedium disponibili per il noleggio, ciò si applica sia alle cassette che ai dvd quindi la metto nella classe più generica (ereditarietà).

\$ number\_currently\_available NON indica il numero di cassette/dischi che sono disponibili per un certo film. Infatti **\$** è un **altro simbolo standard UML** e rappresenta un **attributo statico**: **singolo valore condiviso tra tutti gli oggetti che creo a partire da quella classe**.

L'attributo in realtà indica il numero totale di videomedium: incrementato quando ne creo uno e decrementato quando non è più disponibile.

*Per soddisfare la richiesta del numero disponibile per uno specifico film devo utilizzare le associazioni tra classe Film e VideoMedium (lo vedremo poi)*

- Refer to Example C.1 and consider the following additional information
  - A customer is considered current if there exists a ~~contract~~ with that customer for delivery of our products or services. Contract management is, however, outside the scope of our system.

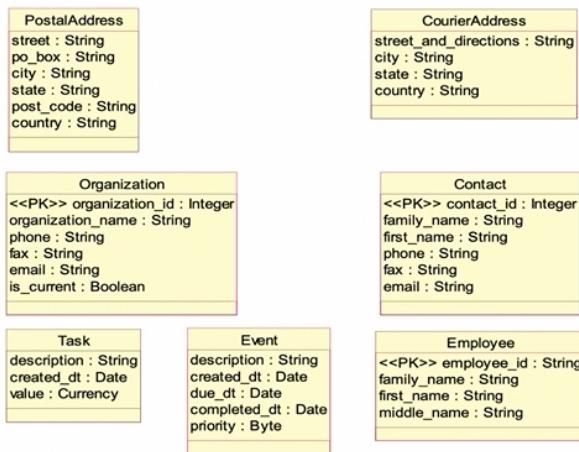


Tornando all'esempio di Contact Management viene detto che un cliente attuale è tale se esiste un contratto, ma che il management del contratto non riguarda il mio sistema (esiste un sistema dedicato a quello) -> contract non è classe importante per il mio software, per stabilire se è attuale aggiungo l'attributo is\_current a organization.

- More requirements:
  - Reports on contacts based on postal and courier addresses (e.g. find all customers by post code)
  - Date and time of the task creation are recorded
  - The "money value" of a task can be stored
  - Events for the employee are displayed on the employee's screen in the calendar-like pages (one day per page).
    - The priority of each event (low, medium or high) is visually distinguished on the screen
  - Not all events have a "due time" - some are "untimed"
  - Event creation time cannot be changed, but the due time can.
  - Event completion date and time are recorded
  - The system stores identifications of employees who created tasks and events, who are scheduled to do the event ("due employee"), and who completed the event

Forse conviene utilizzare informazione strutturata per l'indirizzo creando una classe a parte visto che si vogliono cercare i clienti per post code (primo requisito). L'ultimo requisito sarà importante per la terza iterazione in quanto si vogliono identificare gli impiegati che hanno creato task e eventi, a cui sono stati assegnati e che li hanno completati.

## Lez 14 (20/11)



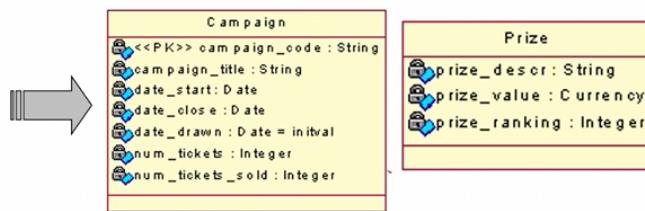
PostalAddress e CourierAddress introdotti come classi a parte come detto, task ed evento hanno attributi created\_dt, due\_dt e completed\_dt di tipo **Date** e value per task di tipo **Currency** (*tipi di dato base in UML*).

Per definire la priorità degli eventi è presente l'attributo priority di tipo Byte (si voleva low medium o high, un boolean non bastava quindi uso byte)

Ci resta da vedere la seconda iterazione per il Telemarketing.

Guardando i nuovi requisiti è conveniente inserire una classe "Prize" per indicare cosa vince chi vince la lotteria.

- Refer to Example D.1
- Consider the following additional information
  - Each campaign
    - Has a **title** that is generally used for referring to it
    - Has also a **unique code** for internal reference
    - Runs over a **fixed period of time**
  - Soon after the campaign is closed, the **prizes** are drawn and the **holders** of winning tickets are advised

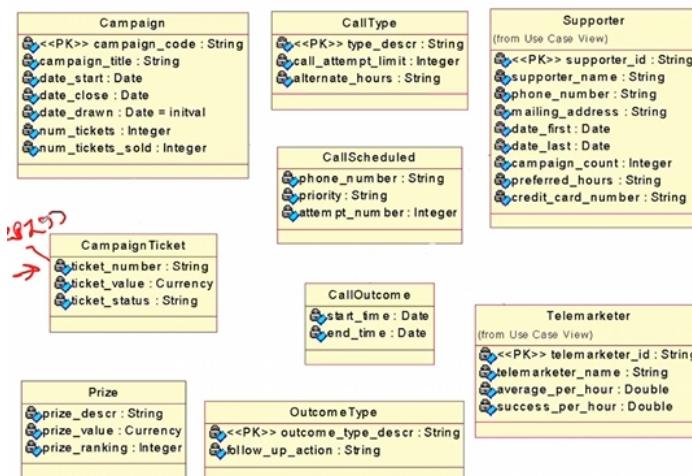


- More requirements:
  - Tickets are uniquely numbered within each campaign
  - The total **number** of tickets in a campaign, **number** of tickets sold **so far**, and the **current status** of each ticket are known (e.g. available, ordered, paid for, prize winner)
  - To determine the **performance** of the society's telemarketers, the **duration of calls** and the **successful call outcomes** (i.e. resulting in ordered tickets) are recorded
  - Extensive information about **supporters** is maintained
    - Contact **details** (address, phone number, etc.)
    - Historical **details** such as the first and most recent dates when a supporter had participated in a campaign
    - Any known supporter's preferences and constraints (e.g. times not to call, usual credit card number)

Per lavorare sul terzo requisito devo lavorare sull'associazione tra telemarketer, chiamata da fare e risultato chiamata (lo faccio per ogni chiamata di un certo telemarketer e capisco così le sue prestazioni).

- More requirements:

- Telemarketing calls are made according to their priorities
- Calls which are unanswered or where an answering machine was found, are rescheduled
  - Times of repeat calls are alternated
  - Number of repeat calls is limited
    - Limits may be different for different call types (e.g. a normal "solicitation" call may have different limit than a call to remind a supporter of an outstanding payment)
- Call outcomes are categorized - success (i.e. tickets ordered), no success, call back later, no answer, engaged, answering machine, fax machine, wrong number, disconnected.



average\_per\_hour e success\_per\_hour sono gli attributi di telemarketer che ne indicano le prestazioni di cui parlavamo prima, ed il valore di essi dipende dall'associazione che esiste tra Telemarketer, CallScheduled e CallOutcome.

Una cosa che probabilmente manca in CallScheduled è la durata della chiamata, fattore che serve per determinare le performance del telemarketer.

Un'osservazione è da fare sulle classi CallOutcome e OutcomeType. Si era ipotizzato di inserire OutcomeType come attributo di CallOutcome, invece è stata creata una classe a parte con un occhio di riguardo proprio all'implementazione.

Sapendo infatti che all'ora vengono fatte ad esempio 500 chiamate tra tutti i telemarketer, la seconda soluzione permette di utilizzare molta meno memoria.

Mettendo gli attributi in una sola classe avrei dovuto memorizzare per ogni CallScheduled 5 attributi, con due classi distinte invece di oggetti OutcomeType non ne creo 500 all'ora, ma solo un numero pari al numero possibile di outcome.

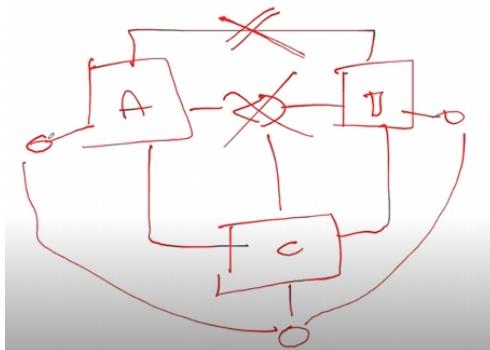
Ora ognuna delle 500 CallOutcome sarà associata al rispettivo OutcomeType -> di outcometype ne ho tipo 15, ho risparmiato un sacco.

**Con la prima iterazione abbiamo identificato le prime entity classes, con la seconda definito eventualmente ulteriori classi e gli attributi, con la terza**

definiamo le associazioni per finire la definizione del modello di dati prima di passare al comportamentale.

Quando si incontra un attributo che ha come tipo di dato una classe, quell'attributo avrà come valore un riferimento a un oggetto di quella classe.  
Quindi in generale un attributo rappresenta **un'associazione** se sono identificati con tipo di dato non elementare ma classe.

Quando si ha un'associazione ternaria questa può essere rimpiazzata con un ciclo di associazioni binarie, rimuovendo una delle associazioni in quanto potrò ad esempio raggiungere la classe A da B in modo transitivo passando da C (risparmio spazio ma perdo in termini di efficienza).



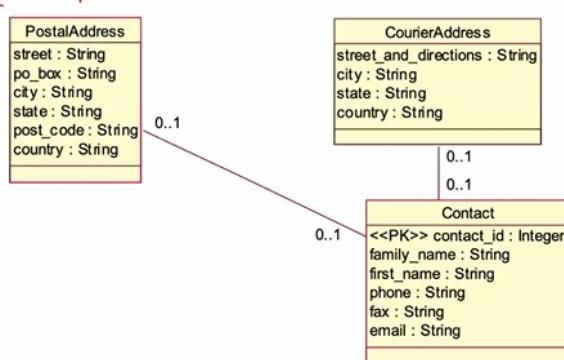
Così come detto per classi e attributi, **anche le associazioni devono avere nomi significativi e devono usare ovviamente la stessa convenzione usata per gli attributi.** Inoltre **è fondamentale esprimere la molteplicità delle associazioni, che come negli schemi ER deve essere espressa ad entrambe le estremità.**

Infine è **importante assegnare i nomi di ruolo (rolename)** alle estremità delle associazioni, *specificano il ruolo giocato dagli oggetti di ciascuna classe che partecipano a quella associazione.*

Stavolta per vedere come si aggiungono le associazioni partiamo dalla terza iterazione di Contact Management.

