

## Ingegneria del software

### Lezione 2

**Prodotto software:** tutta la documentazione che accompagna il software che andiamo a sviluppare

**Processo software:** tutte le attività che ci consentono di partire dai requisiti ed arrivare al prodotto software

#### Produzione software

**Arte:** applicazioni sviluppate da singole persone e utilizzate per loro stesse

**Artigianato:** piccoli gruppi specializzati realizzano dei prodotti software per un cliente

**Industria:** diffusione del software in diversi settori; aumenta la complessità; necessità di migliorare produttività e qualità; non è più un singolo gruppo che produce software, ma è un team coordinato

- **Produttività:** quanto riesco ad ottenere in un'unità di tempo, dice quanto sono stato efficiente nella realizzazione di una certa attività

#### Programma vs Prodotto

- **Programma:** l'autore è anche l'utente
- **Prodotto software:** usato da persone diverse da chi lo ha sviluppato

Software industriale: costo circa 10 volte del costo del corrispondente programma

**Standard IEEE:** il software è l'insieme di programmi, procedure, documentazione associata e tutti i dati che servono per l'operatività di un sistema

Un prodotto software non è solo codice ma sono anche tutti gli artefatti che lo accompagnano: codice, documentazione, casi di test, specifiche di progetto, procedure di gestione, manuale utente

#### Software

- **Prodotto:** invisibile, intangibile, facilmente duplicabile ma costosissimo: opera dell'ingegno protetta dalle leggi
- **Servizio:** ha un servizio e si basa su una infrastruttura

## Ecosistemi software

Sono mercati in cui si vendono prodotti (PlayStore) o componenti e servizi (Amazon Elastic Computing)

La caratteristica principale è che c'è una collezione di prodotti software definiti da un'azienda che vengono sviluppati ed evolvono nello stesso ambiente

## Prodotti software

- **Prodotti generici**

Sistemi prodotti da un'azienda e venduti ad un mercato di massa

- **Prodotti specifici**

Sistemi commissionati da uno specifico utente e che vengono sviluppati e venduti per un certo contraente

La spesa maggiore è nei prodotti generici, ma il maggior sforzo di sviluppo è nei prodotti specifici

## Social software

Software che supporta la conversazione di comunità di utenti

## Software libero

Software può essere eseguito, distribuito liberamente però non è gratis

## Il software è un prodotto industriale

Software è un prodotto industriale, è il risultato di un processo che inizia con un'idea e termina quando il software viene ritirato; questo è ciò che viene chiamato ciclo di vita del prodotto software

Ciclo di vita del software è allora tutto quel tempo che passa dal concepimento di quel prodotto a quando questo prodotto viene ritirato.

Il costo di un prodotto software tende a crescere il quadrato rispetto alle sue dimensioni

## Caratteristiche prodotto software

Rispetto ai prodotti industriali classici:

- E' intangibile: cioè non si tocca

- E' malleabile: che si può cambiare, riusciamo a modificare il software sulla base delle esigenze; i cambiamenti però sono costosi, dobbiamo assicurarc che il cambiamento che andiamo a fare non introduce nuovi errori
- Ad alta intensità di lavoro umano: il lavoro è svolto dalle persone, le persone possono sbagliare, hanno bisogno di riposarsi
- Spesso costruito ad hoc non assemblato
- Manutenzione significa cambiamento: c'è bisogno di effettuare continua manutenzione; aumenta la curva di fallimenti ad ogni cambiamento

## Problemi della produzione software: Costi

Il software ha costi elevati

I costi maggiori sono dati dalle ore lavoro, quelle che chiamiamo manpower, cioè le ore di lavoro delle persone coinvolte nel progetto

Tra tutti i costi quello che è predominante è il manpower che viene espresso in termini di mesi/uomo

Il testing impiega fino al 50% dei costi di sviluppo

La manutenzione costa più dello sviluppo: per sistemi che rimangono a lungo in esercizio, i costi di manutenzione possono essere svariate volte il costo di produzione

## Altri problemi

Sono i ritardi, diciamo che il progetto è "runaway" e sono quei progetti che sono in ritardo o fuori dal budget e lo schedule stimato

## Servizi in perpetuo sviluppo

Ad esempio Facebook o Amazon che sono sistemi che offrono servizi 24 ore tutto il giorno e sono in continuo cambiamento

## Requisito e feature

- Requisito software: intendiamo una funzione che l'utente finale può utilizzare. Quindi è una funzionalità oppure una proprietà; un requisito che è una proprietà è il sistema deve essere usabile, deve essere sicuro, è una proprietà testabile. Questo requisito è importante per il cliente.
- Feature software: insieme di funzioni che permettono di usare un prodotto software in un servizio. Questo è importante per il fornitore

## Dipendenze

Ogni prodotto software dipende da altri prodotti software che a loro volta dipende da altri software. Dobbiamo tenerne conto quando sviluppiamo il prodotto software perché nel momento in cui ci sono queste dipendenze ogni modifica dell'una influenza anche l'altra. Per far questo possiamo associare a ciascun prodotto software un grafo di dipendenza, in cui i nodi rappresentano i pacchetti software

### Quali tipi di dipendenze esistono?

- Dipendenze che noi controlliamo: un mio prodotto software dipende da un altro mio prodotto software
- Dipendenze che noi non controlliamo: relative alle terze parti che noi non possiamo controllare e ogni modifica di queste terze parti interviene anche sul nostro codice
- Dipendenze di dipendenze: andremo a dipendere da altri

## Il software rispetto ad altri prodotti industriali è speciale

- È invisibile e intangibile
- Ogni prodotto ha molte dipendenze
- È facilmente duplicabile e distribuibile su rete
- In Europa non è brevettabile, ma comunque protetto come opera dell'ingegno
- Il software di consumo non è garantito
- Viene acquisito su licenza
  - Che può essere proprietaria
  - Di pubblico dominio
  - Open source

## Protezione legale del software

Il software è un'opera dell'ingegno, chi lo produce è un autore che ha diritto ad un compenso e copiare il software abusivamente è illegale

### SIAE: pubblico registro del software

Possibile essere registrati i software che rispettino i requisiti di originalità e creatività tali da poter essere identificati come opere dell'ingegno

È possibile registrare tutti gli atti che trasferiscono tutto o in parte diritti di utilizzazione economica relativi a programmi per i quali sia già avvenuta la registrazione.

Per registrare un programma, il richiedente deve trasmettere a SIAE una dichiarazione e una descrizione, oltre ad un esemplare del programma da depositare registrato su supporto digitale non riscrivibile.

## La garanzia del software

Il compratore non ha alcuna garanzia rispetto ai difetti del prodotto, perché il software viene venduto così com'è e se ci sono difetti il fabbricante non se ne fa carico. Questo lo dice il contratto, quando si usa per la prima volta un'applicazione escono i termini di uso e tra i termini di uso c'è quello che il software lo prendi così com'è. Il fabbricante non si assume nessun obbligo di intervenire nella modifica del prodotto.

## Garanzie sul software

- Garanzie che si basa sulla verifica: la verifica è la garanzia che quel prodotto è conforme ad una data specifica
- Garanzie che si basa sulla validazione: la validazione consiste nell'accettazione da parte del cliente
- Garanzie che si basa sulla certificazione: la certificazione garantisce l'aderenza a certe specifiche date dalla legge

Questo non lo facciamo sul software commerciale che viene venduto senza garanzie ("as is")

## Altri problemi sul software

La qualità si basa molto su quelli che sono il numero di fault (difetti) rilevati durante l'uso del prodotto software

Altri problemi sono gli abbandoni

Il software è spesso inaffidabile

## Lezione 3

Necessità di applicare principi ingegneristici alla produzione software per sviluppare:

- Il giusto prodotto: fornisce le funzionalità che servono all'azienda stessa
- Al giusto costo: il costo non può essere troppo basso e nemmeno troppo alto
- Nel tempo giusto: ci vuole il tempo adeguato
- Con la giusta qualità: un costo basso non può avere una buona qualità

È ruolo del project manager individuare il giusto livello per ogni tipologia

### Scopo ingegneria del software

Fornire strumenti, metodi, metodologie per la costruzione di software di grandi dimensioni, di notevole complessità, sviluppati tramite lavori di gruppo.

Progetti che quindi hanno sempre: versioni multiple, lunga durata, frequenti cambiamenti che rientrano spesso nella manutenzione possono essere dovuti a necessità di eliminare difetti, adattamento a nuovi ambienti o introduzione di nuove funzionalità o miglioramenti delle funzionalità esistenti.

L'obiettivo finale di chi produce un sistema è quello di creare un prodotto che globalmente soddisfa i requisiti dell'utente.

La capacità di un ingegnere del software di riuscire ad interagire opportunamente non solamente con il team che si occupa del software ma con tutto il team che si occupa dell'intero sistema. Questo sistema di trova ad operare in un certo dominio applicativo

### Fondamenti ingegneria del software

L'ingegneria del software si occupa dei metodi, delle metodologie, dei processi e degli strumenti per la gestione professionale del software (sviluppo, manutenzione, ritiro). Questi metodi, queste metodologie, questi strumenti, si devono basare su dei principi fondamentali.

### Principi ingegneria del software

- Rigore: bisogna essere rigorosi, è un concetto primitivo
- Formalità: qualcosa che va oltre il rigore, si basa su fondamenti matematici

- Separazione di aspetti diversi: il problema complesso lo dobbiamo separare in varie parti per poterlo analizzare da vari punti di vista, questo ci aiuta a superare la complessità
- Modularità: suddividere un sistema complesso in parti più semplici
- Astrazione: processo con cui noi dal basso ci allontaniamo verso l'alto, ci serve per concentrarci solo sugli argomenti più importanti
- Anticipazione del cambiamento: la progettazione deve favorire l'evoluzione del software
- Generalità: tentare di risolvere il problema nella sua accezione più generale
- Incrementalità: lavorare per passi successivi

## Lezione 5

Tre modi per combattere la complessità:

- Astrazione: processo con cui noi dal basso ci allontaniamo verso l'alto, ci serve per concentrarci solo sugli argomenti più importanti
- Decomposizione: scomporre il problema in sottoproblemi più semplici
- Gerarchia: suddivisione per livelli

Decomposizione può essere funzionale e basata sull'object orientation

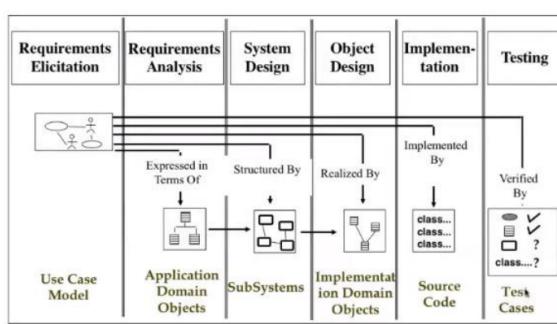
Rispetto alla funzionale, l'object orientation consente di portare a codice di solito più manutenibile

Inizieremo a fornire una descrizione delle funzionalità utilizzando lo Use case model, dopodichè arriveremo ad una progettazione object oriented andando ad individuare gli oggetti che dovranno far parte della nostra soluzione.

