

INGEGNERIA SW – Appunti Dominick Ferraro

Anno 2021-2022

Lo sviluppo SW è un’attività complessa.

Il dominio del problema è difficile e la sua soluzione anche.

Bisogna studiare, capire il gergo e conoscere il dominio applicativo (contesto del sw da implementare).

I sistemi sw si sviluppano in team ed è difficile da gestire.

Il sw inoltre offre una forte flessibilità (può essere modificato continuamente, fattore negativo).

Il sw è un sistema discreto:

- I sistemi continui non hanno problematiche nascoste
- I sistemi discreti hanno problematiche nascoste.

L’attività di sviluppo sw non è principalmente coding ma è fondamentale calarsi nel dominio e individuare una soluzione. La modellazione serve per tenere sotto controllo la complessità di un sistema.

È fondamentale poi l’acquisizione di conoscenza e la gestione delle risorse.

Si usano un insieme di tecniche per arrivare a una realizzazione di un sistema.

- Tecniche: procedure formali per la produzione di risultati usando una nozione definita.
- Metodologie: raccolta di tecniche applicate durante lo sviluppo sw e unificate da un approccio filosofico.
- Tool:
 - strumento o sistemi per realizzare una tecnica.
 - CASE= Computer Aided SW Engineering

L'ingegneria del sw quindi è una collezione di tecniche, metodologie e strumenti che aiutano la produzione di sw di alta qualità sviluppato con un budget fissato entro una scadenza fissata mentre sono in atto continui cambiamenti.

LA CRISI DEL SOFTWARE: le cause della crisi del sw erano collegate alla complessità dei processi sw e alla relativa immaturità nella produzione del wf. La crisi si manifestava in diversi modi:

- Progetti oltre al budget
- Progetti oltre limiti di tempo
- Sw di scarsa qualità
- Sw che non rispetta requisiti

- Progetti ingestibili e codice difficile da mantenere

Nascita dell'ingegneria del SW

Negli anni '50 appaiono i primi linguaggio ad alto livello (COBOL) e i programmi cominciano a diventare sempre più complessi.

Verso gli anni '60-'70 le tecniche di sviluppo adottate non erano scalabili. Gli sviluppatori sembravano incapaci di sviluppare sw in grado di usare a fondo i miglioramenti nell'HW.

Con le conferenze NATO vengono poste le basi per una nuova disciplina (1968-69).

TIPOLOGIE DI PRODOTTI SW

- Prodotti generici: sistemi venduti a un mercato di massa
- Prodotti specifici: sistema commissionato da un utente e sviluppato per questo da un contrante.

La maggior parte della spesa è nei prodotti generici ma il maggior sforzo di sviluppo è nei prodotti specifici.

Infine: l'ingegneria del sw è una disciplina che cerca di fornire le regole per il processo di produzione del sw. Un ingegnere del sw dovrebbe:

- a. Adottare un approccio sistematico e organizzato al proprio lavoro.
 - b. Usare tool e tecniche appropriate, che dipendono dal problema, dal budget e dai vincoli di risorse disponibili.
-

Lezione 2: Modellazione con UML

Abbiamo detto che i sistemi sw sono prodotti industriali, che devono essere progettati secondo rigidi schemi.

I discorsi di modellazione di sistema interessa molti settori. Quello su cui vogliamo puntualizzare è il modo attraverso il quale cerchiamo di rappresentare un sistema complesso.

Abbiamo 3 approcci:

- Atrazione e modellazione: quando dobbiamo rappresentare un sistema, se è complesso si cerca di determinare delle idee che sottendono al sistema, focalizzandosi su aspetti particolari e tralasciando altri. Quando riusciamo ad astrarre dei concetti, possiamo costruirne dei modelli.
- Decomposizione: se abbiamo un problema possiamo suddividerlo in pezzi con l'obiettivo di gestirne la complessità, questi possono essere a loro volta suddivisi in altre parti.
- Gerarchia: quando si è costruito un puzzle di sottosistemi ci deve essere una gerarchia per rappresentare una relazione tra le varie componenti.

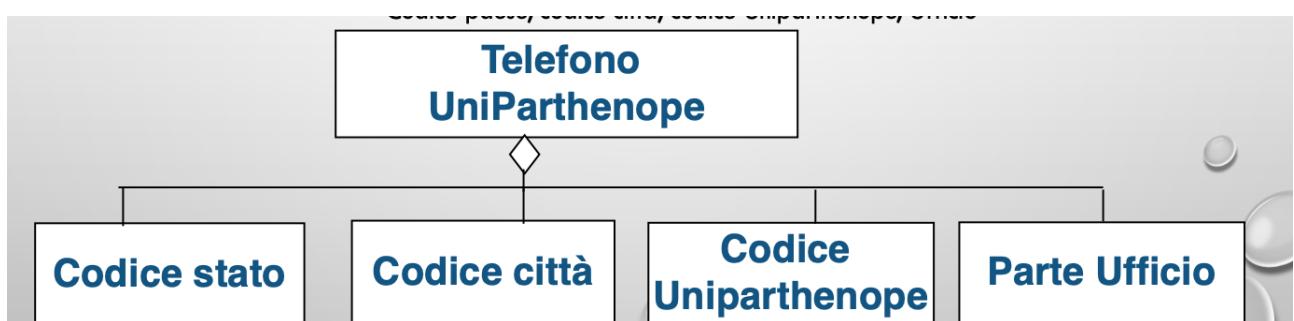
ASTRAZIONE

Si fa un'idea del problema e ci si focalizza su alcuni aspetti. L'astrazione semplifica la visione della parte analizzata. Quando dobbiamo costruire un modello possiamo individuare delle parti comprensibili. I sistemi complessi sono difficile da capire, empiricamente gli umani non riescono a memorizzare +7 / 9 pezzi allo stesso tempo.

Esempio: numero di telefono uni 390815476543: possiamo strutturarla e spezzettarla dando un senso alle varie componenti.

Lo dividiamo in 4 pezzi:

- Codice paese, codice città, codice uni, numero ufficio.



Tramite il processo di astrazione si determinano idee peculiari di un sistema, tralasciando parti meno importanti, rappresentabili tramite un modello.

Ad esempio: supponiamo di dover costruire un aereo, se lo vediamo nella sua totalità sembra impossibile ma con l'astrazione possiamo suddividerlo in più

componenti: proprietà aerodinamiche (costruiamo un modello in scala per analizzarle), i vari sensori e le postazioni di controllo in questo caso sono superflue e quindi vanno tralasciati.

MODELLO

Un modello è una rappresentazione di un sistema che semplifica la realtà. Un modello può essere usato per diversi sistemi:

- Eistenti
 - Scomparsi
 - Futuri
-

Il processo di astrazione e uso di modelli sono usati per sistemi SW.

Quando si descrive un sistema sw si usano più modelli:

- Modelli ad oggetti: qual è la struttura del sistema?
- Modelli funzionali: quali sono le sue funzioni?
- Modelli dinamici: come reagisce il sistema agli eventi esterni?

Unendo i 3 sistemi ottengo quello del sistema SW

Modello del sistema: modello ad oggetti + modello funzionale + modello dinamico

DECOMPOSIZIONE

È una tecnica usata per padroneggiare la complessità (divide and conquer).

Ci sono due tipi di composizioni:

- Funzionale: il sistema è scomposto in una serie di moduli, focalizzata sulle funzioni che deve fornire il sistema.
- Orientata agli oggetti: il sistema è decomposto in classi. Ogni classe è un'entità principale nel dominio applicativo.

Questi due approcci possono essere complementari.

La decomposizione funzionale non è ideale se usata da sola. Se decompongo funzionalmente il sistema, se devo effettuare una modifica a una delle funzioni, bisogna capire tutte le funzionalità del sistema e ciò diventa complesso, crea un codice complesso e la gestione futura diventa complicata.

Una rappresentazione orientata agli oggetti risulta più stabile che consente di effettuare le modifiche. In conclusione possiamo affermare che il miglior approccio è usare l'impostazione ad oggetti combinata con quella funzionale: posso focalizzare l'attenzione sulle entità principali del dominio

applicativo e poi identifico le funzioni per ciascuna classe.

IDENTIFICAZIONE DELLE CLASSI

Per identificare le classi in fase di analisi dobbiamo fare assunzioni di base:

- Costruzione di un sistema ex novo. → Greenfield Engineering
 - Progetto che richiede di modificare un sistema esistente → REENGINEERING
 - Creazione di un sistema esistente ma solo su interfacce → INTERFACE ENGINEERING
-

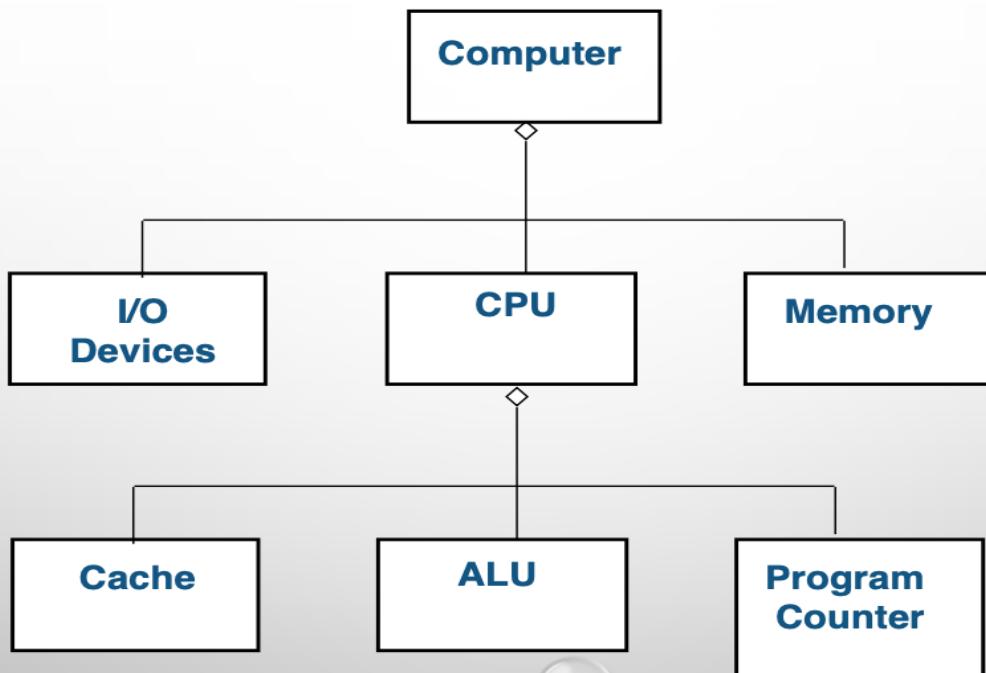
GERARCHIA

Individuate le classi di interesse per rappresentare il dominio applicativo, bisogna rappresentare le relazioni tra le componenti.