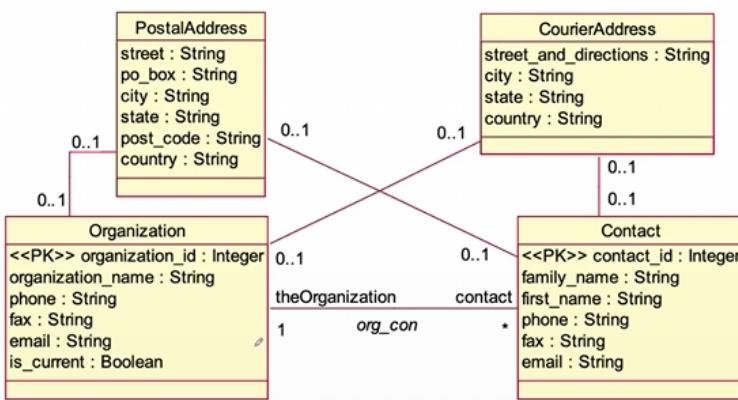
### Example C.3 – Contact Management

- Refer to Examples C.1 and C.2 - specify associations
- Consider, for example, the requirement:
  - The system allows producing various reports on our contacts based on postal and courier addresses



Ad ogni oggetto contatto, per quanto visto con i precedenti requisiti e quello in questo esempio, devo associare un eventuale indirizzo postale e/o un eventuale indirizzo corriere. **La sintassi di UML prevede una linea che collega le classi di interesse, in questo caso non c'è il nome perché è evidente il senso dell'associazione, e sono esplicitate le molteplicità.** **0..1: per ogni indirizzo postale posso avere da 0 a 1 contatti** etc..

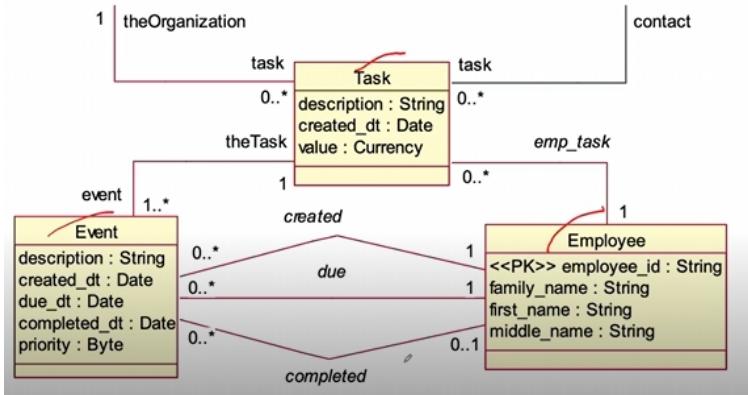
**Vediamo una delle limitazioni di UML se vogliamo porre il vincolo affinché ogni contatto abbia un solo indirizzo tra CourierAddress e PostalAddress (solo uno dei due). Per risolvere la cosa posso farlo informalmente aggiungendo una nota testuale, altrimenti in modo formale si può utilizzare un linguaggio detto **OCL** che permette di associare dei vincoli ai diagrammi UML.**



Qui vediamo l'esempio di un'associazione completamente specificata: abbiamo infatti nella associazione nella partea sotto nome (org\_con), specificità e nome di ruolo. L'associazione mette in relazione Contact e Organization e ha le seguenti molteplicità: *fissato un Contatto questo lavora in una e una sola organizzazione (uno a sx)* mentre *fissata l'organizzazione questa è costituita da almeno 1 ma anche più contatti che vi lavorano (\* a dx, con \* si indica 1:N in UML)*.

Ora vediamo i **role name**, che sono i nomi che esprimono il ruolo con cui gli oggetti di una classe partecipano all'associazione e normalmente sono quelli utilizzati per implementare l'associazione. Per implementarla infatti dovrò inserire nella classe Contatto l'informazione relativa all'organizzazione in cui lavora (quindi aggiungerò a Contact l'attributo theOrganization che è di tipo Organization).

D'altra parte in Organization potrò mettere un attributo contact che indica l'insieme di oggetti Contact che lavorano nell'organizzazione.



Uno dei primi requisiti diceva che un task è un gruppo di eventi -> un evento fa parte di un task e un task ha 1..\* eventi.

Per tener traccia dell'impiegato che ha creato il task: relazione emp\_task dato un task affidato a un solo impiegato, dato un impiegato può aver creato 0..\* task.

Vediamo ora le associazioni che legano la classe impiegato con la classe evento.

L'ultimo requisito richiesto in C2 chiedeva che il sistema memorizzasse gli impiegati che creano task e eventi, a cui è assegnato un evento e che completano l'evento.

Ciò significa che per ogni evento deve esserci un impiegato che crea quell'evento, uno a cui è assegnato e uno che lo completa (possono essere differenti).

Vedendo il modello un Impiegato può aver creato, aver avuto assegnati, aver completato 0..\* eventi.

Fissato evento avrò 1 impiegato che lo ha creato, 1 impiegato a cui è stato assegnato e 0..1 impiegato che l'ha completato (quindi dato evento sicuro qualcuno l'ha creato e a qualcuno è stato assegnato, ma non è detto che sia stato completato).

**Quanto visto riguarda le associazioni standard tra oggetti di classi diverse.**

UML fornisce anche tipi particolari di associazioni, come **l'Aggregazione**.

Aggregazione rappresenta una relazione di tipo contenimento (whole-part) tra una **classe composta** (**superset**, il contenitore) e **una o più classi componenti** (**subset**).

Questo tipo di relazione può assumere 4 significati in base alla “forza” del legame che esiste tra classe contenitore e classe contenuta:

- ExclusiveOwns per cui si hanno le proprietà tra classe contenitore e contenuta di Transitività (se A contiene B e B contiene C allora A contiene C), Asimmetria (se A contiene B B non può contenere A), Existence-dependency (se cancello l'oggetto contenitore elimino anche i contenuti) e Fixed Property (se un'istanza della classe contenuta è messa in relazione con un'istanza della classe contenitore non può esser messa in relazione con nessun'altra istanza contenitore).

Es. associazione tra classe Book e Chapter.

- Owes per cui non si ha la fixed property (Car e Tire, per cui valgono le altre tre proprietà ma non la fixed perché ad esempio potrei spostare la ruota da una macchina all'altra)

- Has per cui *no existence dependency né fixed property* (es. Facoltà e Dipartimento, la facoltà aveva i Dipartimenti. Quando le facoltà sono state cancellate i Dipartimenti sono rimasti)

- **Member** per cui non vi sono proprietà speciali di contenimento eccetto la membership (es. Meeting e Coordinatore, non vi è alcun legame eccetto l'ap0070artenenza al meeting, può essere specificata con una relazione standard)

Vedremo come con UML sia possibile utilizzare due dei suoi costrutti per rappresentare questi 4 possibili significati

## Lez 15 (27/11)

Sappiamo che UML ha soltanto due primitive in termini di relazioni di contenimento:

**Aggregation** (sintassi  $\diamond$ , aggregazione contenimento più debole, corrisponde a

**Has** e **Member**) e **Composition** ( $\blacklozenge$ , corrisponde a **ExclusiveOwns** e **Owns**).

**L'aggregation ha significato per riferimento**, mentre **composition per valore** (ossia oggetti creati a partire da classi in aggregation o composition hanno comportamento cambiato dalla semantica di queste primitive)

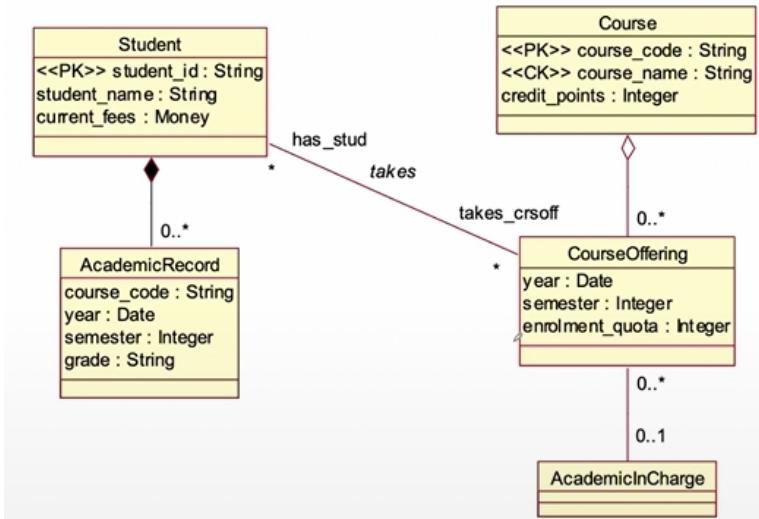
Es. Date *A* e *B* classi dove *B* Composition di *A* allora quando creo i due oggetti da *A* e *B* l'oggetto creato da *A* avrà al proprio interno l'oggetto creato da *B*, e se cancello l'oggetto *A* cancello anche l'oggetto *B* (semantica per valore in quanto l'oggetto contenitore contiene l'oggetto contenuto tra gli attributi).

Stesso scenario ma *A* e *B* legate da Aggregation: semantica per riferimento significa che l'oggetto A avrà tra gli attributi un riferimento all'oggetto B, che vive nel suo spazio di memoria (quindi se cancello l'oggetto A cancello il puntatore ma non B).

Dal punto di vista implementativo quindi un'associazione di tipo aggregation si implementa come associazione standard tra due classi.