### Software Development Lifecycle

È l'insieme di attività e le loro relazioni che supportano lo sviluppo di un sistema software

#### *Software Lifecycle Activities*



- Requirements Elicitation, con la produzione dello *use case model* (ci si concentra sulle funzionalità e sul COSA realizzare con COME realizzarlo)
- Requirements Analysis, traduzione dello *use case model* in diagrammi UML (*object model*) per ottenere una comprensione chiara, completa e non ambigua dei requisiti e delle funzionalità (*Application domain object*)
- System Design, per la progettazione dell'architettura, individuare tutti i sottosistemi coinvolti
- Object Design, per specificare COME ogni sottosistema debba essere realizzato progettandolo in dettaglio (*Implementation domain object*)
- Implementation, per la scrittura del codice precedentemente progettato
- Testing, validazione dell'implementazione

## Primo passo nella ricerca dei requisiti: Identificazione del sistema

Il requirements process consiste di due attività principali:

- **Requirements Elicitation** (Raccolta dei requisiti): definizione del sistema in termini compresi dal cliente col documento PROBLEM DESCRIPTION/functional model. È la fase in cui si “sollecita” il nostro cliente affinché ci fornisca tutte le funzionalità che devono essere realizzate dal sistema. Elicitation infatti è da intendere come un lavoro dialettico tra l'analista e il cliente in cui c'è un passaggio di informazioni e chiarimenti finché non si arriva alla comprensione dei requisiti.
- **Requirements Analysis**: definizione del sistema in termini compresi dallo sviluppatore con il documento PROBLEM SPECIFICATION/analysis object model, che deve essere prodotto con correttezza, completezza e mancanza di ambiguità; è quindi scritto in un linguaggio formale (o semi) come UML

Problem description e problem specification rappresentano concettualmente lo stesso documento (a livello di contenuto), ma da due punti di vista diversi (uno informale per il cliente, l'altro formale per lo sviluppatore)

## System Specification vs Analysis Model

La system specification si differenzia dal modello di analisi perché mentre la system specification utilizza un linguaggio comprensibile dal nostro cliente che è normalmente il linguaggio naturale e questa system specification deriva dal problem statement, il modello di analisi usa linguaggi grafici che sono formali o semi-formali (di solito viene utilizzato UML)

Il punto di inizio di tutto è il problem statement

## Problem Statement – Statement of Work

Di solito il problem statement è sviluppato dal cliente.

Nel caso di prodotti specifici, il problem statement è fornito dal cliente, in cui descrive le caratteristiche principali del prodotto.

Nel caso di prodotti generici non c'è un cliente che ci fornisce il problem statement e quindi ce lo facciamo da soli.

Un buon problem statement deve descrivere:

- Situazione corrente
- Le funzionalità che il nuovo sistema deve supportare
- L'ambiente in cui il sistema sarà eseguito
- Deliverables attesi dal cliente, sono gli artefatti che vengono prodotti durante il processo di sviluppo, non solo per ragioni interne ma che vengono consegnate al cliente
- Date delle consegne dei vari deliverables, ma anche del prodotto finale
- Insieme dei criteri di accettazione, in base a cosa si decide se il prodotto è validato oppure no

La situazione corrente deve evidenziare anche il problema che deve essere risolto e questo deve essere fatto anche attraverso uno o più scenari.

Gli scenari ci consentono di descrivere sia la situazione corrente sia le funzionalità di cui abbiamo bisogno.

Queste funzionalità fanno parte di quelli che si chiamano requisiti funzionali ed insieme a questi requisiti, ci sono anche i requisiti così detti requisiti non funzionali.

### Tipi di requisiti

Requisiti funzionali: descrive le interazioni tra il sistema e il suo ambiente indipendente dall'implementazione

Requisiti non funzionali: sono delle caratteristiche che il sistema deve avere ma che non sono direttamente legati ad un comportamento funzionale. Significa che non è quello che mette a disposizione ma caratteristiche di solito di qualità

Pseudorequisiti/vincoli: imposti dall'utente o dall'ambiente in cui il sistema opera (cliente ci impone una certa tecnologia da utilizzare ecc.)

### Cosa non deve far parte dei requisiti?

Non deve far parte la struttura del sistema, la tecnologia di implementazione, la metodologia di sviluppo, l'ambiente di sviluppo, il linguaggio di implementazione.

È desiderabile che nessuno di questi sia vincolabile dal cliente. Ci sono delle situazioni in cui questo non possiamo evitarlo.

### Validazione dei requisiti

I requisiti devono subire un processo di validazione, avviene di solito dopo la fase di Requirements Analysis e i criteri che devono guidare la validazione dei requisiti sono quelli di:

- **Correttezza:** i requisiti rappresentano effettivamente quello che il cliente ha bisogno, quindi la vista del cliente e non qualcosa di diverso
- **Completezza:** i requisiti devono essere completi, tutte le possibili situazioni di uso sono descritte insieme ai comportamenti eccezionali tenuti sia dall'utente che dal sistema
- **Consistenza:** non ci devono essere due requisiti che si contraddicono
- **Realismo:** i requisiti possono essere concretamente implementati e realizzati
- **Tracciabilità:** per ogni funzionalità siamo in grado di individuare gli artefatti che sono collegati a quella funzionalità e viceversa; tenere traccia di quelli che sono i requisiti che sono relazionati tra di loro

I requisiti cambiano molto velocemente durante la fase di Requirements Elicitation

### Priorità dei requisiti

Si usa assegnare una priorità su questi tre livelli:

- **Alta priorità:** sono quei requisiti che ci aspettiamo che verranno effettivamente realizzati, dovranno essere consegnati nel sistema che consegneremo al nostro cliente
- **Media priorità:** sono quei requisiti che verranno consegnati in una release successiva, ma di norma non vengono implementati in una prima interazione
- **Bassa priorità:** vengono riportati soltanto nelle fasi di analisi

## Tipi di raccolta dei requisiti

- **Greenfield Engineering:** il sistema da realizzare è completamente nuovo, lo sviluppo inizia da zero, i requisiti devono essere derivati dal cliente, devono avere un trigger da parte delle esigenze del cliente
- **Re-engineering:** andare a riprogettare o reimplementare un sistema esistente. In questo caso è il sistema stesso che ci fornisce i requisiti, vogliamo avere le stesse funzionalità implementate in nuova tecnologia oppure che abbia una migliore progettazione, ma le funzionalità che vengono esposte ai clienti finali rimangono invariate e in questo caso possiamo utilizzare il sistema vecchio per derivare i requisiti
- **Interface Engineering:** non andiamo a modificare le funzionalità, ma queste stesse funzionalità le forniamo in un nuovo ambiente.

Abbiamo capito che durante la Requirement Elicitation bisogna far collaborare persone con background diversi: gli utenti con conoscenze nel dominio dell'applicazione (funzionalità che dovrebbe offrire), e gli sviluppatori con conoscenze nel dominio della soluzione (come implementare le funzionalità).

Per ridurre il gap tra utenti e sviluppatori noi utilizzeremo degli strumenti che si chiamano scenari e use case.

- Scenari: sono esempi di uso del sistema espressi come una serie di interazioni tra l'utente e il sistema
- Use case: rappresentano un'astrazione che descrive una classe di scenari

## Perché si utilizzano scenari e use case?

Perché sono strumenti che forniscono un linguaggio comprensibile all'utente finale, consentono di descrivere bene quelli che sono i requisiti funzionali

### Scenario

Uno scenario può essere definito come una descrizione narrativa di cosa le persone fanno e l'esperienza che hanno quando pensano di usare un prodotto software.

Quindi è una descrizione concreta, focalizzata, informale di una singola caratteristica del sistema che viene utilizzato

## Dove troviamo questi scenari?

Non ci aspettiamo che il cliente ci fornisca lo scenario e non ce lo aspettiamo anche se il sistema esiste.

Bisogna utilizzare un approccio dialettico, cioè che evolve nel tempo, che è incrementale.

- Voi aiutate il cliente a formulare i requisiti
- Il cliente vi aiuta a comprendere i requisiti
- Questo ciclo continua fino a quando tutto lo scenario non è ben sviluppato
- I requisiti evolvono mentre gli scenari vengono sviluppati

Euristica è una buona pratica che viene dall'esperienza. Nell'ambito dell'ingegneria del software non abbiamo teoremi, dobbiamo basarci sull'esperienza, sono delle buone pratiche di cui dobbiamo fare tesoro e cercare di applicarle

## EURISTICHE PER TROVARE SCENARI

È opportuno porsi certe domande:

- Quali sono i compiti principali che il sistema deve mettere a disposizione?
- Quali dati l'utente crea, memorizza, cambia, rimuove o aggiunge al sistema?
- Ci sono dei cambiamenti esterni che avvengono nell'ambiente e che devono essere notificati al sistema in maniera che adotti certi comportamenti?
- Di quali cambiamenti o eventi l'utente o il sistema devono essere informati?

Per acquisire queste informazioni dal nostro cliente non possiamo fare dei questionari

Un'operazione importante è la task observation, cioè andare sul campo, osservare come si lavora in quel campo.