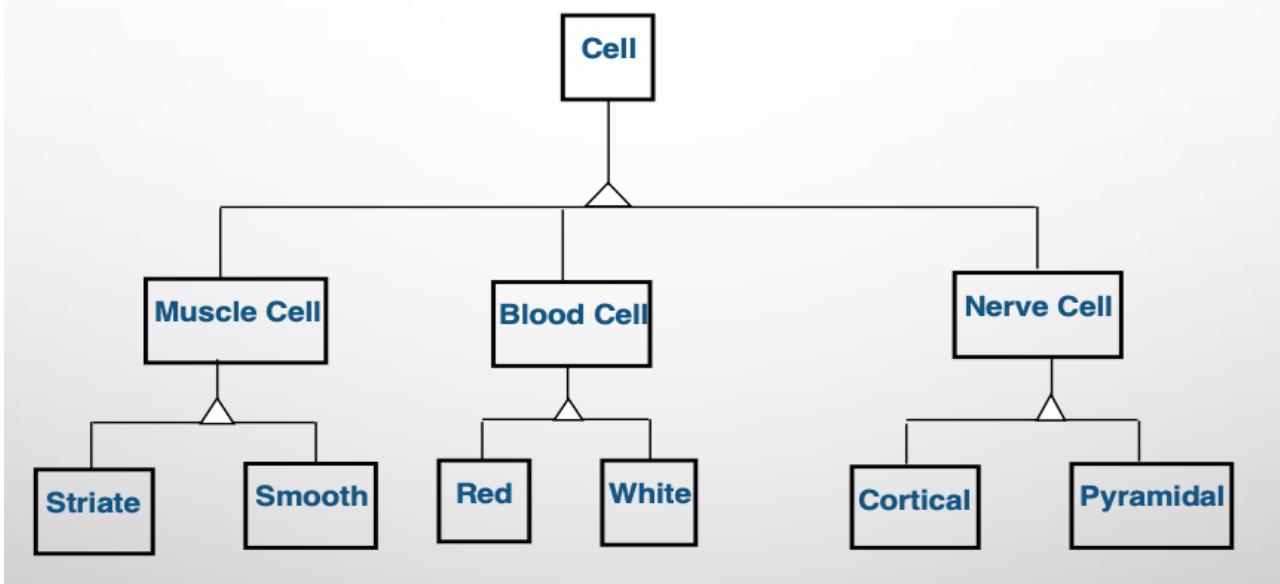
Le gerarchie a disposizione sono:

- “è parte di” → ci sono dei livelli dove il più alto è quello più importante rispetto a quelli a livello inferiore.
- “è tipo di” → non prevede una relazione di superiorità di livelli ma di inclusione.

GERARCHIA 'PARTE DI'



GERARCHIA 'È TIPO DI'



Come procediamo nel modo corretto?

Iniziamo con una descrizione delle funzionalità di un sistema, descriviamo poi la sua struttura.

Quando si parla di ciclo di vita del SW parliamo delle fasi di sviluppo dall'analisi, coding, testing e pubblicazione.

Dopo aver costruito il modello, prima di consegnarlo deve essere **validato**. Come capiamo se il progetto va bene o meno?

Potrebbe non andare male perché crasha, perché non è conforme alle richieste del cliente ecc...

Quindi dopo aver costruito il modello del sistema dobbiamo valutarlo, cioè deve essere *falsificabile*: un modello deve rappresentare la realtà e a priori non si può dire che quel modello sia valido o meno, si assume che quel modello sia ok finché non si trovi una prova contraria. La falsificazione avviene durante il *testing*: si usa il modello in modo da farlo sbagliare in tutti i modi per cercare ulteriori bug o errori da risolvere per stabilire se il sistema è valido o no.

CONCETTI E FENOMENI

- Un fenomeno è un oggetto così come lo percepiamo.

- Un concetto descrive le proprietà comuni di un fenomeno, quindi la generalizzazione dei fenomeni.

Un concetto è una tri-tupla: NOME SCOPO MEMBRI



SISTEMI, MODELLI E VIEW

Un modello è quindi un'astrazione che descrive un sistema.

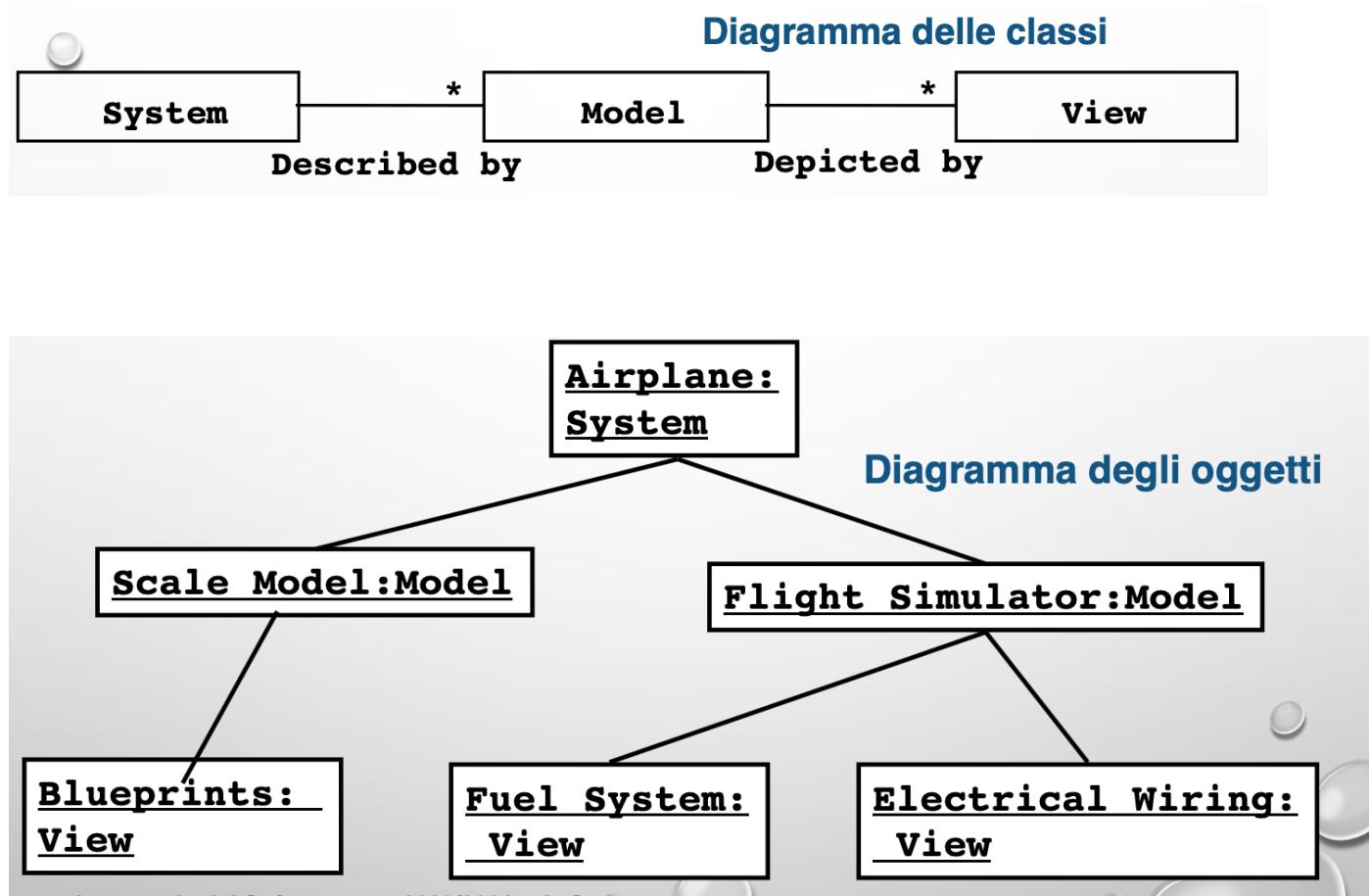
Una view raffigura aspetti selezionati di un modello.

Una notazione è un insieme di regole grafiche o testuali per illustrare modelli e view.

Esempio:

- Sistema : aereo
- Modelli: simulatore di volo, modello in scala
- View: planimetria, diagramma di cavi elettrici.

SISTEMI, MODELLI E VIEW (UML)



DOMINIO DELL'APPLICAZIONE VS DOMINIO DELLA SOLUZIONE

- Dominio dell'applicazione: l'ambiente in cui il sistema sta funzionando. (analisi)
- Dominio della soluzione: tecnologie usate per la risoluzione del sistema. (progettazione e implementazione)

UML (Unified Modelling Language)

È uno standard non proprietario per modellare sistemi software.

DIAGRAMMI UML:

- Diagrammi di casi d'uso: descrivono il comportamento funzionale del sistema come sono visti dagli utenti
- Diagrammi delle classi: descrivono la struttura statica del sistema: oggetti, attributi, associazioni
- Diagrammi delle sequenze: descrivono il comportamento dinamico tra gli oggetti del sistema
- Diagramma degli stati: descrivono il comportamento dinamico di un singolo oggetto
- Diagrammi delle attività: descrivono il comportamento dinamico di un sistema, in particolare il flusso di lavoro.

NOTAZIONE UML

Tutti i diagrammi UML denotano grafi di nodi e vertici.

- I nodi sono rappresentati come rettangoli o ovali
- I rettangoli denotano classi o istanze
- Gli ovali determinano funzioni

- I nomi delle classi non sono sottolineati.
 - SimpleWatch
- I nomi delle istanze sono sottolineati.
 - myWatch:SimpleWatch

Un arco tra due nodi indica una relazione tra le entità corrispondenti.

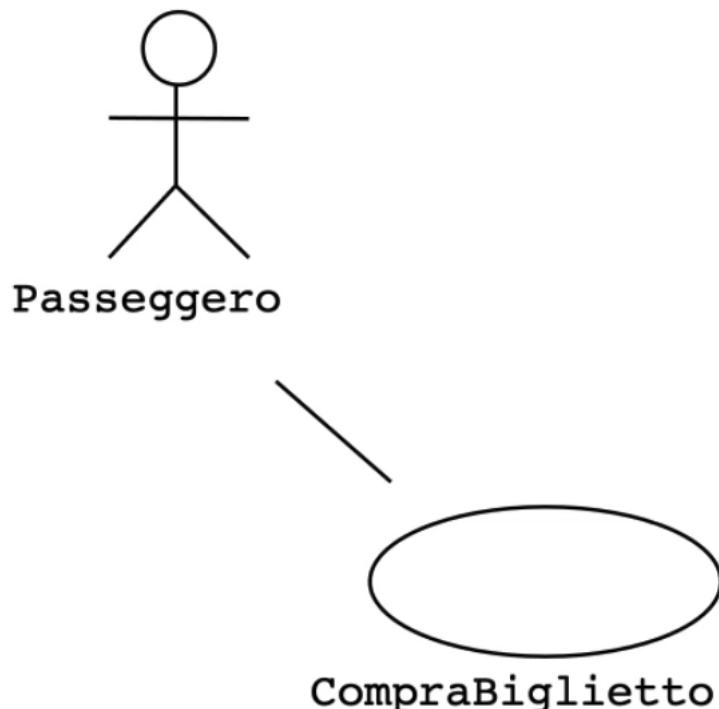
Lezione 3

UML

Diagramma dei casi d'uso: usati durante la fase di scoperta dei requisiti per rappresentare il comportamento esterno.