## Example A.3 – University Enrolment

- Refer to Examples A.1 and A.2
- Consider the following additional requirements:
  - The student's academic record to be available on demand
  - The record to include information about the student's grades in each course that the student enrolled in (and has not withdrawn without penalty)
  - Each course has one academic in charge of a course, but additional academics may also teach in it
    - There may be a different academic in charge of a course each semester
    - There may be different academics for each course each semester



AcademicRecord come classe per registrare la carriera universitaria dello studente dove ogni volta che lo studente supera l'esame ne creo un nuovo oggetto.

Fissato lo studente, questo può fare da 0 a N esami (0..\*).

Takes è l'associazione tra Student e i corsi che sta seguendo in quel momento (\* sta per molteplicità 1:N). AcademicInCharge sta per docente, fissato il docente questo può insegnare 0..\* corsi “offerti, fissato il corso questo (secondo l'ultimo requisito sopra) in realtà dovrebbe poter essere insegnato anche da più docenti e sicuramente da almeno 1, quindi non 0..1 ma 1.\*.

*Analizziamo ora l'uso di Composition e Aggregation.*

La composition a sinistra indica un legame molto forte tra gli oggetti studente e gli oggetti AcademicRecord, se si cancella dall'archivio uno studente si cancellano anche i dati relativi agli esami da lui sostenuti (dichiarazione forte, ma non è errore).

Riguardo l'aggregation a destra, ci sta dicendo che se cancello ad es. il corso ISW allora non saranno cancellate tutte le istanze del corso negli anni passati (corretto, se si cancellassero si perderebbero formazioni importanti legate anche agli studenti che hanno seguito e sostenuto l'esame).

**Quindi quando un requisito implica il contenimento di una classe in un'altra uso relazioni di associazione di contenimento, da decidere se Aggr. o Composition.**

L'altra associazione molto importante che ha anche portato il successo all'OOP è **l'Ereditarietà (generalizzazione)**. Si parla di generalizzazione in realtà perché in UML non esiste davvero il concetto di ereditarietà ma esiste una primitiva detta **generalizzazione**.

Si tratta di un meccanismo per *rappresentare la condivisione tra attributi e operazioni tra classi, le caratteristiche comuni sono modellate in una classe generica (superclasse) che viene specializzata nell'insieme di sottoclassi che ne ereditano attributi e operazioni.*

Le **caratteristiche della generalizzazione** sono:

- **Sostituibilità**: un oggetto della sottoclasse può essere utilizzato ovunque sia richiesto un oggetto della superclasse e non viceversa.

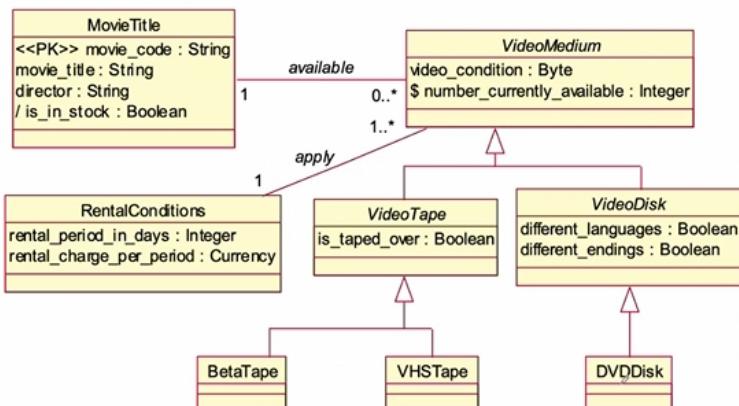
Immaginiamo di avere una classe A specializzata in una sottoclasse B e di creare i rispettivi oggetti a e b. Dire a = b è corretto secondo la sostituibilità in quanto la classe b sa fare le stesse cose che fa a (le eredita) e anche altro in più, mentre b = a non va bene in quanto la classe a non sa fare tutto ciò che fa b quindi se chiedessi a quel punto a b un metodo di B sarebbe la fine per l'umanità.

- **Polimorfismo**: *una stessa operazione può avere implementazioni diverse nelle sottoclassi (una sottoclasse può ridefinire il comportamento di alcuni metodi ereditati in base alle proprie necessità).*

**Per rappresentare la generalizzazione in UML si usa una linea avente freccia diretta verso la superclasse.** Capiamo di doverla usare quando nei requisiti leggiamo tipo “può essere” o “è del tipo di”...

### Example B.3 – Video Store

- Refer to Examples B.1 and B.2
- The classes identified in Example 4.5 imply a generalization hierarchy rooted at the class `VideoMedium`
- Extend the model to include relationships between classes, and specify generalization relationships
- Assume that the Video Store needs to know if a `VideoTape` is a brand new tape or it was already taped over (this can be captured by an attribute `is_taped_over`)
- Assume also that the storage capacity of a `VideoDisk` allows holding multiple versions of the same movie, each in a different language or with different endings



Generalizzazioni piuttosto chiare, relazione tra `MovieTitle` e `VideoMedium` `available` permette di capire quanti e quali supporti video ho legati a un film specifico, si associa in particolare un oggetto `MovieTitle` a `0..*` oggetti `VideoMedium` utilizzabili per noleggiarlo. Se 0 il film non è disponibile per il noleggio -> attributo di `MovieTitle` `is_in_stock` diventa falso.

`RentalConditions`, per ogni `VideoMedium` che viene noleggiato si applica una specifica `RentalCondition`, mentre fissato un oggetto `RentalCondition` questa può essere associata a uno o più `VideoMedium` (più perché più clienti possono scegliere la stessa).

Si noti come il nome della classe `VideoMedium`, `VideoTape`, `VideoDisk` siano in corsivo. Ciò non è casuale, si indicano delle **Classi Astratte** (ossia non si possono creare oggetti a partire da quelle classi, posso creare solo in questo caso `BetaTape`, `VHSTape` e `DVDDisk` unici oggetti effettivamente concreti).

Perché allora non utilizzarle direttamente? Se l'avessi fatto avrei dovuto però specificare l'associazione available con MovieTitle e apply con RentalConditions per ognuna delle tre classi, sarebbe quindi stato corretto sintatticamente ma non avrebbe espresso il significato dietro la generalizzazione per cui quelle associazioni valgono per tutti.

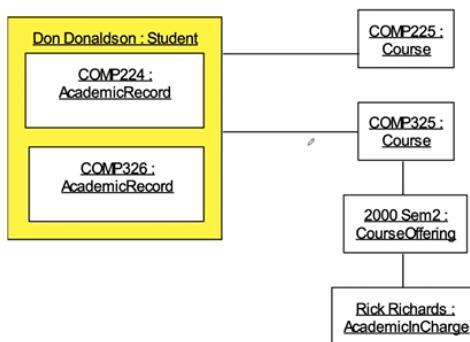
Con questa terza iterazione abbiamo concluso la parte relativa alle associazioni e in generale il primo step sul **Modello dei Dati**. Passiamo al **Modello Comportamentale**, vedremo che lavorare su di esso ci permetterà di tornare al Class Diagram e aggiungere anche le operazioni.

Prima di far ciò vediamo un altro diagramma UML che può essere utile nel Modello dei Dati come complemento del Class Diagram: [\*\*l'Object Diagram\*\*](#).

Esso è per la rappresentazione grafica di istanze di classi ed è usato per **mostrare esplicitamente relazioni complesse tra classi** (es. sopra mostrare la relazione available direttamente tra oggetto MovieTitle e DVDDisk), **illustrare le modifiche ai singoli oggetti durante l'evoluzione del sistema** (ogni oggetto creato da una classe ha uno stato definito dai valori dei suoi attributi, che possono cambiare nel tempo), **illustrare la collaborazione tra oggetti durante l'evoluzione del sistema**.

#### Example A.4 – University Enrolment

- Show an object diagram with few objects representing the classes in Example A.3



Lo studente Don Donaldson ha superato due esami: COMP225 e COMP326, *il fatto che esistesse la relazione di Composition viene esplicitato dal fatto che gli AcademicRecord sono contenuti in Student* (se cancello lo studente cancello anche gli esami che ha fatto). Dall'altra parte *l'associazione tra l'oggetto corso e courseoffering è rappresentata normalmente (semantica di riferimento come visto)*. In realtà questo esempio sarebbe sbagliato in riferimento all'esempio A.3 in quanto in tale diagramma non vi era relazione diretta tra studente e corso, ma solo con CourseOffering.

Si ricorda che stiamo usando una specifica semiformale quindi vogliamo costruire un modello del software che lo rappresenti da tutti i punti di vista: statico (**dati**), funzionale (**comportamentale**) e di controllo (**dinamico**).

*Il modello comportamentale ci dice come le informazioni (fornite dal modello dei dati) vengono elaborate al fine di fornire agli utenti i servizi offerti dal software.*

*Poiché stiamo costruendo il modello dal punto di vista object oriented, alla fine il software sarà popolato da un certo numero di oggetti che collaborano scambiandosi messaggi (**messaggio == metodo che chiede a un altro oggetto di eseguire un metodo**)*

A differenza del **Modello di Dati** in cui si utilizzava principalmente **Class Diagram** e **Object Diagram**, nel Modello Comportamentale si utilizzano:

- **Diagramma di Casi d'Uso** (per descrivere i possibili scenari di funzionamento)
- **Activity Diagram** (per descrivere il flusso di elaborazione)
- **Sequence Diagram** e **Collaboration Diagram** detti **diagrammi d'interazione**

poiché descrivono l'interazione tra oggetti.