## Lezione 6

- Scenario: È un esempio di uso del sistema in termini di una serie di interazioni tra gli utenti e il sistema che deve essere utilizzato
- Use case: Rappresenterà una classe di scenari

Per la Requirements Elicitacion useremo diversi approcci

| Tecnica  | Serve per   | Vantaggi   | Svantaggi   |
|--|---|--|---|
| Questionari                                      | Rispondere a domande specifiche.  | Si possono raggiungere molte persone con poco sforzo.  | Vanno progettati con grande accuratezza, in caso contrario le risposte potrebbero risultare poco informative.<br><br>Il tasso di risposta può essere basso. |
| Interviste individuali                           | Esplorare determinati aspetti del problema e determinati punti di vista.                    | L'intervistatore può controllare il corso dell'intervista, orientandola verso quei temi sui quali l'intervistato è in grado di fornire i contributi più utili. | Richiedono molto tempo.<br><br>Gli intervistati potrebbero evitare di esprimersi con franchezza su alcuni aspetti delicati.                                 |
| Focus group                                      | Mettere a fuoco un determinato argomento, sul quale possono esserci diversi punti di vista. | Fanno emergere le aree di consenso e di conflitto.<br><br>Possono far emergere soluzioni condivise dal gruppo.   | La loro conduzione richiede esperienza.<br><br>Possono emergere figure dominanti che monopolizzano la discussione.  |
| Osservazioni sul campo                           | Comprendere il contesto delle attività dell'utente.   | Permettono di ottenere una consapevolezza sull'uso reale del prodotto che le altre tecniche non danno.   | Possono essere difficili da effettuare e richiedere molto tempo e risorse.  |
| Suggerimenti spontanei degli utenti              | Individuare specifiche necessità di miglioramento di un prodotto.                           | Hanno bassi costi di raccolta.<br><br>Possono essere molto specifici.  | Hanno normalmente carattere episodico.  |
| Analisi della concorrenza e delle best practices | Individuare le soluzioni migliori adottate nel settore di interesse                         | Evitare di "reinventare la ruota" e ottenere vantaggio competitivo   | L'analisi di solito è costosa (tempo e risorse)   |

## Use case

Permettono di ridurre il *gap* tra sviluppatore e utente finale. Sono un importante strumento per la gestione dei progetti e vengono utilizzati in maniera trasversale durante tutto il processo di sviluppo.

**Uno use case rappresenta in generale una funzionalità messa a disposizione dal sistema.**

Gli use cases furono introdotti da Ivar Jacobson per la *reingegnerizzazione dei processi di business*: un processo di business è un insieme di attività effettuate da un'organizzazione per uno specifico scopo.

es: "riconoscimento attività lavorativa"

Io studente invia la richiesta > la segreteria prende in carico > trasmette la richiesta al consiglio didattico > il consiglio analizza > il consiglio informa lo studente

Jacobson usava gli use cases per andare a modificare questi business process, aggiungendo/eliminando/modificando alcuni passaggi, per ottenere lo stesso scopo per cui era stato sviluppato il processo di business. (reengineering)

## Tipi di scenari

- **Scenari as-is:** sono com'è, usati per descrivere la situazione attuale
- **Scenari visionari:** usato per descrivere un sistema futuro, usato per descrivere un sistema che non è ancora stato realizzato, possono essere realizzati dall'utente o dallo sviluppatore a seconda della situazione
- **Scenari di valutazione:** serve per descrivere quelli che sono i task dell'utente rispetto al quale il sistema deve essere valutato, quindi indico come vorrei che fosse e vado a verificare che il sistema realizzato si comporta come io ho descritto nello scenario di valutazione
- **Scenari di training:** viene fatto per fare il training di utenti novizi

Dopo che gli scenari sono stati formulati:

Raggruppiamo i vari scenari in uno use case

- Come prima cosa gli diamo un nome

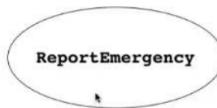
Per descrivere gli use case cerchiamo di analizzare le varie informazioni presenti nello scenario andando ad individuare alcuni dettagli:

- **Entry condition:** rappresenta la condizione che deve essere soddisfatta affinchè un certo use case possa essere attivato (piano di studi: l'utente deve essere registrato su esse3)
- **Exit condition:** condizione affinchè uno use case possa essere terminato (il piano di studi viene sottomesso)
- **Flusso di eventi:** rappresenta tutto quello che è in dialogo che avviene tra la entry condition e la exit condition (piano di studi: scelta dei 6 cfu, scelta dei 18 cfu, scelta attività extra, ecc)
- **Eventuali eccezioni:** gestire e trattare errori o situazioni inattese con flussi di esecuzione diversi (piano di studi: scegliere come trattare studenti con tasse non regolari, studenti fuori corso)
- **Requisiti speciali (vincoli, requisiti non funzionali):** requisiti speciali da soddisfare (piano di studi: questo deve essere possibile farlo con tempo non superiore a un tot, questa funzionalità deve essere particolarmente facile da utilizzare)

## Use case

**Uno use case è una funzionalità che il sistema mette a disposizione e che ha valore per un attore**

- Uno use case può essere visto come un flusso di eventi nel sistema che include l'interazione con gli attori.
- È iniziato con l'attore
- Ogni use case deve avere un nome, deve essere un verbo, che ricorda la funzionalità
- Ogni use case ha una condizione di terminazione
- Rappresentato dal punto di vista grafico con un ovale con il nome dello use case all'interno, **lo use case rappresenta una funzionalità messa a disposizione dal sistema vista dal punto di vista dell'attore, da chi usa quella funzionalità**



**Use Case Model:** è l'insieme di tutti gli use case che specificano le funzionalità complete del sistema

## Lezione 8

### Glossario

Una specie di dizionario, tutti i termini nuovi di quel dominio andranno inseriti e ci verrà come in un dizionario una specie di descrizione

### EURISTICHE PER TROVARE USE CASE

Consideriamo uno scenario, lo discutiamo con gli utenti per capire lo stile di interazione preferito, selezioniamo vari scenari che si riferiscono alla stessa funzionalità, discutiamo di questi scenari con gli utenti finali facendoci supportare dai mock-up che rappresentano i prototipi della nostra interfaccia.

Ci possiamo far aiutare dalla task observation o dai questionari anche se non è il modo preferito per procedere

- Primo passo: dargli un nome, questo nome deve avere un'azione all'interno, quindi un verbo
- Secondo passo: trovare gli attori, cioè gli utenti che interagiscono con il sistema, specificano qual è il ruolo che hanno in questa interazione
- Terzo passo: definire flusso di eventi utilizzando il linguaggio naturale

### Linee guida per la formulazione degli use case

- Ogni use case deve essere nominato con un verbo significativo nel contesto
- Gli use case devono avere una lunghezza adeguata, non devono eccedere le ½ pagine

- Uno use case deve descrivere un'iterazione completa, un output definito
- Il flusso di eventi di uno use case può essere descritto in un linguaggio naturale anche molto narrato, ma utilizzeremo approccio più strutturale
  - Utilizzare la voce attiva (relazione causa-effetto ben chiara per non lasciare ambiguità)
  - Non dobbiamo limitarci al flusso di eventi principale
  - Capire quali sono i limiti del sistema (boundary -> confine del sistema), tutto quello che deve essere realizzato da noi e tutto quello che non fa parte del sistema che dovremmo realizzare
  - Tutti i termini nuovi del dominio li andiamo a definire nel glossario

## Associazioni tra use case

Servono per ridurre la complessità, quando ad esempio uno use case è troppo lungo, riduciamo il flusso di eventi attraverso l'invocazione di altri use case; oppure possono essere dei flussi di evento alternativi; o per un raffinamento degli use case attraverso una specializzazione degli use case

Per la decomposizione utilizziamo l'associazione di tipo include

Per esprimere dei flussi eccezionali, alternativi, utilizziamo l'extend

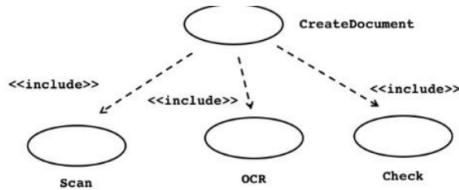
Per raffinare gli use case astratti e quindi rappresentare astrazioni e specializzazioni di use case utilizziamo la generalizzazione

Uno use case model consiste di tutti gli use case e le relazioni tra use case che sono di questi tre tipi

### **<<Include>> ci serve per la decomposizione funzionale**

Risolve il seguente problema: nella formulazione originale è troppo complesso e lungo e quindi lo andiamo a scomporre

Descriviamo la funzione come l'aggregazione di funzioni più semplici



### <<Extend>>

Non è una invocazione funzionale, con l'extend andiamo a descrivere comportamenti che avvengono in maniera imprevista in certi punti del flusso di eventi e continueranno su quel flusso



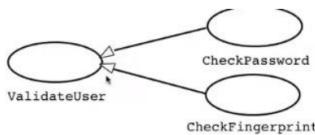
### Extend vs Include

- Include
  - All'interno del flusso di eventi nel caso di uso base ci deve essere l'invocazione in qualche punto del caso d'uso in uso
- Extend
  - Capiamo che il caso d'uso che estende si deve attivare quando vediamo nella condizione di attivazione che è presente nella entry condition dello use case che estende
  - Rappresentano situazioni eccezionali (fallimenti, ecc)

### Generalizzazione

Spesso utilizzata nelle ultime fasi di descrizione del modello degli use case.

È quel processo per cui ci rendiamo conto che ci sono delle situazioni che possono essere generalizzate in uno use case che rappresenta entrambi questi use case



Il caso d'uso padre è una generalizzazione dei casi d'uso figli. Quindi il flusso di eventi dei casi d'uso figli va a sovrascrivere il flusso di eventi del caso d'uso padre

## Lezione 9

### Flussi di eventi alternativi/errore vs Extends

Si usano i flussi alternativi quando:

- Conosciamo il punto preciso di diramazione: si specifica nel documento il punto in cui potrebbe avvenire l'eccezione e come il sistema reagisce
- Il flusso alternativo risulta breve e non troppo complesso

Si usa l'extend quando:

- La diramazione può avvenire in un punto qualsiasi del flusso principale
- Il flusso di eventi che viene fuori non è piccolo, non è minimale rispetto al flusso di eventi principale
- Potrebbe essere che quel tipo di flusso alternativo vada bene non solo per questo use case ma anche per altri use case

## Lezione 10

### Requisiti

- Requisiti funzionali: sono le funzionalità che mette a disposizione il sistema e che verranno utilizzati da qualche utente
- Requisiti non funzionali: non sono delle funzionalità ma forniscono delle caratteristiche di qualità al sistema
- Vincoli: pongono delle descrizioni su come queste funzionalità debbano essere realizzate

Per fare la raccolta dei requisiti partiamo dagli scenari e quando avremo raggiunto un certo numero di scenari per certe funzionalità passeremo alla definizione degli use case che andranno ad astrarre vari scenari. Assoceremo agli use case anche un diagramma degli use case che fungerà da indice perché fornisce le funzionalità principali e quelli che sono gli attori principali. Gli attori rappresentano coloro che interagiscono con il sistema: possono essere utenti finali o anche altri sistemi.