- Gli attori rappresentano ruoli, cioè un tipo di utente del sistema
- I casi d'uso rappresentano una sequenza di interazioni per un tipo di funzionalità.

Il modello dei casi d'uso è l'insieme di tutti i casi d'uso. È una descrizione completa della funzionalità del sistema e del suo ambiente.



Un attore non è solo l'utente ma anche componenti coi quali il sistema interagisce all'esterno delle funzioni.

Nell'esempio il passeggero è l'attore e un caso d'uso potrebbe essere la funzione “compra biglietto”.

Tutti gli aspetti tecnici e implementativi non entrano in gioco nei diagrammi di caso d'uso.

ATTORI

Un attore modella un'entità esterna che comunica col sistema:

- Utente
- Sistema esterno
- Ambiente fisico

Un attore ha un nome unico e una descrizione opzionale.

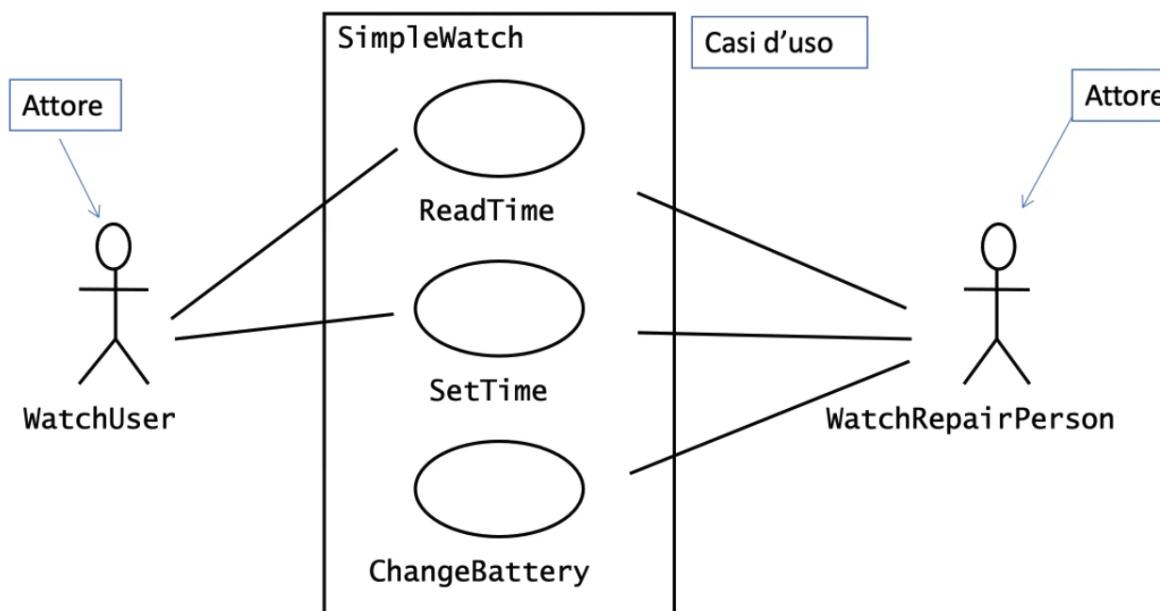
Esempio: Passeggero: Una persona nel treno.

Satellite GPS: fornisce al sistema le coordinate del GPS

CASO D'USO

Si rappresenta con un ovale. Rappresenta una classe di funzionalità fornite dal sistema. I casi d'uso possono essere descritti testualmente, con attenzione sul flusso di eventi tra attore e sistema.

ESEMPIO



Gli attori sono il proprietario dell'orologio e l'orologiaio che lo ripara. Le funzionalità dell'orologio sono:

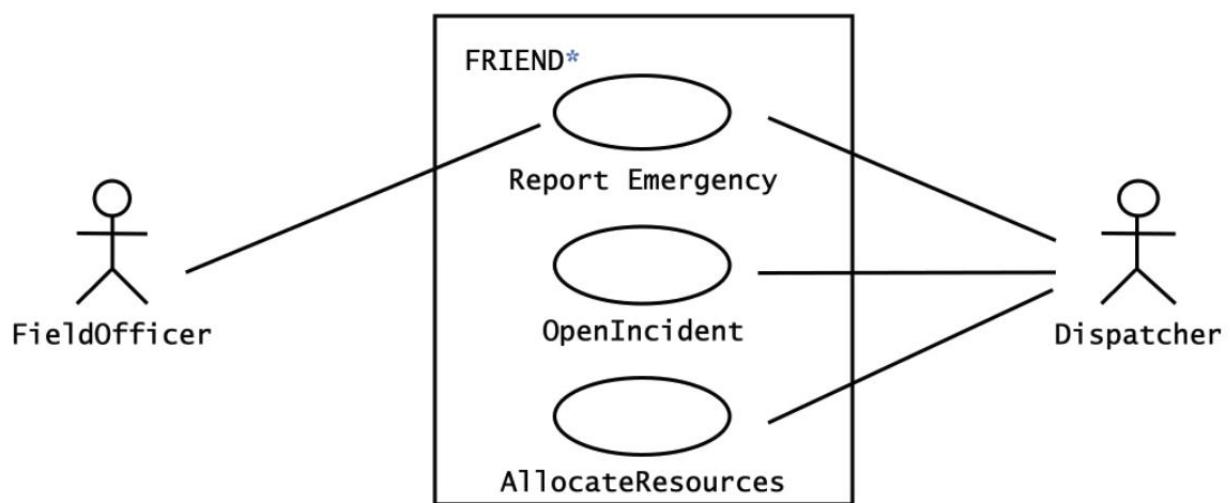
- Leggere l'ora → utente e orologiaio
- Modifica l'ora → utente e orologiaio
- Cambiare batteria → orologiaio

Come vediamo in figura, all'interno del rettangolo abbiamo tutte le funzioni del sistema, all'esterno gli attori con le frecce che collegano essi con i loro casi d'uso.

ESEMPIO: SISTEMA GESTIONE INCIDENTI

- I *field officer* (*poliziotto o pompiere*), hanno accesso a un computer wireless che consente di interagire con un Dispatcher.
- Il Dispatcher può visualizzare a video lo stato corrente di tutte le risorse, come le macchine di polizia o autocarri e inviare una risorsa inserendo comandi da una stazione di lavoro.
Es: c'è un incidente: le risorse saranno i mezzi e gli operatori di soccorso.
 - In questo esempio FieldOfficer e Dispatcher sono attori.

Il sistema prende il nome di FRIEND.



* First Responder Interactive Emergency Navigational Database

DESCRIZIONE TESTUALE DEI CASI D'USO

È possibile descrivere testualmente un caso d'uso specificando:

1. Nome univoco

2. Attori partecipanti

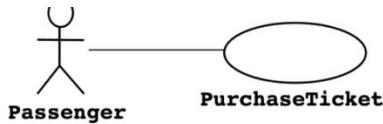
3. Condizioni di ingresso (contengono info sullo stato del sistema nel momento in cui il sistema viene invocato)

4. Condizioni di uscita (riportano lo stato del sistema quando il caso d'uso è completato).

5. Flusso degli eventi (deve dettagliare tutto ciò che accade quando invochiamo il caso d'uso. Se devo comprare un biglietto con la macchinetta devo eseguire diversi step: seleziono località, fascia di prezzo, fascia oraria, pagamento, ... , emissione del biglietto. Questi passi vanno esplicitati nel flusso degli eventi per capire cosa e come deve fare il sistema).

6. Requisiti speciali (requisiti non funzionali: vincoli HW e di sistema).

ESEMPIO DI DESCRIZIONE TESTUALE DEI CASI DI USO



1. Nome: PurchaseTicket

2. Attore partecipante:

Passenger

3. Condizione di ingresso

- Passenger si trova di fronte al Distributore di biglietti
- Passenger ha soldi sufficienti per acquistare il biglietto

4. Condizione di uscita:

Passenger ha il biglietto

5. Flusso degli eventi

1. Passenger seleziona l'area di destinazione (numero)
2. Il Distributore dei biglietti visualizza l'importo dovuto
3. Passenger inserisce i soldi, almeno l'importo dovuto
4. Il Distributore di biglietti restituisce il resto
5. Il Distributore di biglietti emette il biglietto

6. Requisiti speciali: nessuno

TEMPLATE PER I CASI D'USO

Per descrivere un caso d'uso è possibile usare un template composto dai sei campi sopra descritti. Di solito sono sottoforma di tabella. Ecco un esempio:

Nome	ReportEmergency
Attori Partecipanti	Iniziato dal FieldOfficer Comunica con il Dispatcher
Flusso eventi:	<ol style="list-style-type: none">1. Il FieldOfficer attiva la funzione "Report Emergency del terminale2. FRIEND risponde presentando una form al FieldOfficer3. Il FieldOfficer riempie la form selezionando il livello di emergenza, tipo, località, breve descrizione della situazione. Il FieldOfficer descrive anche possibili risposte alla situazione di emergenza. Appena finito Il FieldOfficer invia la form4. FRIEND riceve la form e notifica al Dispatcher5. Il Dispatcher rivede le informazioni sottomesse e crea un Incidente nel DB invocando il caso d'uso OpenIncident. Il Dispatcher seleziona una risposta e accetta il rapporto.6. FRIEND visualizza l'accettazione e la risposta selezionata al FieldOfficer
Condizioni di entrata	Il FieldOfficer è loggato in FRIEND
Condizioni di uscita	Il FieldOfficer ha ricevuto un'accettazione e una risposta selezionata dal Dispatcher, OR Il FieldOfficer ha ricevuto una spiegazione indicante perché la transazione non è stata eseguita
Requisiti di qualità	Il rapporto del FieldOfficer è accettato entro 30 secondi La risposta selezionata arriva non più tardi di 30 secondi dopo che è stata inviata dal Dispatcher

- Negli attori si specifica chi inizia l'azione e con chi comunica
 - Col flusso di eventi si descrivono tutti i passi dell'applicazione. Per semplificare i passi eseguiti dall'attore e quello del sistema si usa un'indentazione: quella a sinistra sono svolti dall'attore. Quelli più indentati a destra sono fatti dal sistema.
-
- Condizioni di entrata
 - Condizione di usciti
 - Requisiti di qualità
-

SCENARI

Un caso d'uso è un'astrazione che descrive tutti i possibili scenari che coinvolgono la funzionalità descritta.

Uno *scenario* è un'istanza del caso d'uso che descrive un insieme concreto di azioni:

- Sono usati come esempi per illustrare i casi comuni
- I casi d'uso descrivono tutti i possibili casi. L'obiettivo è la completezza.

FRIEND potrebbe essere usato per un incidente, incendio, attacco terroristico, sparatoria... Uno scenario è un evento tra questi **specifico**.

Anche gli scenari hanno un template.