*Ancora il modello è costruito in modo incrementale e iterativo, si sfruttano di volta in volta le informazioni del modello dei dati che a sua volta fa uso del modello comportamentale per aggiungere altre classi (**inizialmente ci siamo soffermati sulle entity classes, arriveranno le control (per esecuzione) e boundary (per interfaccia) classes**).*

Iniziamo dall'Use Case Diagram. Già iniziato a vedere per il Modello dei Dati, sappiamo di poterlo usare a diversi livelli di astrazione (sia in fase di OOA che OOD) **Durante OOA si concentra su COSA il sistema deve fare.**

Un caso d'uso rappresenta:

- **una funzionalità completa** (un caso d'uso deve rappresentare **un'intera funzionalità**, comprensiva del **flusso principale** e di eventuali **flussi alternativi o di errore**)
- **una funzionalità visibile dall'esterno**, ossia visualizzabile da un attore/utente
- **comportamento ortogonale** (ogni caso d'uso deve essere indipendente, ciò segue direttamente dal fatto che ogni caso d'uso deve rappresentare funzionalità completa)
- **una funzionalità originata da un attore del sistema** (si deve avere almeno un attore che attiva il caso d'uso, se nessun attore lo attiva allora non è caso d'uso)
- **una funzionalità che produce un risultato significativo per un attore.**

*I casi d'uso vengono identificati dall'insieme di requisiti utente e soffermandosi sugli utenti principali dei casi d'uso: gli attori (e le loro necessità)*

- L'identificazione può essere facilitata facendosi guidare dalle seguenti domande:
  - Quali sono i compiti principali svolti da ciascun attore?
  - Un attore accede o modifica l'informazione nel sistema?
  - L'attore rappresenta il tramite mediante cui il sistema viene informato di modifiche apportate in altri sistemi?
  - L'attore deve essere informato di eventuali cambiamenti avvenuti nel sistema?
- Durante la fase di OOA, i casi d'uso identificano le **necessità degli attori** del sistema

La **sintassi fornita da UML** per la specifica dei casi d'uso prevede graficamente come visto *due primitive principali*: **attore** == omino e **caso d'uso** == ellisse.

Tali elementi sono collegati tra loro tramite relazioni, quattro tipi:

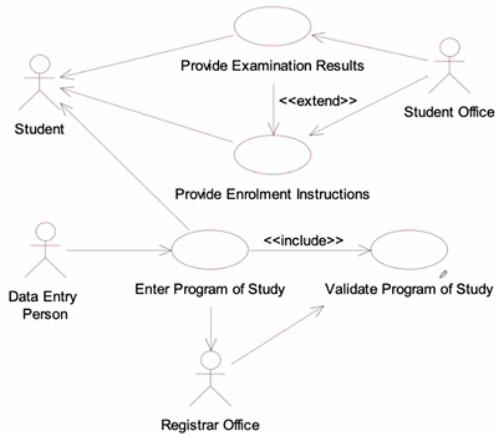
- **Associazione**: è la relazione più immediata e lega attore e caso d'uso. Nella fase di requisiti ad alto livello bastava la linea, ora che siamo più a basso livello in fase di specifica dobbiamo necessariamente usare archi orientati (se da attore a caso d'uso l'attore l'ha attivato, altrimenti attore coinvolto nello svolgimento del caso d'uso)

- **Include ed Extend**: si identificano tra due casi d'uso. Nella fase requisiti linea tratteggiata per identificare significato generico, ora in specifica devo specificare se si tratta di **Stereotipo Extend** o **Include**. Se caso d'uso A <<include>> caso d'uso B, allora se un attore attiva A affinché A venga completato dovrò necessariamente attivare (da un attore differente o lo stesso) e completare B.

Se A <<extend>> B, allora quando un attore attiva il caso d'uso B potrei (ma non necessariamente!) attivare (da un attore differente o lo stesso) anche A.

- **Generalizzazione**: riguarda la relazione tra coppie di attori. Stesso meccanismo visto per le classi: se A viene specializzato da un attore B (come per le classi freccia da sottoattore a superattore) allora B potrà attivare tutti i casi d'uso attivati da A e in più anche altri casi d'uso suoi specifici.

### Example A.5 – University Enrolment



Abbiamo detto che il software per l'università deve prima dell'iscrizione dello studente inviargli le istruzioni per iscriversi insieme ai risultati dell'ultimo semestre. Durante l'iscrizione lo studente consegna il piano di studi e la tizia in segreteria lo inserisce nel software che lo convalida.

Questi scenari di funzionamento sono descritti dai quattro casi d'uso presenti in figura. Gli attori sono gli impiegati della segreteria studenti (student office) che manda le istruzioni insieme ai voti, impiegati segreteria didattica per i corsi di studio specifici (registrar office) che riceve i piani di studio e li convalida, data entry person che fa parte della segreteria e inserisce i dati dello studente nel sistema ed infine lo studente in sé.

I primi tre sono attori che attivano casi d'uso, lo student è attore coinvolto

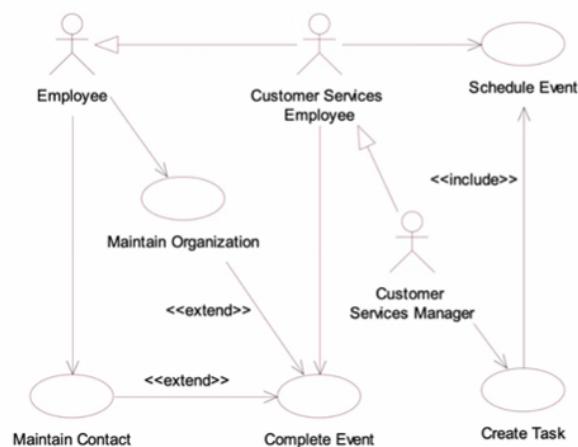
dall'esecuzione di alcuni casi d'uso (es. provide examination results verranno mandati per email allo studente).

Vediamo ora extend e include. Enter program study include validate program study significa che per completare l'enter devo completare il validate, starebbe a significare che non posso inserire un piano di studi e lasciarlo là senza sapere se è valido o meno, corretto.

Provide Examination Result extend provide enrolment instructions, ossia se mando il secondo potrei anche inviare i dati legati ai risultati dell'ultimo semestre (ma non necessariamente). Corretto in quanto se uno studente si iscrive al primo anno sicuramente riceve le istruzioni sul come iscriversi ma non ha risultati dello scorso semestre quindi il primo caso d'uso non si attiva, al contrario potrebbe attivarsi se si parla di uno studente che si iscrive dal secondo anno in poi.

## Lez 16 (30/11)

### Example C.3 – Contact Management



Ricordiamo che il software di contact management aveva l'obiettivo di gestire i contatti dell'azienda che si occupa di ricerca di mercato.

Gestire un contatto nel software significava creare task relativi ad attività di contact management, ogni task presenta diversi eventi.

Quindi casi d'uso Create Task, Schedule Event, Complete Event, Maintain Organization e Maintain Contact (le ultime due tutte le operazioni di base (CRUD) che possono essere fatte sia sulle operazioni che sui contatti).

Create Task include Schedule Event, quindi quando attivo il primo deve essere attivato e portato a termine anche il secondo (corretto perché quando creo un task questo è formato da eventi che devono essere schedulati).

Maintain Contact e Maintain Organization sono in extend con Complete Event, anche questo ha senso perché quando attivo Complete Event (mentre sto completando un evento) potrei aver necessità di gestire le informazioni di una contact person o di un'organizzazione a cui sto facendo riferimento.

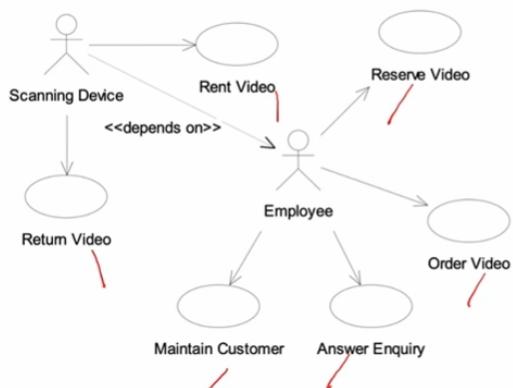
Riguardo gli attori, questo software poteva essere usato da tutti gli impiegati con accessibilità diversa in base al grado.

Si osserva quindi come l'impiegato generico possa attivare i casi d'uso Mantain Contact e Organization per recuperare le informazioni basi di organizzazioni/contact person. Poi l'attore Customer Services Employee (già più specifico) che può schedulare gli eventi e completarli. Vi è infine un ultimo attore, il Customer Services Manager, l'unico che può creare task.

Generalizzazioni: l'impiegato può essere specializzato in un impiegato delle risorse clienti, che a sua volta può essere manager del reparto. Gli attori più specifici in questa gerarchia possono fare quanto fanno gli attori generici più alcune cose specifiche che fa in quanto specializzato (da qui l'attivazione di certi casi d'uso), quindi un sottoattore come CSM può non solo creare task, ma anche schedulare eventi e accedere a informazioni di organizzazioni e contact person (ereditarietà).

Similmente a quanto visto nel modello dei dati anche qui avrei potuto evitare la generalizzazione collegando tutto e facendo un macello, ma avrei mancato di semantica per capire le relazioni tra attori coinvolti nei casi d'uso.

### Example B.4 – Video Store



Tornando ora al caso di studio VideoStore, l'attore principale è sicuramente Employee che usa il software per gestire tutto, ma vediamo un secondo attore Scanning Device che è collegato all'impiegato con una relazione **<<depends on>>** (anche questa associazione generale, tratteggiata in realtà e non continua come si vede). *Tale relazione indica che la pistola per scannerizzare è “dipendente” dall'impiegato, necessita l'impiegato perché si attivino e completino i casi d'uso associati ad essa, nb non è generalizzazione!*

Quindi è come se fosse l'impiegato ad essere associato a Rent e Return Video, ma serve lo strumento/attore di scanning per farlo.

**Ora, il diagramma di casi d'uso in sé non fornisce informazione completa dal punto di vista comportamentale.** Ciò che dobbiamo fare sarà quindi estendere la descrizione specificando i dettagli dietro ciascun caso d'uso.

**Due approcci** per far ciò: **Informale** per cui in un template inserisco i dettagli in linguaggio naturale per specificare il caso d'uso, **Formale** che fa uso del **Activity Diagram** (UML) che fa capire esattamente cosa fa il software quando si attiva il caso d'uso.