Requisiti rappresentano una descrizione testuale di quello che mette a disposizione il sistema devono essere scritti bene, devono essere ben formati

- Devono essere verificati
- Devono essere misurabili: ha a che fare con requisiti non funzionali
- Espressi in linguaggio naturale strutturato semi-formale: è bene evidenziare soggetto - verbo - complemento, usiamo il verbo per descrivere anche il livello di priorità
  - Le priorità si possono indicare con il verbo che si utilizza:
    - Deve: requisiti obbligatori
    - Dovrebbe: preferenze, desideri non obbligatori
    - Può: suggerimento, non obbligatorio
- Evitiamo frasi negative e utilizzare forme attive
- Evitare sinonimi, essere consistenti

Secondo lo standard IEEE i requisiti dovrebbero essere espressi secondo questa sintassi

♦ [Condition][Subject][Action][Object][Constraint]

- ♦ When signal x is received [Condition], the system [Subject] shall set [Action] the signal x received bit [Object] within 2 seconds [Constraint].

Un requisito deve essere:

- Necessario: definisce delle caratteristiche essenziali, se lo eliminate togliete una cosa essenziale non offerto da altre funzionalità, non deve essere obsoleto
- Implementation free: il requisito è su cosa deve essere fornito e non sul come deve essere fornito
- Non ambiguo: ci deve essere un'unica interpretazione e deve essere facile da comprendere
- Consistente: non deve essere in conflitto con altri requisiti
- Completo: descrive in maniera sufficiente una certa caratteristica e misurabile (misurabile: prenotazione posto a mensa deve poter avvenire ad al più due click)
- Singolare: ci deve essere un unico requisito, non congiunzioni
- Ammissibile: deve essere realizzabile sulla base dei vincoli che mi sono stati posti
- Tracciabile: in avanti e in indietro, cioè se parto dal requisito riesco a dire in quale modulo del sistema è preso in considerazione, poi nel modulo in quale componente dell'object design è preso in considerazione, poi qual è il codice corrispondente all'implementazione di questo requisito, quali sono i test case che vanno a testare questo requisito, quindi devo poter da questo requisito riuscire ad individuare tutti gli artefatti che sono relativi a questo requisito, in indietro invece parto dal test case e arrivo al requisito
- Verificabile

L'insieme dei requisiti invece deve essere:

- Completo: tutti i requisiti che sono stati descritti rappresentano tutte le esigenze
- Consistente: non ci devono essere requisiti in conflitto nell'insieme

- Affordable: possono essere realizzati da una soluzione ottenibile all'interno del ciclo di vita, i vincoli imposti sul sistema
- Bounded: l'insieme rimane all'interno delle necessità del sistema, senza allargarsi aggiungendo requisiti che non rispondono ad esigenze del cliente

## Matrice di tracciabilità

Un modo per realizzare la tracciabilità tra gli artefatti di un prodotto software è quello di utilizzare una matrice di tracciabilità

Ci sono vari modi per implementarla

Nei requisiti è importante evitare termini come

- Superlativi: è il migliore
- Termini che possono essere soggettivi: facili da usare
- Pronomi vaghi: suo, questo, quello
- Avverbi e aggettivi ambigui: significativo, minimale
- Termini non verificabili: fornire supporto, come minimo, non limitato a..
- Frasi comparative: meglio di, di una migliore qualità
- Espressioni: se possibile, per quanto appropriato, per quanto applicabile
- Evitare statement negativi

## Tipo funzionale vs non funzionale

Un requisito funzionale è l'affermazione di una qualche funzione che deve essere implementata in un sistema **[cosa deve essere realizzato?]**

Un requisito non funzionale è l'affermazione di un vincolo che si applica ad un sistema **[quanto velocemente? Quanti fallimenti? Quanto accurato? Quanto sicuro deve essere il nostro sistema?]**

Requisito funzionale e non funzionale costituiscono quello che può essere visto come un accordo tra il cliente e l'ingegnere del software

### Attributi di qualità popolari

- affidabilità: intendiamo disponibilità (del servizio offerto), fault tolerance (capacità di recupero di un fault), recuperabilità (tempo di recupero da un fault)
- performance: intendiamo tempo di risposta, utilizzo efficiente di risorse
- operabilità: intendiamo la facilità d'uso, l'estetica dell'interfaccia utente, ...
- sicurezza
- compatibilità: interoperabile con un sistema esistente o da realizzare
- manutenibilità: facilmente aggiungere nuove funzionalità, facile da modificare, facile da testare, facile da riutilizzare
- portabilità: installare il sistema in un nuovo ambiente

Spesso lo stesso requisito può essere espresso come requisito funzionale e non funzionale

## Lezione 11

### FURPS+: categorie di requisiti non funzionali

Il modello FURPS+ fornisce un'organizzazione dei requisiti non funzionali e dei pseudorequisiti (vincoli). In particolare considera come requisiti non funzionali le seguenti categorie:

- usabilità: tutto quello che può aiutare l'utente ad utilizzare al meglio il prodotto software e che possono aiutare a imparare ad utilizzare il sistema e a farlo senza commettere errori
- affidabilità: prende in considerazione la capacità del sistema di eseguire le funzioni sotto delle condizioni specificate per un tempo specificato
  - dependability
  - robustezza: tolleranza ai guasti, a input non previsti
  - safety: capacità del sistema di non recare danno a cose o persone
- performance: tutti i requisiti che hanno a che fare con il tempo di risposta, throughput, disponibilità di certe informazioni e accuratezza.
  - Accuratezza: la precisione con cui ci vengono forniti certi risultati
  - Throughput: quante operazioni riesce al fare
- Supportabilità: requisiti che hanno a che fare con la facilità di cambiamento dopo che è avvenuta la consegna del sistema ed include la manutenibilità (capacità di apportare modifiche) e che a sua volta include l'adattabilità (capacità di adattarsi a diversi ambienti hardware e software) a nuovi ambienti

### FURPS+: categorie di pseudo requisiti

- **Requisiti di implementazione** se pongono vincoli su come deve essere realizzato il sistema (uso di linguaggi di programmazione, uso di diverse piattaforme hardware)
- **Vincoli di interfaccia** sono vincoli su interfaccia con sistemi esterni, che potrebbero essere legacy cioè ereditati e hanno a che fare con il formato dello scambio dei dati
- **Requisiti operativi** che includono la tipologia di gestione e amministrazione che deve essere fatta
- **Requisiti di packaging** come dovrà avvenire la consegna del sistema, che strumenti sono necessari per l'installazione del sistema

- **Requisiti legali** vincoli di certificazioni legali, rispetto delle leggi ecc..

## EURISTICHE REQUISITI NON FUNZIONALI

### Usabilità:

- Qual è il livello di esperienza dell'utente?
- A quali standard di interfaccia utenti sono abituati?
- Quale documentazione sarebbe utile fornire all'utente?

### Reliability (include robustezza, safety e security):

- Come affidabile, disponibile e robusto deve essere il sistema?
- È accettabile che riavvio il sistema dopo che c'è stato un fallimento oppure il comportamento deve essere diverso?
- Fino a che punto è accettabile la perdita dei dati?
- Come dovrebbero essere gestite le eccezioni?
- Quali requisiti in termini di sicurezza delle persone?
- Quali requisiti in termini di accessi non autorizzati?

Ecc....

| <i>Identifying nonfunctional requirements (1)</i>                      |   |
|--|---|
| <b>Usability</b>   | What is the level of expertise of the user?<br>What user interface standards are familiar to the user?<br>What documentation should be provided to the user?  |
| <b>Reliability<br/>(including robustness, safety, and security)</b>    | How reliable, available, and robust should the system be?<br>Is restarting the system acceptable in the event of a failure?<br>How much data can the system loose?<br>How should the system handle exceptions?<br>Are there safety requirements of the system?<br>Are there security requirements of the system ? |
| <b>Performance</b>   | How responsive should the system be?<br>Are any user tasks time critical?<br>How many concurrent users should it support?<br>How large is a typical data store for comparable systems?<br>What is the worse latency that is acceptable to users?  |
| <b>Supportability<br/>(including maintainability and portability )</b> | What are the foreseen extensions to the system?<br>Who maintains the system?<br>Are there plans to port the system to different software or hardware environments?  |

  105

### ***Identifying nonfunctional requirements (2)***

|                       |  |
|-----------------------|--|
| <b>Implementation</b> | <p>Are there constraints on the hardware platform?</p> <p>Are constraints imposed by the maintenance team?</p> <p>Are constraints imposed by the testing team?</p>   |
| <b>Interface</b>      | <p>Should the system interact with any existing systems?</p> <p>How are data exported/imported into the system?</p> <p>What standards in use by the client should be supported by the system?</p>          |
| <b>Operation</b>      | <p>Who manages the running system?</p>   |
| <b>Packaging</b>      | <p>Who installs the system?</p> <p>How many installations are foreseen?</p> <p>Are there time constraints on the installation?</p>   |
| <b>Legal</b>          | <p>How should the system be licensed?</p> <p>Are any liability issues associated with system failures?</p> <p>Are any royalties or licensing fees incurred by using specific algorithms or components?</p> |



## **Nonfunctional Requirements (Questions to overcome “Writers block”)**

User interface and human factors

- What type of user will be using the system?
  - Will more than one type of user be using the system?
  - What training will be required for each type of user?
  - Is it important that the system is easy to learn?
  - Should users be protected from making errors?
  - What input/output devices are available?

## Documentation

- What kind of documentation is required?
  - What audience is to be addressed by each document?

## Nonfunctional Requirements (2)

### Hardware considerations

- What hardware is the proposed system to be used on?
- What are the characteristics of the target hardware, including memory size and auxiliary storage space?

### Performance characteristics

- Are there speed, throughput, response time constraints on the system?
- Are there size or capacity constraints on the data to be processed by the system?

### Error handling and extreme conditions

- How should the system respond to input errors?
- How should the system respond to extreme conditions?

## Nonfunctional Requirements (3)

### System interfacing

- Is input coming from systems outside the proposed system?
- Is output going to systems outside the proposed system?
- Are there restrictions on the format or medium that must be used for input or output?

### Quality issues

- What are the requirements for reliability?
- Must the system trap faults?
- What is the time for restarting the system after a failure?
- Is there an acceptable downtime per 24-hour period?
- Is it important that the system be portable?

## Nonfunctional Requirements (4)

### System Modifications

- What parts of the system are likely to be modified?
- What sorts of modifications are expected?

### Physical Environment

- Where will the target equipment operate?
- Is the target equipment in one or several locations?
- Will the environmental conditions be ordinary?

### Security Issues

- Must access to data or the system be controlled?
- Is physical security an issue?