Nome Scenario	<u>warehouseOnFire</u>
Istanze attori partecipanti	<u>bob, alice: FieldOfficer</u> <u>john: Dispatcher</u>
Flusso di eventi	<ol style="list-style-type: none">1. Bob, guidando lungo la strada principale nella sua patrol, osserva del fumo proveniente da un magazzino. Il suo partner, Alice, attiva la funzione "Report Emergency" dal laptop del FRIEND.2. Alice immette l'indirizzo dell'edificio, una breve descrizione della sua ubicazione (angolo nord-ovest) ed un livello di emergenza. In aggiunta ad un'unità dei vigili del fuoco, richiede diverse unità paramediche poiché la zona non è molto trafficata. Conferma l'input ed attende l'accettazione.3. John, il Dispatcher, è allertato per l'emergenza da un beep dalla sua stazione di lavoro. Rivede le informazioni sottomesse da Alice e accetta il rapporto. Alloca un'unità dei vigili del fuoco e due unità paramediche sul sito dell'Incidente ed invia la stima del tempo di arrivo (ETA) ad Alice.4. Alice riceve l'accettazione e l'ETA.

RELAZIONI TRA I CASI D'USO

Quando i casi d'uso si scambiano info, stanno comunicando.

Gli scambi di info possono essere rappresentati con le relazioni di comunicazione.

I diagrammi dei casi d'uso possono includere 4 tipi di relazioni: comunicazione, inclusione, estensione, ereditarietà.

Le relazioni di **comunicazione** sono illustrate con linee solide tra i simboli dell'attore e del caso d'uso.

- In riferimento a FRIEND di gestione :
 - Gli attori Field e Dispatcher comunicano col caso d'uso Report Emergency.
 - Solo il dispatcher comunica con i casi OpenIncident e AllocateResources.
-

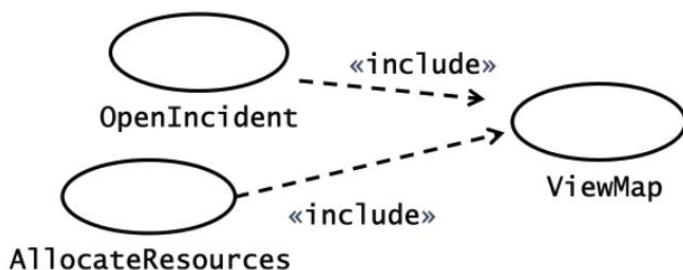
RELAZIONE DI INCLUSIONE

È possibile ridurre la complessità di un modello identificando degli aspetti in comune tra differenti casi d'uso.

Esempio: assumiamo che il Dispatcher premendo un tasto abbia accesso a una mappa, ciò può essere modellato da un caso d'uso “ViewMap” **incluso** nei casi d'uso OpenIncident e AllocateResources.

Il modello risultante descrive ViewMap una sola volta, riducendo di fatto la complessità del modello dei casi d'uso complessivo.

Esempio di relazione di inclusione:



«**include**» in figura si chiama “stereotipo”.
Esempio: relazione «**include**» testuale.

Nome	AllocateResources
Attori Partecipanti	Iniziato da Dispatcher
Flusso eventi:	...
Condizioni di entrata	Il Dispatcher apre un Incidente
Condizioni di uscita	Sono assegnate Risorse aggiuntive ad un incidente Le risorse sono notificate dei nuovi assegnamenti Il FieldOfficer che ha a carico l'incidente è notificato delle nuove risorse
Requisiti di qualità	In qualsiasi punto del flusso di eventi, questo caso d'uso può includere il caso d'uso ViewMap. Il caso d'uso ViewMap è iniziato quando il Dispatcher invoca la funzione “map”. Quando invocato all'interno di questo caso d'uso, il sistema scorre la mappa in modo che il luogo dell'incidente sia visibile al Dispatcher

Il caso d'uso viene descritto nei Requisiti di Qualità.

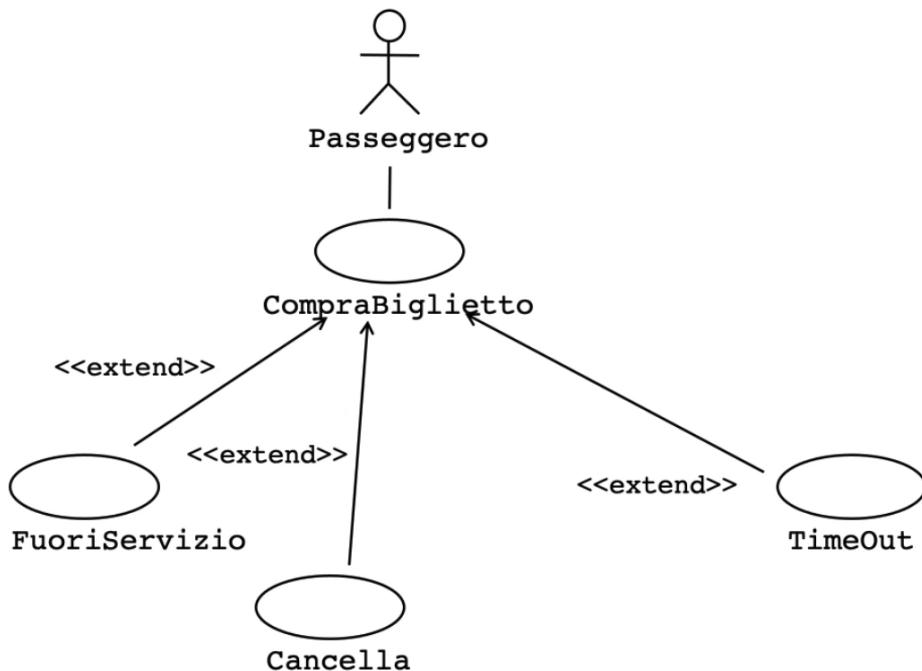
RELAZIONE DI ESTENSIONE

Le relazioni di estensione sono modi alternativi per ridurre la complessità.

Un caso d'uso può estendere un altro caso d'uso aggiungendo eventi.

Mentre l'inclusione si usa per casi d'uso condivisi da altri casi d'uso, le estensioni sono funzionalità associate ad eventi rari, come eccezioni (errori).

Esempio della relazione <<extended>>



A differenza dell'inclusione la freccia va dal caso d'uso che estende a quello esteso (al contrario dell'inclusione).

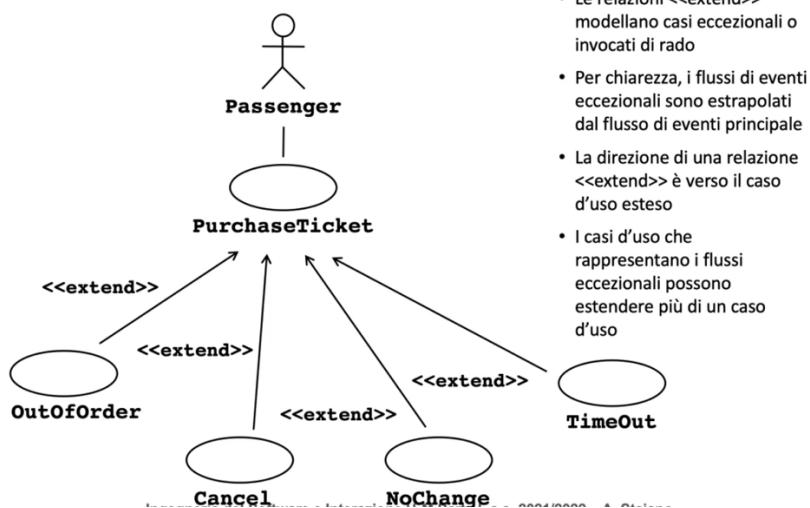
Esempio_ relazione <<extended>> testuale

Nome	ConnectionDown
Attori Partecipanti	Comunica con FieldOfficer e Dispatcher
Flusso eventi:	...
Condizioni di entrata	Questo caso d'uso estende i casi d'uso OpenIncident e AllocateResources. E' iniziato dal sistema quando la connessione di rete tra il FieldOfficer e il Dispatcher è persa.
Condizioni di uscita	...

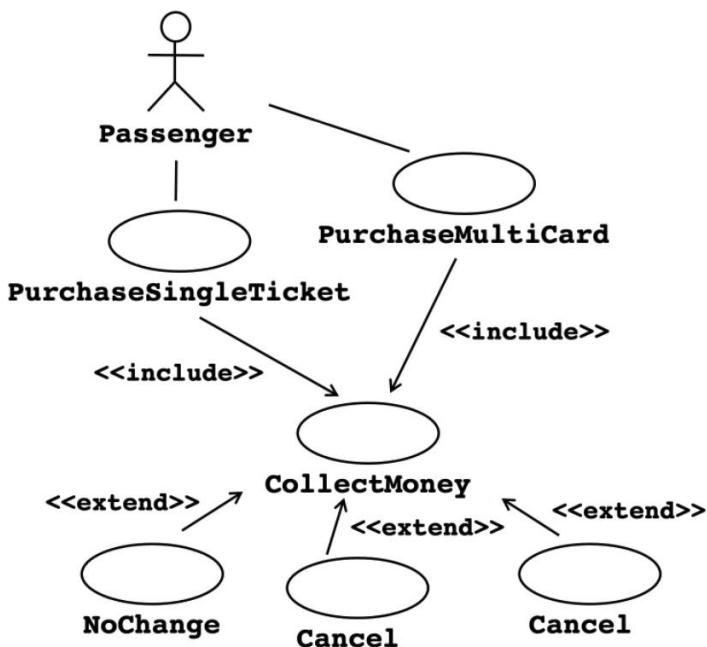
Il caso d'uso estensione va in una tabella a parte.

Ancora esempi sull'estensioni

Relazione <<extend>>



Estensioni e inclusioni insieme: esempio



RELAZIONI DI EREDITARIETA'

Terzo meccanismo per ridurre la complessità di un modello.

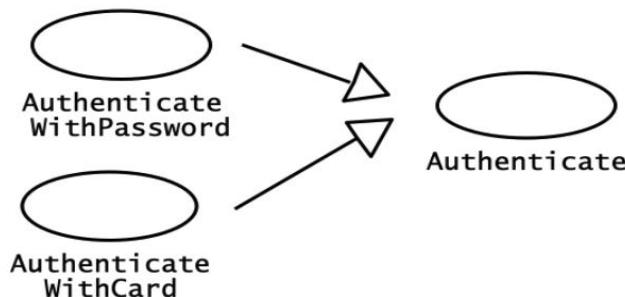
Un caso d'uso può specializzare un altro caso più generale aggiungendo più dettagli.

Ad esempio, ai FieldOfficeer è richiesto di autenticarsi prima di usare Friend:

- Durante le prime fasi della scoperta dei requisiti, l'autenticazione è modellata come un caso ad alto livello Authenticate.
- In seguito, gli sviluppatori descrivono Authenticate con più dettaglio e consentono diverse piattaforme HW.
- Il risultato sono due ulteriori casi, AuthWithPswd e AuthWithCard.

I due nuovi casi sono rappresentati come specializzazioni del caso d'uso Authenticate.

Nella rappresentazione testuale, i casi specializzati ereditano l'attore che inizia e le condizioni di entrate e uscita dal caso generale.