Vediamo ora come funziona ciò focalizzandoci sul singolo caso d'uso Rent Video. Iniziamo con **L'approccio Informale:**

Brief Description	<p><i>A customer wishes to rent a video tape or disk that is picked from the store's shelves or that has been previously reserved by the customer.</i></p> <p><i>Provided the customer has a non-delinquent account, the tape is rented out once the payment has been received.</i></p> <p><i>If the tape is not returned in a timely fashion, an overdue notice is mailed to the customer.</i></p>
Actors	<i>Employee, Scanning Device</i>
Preconditions	<p><i>Video tape or disk is available to be hired.</i></p> <p><i>Customer has a membership card.</i></p> <p><i>Scanner devices work correctly.</i></p> <p><i>Employee at the front desk knows how to use the system.</i></p>

**Brief Description** == molto simile (a livello d'astrazione) di quanto si fa in fase di definizione dei requisiti use case driven, dove per ogni caso d'uso fornisco una descrizione che mi dica a grandi linee ciò che succede.

**Attori** == si inseriscono entrambi gli attori per la dipendenza di cui abbiamo parlato prima (scanning device non è attore autonomo)

**Precondizioni** == condizioni che devono essere vere per abilitare la funzione sul software, in questo caso il cliente deve avere la card, il film deve essere pronto per il noleggio, lo scanner funziona, l'impiegato sa usare il software.

Main Flow	<p><i>A customer may inquire an employee about video availability (including a reserved video) or may pick a tape or disk from the shelves. The video and membership card are scanned and any delinquent or overdue details are brought up for the employee attention. If the customer does not have a delinquent rating, then he/she can hire up to a maximum of eight videos. However, if the rating of the customer is "unreliable" then a deposit of one rental period for each tape or disk is requested. Once the amount payable is received, the stock is updated and the tapes and disks are handed out to the customer together with the rental receipt. The customer pays by cash, credit card or electronic transfer. Each rental record stores the check-out and due-in dates together with the identification of the employee. A separate rental record is created for each video hired.</i></p> <p><i>The use case will generate an overdue notice to the customer if a video has not been returned within two days of the due date, and a second notice after another two days (and at that time the customer is noted as "delinquent").</i></p>
-----------	---

**Main Flow** == flusso di attività principale legato al caso d'uso. Delinquent sta ad indicare il fatto che il cliente non ritorni i video che ha noleggiato, in caso ne abbia non potrà noleggiare i video. Se è un unreliable gli posso chiedere un deposito (se il film costa 2 euro me ne faccio dare 4 e gliene ridarò 2 se lo riporta in tempo).

Una volta prestati i film è importante aggiornare il database per le occorrenze.

Ogni prestito genera data prestito e data ritorno con impiegato correlato.

Si manda una overdue notice due giorni dopo la data di ritorno, e se non ritorna il cliente diventa delinquente!!!

(nb gradi di delinquenza: unreliable se poco delinquente e posso chiedere deposito, se invece grado più alto non gli presto più niente)

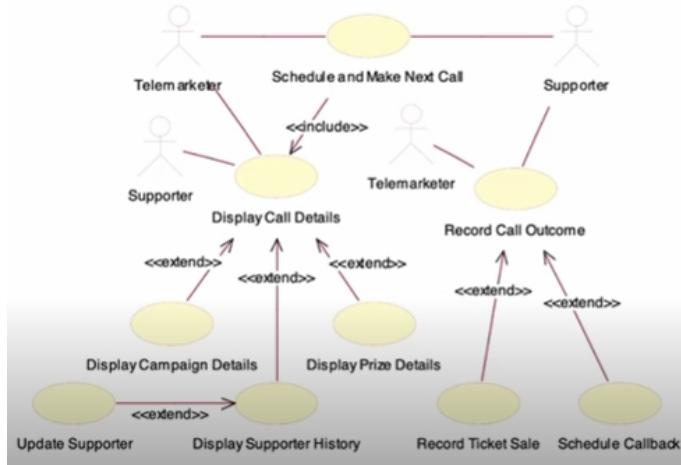
Alternative Flows	<p><i>A customer does not have a membership card. In this case, the Maintain Customer use case may be activated to issue a new card.</i></p> <p><i>An attempt to rent too many videos.</i></p> <p><i>No videos can be rented because of the customer's delinquent rating.</i></p> <p><i>The video medium or membership card cannot be scanned because they are damaged</i></p> <p><i>The electronic transfer or credit card payment is refused.</i></p>
Postconditions	<p><i>Videos are rented out and the database is updated accordingly.</i></p>

**Alternative Flows** == si specificano eventuali flussi alternativi. Es. se il cliente non ha la membership card allora potrei aver la necessità di attivare Mantain Customer, in questo senso aveva senso legare i due casi d'uso con <<extend>> con freccia verso Rent Video (mentre si fa rent video potrei necessitare di Mantain Customer)

**Postconditions** == condizioni vere dopo che il caso d'uso completato con successo

Prima di vedere l'approccio formale vediamo per completezza il diagramma casi d'uso del Telemarketing.

### Example D.3 – Telemarketing



Schedule and Make Next Call e Record Call Outcome due casi d'uso fondamentali che fa il Telemarketer. Display Call Details attivato a fronte di una telefonata perché il telemarketer sappia chi sta chiamando e perché. Diversi casi d'uso in extend: per quel che riguarda i risultati telefonata Record Ticket Sale (se il cliente acquista biglietti) e Schedule Callback (se si vuole ripianificare la chiamata al supporter).

Per quanto riguarda invece i dettagli chiamata potrei necessitare dei dettagli campagna, dettagli premi e dettagli storia supporter e eventualmente aggiornare dati supporter. Tutti questi extend sono chiaramente convincenti.

Esiste infine un include tra schedule and make next call e display call details anch'esso convincente, infatti quando faccio una nuova chiamata sono necessari i

dettagli di chi chiamo e il perché.

Un problema nel diagramma tuttavia è che a questo livello di specifica bisogna sempre specificare archi orientati tra attori e casi d'uso, e ciò qui non avviene.

Si osserva poi (non è un errore) che vi sono attori duplicati.

**Altro importante errore è che abbiamo detto definendo il concetto di caso d'uso che ogni caso d'uso deve avere un attore che lo attiva e in questo caso diversi sono lasciati a se stessi, è chiaro come questi dovranno essere attivati dal telemarketer.**

Risolvo il problema o utilizzando associazioni esplicite o aggiungendo una nota al diagramma.

Ora passiamo **all'approccio formale**, per ogni caso d'uso dovrei definire il **Diagramma delle Attività**.

*Il diagramma delle attività ha grande potere espressivo, poiché permette di rappresentare il flusso di esecuzione sia sequenziale che concorrente, inoltre utilizzabile a diversi livelli di astrazione (OOA e OOD).*

*Esso rappresenta in UML una variante degli state diagram, dove in questo caso i nodi (ex stati) rappresentano le attività mentre le transizioni (ex passaggio di stato) rappresentano il completamento di un'attività e il passaggio alla successiva.*

*In realtà questo diagramma si utilizza principalmente in fase di OOD per rappresentare un algoritmo, ossia un flusso di esecuzione di operazioni definite nel class diagram, ma in fase di OOA si usa per rappresentare il flusso di attività nell'esecuzione di un singolo caso d'uso.*

*Dal momento che in OOA non vengono rappresentati gli oggetti che eseguono le attività, posso usare il class diagram in questo senso anche senza class diagram (tanto che in alcuni approcci si parte dal modello comportamentale con questo diagramma).*

Le primitive fornite da UML per specificare il diagramma delle attività: si parte da un **nodo iniziale che rappresenta l'evento d'inizio**, che *corrisponde all'attivazione del caso d'uso*. Le **varie attività da svolgere (altri nodi)** si identificano pensando a uno specifico scenario di funzionamento del caso d'uso, ed **ognuno di questi nodi sono collegati da transizioni che possono essere associate a specifiche condizioni di guardia** (posso passare da un nodo a un altro sse la condizione sulla transizione è vera). *Si passa ovviamente a una transizione in uscita se l'attività è stata completata, ossia è stata terminata la sua elaborazione.*

Per rappresentare **flussi concorrenti** si sfruttano delle barre di sincronizzazione dette **fork-join** (simili a quelle viste nelle reti di Petri) mentre per i **flussi alternativi** si sfruttano **nodi decisionali** (branch/merge diamonds, rombi).

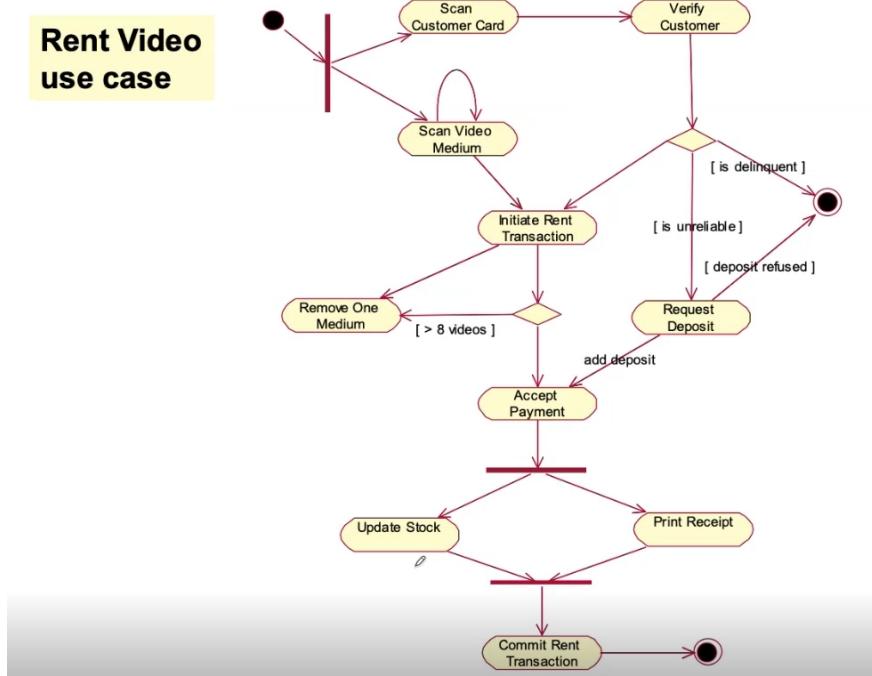
*Importante l'utilizzo di questi diagrammi in quanto mentre con il linguaggio naturale è possibile ambiguità qui l'interpretazione è una.*

Descriviamo l'esempio sotto. Nodo iniziale cerchietto pieno (corrisponde all'attore che attiva il caso d'uso), finale cerchietto pieno cerchiato.