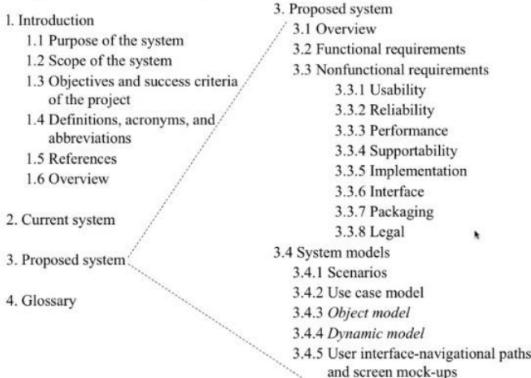
## Nonfunctional Requirements (5)

### Resources and Management Issues

- How often will the system be backed up?
- Who will be responsible for the back up?
- Who is responsible for system installation?
- Who will be responsible for system maintenance?

## RAD risultato della raccolta e analisi dei requisiti

### ***Requirements Analysis Document***



- **Glossario:** sorta di vocabolario, definire tutti i termini del dominio del problema che è importante conoscere per capire questi requisiti
- **Object model e dymanic model:** fanno parte della fase di analisi, mentre tutto il resto della fase di requirements elicitation
- **Purpose of the system:** obiettivi di business che il sistema deve soddisfare, per cosa viene realizzato questo sistema
- **Scope of the system:** fornisce il confine, quali attività fornisce il sistema
- **Criteri di accettazione:** andremo a definire quando questo progetto avrà raggiunto il successo, in base a quale criterio sarà valutato il successo o meno del progetto
- **acronimi, abbreviazioni, sigle** che noi utilizzeremo all'interno del documento per aiutarci nella comprensione del documento stesso
- **Referenze:** link a altri documenti che è importante conoscere e che vengono citati per la comprensione di questo documento
- **Panoramica:** si dice come è organizzata la parte restante del documento
- Nella sezione 2 viene descritto il sistema attuale, ponendo attenzione ai limiti del sistema sw esistente. (se non c'è sistema sw, descrivere l'attività manuale e come il nostro sw può rimpiazzarla)
- Nella sezione 3 viene descritto il sistema proposto
  - 3.1 Overview: descrizione di come la sezione 3 è organizzata
  - 3.2 requisiti funzionali: o elencare i requisiti o un link ad un excel con i requisiti riportati

- 3.3 requisiti non funzionali secondo FURPS+: stessa cosa dei funzionali
- 3.4 modelli del sistema
  - Scenari
  - Use case
  - ...
  - ...

## Activity Diagrams: linguaggio UML

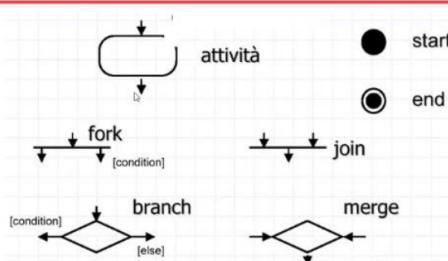
Questo diagramma fornisce una sequenza di operazioni la realizzazione di attività complesse. E' possibile rappresentare dei processi paralleli e la loro sincronizzazione.

(Activity Diagram può essere considerato un'estensione del flow chart e si basa sulla scomposizione funzionale)

Un activity diagram può essere associato ad una classe, all'implementazione di un'operazione, ad uno use case, a tutto il sistema, ai *processi di business*.

Li utilizziamo per rappresentare processi di business e path navigazionali del sistema  
**Nel RAD lo utilizzeremo per descrivere current system overview, proposed system overview e path navigazionali**

## Elementi Grafici



*Le attività possono essere gerarchiche*

- Activity: rappresenta una qualche operazione che deve essere fatta che non è atomica, cioè non istantanea, richiede un certo tempo. Possono essere a loro

volta scomposte in altre attività e queste attività possono essere se stesse delle activity diagram

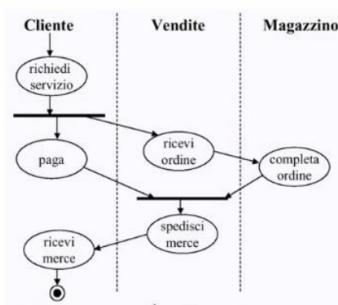
- Transizioni: rappresentano il flusso tra due action successive
- Espressioni di guardia: espressioni booleane, delle condizioni che devono essere verificate per poter attivare una transizione
- Branch: con cui descriviamo percorsi alternativi
- Barre di sincronizzazione: fork e join

## Swimlanes

Costrutto grafico che rappresenta un partizionamento delle activity e individua chi ha la responsabilità relativamente a certe attività. Sono particolarmente importanti quando è che vogliamo descrivere i processi di business perché identificano quali sono le unità organizzative

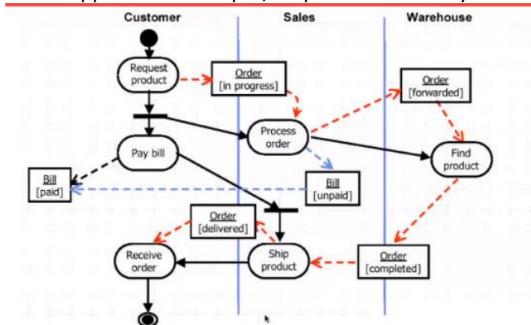
Si assegna uno spazio dell'activity diagram a ciascun attore e per ciascuno spazio le attività di quello specifico attore.

### Esempio: Swimlanes



## Object flow

All'interno di questi diagrammi possiamo andare a considerare gli oggetti: object flow. Rappresentano l'input/output di una activity.



## Lezione 12

### Requirements Elicitation

L'analista dei requisiti deve identificare il problemaopportunità andando a identificare quelli che sono i confini del problema:

- Quale problema deve essere risolto?
- In quale contesto? (problem domain)
- Di chi è il problema? (identificare stakeholders)
- Perché è necessario risolvere questo problema? (quali sono gli obiettivi degli stakeholder?)
- Come questo problema può essere supportato da un sistema software? (collezionando degli scenari appropriati)
- Entro quando questo problema deve essere risolto? (definiamo vincoli di sviluppo)
- C'è qualcosa che mi impedisce di risolvere questo problema? (quali potrebbero essere i rischi principali e se questo problema che mi è stato assegnato può essere ammissibile, realizzabile oppure no)

Tutti questi elementi sono presenti nello SOW, tranne la valutazione dei rischi (oggetto di valutazione successiva nel documento di *business case* in cui si decide se avviare il progetto)

### Tecniche di Elicitation

- Interviste: con una persona sola o più persone
- Osservazione sul campo
- Meeting con cliente o utente: si mostrano prototipi, che possono essere scenari, aiutati con mock-up dell'interfaccia
- Brainstorming: obiettivo è far venir fuori tutte le idee di tutti, si utilizza quando abbiamo a che fare con problemi difficili, nuovi, obiettivo far venir fuori nuovi modi di pensare

### Interviste

Possono essere semi-strutturate con domande più o meno aperte oppure domande aperte.

Si possono collezionare molte informazioni, si può avere anche il feeling, si può scoprire le opinioni che diversi attori possono avere sul prodotto, possono far entrare in profondità e possono aiutare ad adattare le domande in base a quello che è il risultato delle risposte finora raggiunte, cosa che con il questionario non è

possibile fare. Vengono fuori un gran numero di dati qualitativi (non misurabili, senza numeri quindi difficili da categorizzare, catalogare, confrontare e analizzare)

## Meeting

Sono costosi in termini di tempo, riservati per certi momenti, per avere dei feedback, utile per lavoro di sintesi che informa tutti sullo stato di uno stadio, quindi a che punto siamo e utile alla fine di ogni stadio, in maniera tale che si discute dei risultati della raccolta di informazioni che si sono realizzate, concludere un insieme di requisiti. Alla fine del meeting mi aspetto di aver raggiunto il risultato sperato. Durante il meeting bisogna tener conto dell'agenda e tener conto dell'orario di inizio e di fine ed evitare di sfornare. Alla fine del meeting ci sarà la minuta distribuita a tutti coloro che hanno partecipato. Durante il meeting c'è il facilitatore che si avvale del timekeeper, colui che tiene traccia del tempo e ci sta colui che si occupa della minuta che darà al facilitatore la bozza del verbale e il facilitatore una volta che l'ha sistemata la invierà a tutti i partecipanti.

## Requisiti funzionali

Questo insieme di requisiti dovrà essere in grado di fornirci le seguenti informazioni:

- **Perché** viene realizzato il sistema
- **Per** chi sarà utile
- **Cosa** sarà realizzato in termini di funzionalità

## Lezione 13

### User stories

Hanno un formato del tipo



Sono simili ai requisiti visti finora, ma si focalizzano di più su quelli che sono gli utenti e sui benefici che gli utenti possono ricavare

È lo strumento principale utilizzato nei **metodi agili**

Possono essere utilizzate sia per i requisiti funzionali e non

### EURISTICHE PER COMPLETEZZA DEI REQUISITI

- Per ogni entità se gli utenti hanno bisogni di cercare un'entità all'interno di una collezione
- Devono creare una nuova entità
- Leggere un'entità esistente
- Aggiornare una o più entità
- Eliminare una o più entità

## Lezione 14

### Linee guida per use case

- I nomi dei casi d'uso devono includere verbi e devono richiamare esattamente l'operazione che l'utente cerca di ottenere
- I nomi degli attori devono essere dei sostantivi perché rappresentano dei ruoli all'interno e nell'uso del sistema
- I confini del sistema dovrebbero essere chiari (definire cosa è di nostra competenza e cosa non lo è)
- Le relazioni causali tra passi successivi dovrebbero essere chiari (ci riferiamo alla descrizione degli use case perché abbiamo un dialogo, l'attore fa una richiesta e il sistema risponde; il sistema si comporta in quel modo sulla base di una specifica richiesta)
- Non bisogna usare la forma passiva perché consente di omettere il soggetto che compie una determinata azione e può risultare ambiguo
- Un caso d'uso dovrebbe descrivere una transizione utente completa (il flusso degli eventi principali dovrebbe essere tale che partendo dalla entry condition raggiungiamo la exit condition)
- Le eccezioni dovrebbero essere descritte separatamente, tramite flussi alternativi eccezioni semplici oppure tramite l'estensione dei casi d'uso.
  - Il flusso alternativo è più utile quando l'eccezione è più piccola
  - L'estensione è più utile quando l'eccezione è più grande e quando potrebbe essere utile anche da qualche altra parte, ma non sappiamo quando questa extend si attiverà
- Un caso d'uso non dovrebbe descrivere un'interfaccia del sistema (meglio attraverso prototipi mock-up)
- Un caso d'uso non dovrebbe superare le 2/3 pagine, altrimenti usiamo l'include o l'extends a rendere il caso d'uso più piccolo