Esempio: relazione di ereditarietà testuale

Nome	AuthenticateWithCard
Attori Partecipanti	Ereditato dal caso d'uso Authenticate
Flusso eventi:	1. Il FieldOfficer inserisce la propria carta nel Terminale 2. Il terminale di campo accetta la carta e richiede all'attore il PIN 3. Il FieldOfficer immette il PIN con il tastierino numerico 4. Il Terminale controlla il PIN immesso con il PIN memorizzato sulla carta. Se i PIN sono uguali, il FieldOfficer è autenticato. Altrimenti, il Terminale rifiuta il tentativo di autenticazione
Condizioni di entrata	Ereditato dal caso d'uso Authenticate
Condizioni di uscita	Ereditato dal caso d'uso Authenticate

Anche in questo caso la tabella è indipendente.

EXTEND E INHERITANCE

Le relazioni extend e inheritance sono differenti:

- In extend ogni caso descrive un differente clusso di eventi per fare un compito differente.
 - o Il caso OpenIncident descrive le azioni che si verificano quando il Dispatcher crea un nuovo incidente, mentre ConnectionDown descrive le azioni che occorrono durante l'interruzione della connessione.
- Nel caso della relazione di ereditarietà, AuthWithPasswd e Authenticate descrivono entrambe le azioni che si verificano durante l'autenticazione, anche se differenti a livelli di astrazione.

APPLICAZIONE DEI EDIAGRAMMI DI CASI D'USO

I casi d'uso e gli attori definiscono i confini del sistema.

Sono sviluppati durante la fase di scoperta dei requisiti, spesso con i clienti e gli utenti.

Durante la specifica, i casi sono rifinati e corretti poiché sono rivisti anche dagli sviluppatori e validati rispetto ai casi reali.

Lezione 4

DIAGRAMMI DELLE CLASSI

I diagrammi delle classi descrivono la struttura del sistema in termini di classi e oggetti.

Le classi sono astrazioni che specificano gli attributi e il comportamento di un insieme di oggetti.

Gli oggetti sono entità che encapsulano lo stato e il comportamento.

In UML, classi e oggetti sono rappresentati da riquadri composti da 3 compartimenti:

1. Parte alta: nome della classe o dell'oggetto
2. Centro: attributi
3. Parte bassa: operazioni

La parte relativa agli attributi e alle operazioni possono essere omesse per chiarezza.

CONVENZIONI UML PER CLASSI E OGGETTI

I nomi degli oggetti sono sottolineati per indicare che sono istanze.

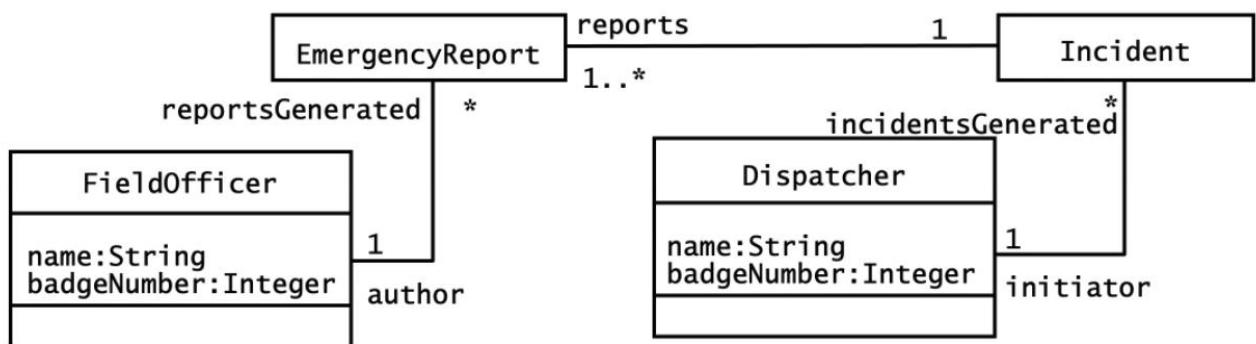
I nomi delle classi iniziano con le lettere **maiuscole**.
Agli oggetti possono essere assegnati dei nomi.

ATTORE VS CLASSE VS OGGETTO

- Attore: un'entità da modellare esterna al sistema, interagisce col sistema (passeggero).
 - Classe: astrazione che modella un'entità nel dominio applicativo o del dominio della soluzione (user, server, distributore biglietti)
 - Oggetto: specifica istanza di una classe (joe, il passeggero che sta comprando un biglietto dal distributore).
-

ESEMPIO DIAGRAMMA DELLE CLASSI

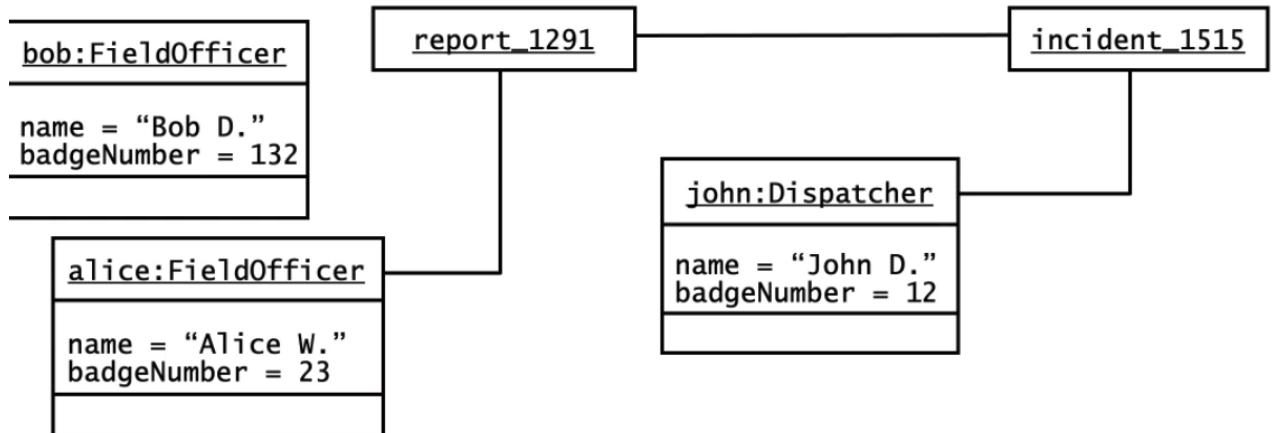
Classi partecipanti nel caso d'uso *ReportEmergency*



Rappresentiamo il FieldOfficer che ha un nome e un #tesserino, lo stesso per il Dispatcher. Il FOfficer avvia

il caso d'uso che consiste nella creazione della stesura di un rapporto. Rappresentiamo il rapporto come una classe *EmergencyReport* e poi una classe *Incident* che è connesso col dispatcher e con la classe *EmergencyReport*. Le linee rappresentano le associazioni tra le classi.

Esempio di diagramma degli oggetti: oggetti partecipanti nello scenario [warehouseOnFire](#)



Questo diagramma rappresenta uno scenario specifico.

Abbiamo uno specifico incidente, uno specifico dispatcher e due FO.

ASSOCIAZIONI E COLLEGAMENTI (LINK)

Un collegamento rappresenta una connessione tra due oggetti.

Le associazioni sono relazioni tra classi e rappresentano gruppi di link.

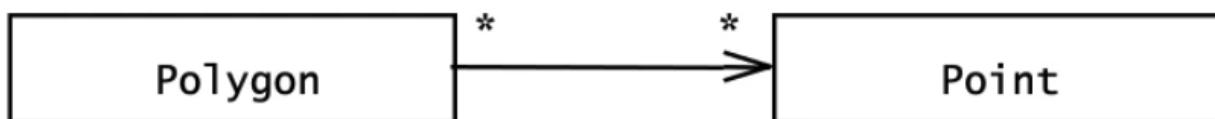
Nell'esempio FRIEND, ogni oggetto FO ha una lista di *EmergencyReport* che sono stati scritti dal FO.

ASSOCIAZIONI SIMMETRICHE E ASIMMETRICHE

Le associazioni possono essere bidirezionali (simmetriche) o unidirezionali (asimmetriche).

Un'associazione asimmetrica è quella tra le classi *Polygon* e *Punto*:

- La freggia di navifazione indica che il sistema supporta solo il verso da poligono a punto.
 - Dato un poligono posso determinare tutti i punti che appartengono al poligono e non viceversa.



Gli sviluppatori in fase di analisi tendono a non fare associazioni asimmetriche, rimandandole in fase

progettuale per evitare che il cliente possa avere difficoltà di comprensione.

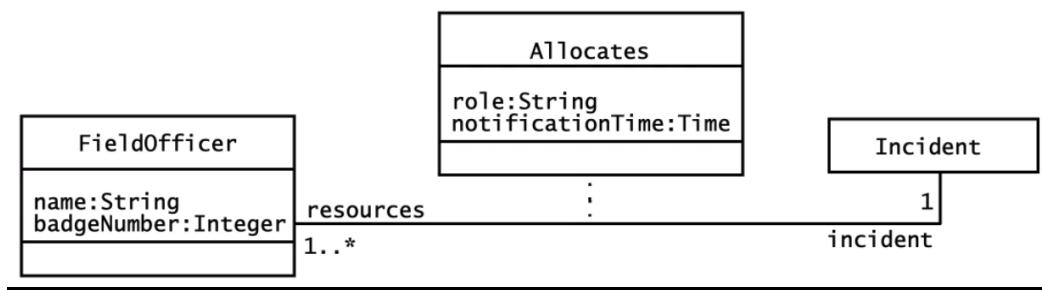
CLASSE ASSOCIAZIONE

Le associazioni sono simili alle classi poiché possono avere attributi e operazioni.

Una tale associazione è chiamata *classe associazione*:

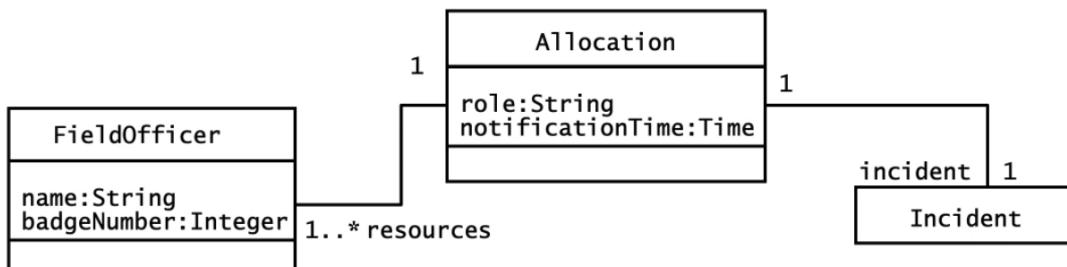
- Rappresento l'associazione con una classe e le collego con una linea tratteggiata

Esempio di classe di associazione



L'uso di una classe di associazione o di una classe dipendono dalle applicazioni e dell'esperienza acquisita.

Qualsiasi classe di associazione può essere trasformata in una classe e associazioni semplici.