Troviamo subito una barra che è di fork (in quanto il numero di archi in uscita è maggiore degli entranti) che quindi ci dice che vi sono due flussi in parallelo, il primo

scannerizza i video medium noleggiati dal cliente e da qui due possibili uscite, la prima self transaction (se il cliente arriva con 5 video devo scannerizzarli tutti) e la seconda mi porta all'attività successiva. La seconda attività sopra è scannerizzare la carta del cliente per poi eseguire l'attività verifica cliente (se è delinquente o meno).

## Example B.5 – Video Store



Dopo verify customer c'è un nodo decisionale: cliente affidabile allora inizio la transazione di prestito, se delinquente arrivo al nodo finale e finisce la storia, se inaffidabile richiedo il deposito, se rifiutato allora arrivo allo stato finale, se accettato allora lo faccio pagare. NB ogni arco uscente da un rombo (flusso alternativo) deve specificare una condizione, al limite posso lasciarne uno vuoto, che ha condizione implicita dalle altre definite (es. in questo caso cliente affidabile o  $\leq 8$  videos). Tornando al cliente affidabile dopo aver inizializzato la transazione devo verificare che il numero in noleggio sia  $\leq 8$ , se così non è allora ne rimuovo finché e torno a Initialize Rent Transaction finché la condizione è verificata e scendo a Accept Payment. A questo punto di nuovo **fork**: il software aggiorna lo stock e allo stesso tempo stampa la ricevuta. Dopodiché **join**: ciò significa che anche se l'update stock fa prima della stampa ricevuta prima di passare alla fase successiva si deve aspettare. C'è infine la conferma della transazione che porta allo stato finale.

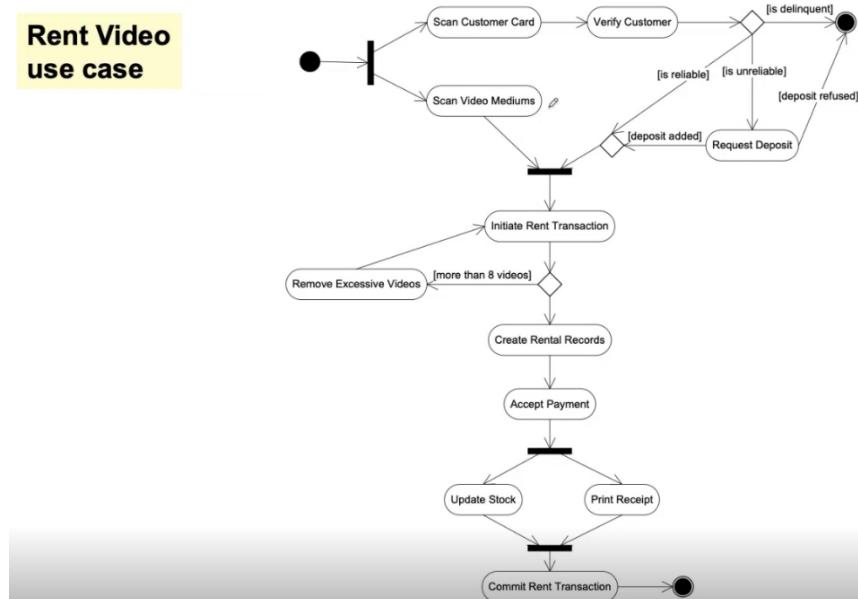
Il diagramma d'esempio presenta vari problemi sia sintattici che semanticci: chiaramente la freccia del ciclo deve andare da Remove One Medium a Inizialize Rent Transiction e non viceversa.

Un altro problema è che se il cliente è considerato inaffidabile non effettuo il controllo sul numero di video, altro problema sta in Scan Video Medium: non esistono condizioni di guardia che mi dicano quando devo ciclare e quando devo passare all'attività successiva. Normalmente per rappresentare ciò mi serve un nodo di branch (rombo che torna all'attività se devo riscannerizzare altrimenti prosegue),

in UML posso rappresentare ciò con la forma vista sopra ma devo necessariamente in tal caso anche esplicitare la condizione di guardia che mi permetta di capire come comportarmi.

Ulteriore problema sta nel fatto che Initialize Rent Transaction è attivata due volte: sia quando il flusso arriva da Scan Video Medium che quando arriva dal branch post Verify Customer per cui il cliente è affidabile. Serviva sfruttare una join affinché l'attività avvenisse solo quando entrambi erano conclusi.

Da questi ragionamenti si deriva la versione corretta:



Si noti la differenza tra **merge** e **join**: uso **merge** con il rombo quando o vale una condizione o vale l'altra, la **join** invece con la barra se devo aspettare entrambi i flussi.

Questa versione è sicuramente corretta ma potrebbe essere ancora migliorata, es. il fatto di definire le attività concorrenti iniziali di scan non è conveniente visto che l'impiegato deve scannerizzarle una alla volta



Tornando al modello comportamentale ciò che abbiamo fatto è stato identificare i casi d'uso e gli attori, scrivere il diagramma dei casi d'uso e per ogni caso d'uso fornire un diagramma di attività o usare notazione informale di specifica.

Stiamo tuttavia utilizzando un linguaggio Object Oriented, tutte le attività presenti nei nodi del diagramma di attività saranno realizzate concretamente al tempo di esecuzione dall'interazione (scambio di messaggi ossia richiesta di far partire metodi) tra oggetti creati dalle classi.

Quindi a questo punto ho necessità dell'utilizzo del Modello dei Dati per andare avanti.

Si definiscono a questo punto i [Diagrammi di Interazione](#).

UML li mette a disposizione tramite due diagrammi equivalenti in termini di potere espressivo: **Sequence Diagram** e **Collaboration Diagram**.

Come già accennato il diagramma di sequenza è usato principalmente in fase di specifica (OOA) e descrive lo scambio di messaggi tra oggetti in ordine temporale mentre quello di collaborazione in fase di progettazione (OOD) e descrive lo scambio di messaggi tra oggetti mediante relazioni tra gli oggetti stessi.

Come detto si tratta di rappresentazioni equivalenti e possono essere generati automaticamente l'uno dall'altro. Esiste la differenza perché il primo rende esplicito l'ordine temporale, mentre il secondo si focalizza proprio sui messaggi che si scambiano gli oggetti. Si usa il primo in fase di specifica perché diventa ingombrante all'aumentare degli oggetti, e in OOD molti oggetti.

Questi diagrammi sono importanti perché è grazie a loro che siamo in grado di identificare le operazioni delle classi nel class diagram (ed eventuali classi aggiuntive).

Ci concentreremo ora sui diagrammi di sequenza, vedremo quelli di collaborazione nella seconda parte del corso.

**Specifiche di sequence diagram:** il diagramma di sequenza è un diagramma che mostra in modo esplicito le interazioni tra oggetti, che avvengono tramite uno scambio di messaggi (di tipo "richiesta esecuzione attività").

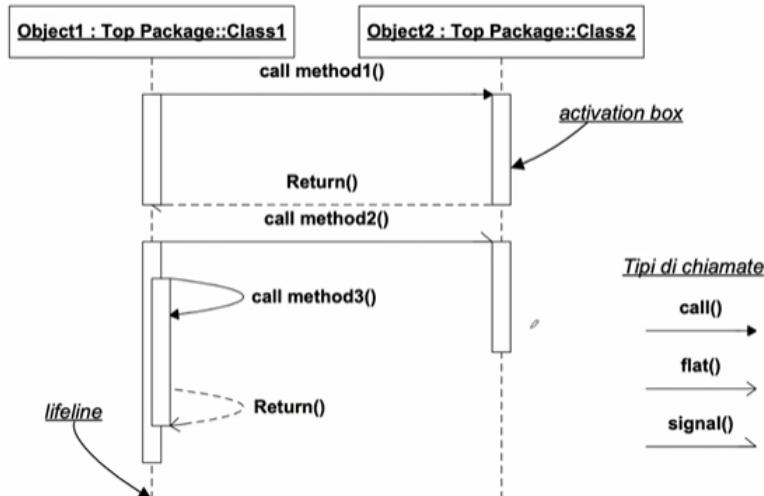
In particolare le activity dell'activity diagram possono essere mappate come messaggi in un sequence diagram.

I messaggi possono essere di due tipi:

- **Signal (asincrona)** == denota una chiamata di tipo asincrono, rappresenta il fatto che l'oggetto mittente continua l'esecuzione dopo aver inviato il messaggio asincrono
- **Call (sincrona)** == denota una chiamata di tipo sincrono, rappresenta una richiesta di tipo send-reply in quanto l'oggetto mittente blocca l'esecuzione dopo aver inviato il messaggio in attesa di risposta da parte dell'oggetto destinatario.

Dal punto di vista visuale la notazione è la seguente:

## Notazione per sequence diagram



I rettangoli rappresentano gli oggetti che interagiscono, per ognuno devo dirne nome e classe che lo genera (per ora si tralascia il package).