### Team contract

Per organizzare una squadre (insieme di persone che devono raggiungere un obiettivo comune) è necessario stabilire delle regole: lo si fa con il **Team Contract**

## Analisi dei Requisiti

Obiettivo è fornire un modello del sistema che sia:

- Corretto: che fa quello che serve effettivamente
- Completo: che tutto quello che serve è descritto
- Consistente: non ci sono delle parti che si contraddicono l'un l'altro
- Non ambiguo: chi prenderà questo modello per farne la progettazione non si troverà a decidere qual è il significato e come deve essere fatto il sistema, ma quale funzionalità dovrà mettere a disposizione il sistema, non dà luogo a altre interpretazioni

Gli sviluppatori

- Andranno a formalizzare i requisiti prodotti durante la fase di raccolta dei requisiti
- Andranno a validare, correggere, chiarire dei dubbi, eventualmente coinvolgendo il cliente o l'utente finale
- Andranno ad esaminare più in dettaglio le condizioni limite e i casi eccezionali

In questa fase viene prodotto il modello ad oggetti che descrive il dominio del problema: il **Class Diagram**

## Concetti di Analisi

- **Il modello degli oggetti di analisi**
  - rappresenta il sistema dal punto di vista dell'utente
  - si focalizza sui concetti che sono manipolati dal sistema, le loro proprietà e relazioni
  - è un dizionario visuale dei concetti principali visibili all'utente
  - l'UML class diagram include classi, attributi e operazioni
- **Il modello dinamico**
  - si focalizza sul comportamento del sistema
    - I sequence diagram rappresentano le interazioni tra un insieme di oggetti durante un singolo use case
    - Gli statechart rappresentano il comportamento di un singolo oggetto o di alcuni oggetti strettamente accoppiati
  - consente di assegnare le responsabilità alle classi e quindi individuare nuove classi che sono aggiunte al modello degli oggetti dell'analisi
- **Ricorda:** sia il modello dinamico che il modello degli oggetti rappresentano concetti a livello utente non a livello di componenti e classi software
  - Le classi di analisi rappresentano astrazioni che saranno realizzati con molti più dettagli successivamente

## Lezione 15

### Class Diagram

Definisce la visione statica del sistema

- Classi (di oggetti)
- Relazioni tra classi

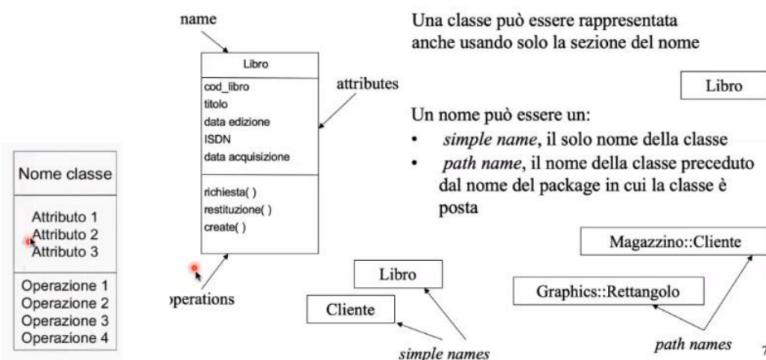
Oggetto: è una rappresentazione di un qualcosa di reale o astratto, ha delle caratteristiche ed uno stato

Classe: rappresenta un'astrazione di oggetti con caratteristiche comuni

### Classi in UML

In UML una classe è composta da tre parti

- Nome
- Attributi (lo stato)
- Metodi o operazioni (il comportamento)



## Attributi

Un attributo è una proprietà statica di un oggetto

- Nome, età, peso sono attributi della classe Persona

L'attributo contiene dei valori per ogni istanza

I nomi degli attributi devono essere unici

Il valore di un attributo non ha un'identità

Per ciascun attributo si può specificare il tipo ed un eventuale valore iniziale

Durante la scrittura degli attributi non specificare l'id, poiché gli oggetti hanno già una loro identità

Candidati per attributi

- Nomi che non sono diventati classi

## Attributi derivati

Una serie di informazioni che si possono ricavare da altre già specificate

|                    |
|--------------------|
| Persona            |
| dataNascita : Date |
| età : int          |

{età = oggi - dataNascita}

## Operazioni

Sono i comportamenti che gli oggetti devono mettere a disposizione per realizzare la nostra soluzione

Tipi di operazione:

- Query: accedono all'informazione dell'oggetto e non le alterano

- Modificatore: cambiano lo stato dell'oggetto

Operazioni che riguardano la creazione e distruzione

- Costruttore: crea un nuovo oggetto e inizializza il suo stato
- Distruttore: distrugge un oggetto e libera il suo stato

Per ciascuna operazione si può specificare il nome, il tipo, i parametri...

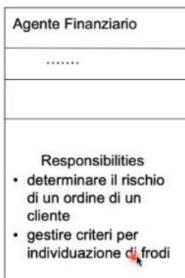
È possibile categorizzare le operazioni e lo si fa con gli stereotipi



Gli stereotipi aggiungono della semantica al Class Diagram rendendolo più comprensibile, organizzando le operazioni sotto delle categorie

## Responsabilità

Una responsabilità è un contratto di una classe (una classe ben strutturata ha poche responsabilità)



## Visibilità

- + (public)
- # (protected)
- - (private)

|                 |
|-----------------|
| Libro           |
| # cod_libro     |
| - titolo        |
| - data edizione |
| - ISDN          |
| + richiesta( )  |
| restituzione( ) |
| + create( )     |

## Molteplicità

È usata per indicare il numero di istanze di una classe

- Una molteplicità pari a zero indica una classe astratta
- Pari a 1 è una singleton class
- Per default è assunta una molteplicità maggiore di uno

La molteplicità è indicata con un numero intero posto nell'angolo in alto a destra del simbolo della class

|                          |   |
|--------------------------|---|
| NetworkController        | 1 |
| consolePort [2..*]: Port |   |
|                          |   |

## Specifica attributi

Per specificare un attributo in UML la sintassi è

[visibility] name [ [multiplicity] ] [: type] [= initial-value] [{property-string}]

Property-string può assumere uno dei seguenti valori:

- Changeable: nessuna limitazione per la modifica dell'attributo
- addOnly: per attributi con molteplicità maggiore di 1 possono essere aggiunti ulteriori valori, ma una volta creato un valore non può essere né rimosso né modificato
- frozen: il valore non può essere modificato dopo la sua inizializzazione

## Specifica operazioni

La sintassi per specificare un'operazione in UML è

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

Property-string può assumere uno dei seguenti valori:

- isQuery: l'esecuzione dell'operazione lascia lo stato del sistema immutato
- sequential
- guarded
- concurrent

La lista dei parametri ha questa sintassi

[direction] name: type [=default-value]

Dove direction può assumere uno di questi valori:

- in: parametro di input
- out: parametro di output
- inout: parametro di input/output

## Legami e associazioni

Un legame (link) indica una relazione (fisica o concettuale) tra oggetti la cui conoscenza deve essere preservata per un certo periodo di tempo.

Un'associazione invece descrive un gruppo di legami aventi struttura e semantica comuni

Una associazione indica una relazione tra le classi

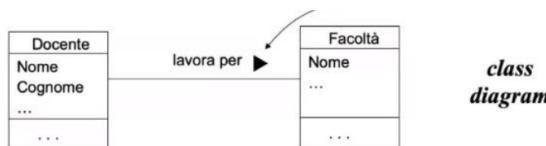
Un'associazione deve avere un nome

- il nome è un verbo
- le associazioni sono bidirezionali

I link sono istanze delle associazioni

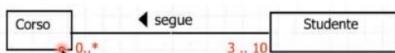
- un link connette due oggetti
- un'associazione connette due classi

Possiamo specificare una direction (freccia) per indicare da quale parte iniziare a leggere



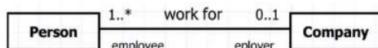
22

## Molteplicità delle associazioni



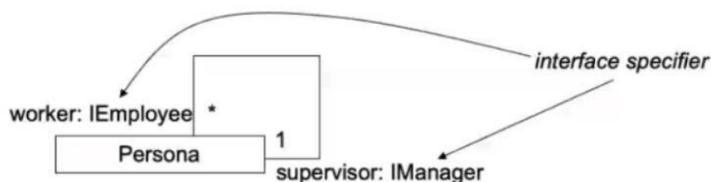
## Ruoli

Forniscono una modalità per attraversare relazioni da una classe all'altra



## Interface Specifier

Specifica quale parte dell'interfaccia di una classe è mostrata da questa nei confronti di un'altra classe della stessa associazione



## Classi associative

Sono utilizzate per modellare proprietà delle associazioni

Questo accade quando ci sono delle proprietà che non sono di nessuna classe coinvolta nell'associazione, ma sono dell'associazione stessa (che diventa una classe)



## Aggregazioni

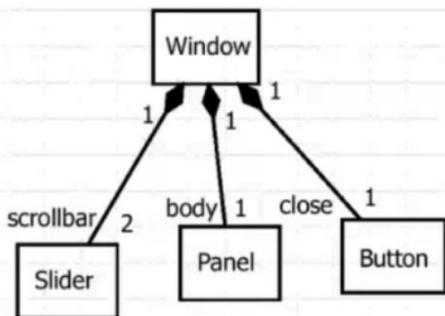
È possibile specificare delle aggregazioni: è un'associazione speciale che aggrega oggetti di una classe in un unico oggetto (composto da, è parte di)



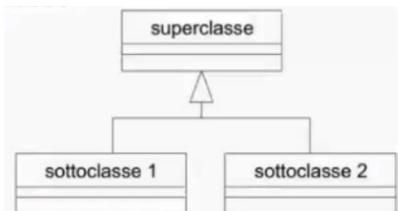
## Composizione

È un'aggregazione forte

- le parti componenti non esistono senza il contenitore
- ciascuna parte componente ha la stessa durata di vita del contenitore
- una parte può appartenere ad un solo tutto per volta



## Generalizzazione



Ogni oggetto di una sottoclasse è anche oggetto della sua superclasse

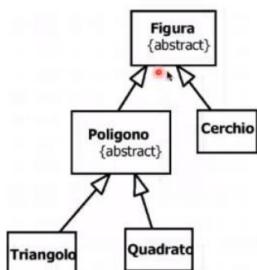
## Ereditarietà

È un meccanismo di condivisione delle proprietà degli oggetti in una gerarchia di generalizzazione