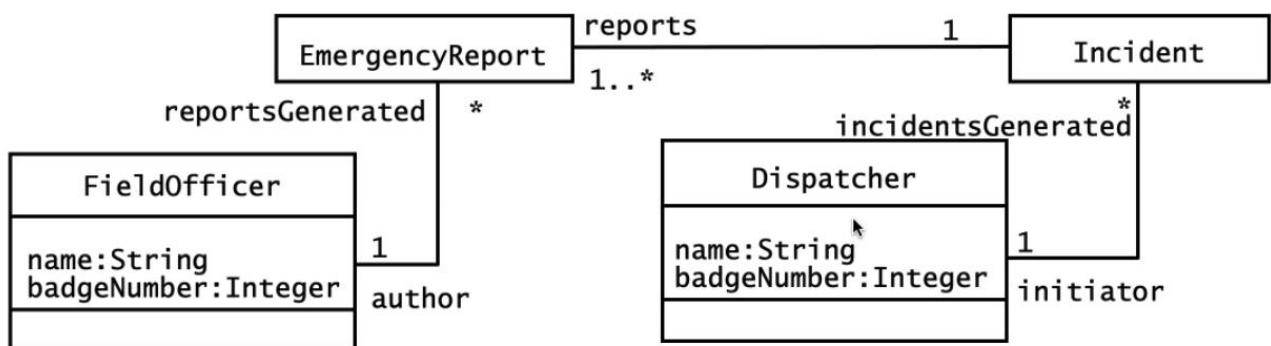


RUOLI

Ciascuna estremità di un'associazione può essere etichettata con una stringa chiamata **ruolo**.

I ruoli dell'associazione tra le classi *EmergencyReport* e *FO* sono autore e rapporto generato.

Etichettare le estremità con i ruoli consente di distinguere tra le multiple associazioni che si originano da una classe. Inoltre i ruoli chiariscono lo scopo dell'associazione.



AGGREGAZIONE

Le associazioni sono usate per rappresentare un'ampia gamma di connessione tra un insieme di oggetti.

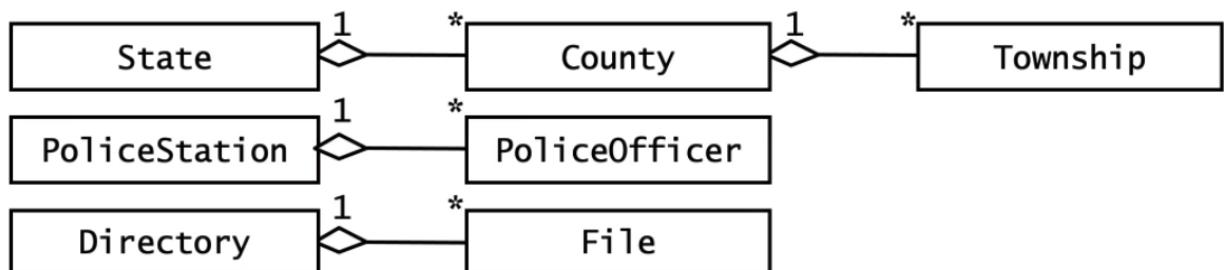
Un tipo speciale di associazione si presenta spesso: le aggregazioni (denotate con una linea con testa di diamante):

- Esempi: uno stato contiene molti Paesi che a loro volta contengono molte città

- Una stazione di polizia è costituita da più poliziotti
- Una directory contiene un certo numero di file

Tali relazioni possono essere modellate con associazioni 1-a-molti ma denota anche associazioni molti a molti.

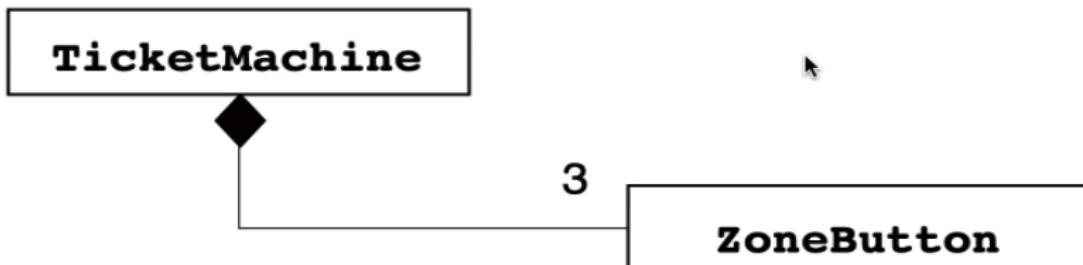
Esempi di aggregazioni



COMPOSIZIONE

Un rombo solito denota una composizione.

Una forma forte di aggregazione dove il tempo di vita delle istanze è controllato dall'aggregato.



Se il distributore non esiste, il bottone non esisterà mai.

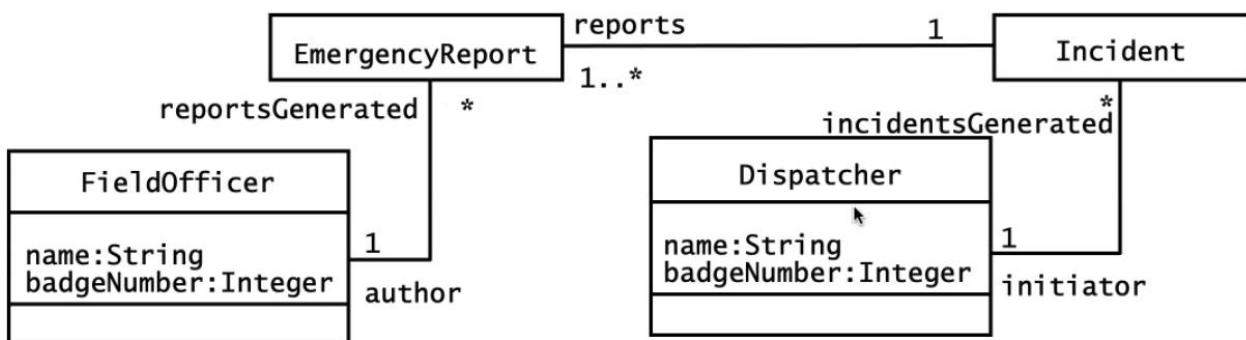
Le entità componenti non hanno vita propria e sono dipendenti all'entità di cui fa parte.

MOLTEPLICITA'

Ogni estremità di un'associazione può essere etichettata con un insieme di interi che indicano il numero di link che si originano da un'istanza della classe connessa all'estremità dell'associazione. Questo insieme di interi si chiama molteplicità.

Esempio molteplicità

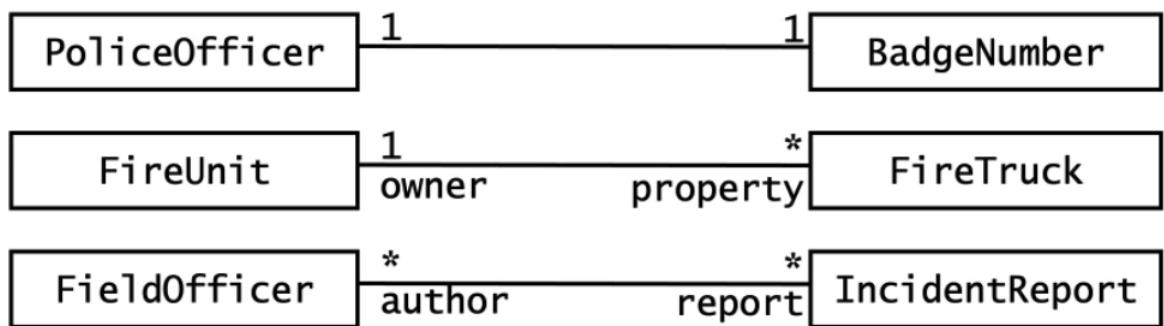
- L'estremità dell'associazione *author* ha una molteplicità pari ad 1
 - Significa che tutti gli *EmergencyReport* sono scritti esattamente da un *FieldOfficer* vale a dire, ogni oggetto *EmergencyReport* ha esattamente un link ad un oggetto della classe *FieldOfficer*
 - La molteplicità dell'estremità dell'associazione *reportsGenerated* è "molti" ed indicata con un asterisco (*) che indica 0..n



TIPI DI MOLTEPLICITA'

- Uno a uno
- Uno a molti
- Molti a molti

Esempi di molteplicità



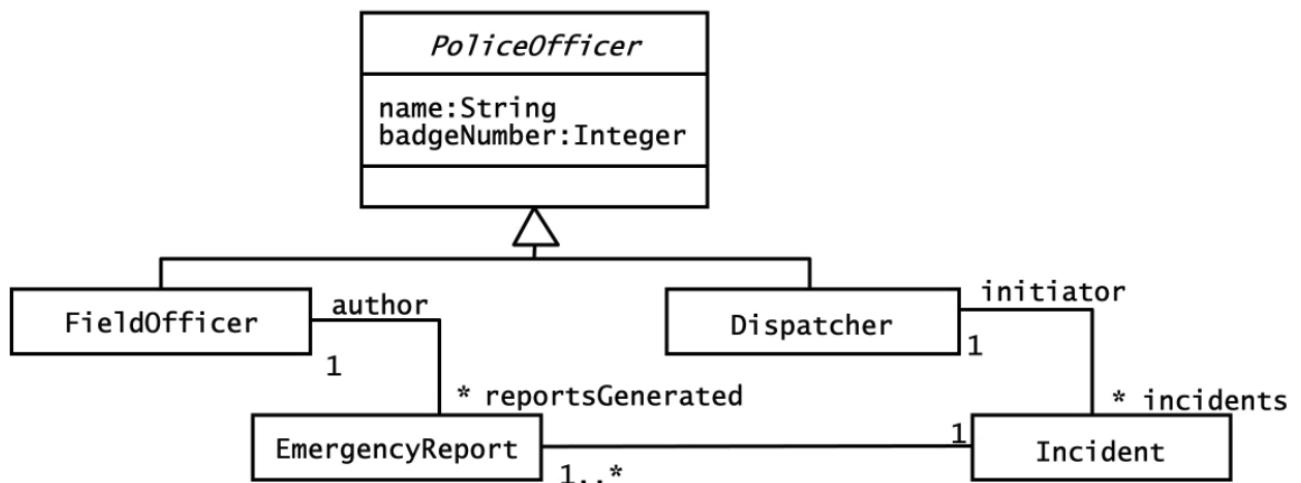
EREDITARIETA'

L'ereditarietà è la relazione tra una classe generale e una o più classi specializzate.

L'ereditarietà consente di descrivere tutti gli attributi e le operazioni che sono comuni a un insieme di classi.

- Esempio: *FieldOfficer* e *Dispatcher* hanno entrambi gli attributi *name* e *badgeNumber*. Tuttavia, *FieldOfficer* ha un'associazione con *EmergencyReport*, mentre *Dispatcher* ha un'associazione con *Incident*. Gli attributi comuni di *FieldOfficer* e *Dispatcher* possono essere modellati introducendo una classe *PoliceOfficer* che è specializzata da *FieldOfficer* e *Dispatcher*
- *PoliceOfficer* è chiamata la **superclasse**; *FieldOfficer* e *Dispatcher* sono chiamate **sottoclassi**
- Le sottoclassi ereditano gli attributi e le operazioni della loro superclasse

Esempio di generalizzazione



CLASSI ASTRATTE

La classe POfficer è astratta perché non ha una corrispondenza immediata col dominio dell'applicazione: la si riconosce perché è scritta in corsivo.