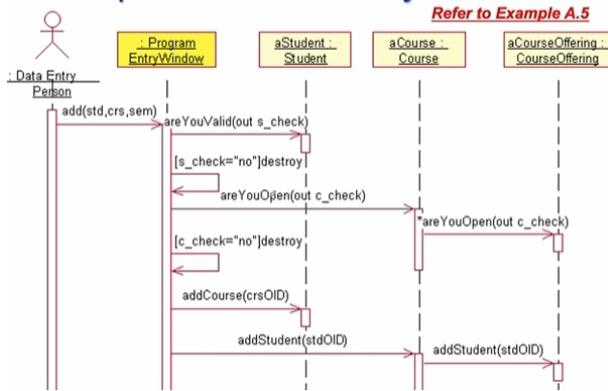
Per ogni oggetto si identifica la **lifeline** (corso di vita dell'oggetto), dove l'oggetto scambia informazioni con altri, e questo scambio si rappresenta con archi orientati. es. call method1() significa che l'oggetto1 chiede a oggetto2 di eseguire metodo1, ciò viene fatto tramite **call() freccia piena**, quindi attende la risposta e non fa niente nell'attesa. Il secondo oggetto invia la risposta Return() al primo tramite **signal (freccia con aletta a metà)** per cui continua l'esecuzione dopo aver risposto, il primo oggetto allora riprende l'esecuzione. Poi il primo manda una signal method2() per cui non si ferma e come successiva esecuzione call method3() call su se stesso.

UML da anche la possibilità di definire interazioni senza specificare se siano call o signal tramite flat() se ancora in fase di specifica per certe interazioni non si è scelto in modo definitivo.

Altra cosa che si nota è che **quando un oggetto è attivo** (in esecuzione/manda messaggi etc..) *allora ciò è specificato tramite gli activation box*. **Importante il fatto che un activation box più piccolo NON implica il fatto che questo rappresenti un arco temporale minore poiché in UML non è definito il concetto di tempo** (infatti esistono profili separati per estendere UML con caratteristiche temporali).

Con il sequence diagram mi limito a conoscere le sequenze temporali.

### Example A.6 – University Enrolment



Gli oggetti coinvolti dal sequence diagram sono creati dalle classi che conosciamo Student, Course e CourseOffering. Viene anche creato un altro oggetto Program EntryWindow da una classe senza nome che non abbiamo considerato in quanto si tratta di una *classe boundary*, ossia classi che non gestiscono informazioni ma che si occupano di implementare l'interfaccia tra software e utente.

Infine si ha l'attore che scatena la sequenza di interazioni tra oggetti.

Si descrive ciò che avviene quando dopo che lo studente ha consegnato il piano di studi il data entry person deve inserirlo nel sistema.

L'attore chiede al software tramite l'interfaccia di aggiungere un certo studente std a un certo corso crs in un certo semestre sem. Per rispondere alla richiesta l'oggetto di interfaccia interagirà con gli altri oggetti entity che conosciamo.

Per prima cosa si verifica che lo studente sia correttamente iscritto (valid), poi se il corso accetta ancora studenti. Se le cose vanno bene aggiungerò lo studente.

Anzitutto l'oggetto interfaccia (boundary) chiede all'oggetto studente se è valido, quindi chiamata al metodo areYouValid con parametro di output s\_check (lo passo vuoto, mi aspetto risposta s\_check booleana).

If s\_check no allora boundary fa una call a se stesso “destroy” di modo che l'attore capisca di non poter inserire i dati (ovviamente apparirà un messaggio a schermo), else boundary chiede all'oggetto corso se è ancora aperto (out c\_check) (condizione di guardia!).

Ora l'oggetto corso non mantiene le informazioni di tutte le istanze del corso quindi a sua volta chiede all'istanza del corso per quel semestre CourseOffering se è o meno aperto inoltrando la richiesta iniziale. If c\_check no then destroy, else boundary fa

aggiungere il corso tra quelli seguiti dallo studente tramite interazione con Studente e aggiunge lo studente al corso tramite interazione con Course che a sua volta inoltrerà la richiesta a CourseOffering.

*NB qui si usa UML1, molto limitato. In realtà le condizioni di guardia qui presenti (es if s\_check no .. else..) si dovrebbero usare nei diagrammi di attività e non nei sequence diagram, andrebbe quindi creato un sequence diagram per ogni possibile interazione.*

Ciò è superato in UML 2 grazie all'introduzione di primitive apposite.

Ciò che comunque a noi interessa guardando questo sequence diagram è che possiamo estrapolare informazioni molto utili per raffinare il class diagram in termini di operazioni.

Infatti ad es. vediamo come boundary chieda all'oggetto studente di eseguire il metodo areYouValid(), quindi Student deve mettere a disposizione quel metodo. Secondo questa logica capisco meccanicamente a partire dal sequence diagram quali metodi devo assegnare alle rispettive classi.

## Lez 17 (4/12)

Torniamo quindi al modello dei dati per aggiungere nuove operazioni ed anche eventuali nuove classi (come le boundary), dall'insieme di operazioni ci sarà quindi possibile definire **l'Interfaccia Pubblica di Classe**.

Perché pubblica? Un concetto importante nella programmazione a oggetti è **l'information hiding** per cui si vuole far vedere all'esterno solo ciò che è necessario per gli oggetti conoscere, nascondendo quanto non è necessario.

**In generale si vuole che, piuttosto che rendere immediatamente accessibile agli oggetti gli attributi di una classe** (e quindi la possibilità per lui di modificarli direttamente), **si definisce un'interfaccia di classe attraverso degli accessor method** (metodi di accesso, *i getter e setter*) **che permettono di recuperare e aggiornare i valori degli attributi di una certa classe** (es. definisco solo getter per attributi che non voglio vengano modificati).

Con la notazione UML siamo abituati a vedere le classi come un rettangolo diviso in tre compartimenti nome, attributi e operazioni mentre un'altra notazione poi scartata rappresentava meglio l'information hiding, due cerchi dove dentro quello più interno gli attributi e nella parte esterna le operazioni: bisogna passare per le operazioni affinché si raggiungano gli attributi.

**Quindi l'Interfaccia Pubblica di Classe definisce l'insieme di operazioni che la classe mette a disposizione delle altre classi.** Durante la fase di OOA (dove ci si concentra sul COSA), si determina solo la **signature dell'operazione** (dichiarazione di funzione) : **nome, lista di parametri e il tipo di ritorno**.

Sarà in fase di OOD che si definirà l'algoritmo che implementa l'operazione.

Un'operazione può avere **Instance Scope** se usata su singolo oggetto, o

**Class (static) scope** se opera su attributi statici (condivisi da tutti gli oggetti creati a partire da quella classe). La seconda operazione è rappresentata con un char \$ che precede il nome dell'operazione.

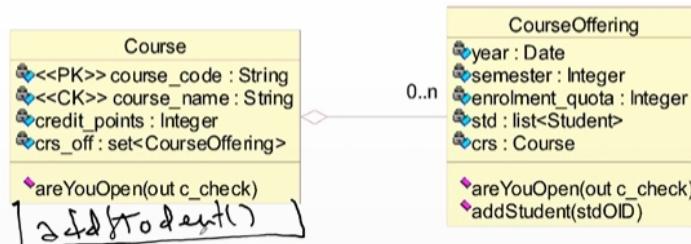
Per identificare le operazioni come detto partiamo dal sequence diagram e banalmente ogni messaggio inviato ad un oggetto identifica un metodo della classe a cui appartiene tale oggetto.

Inoltre ad ogni classe vanno aggiunte tutte quelle operazioni che appartengono al criterio CRUD, ossia ogni oggetto deve supportare operazioni di **create** (crea nuova istanza, almeno un costruttore), **read** (stato di un oggetto, getter), **update** (lo stato di un oggetto, setter), **delete** (l'oggetto stesso, serve a sistemare anche quanto fatto dall'oggetto es. prima di eliminarlo chiudo i file etc...).

Vediamo l'applicazione da A.6 sequence diagram per l'università all'aggiornamento del diagramma di dati. L'unica cosa da aggiungere qua sotto era addStudent come metodo anche di Course

## Example A.7 – University Enrolment

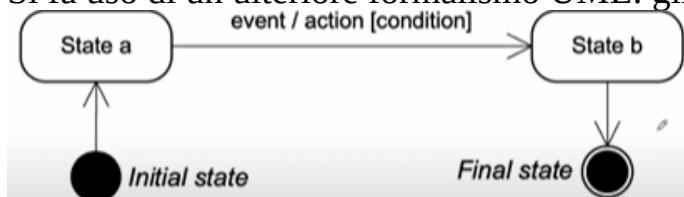
- Refer to Examples A.3 and A.6 and to the classes Course and CourseOffering
- Derive operations from the Sequence Diagram and add them to the classes Course and CourseOffering



Oltre che costruire i modelli comportamentale e di dati tuttavia, è necessario costruire anche il **modello dinamico** affinché la specifica con approccio semiformale mi permetta di costruire un modello completo.

Il **Modello Dinamico** rappresenta il **comportamento dinamico (evoluzione)** degli oggetti creati da una certa classe, in termini di **stati possibili ed eventi e condizioni** che originano **transizioni di stato** (oltre che le eventuali azioni da svolgere a seguito dell'evento verificatosi).

Si fa uso di un ulteriore formalismo UML: gli **State Diagrams**.



**Evento/azione che comporta il cambiamento di stato dell'oggetto ed un'eventuale condizione di guardia che deve esser vera affinché la transizione termini correttamente.** Somiglia molto a un Activity Diagram ma in realtà è l'activity diagram ad essere un caso particolare di State Diagram.

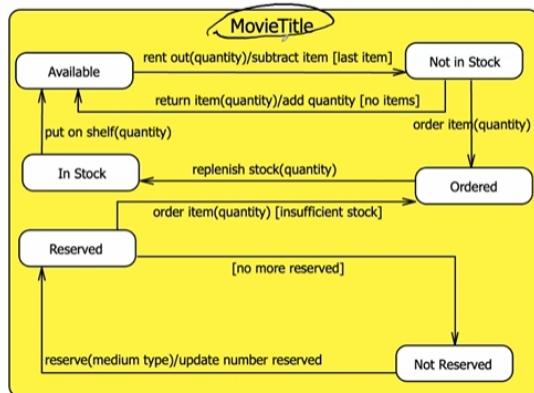
Normalmente il Modello dinamico viene costruito solo per le classi per cui è interessante descriverne il comportamento dinamico, come le **classi di controllo** (finora abbiamo visto **entity classes** che gestiscono l'accesso ai dati e **boundary classes** per l'interfaccia utente, le **control classes** gestiscono la logica

**dell'applicazione** es. che fare a fronte di una richiesta utente verso un oggetto boundary, l'oggetto di controllo è quello che conosce la policy di funzionamento).