Tutte le proprietà (attributi e operazioni) di una superclasse possono essere applicati alle sottoclassi

## Classi astratte

Definisce un comportamento “generico”



## Root, leaf and polymorphic elements

---

- ◆ Per specificare che una classe non può avere discendenti si indicherà per questa la proprietà *leaf* sotto il nome della classe
- ◆ Per specificare che una classe non può avere antenati si indicherà per questa la proprietà *root* sotto il nome della classe
- ◆ Per indicare che una operazione è astratta il suo nome sarà scritto in corsivo
- ◆ Una operazione per la quale esiste la stessa signature in più classi di una gerarchia è *polimorfica*

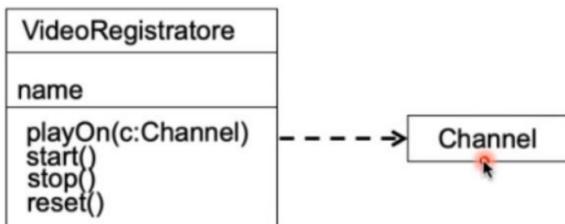
## Gerarchia di classi

UML definisce

- 1 stereotipo
  - Implementation: la sottoclasse eredita l'implementazione della superclasse ma non rende pubblica né supporta la sua interfaccia
- 4 constraints
  - Complete: tutte le sottoclassi sono state specificate, nessun'altra sottoclasse è permessa
  - Incomplete: non tutte le sottoclassi sono state specificate, altre sottoclassi sono messe a disposizione
  - Disjoint: oggetti del genitore possono avere non più di un figlio come tipo
  - Overlapping: oggetti del genitore possono avere più di un figlio come tipo

## Dipendenza

Relazione semantica in cui un cambiamento sulla classe indipendente può influenzare la semantica della classe dipendente



## Realizzazione

Una relazione tra le classi in cui una specifica un contratto che l'altra garantisce di compiere



## Class Diagram

---

- ◆ Un class diagram rappresenta un insieme di class, interface, collaboration e le loro relationship
- ◆ Un class diagram è tipicamente usato per modellare
  - la vista di static design di un sistema, che supporta principalmente i requisiti funzionali del sistema
  - se include active class descrive la vista statica di un processo di un sistema
  - il glossario di un sistema: sono prese decisioni relativamente alle astrazioni da considerare
  - semplici collaborazioni
  - lo schema concettuale di un database

## Lezione 16

**Software lifecycle:** è il periodo di tempo che inizia quando il software viene concepito e finisce quando il software non è più disponibile all'uso

**Software development cycle:** periodo di tempo che inizia con la decisione di sviluppo del software e termina con la prima consegna del prodotto

**Modello di ciclo di vita del software (Modello CVS):** è una caratterizzazione descrittiva di come un sistema software dovrebbe essere sviluppato (in che modo si organizza lo sviluppo di un software)

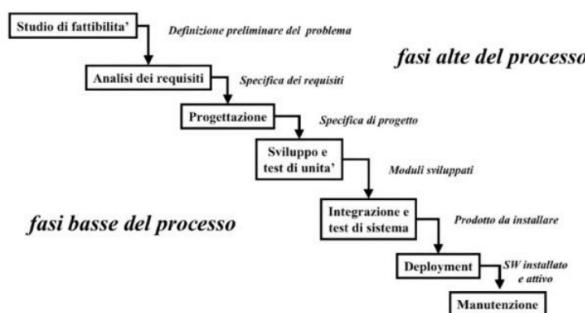
### ESISTONO VARI MODELLI CVS

#### Modelli a Cascata (Waterfall)

Nasce negli anni 70, anni in cui si procedeva con un approccio “code and fix”: si scrive direttamente il codice e fixare gli eventuali problemi quando era necessario.

Questo modello è **sequenziale lineare**: si procede sequenzialmente tra più fasi (senza ricicli, senza tornare indietro), non c'è overlap fra le fasi e si introducono dei semilavorati tra le fasi (documentazione, test cases, beta versions ...)

#### Modello a Cascata



Fasi alte si occupano del cosa deve essere realizzato e le fasi basse si occupano di come deve essere realizzata la soluzione

- Studio di fattibilità: si occupa di fare una valutazione preliminare di costi e benefici
  - Obiettivo: stabilire se avviare il progetto, individuare le possibili opzioni e le scelte più adeguate, valutare le risorse umane e finanziarie necessarie
  - Output: documento di fattibilità
    - Definizione preliminare del problema: viene realizzato il business case del progetto
    - Strategie diverse di soluzione
    - Costi, tempi, modalità di sviluppo per ognuna delle alternative
- Analisi dei requisiti: analisi completa dei bisogni dell'utente e dominio del problema, richiede coinvolgimento di committente e ingegneri del software per riuscire a capirsi l'un l'altro
  - Obiettivo: che cosa deve essere realizzato ma non il come, quindi deve fornire le funzionalità che deve mettere a disposizione il sistema e caratteristiche di qualità come i requisiti non funzionali
  - Output: documento di specifica dei requisiti accompagnato da un manuale utente e da un piano per il test di accettazione del sistema, cioè quali saranno i criteri e i test per accettare il sistema
- Progettazione: bisogna individuare la struttura per il software, quella che è il design architettonico andando a individuare quali sono le componenti di cui si comporrà il prodotto software, quali relazioni ci sono fra queste componenti per poi passare alla scomposizione di queste componenti per vedere i dettagli interni di ciascuna componente e questo viene fatto con il design dettagliato
  - Obiettivo: non è il che cosa ma il come bisogna andarlo a realizzare
  - Output: documento di specifica del progetto che farà uso di linguaggio naturale e di formalismi specifici di progettazione (UML)
- Programmazione e test di unità: ogni modulo che è stato individuato con il progetto dettagliato viene codificato nel linguaggio scelto e testato singolarmente in maniera da assicurare che ogni modulo non presenti "errori"
- Integrazione e test e successivamente testare l'intero sistema
  - Alfa test: sistema rilasciato internamente al produttore
  - Beta test: sistema rilasciato a pochi selezionati utenti
- Deployment: distribuzione e gestione del software presso l'utenza
- Manutenzione

## Modello a cascata: vantaggi e svantaggi

### ◆ Pro

- ha definito molti concetti utili (semilavorati, fasi ecc.)
- ha rappresentato un punto di partenza importante per lo studio dei processi SW
- facilmente comprensibile e applicabile

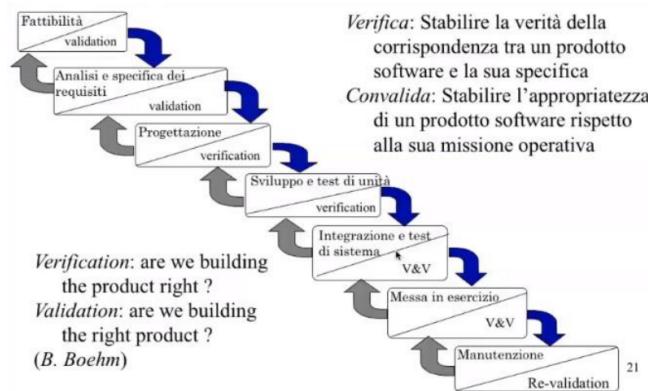
### ◆ Contro

- interazione con il committente solo all'inizio e alla fine
  - » requisiti congelati alla fine della fase di analisi
  - » requisiti utente spesso imprecisi: "l'utente sa quello che vuole solo quando lo vede"
- il nuovo sistema software diventa installabile solo quando è totalmente finito
  - » né l'utente né il management possono giudicare prima della fine dell'adesione del sistema alle proprie aspettative

18

Sono stati proposti altri modelli che vanno ad aggirare alcuni problemi (instabilità dei requisiti, assenza di ricicli, maggiore importanza sulla manutenzione, ...)

## V&V con Retroazione



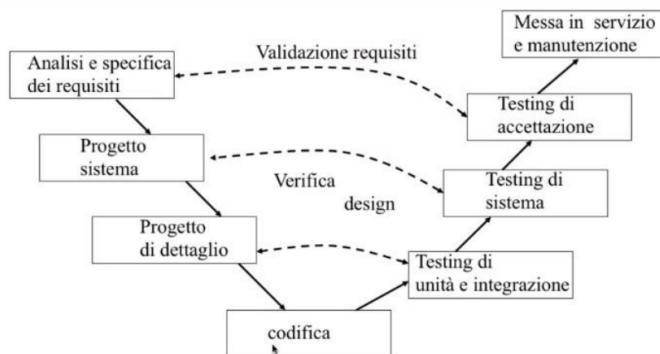
21

V&V sta per Verification & Validation: significano due cose diverse

- Verification tende a stabilire che ci sia corrispondenza tra un prodotto software e una sua specifica
- Validation mira a stabilire che quel prodotto è appropriato rispetto a quelle che sono le esigenze

In questa variante viene data la possibilità di “tornare indietro” se Verification o Validation non sono state superate. Si prevede la possibilità di tornare indietro fino alla prima fase di fattibilità: questo perché si deve rendere consistente la documentazione con le modifiche apportate al progetto (la documentazione sarà rivista poi durante la manutenzione)

## Modello a V



È il modo di riportare il modello a cascata in un'altra forma

## Modelli basati su prototipo

Un prototipo è un artefatto non completo che noi facciamo come prima approssimazione e che sottoponiamo al giudizio. In questo caso il prototipo è utilizzato per far comprendere bene i requisiti e/o valutare la fattibilità:

- Forniamo al cliente un prototipo di sistema ed usandolo riesce a capire se i requisiti sono soddisfatti, cosa aggiungere/migliorare

Col prototipo si **validano i requisiti e si accetta la fattibilità del prodotto**

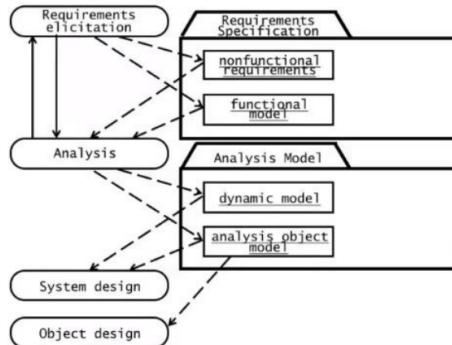
Dopo aver utilizzato il prototipo e dopo che il cliente lo ha accettato possiamo partire con un approccio Waterfall. (in questo modo eliminiamo il rischio che il cliente non abbia ben compreso i requisiti che sta per accettare)

**mock-ups:** produzione completa dell' interfaccia utente. Consente di definire con completezza e senza ambiguità i requisiti (si può, già in questa fase, definire il manuale di utente)

**breadboards:** implementazione di sottoinsiemi di funzionalità critiche del SS, non nel senso della fattibilità ma in quello dei vincoli pesanti che sono posti nel funzionamento del SS (carichi elevati, tempo di risposta, ...), senza le interfacce utente. Produce feedbacks su come implementare la funzionalità (in pratica si cerca di conoscere prima di garantire).