OGGETTI E OPERAZIONI

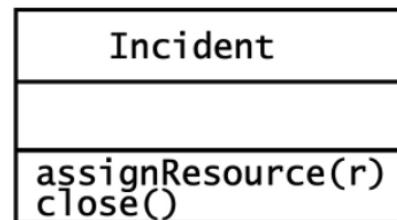
Il comportamento di un oggetto è specificato dalle operazioni.

Un oggetto richiede l'esecuzione di un'operazione a un altro oggetto inviando un messaggio.

Il messaggio è confrontato col metodo definito dalla classe.

ESEMPIO di operazioni della classe *Incident*

- L'operazione *assignResource(r)*, dato un FieldOfficer, crea un'associazione tra l'Incidente ricevente e la Risorsa specificata
- L'operazione *close()* chiude l'incidente
 - Attraversa tutte le risorse assegnate all'incidente e raccoglie i relativi rapporti



APPLICAZIONE DEI DIAGRAMMI DELLE CLASSI

Usati per descrivere la struttura del sistema.

Durante la fase di analisi gli ingegneri SW costruiscono diagrammi delle classi per formalizzare la conoscenza del dominio dell'applicazione.

Le classi rappresentano gli oggetti partecipanti individuati nei diagrammi dei casi d'uso e di interazione descrivendo i loro attributi e le operazioni.

Lo scopo dei modelli di analisi è descrivere lo scopo del sistema e scoprire i suoi confini: ad esempio un analista può esaminare la molteplicità dell'associazione tra FO e EmergencyReport e chiedere se all'utente se ciò è corretto.

La fase di analisi tiene fuori i concetti implementativi.

DIAGRAMMI DI INTERAZIONI E DI STATO

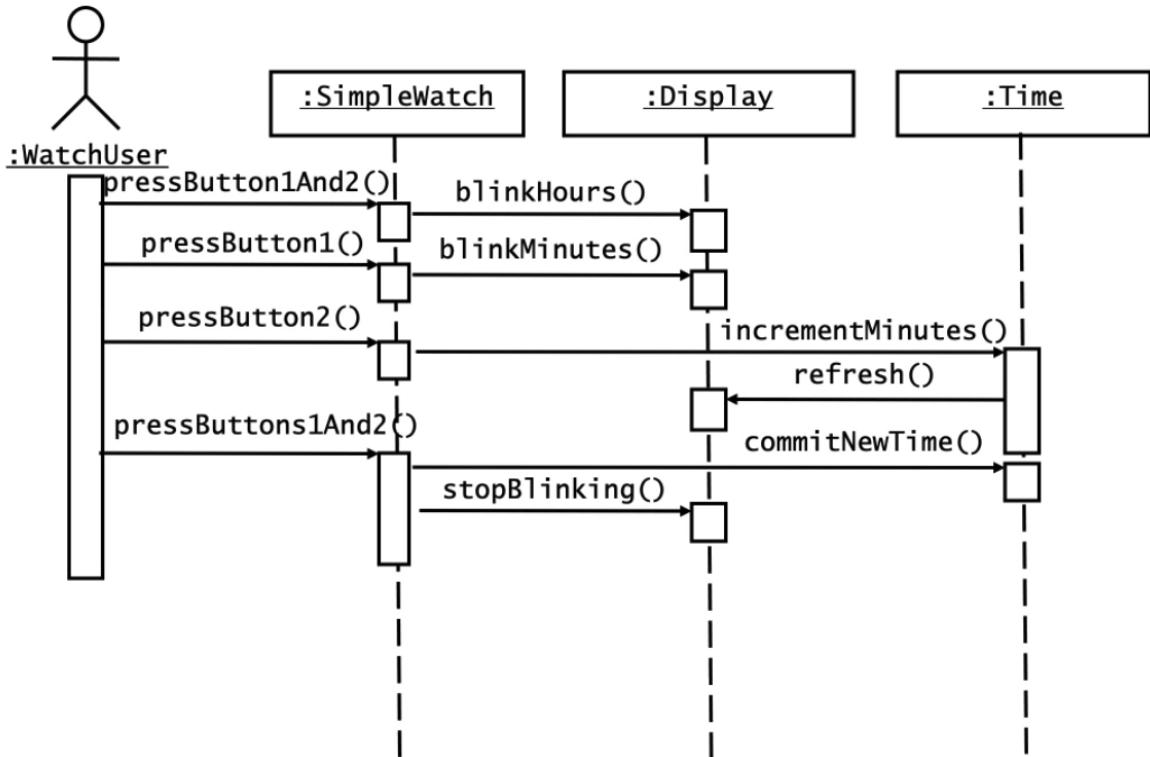
Diagrammi di INTERAZIONE: descrivono le modalità di comunicazione tra un insieme di oggetti interagenti. Un oggetto interagisce con un altro inviando messaggi:

- La ricezione di un messaggio aziona l'esecuzione di un metodo
 - Possono essere inviati degli argomenti insieme al messaggio compatibile coi parametri del metodo di cui si richeide l'esecuzione.
-

DIAGRAMMA DELLE SEQUENZE

Sono usati durante l'analisi dei requisiti per rifinire le descrizioni dei casi d'uso e per trovare oggetti aggiuntivi e durante la progettazione del sistema per rifinire le interfacce dei sottosistemi.

I diagrammi delle sequenze rappresentano orizzontalmente gli oggetti partecipanti nell'interazione e verticalmente il tempo.
Esempio: un orologio con 2 pulsanti (2Bwatch).

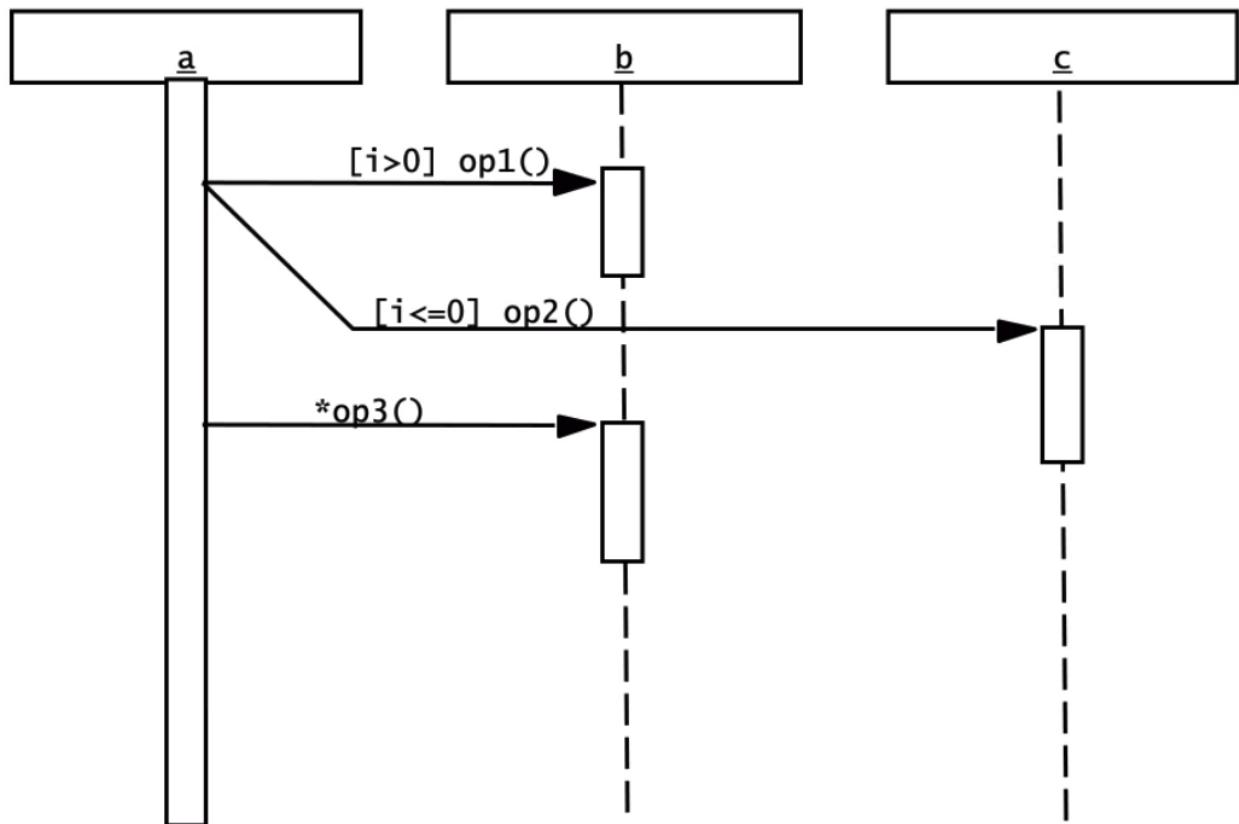


Vedere notazioni dalle slide

CONDIZIONI E ITERAZIONI

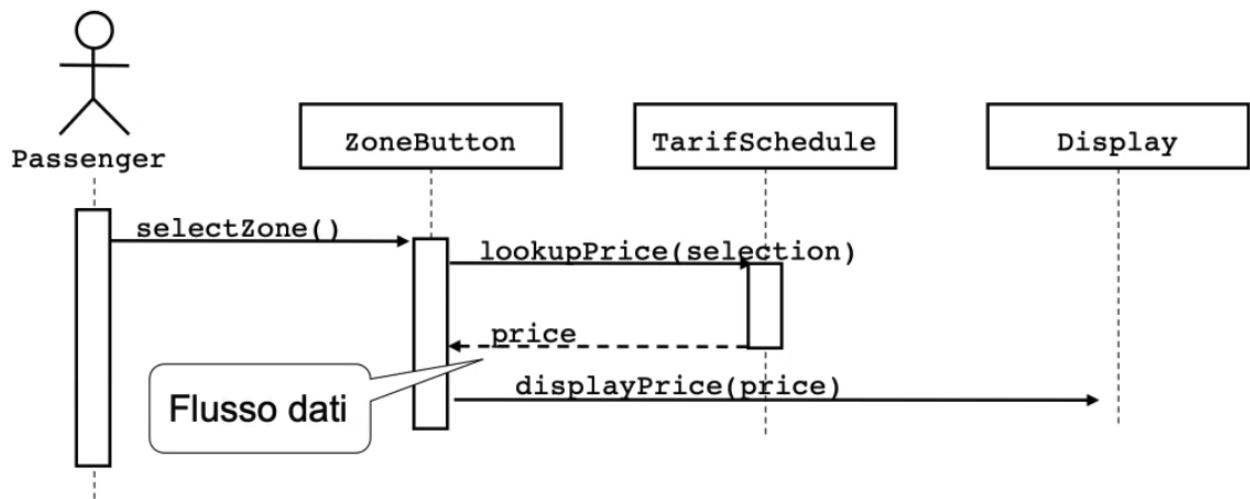
I diagrammi delle sequenze possono essere usati per descrivere una sequenza astratta o concreta.
Sono disponibili notazioni per esporre condizioni.