**Principalmente quindi questo modello è usato per software per cui è importante conoscere l'evoluzione, come applicazioni real-time e scientifiche** (mentre è meno frequente nello sviluppo di applicazioni gestionali).

Vediamo soltanto un esempio visto che nei nostri casi di studio abbiamo visto software di stampo gestionale per capire come viene applicato il diagramma degli stati.

### Example B.5 – Video Store



Si vuole quindi rappresentare l'evoluzione dei possibili stati in cui si può trovare l'oggetto creato dalla classe MovieTitle. Qui manca in realtà nodo iniziale e finale, è facile immaginare l'iniziale collegato a Available (creo un oggetto film perché è arrivato e disponibile in magazzino). L'unico arco in uscita da Available porta a Not In Stock con transizione avente notazione completa in quanto ha evento, azione e condizione. Quindi affinché si passi da Available a Not In Stock è necessario che sia noleggiato l'ultimo elemento disponibile, quando ciò accade esso viene sottratto. Si resta in Not In Stock finché o viene restituito il film noleggiato (in tal caso si torna in available) oppure se ne vengono ordinati di nuovi (si passa da ad Ordered). Da Ordered al magazzino stato In Stock e poi si torna ad Available dopo averli messi sullo scaffale.

Si poteva passare anche allo stato Ordered nel caso in cui un cliente lo abbia prenotato.

Con ciò abbiamo concluso la panoramica sull'analisi orientata a oggetti (abbiamo introdotto un metodo che non segue proprio uno dei vari metodi proposti nel corso

degli anni ma adattato per mostrare i diagrammi principali e far capire come procedere in fase di specifica).

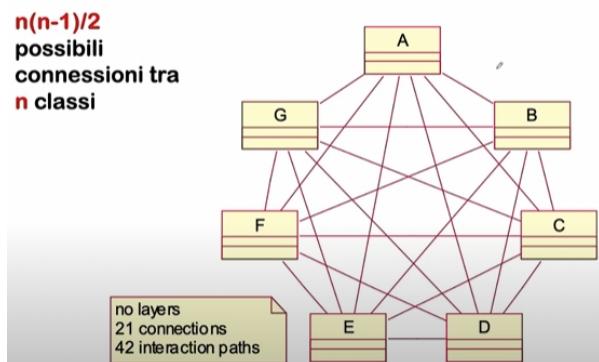
Prima di passare alla parte di *Pianificazione* ragioniamo su un aspetto molto rilevante quando si sviluppa un software anche di dimensioni medie: la **Gestione della Complessità nei modelli di OOA**.

In casi reali i modelli che costruiamo possiamo arrivare ad avere un numero enorme di classi. Nella fase di OOA per sistemi software di grandi dimensioni occorre quindi gestire opportunamente l'intrinseca complessità dei modelli.

Le associazioni tra classi nel modello dei dati generano complesse reti di interconnessione, in cui i cammini di comunicazione crescono in modo esponenziale con l'aggiunta di nuove classi.

**L'approccio utilizzato per far fronte alla complessità** è la **stratificazione**: meccanismo che permette di isolare elementi tenendo conto di come essi devono interagire. In particolare gli elementi che fanno parte di un certo strato possono interagire solo con elementi dello stesso strato o di strati adiacenti. Tramite l'introduzione di questa gerarchia di classi, si passa da complessità esponenziale a polinomiale.

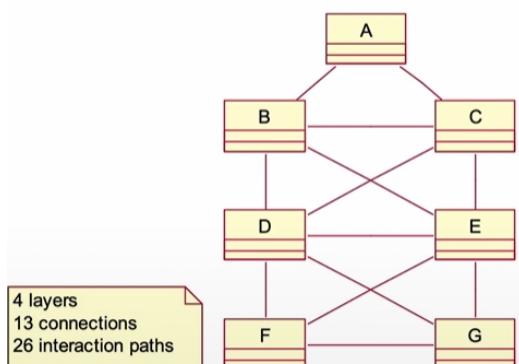
### Class diagram non stratificato



In questo caso 21 possibili connessioni. Con connessione si intende associazione e sappiamo che un'associazione può esser navigata in entrambi i versi, quindi in totale in realtà 42 ( $n(n-1)$ ) percorsi d'interazione -> con solo 7 classi nel caso peggiore 21 connessioni e 42 interazioni tra classi.

Con 4 strati:

### Class diagram stratificato



Si scende a 13 possibili connessioni e quindi 26 interazioni.

Bisogna ora capire come definire gli strati via UML e quale criterio usare per raggruppare le classi in uno strato.

Per poter raggruppare gli elementi in UML si sfrutta la nozione di **Package** (in un package raggruppo quindi non solo classi ma anche altri elementi UML come use case).

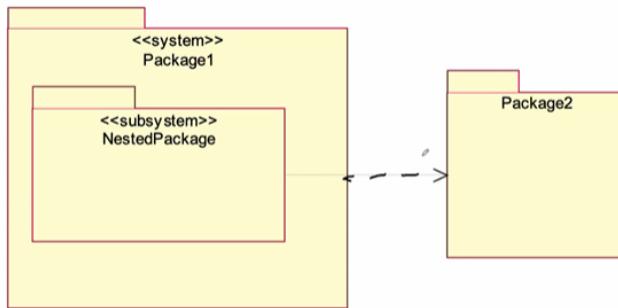
I package possono anche essere annidati (gerarchie di package, il più esterno ha accesso a quelli interni), una classe può appartenere ad un solo package ma comunicare con classi appartenenti ad altri package.

Si effettua una dichiarazione di visibilità (public, private, protected) per definire la visibilità delle classi presenti nel package.

Si usa il **simbolo cartella** per identificare i package, e si possono definire relazioni di dipendenza tra package con la freccia tratteggiata (relazione generica), quando si usa è opportuno come al solito definire lo stereotipo per capire che tipo di relazione sia.

In generale si possono specificare **due tipi di relazioni tra package**:

Generalizzazione (implicano anche dipendenza) e Dipendenza (di uso, accesso, visibilità etc..)



la relazione di dipendenza non è specificata, ma sta ad indicare che eventuali modifiche a *Package2* potrebbero richiedere modifiche di *NestedPackage*

In UML non esiste il concetto di Package Diagram, infatti come già anticipato i package possono essere creati all'interno di class diagram e use case diagram.

Il criterio che ci permette di definire gli strati, quanti sono e come raggruppare le classi è l'approccio **BCE (Boundary Control Entity)**.

Le classi sono raggruppate in funzione delle loro responsabilità:

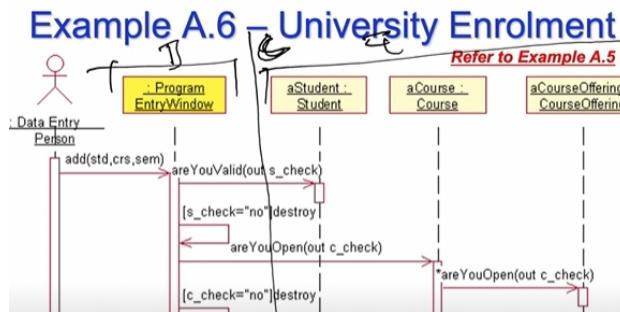
- nel **Boundary Package** inseriamo solo le classi i cui oggetti gestiscono l'interfaccia tra attore e sistema

- nel **Control Package** inseriamo le classi che rappresentano la logica applicativa del software, ossia i cui oggetti intercettano l'input dell'utente e controllano l'esecuzione di uno scenario di funzionamento del sistema

- nel **Entity Package** inseriamo le classi entity che abbiamo identificato fin dall'inizio, i cui oggetti gestiscono la parte di accesso ai dati

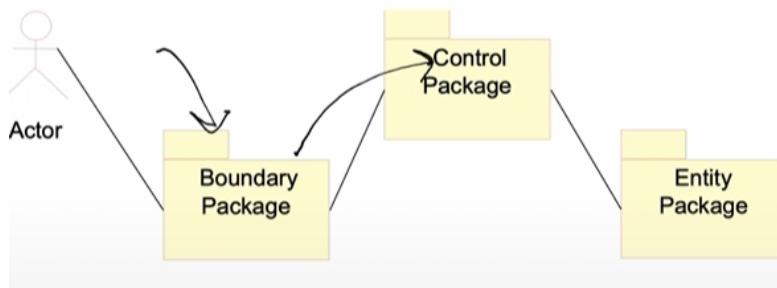
(BCE simile al paradigma in programmazione MVC model view controller dove model rappresentazione dati, view interfaccia e controller gestione logica applicativa)

Ma come stratificarli? Semplicemente a fronte di una richiesta di un utente esterno al sistema, questa è “catturata” dall’oggetto Boundary, inoltrata all’oggetto di Controllo che per esaudire la richiesta potrà necessitare l’uso di oggetti entity per accedere alle informazioni.



Questo sequence diagram in questo senso non era corretto secondo questo meccanismo di stratificazione in quanto mancava l’oggetto di controllo, infatti le funzionalità di controllo erano state assegnate allo stesso oggetto boundary che sapeva esattamente come comportarsi e quali oggetti contattare.

Quello che facciamo secondo BCE è invece creare una gerarchia di package boundary-control-entity.



Un oggetto boundary non può comunicare direttamente con Entity e viceversa! Vantaggioso anche dal punto di vista di manutenzione: se cambia ad es. la policy legata all’iscrizione saprà subito risalire all’oggetto responsabile della logica da seguire presente nel package di controllo.