## Lezione 17

### Prodotti della raccolta dei requisiti e dell'analisi dei requisiti.



Siamo in fase di Analisi e per tanto vanno individuati 2 concetti:

- **Il modello degli oggetti di analisi:** fornire una rappresentazione di quello che è il dominio del problema e le esigenze dell'utente e non si focalizza sul come deve essere realizzato. Andremo a realizzare un dizionario visuale dei concetti principali visibili all'utente e lo rappresentiamo con il Class Diagram di UML che include classi, attributi e operazioni
- **Il modello dinamico:** l'obiettivo è focalizzarsi sul comportamento del sistema andando a individuare i sequence diagram e i statechart diagram e soprattutto assegnare delle responsabilità alle classi
  - I sequence diagram rappresentano le interazioni tra un insieme di oggetti per la realizzazione delle funzionalità rappresentate da un singolo use case.
  - Gli statechart verranno utilizzati per rappresentare il comportamento di un singolo oggetto

Poiché siamo in fase di analisi, il tutto deve continuare a mantenersi a **livello utente** (non a livello di classi software)

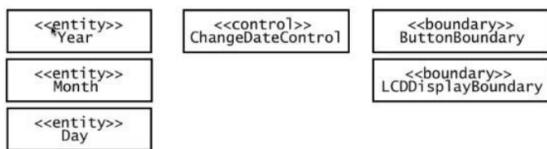
## Oggetti Entity, Boundary e Control

- Entity: rappresentano l'informazione persistente (oggetti del dominio di applicazione)
- Boundary: rappresentano gli oggetti con cui avviene l'interazione tra l'attore e il sistema (oggetti relativi all'interfaccia utente, dei dispositivi, del sistema)
- Control: rappresentano gli oggetti che si occupano di coordinare gli altri oggetti per la realizzazione di uno use case, cioè di una funzionalità

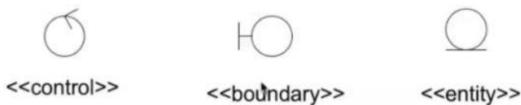
Ese. 2Bwatch:

- Year, Month e Day sono oggetti **Entity**
- Button e LCDDisplay sono oggetti **Boundary**
- ChangeDateControl è un oggetto **Control**, che rappresenta l'attività di cambiare la data premendo combinazioni di pulsanti

Per rappresentare questi tipi di oggetti a livello del class diagram si possono utilizzare gli stereotipi:



Ottobre con le nozioni alternative



## Attività di Analisi: dagli use case agli oggetti

- Le attività che consentono di trasformare gli use case e gli scenari della raccolta dei requisiti in un modello di analisi sono:
  - Identificare gli Oggetti Entity
  - Identificare gli Oggetti Boundary
  - Identificare gli Oggetti Control
  - Mappare gli Use Case in Oggetti con Sequence Diagram
  - Identificare le Associazioni
  - Identificare gli Aggregati
  - Identificare gli Attributi
  - Modellare il Comportamento dipendente dallo stato degli Oggetti individuali (statechart)
  - Modellare le Relazioni di Ereditarietà
  - Rivedere il Modello di Analisi

### Identificare gli oggetti Entity

Si analizzano gli use case e scenari e si utilizza un'euristica che potrebbe aiutarci: l'euristica di Abbott che si basa sull'analisi del linguaggio naturale per identificare oggetti, attributi ed associazioni (si mappa il parlato con componenti del modello)

| Parti del parlato | Componente del modello | esempio                    |
|-------------------|------------------------|----------------------------|
| Nome proprio      | Istanza                | Alice                      |
| Nome comune       | Class                  | Funzionario(FieldOfficer)  |
| Verbo fare/azione | Operazione             | Crea, Sottoponi, Seleziona |
| Verbo essere      | gerarchia              | È un tipo di, è uno di     |
| Verbo avere       | aggregazione           | Ha, consiste di, include   |
| aggettivo         | attributo              | Descrizione dell'incidente |

Questo è utile per una prima analisi, perché il linguaggio naturale è spesso impreciso (quindi possono esserci errori)

Insieme all'euristica di Abbott si può usare anche un'altra euristica che dice:

- **Termini** che gli sviluppatori e gli utenti hanno bisogno di chiarire per comprendere gli use case
- **Sostantivi** ricorrenti negli use case
- **Entità** del mondo reale che il sistema deve considerare
- **Attività** del mondo reale che il sistema deve considerare
- **Sorgenti o destinazioni** di dati

Per ogni oggetto identificato

- Si deve assegnare un nome e una breve descrizione
  - È opportuno utilizzare gli stessi nomi utilizzati dagli utenti e dagli specialisti del dominio applicativo
- Si individuano attributi e responsabilità (non tutti, soprattutto non quelli ovvii)

## Identificare gli oggetti Boundary

- In ogni use case, ogni attore interagisce almeno con un oggetto Boundary
- L'oggetto Boundary colleziona l'informazione fornita dall'attore e la traduce in una forma che può essere usata sia dagli oggetti Control che Entity

Gli oggetti Boundary modellano l'interfaccia ma non descrivono gli aspetti visuali

## EURISTICHE PER IDENTIFICARE GLI OGGETTI BOUNDARY

- Identifica i controlli della UI di cui l'utente ha bisogno per iniziare lo use case (button)
- identifica form di cui l'utente ha bisogno per inserire dati nel sistema (form)
- identifica avvisi e messaggi che il sistema usa per rispondere all'utente (AcknowledgmentNotice)
- quando più attori sono coinvolti in uno use case, identifica gli attori che sono terminali per riferirsi alla UI in considerazione
- non modellare aspetti visuali della UI con oggetti Boundary
- usare sempre i termini dell'utente finale per descrivere l'interfaccia

## Identificare gli oggetti Control

### EURISTICHE PER IDENTIFICARE GLI OGGETTI CONTROL

- identifica un oggetto Control per ogni use case
- identifica un oggetto Control per ogni attore in uno use case
- la vita di un oggetto Control deve corrispondere alla durata di uno use case.  
Se è difficile identificare l'inizio e la fine dell'attivazione di un oggetto Control, il corrispondente use case probabilmente non ha delle entry e exit condition ben definite

### Mappare Use case in Oggetti con Sequence Diagram

- Un Sequence Diagram
  - mostra come il comportamento di uno use case (o scenario) è distribuito tra i suoi oggetti partecipanti
    - infatti vengono assegnate responsabilità a ogni oggetto in termini di un insieme di operazioni
  - Illustra la **sequenza di interazioni** tra gli oggetti necessaria per realizzare uno use case
    - non ci occupiamo di questioni di implementazioni, come l'efficienza!
- Non è adatto alla comunicazione con il cliente
- Per gli esperti è intuitivo e più preciso degli use case
- Fornisce una prospettiva diversa che consente di **individuare oggetti mancanti e aree non chiare** nelle specifiche

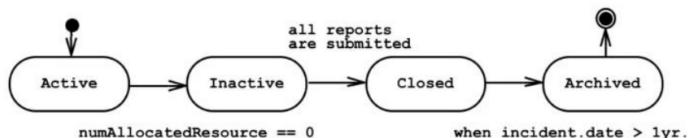
## Sequence Diagram

- Le colonne rappresentano gli oggetti che partecipano nello use case
  - La colonna più a sinistra rappresenta l'attore che inizia lo use case
  - La seconda colonna -> oggetto Boundary con cui l'attore interagisce per iniziare lo use case
  - La terza colonna -> oggetto Control che gestisce il resto dello use case
  - Gli oggetti Control creano altri oggetti Boundary e possono interagire con altri oggetti Control
- Le frecce orizzontali tra le colonne rappresentano messaggi o stimoli inviati da un oggetto ad un altro
- La ricezione di un messaggio determina l'attivazione di un'operazione
  - L'attivazione è rappresentata da un rettangolo da cui altri messaggi possono prendere origine
  - La lunghezza del rettangolo rappresenta il tempo durante il quale l'operazione è attiva
  - Un'operazione è un servizio fornito ad altri oggetti
- La vita degli oggetti
  - Il tempo procede verticalmente dal top al bottom
  - Al top del diagramma si trovano gli oggetti che esistono prima del 1° messaggio inviato
  - Oggetti creati durante l'interazione sono illustrati con il messaggio <<create>>
  - Oggetti distrutti durante l'interazione sono evidenziati con una croce
  - La linea tratteggiata indica il tempo in cui l'oggetto può ricevere messaggi

## Modellare il comportamento dei singoli oggetti

- I sequence diagram sono usati per “distribuire” il comportamento tra i diversi oggetti del sistema e rappresentano il comportamento del sistema dal punto di vista dell’utente
- Gli statechart rappresentano il comportamento dei singoli oggetti**
- Gli sviluppatori costruiscono una descrizione formale del comportamento degli oggetti, e di conseguenza identificano use case “tralasciati” nella prima fase
- Non è necessario costruire statechart per tutti gli oggetti del sistema (solo oggetti con una durata del ciclo di vita significativa):
  - è il caso degli oggetti control,
  - meno degli oggetti entity,
  - no per oggetti boundary.

## UML statechart for Incident.



## UML Statechart Diagram

### Notation



- sugli archi è inserito l'evento (con attributi), con eventuale condizione (se si verifica l'evento e la condizione è vera allora avviene la transizione)/azione
- in uno stato possono essere specificate delle entry/exit condition con delle azioni (quando si entra/esce dallo stato si eseguono delle azioni)
- action è diverso da activity perché le action sono azioni eseguite istantaneamente, le activity no

## Rivedere il modello di analisi

- Il modello di analisi è costruito **incrementalmente e iterativamente** (molte iterazioni con il cliente e gli utenti sono necessarie)
- Quando il modello di analisi diventa **stabile** (non si hanno più cambiamenti, oppure sono minimali), il modello è riesaminato, prima dagli sviluppatori e poi insieme con il cliente e gli utenti
- L'obiettivo di questa attività di revisione è stabilire che la specifica risulta essere: corretta, completa, consistente e chiara