Esempi di condizioni e iterazioni

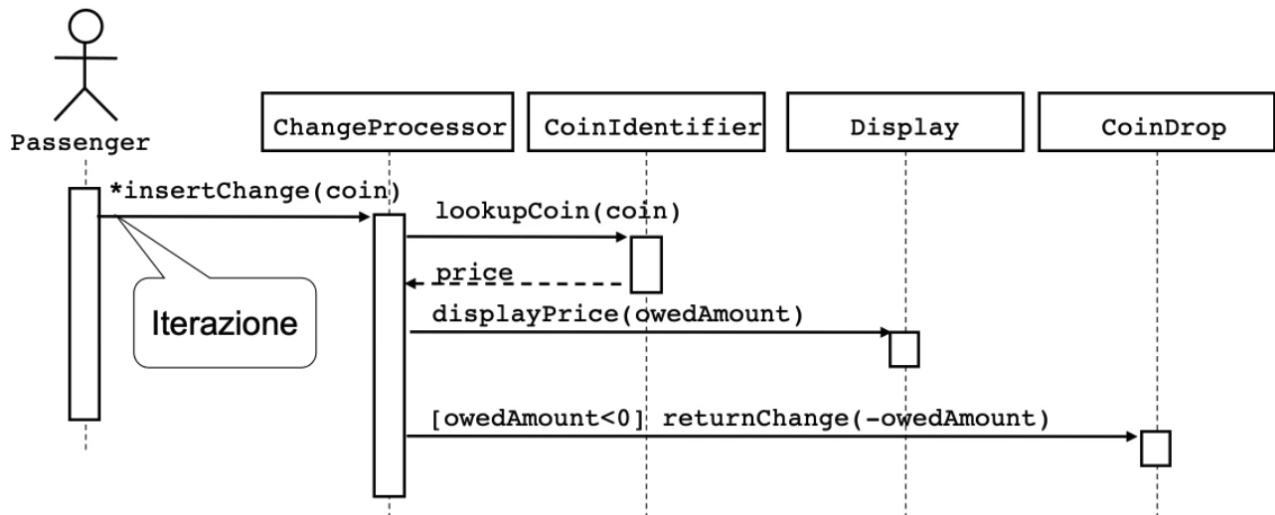


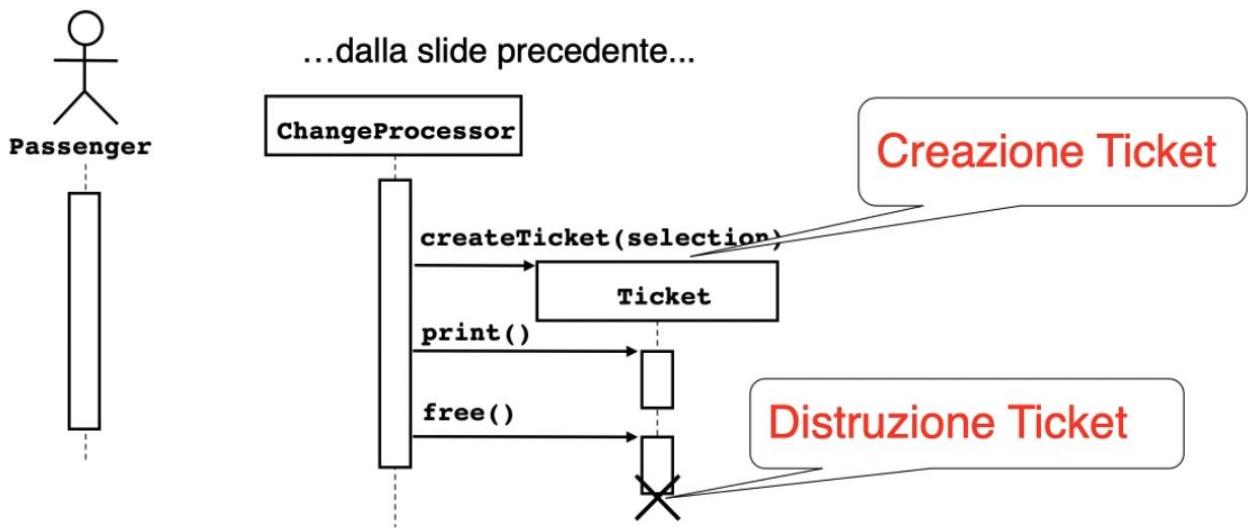
A invia un messaggio all'oggetto B che invoca `op1()`.
Questo messaggio verrà inviato a B solo se la
variabile è maggiore di zero, oppure viene inviato un
altro messaggio all'oggetto C.
Mentre un loop è indicato con un * prima del nome
del messaggio.

I diagrammi delle sequenze: flusso dati



In questo caso *price* è un dato che viene restituito dalla classe TariffSchedule.





- La creazione è denotata da un messaggio freccia che punta all'oggetto
- La distruzione è denotata da una x alla fine dell'attivazione

PROPRIETA' DIAGRAMMI DELLE SEQUENZE

Rappresentano il comportamento in termini di interazioni.

Utile per identificare o trovare oggetti mancanti.

Comporta tempo per la preparazione.

Complementari al diagramma delle classi.

DIAGRAMMA DELLE COLLABORAZIONI

Descrivono le stesse info dei diagrammi delle sequenze.

Rappresentano la sequenza dei messaggi numerando le interazioni. Ciò elimina la necessità di vincoli geometrici sugli oggetti col risultato di diagrammi compatti. Tuttavia, la sequenza di messaggi diventa

più difficile da seguire. Vale quindi la pena avere un diagramma più lineare.

Esempio: diagramma collaborazioni per 2Bwatch

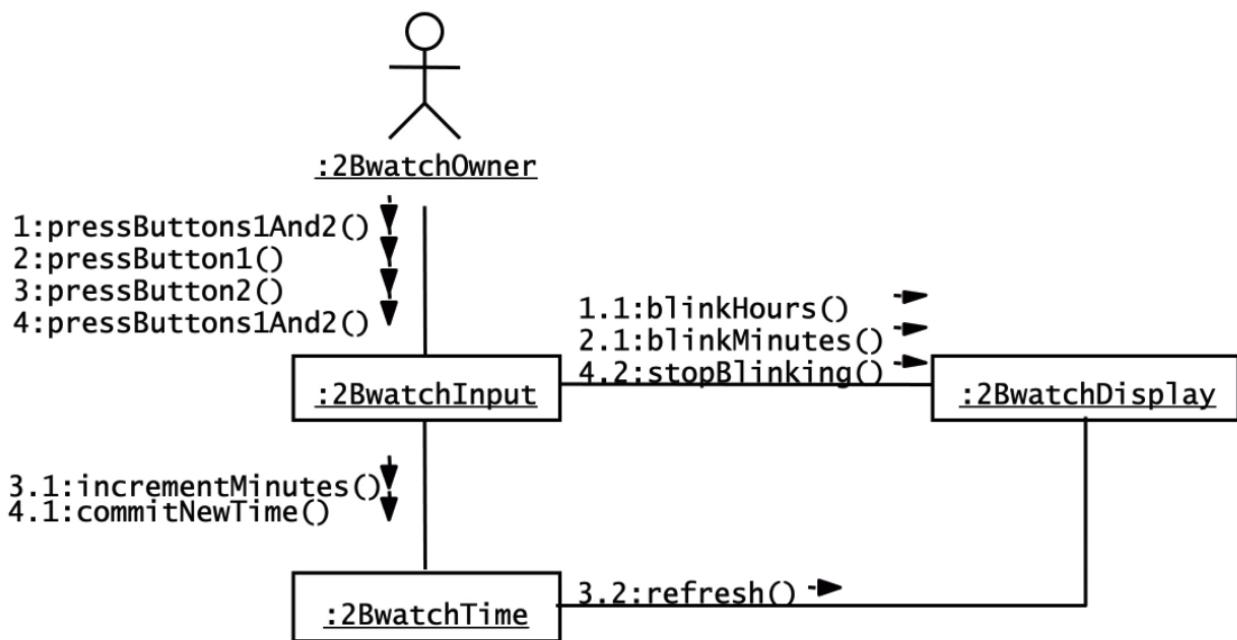


DIAGRAMMA DEGLI STATI

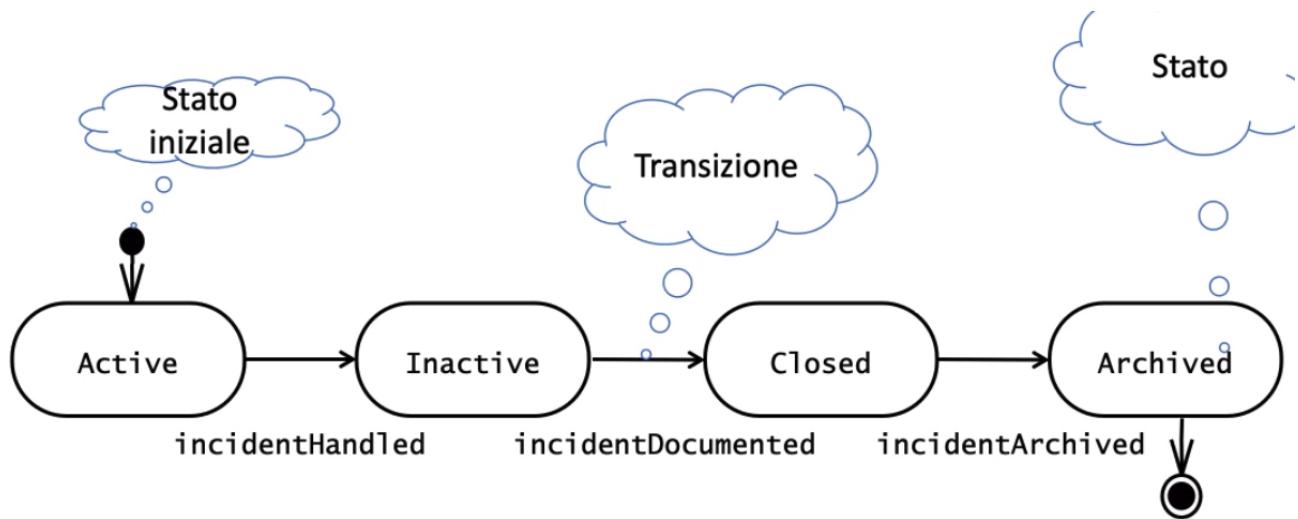
Descrivono la sequenza di stati di un oggetto in seguito all'occorrenza di eventi esterni.

Sono estensioni del modello della macchina a stati finiti: forniscono le notazioni per innestare stati e forniscono notazioni per legare le transizioni agli invii di messaggi e le condizioni sugli oggetti.

Uno stato è una condizione soddisfatta dagli attributi di un oggetto.

AD esempio: un oggetto Incident IN FRIEND può trovarsi in uno di 4 stati:

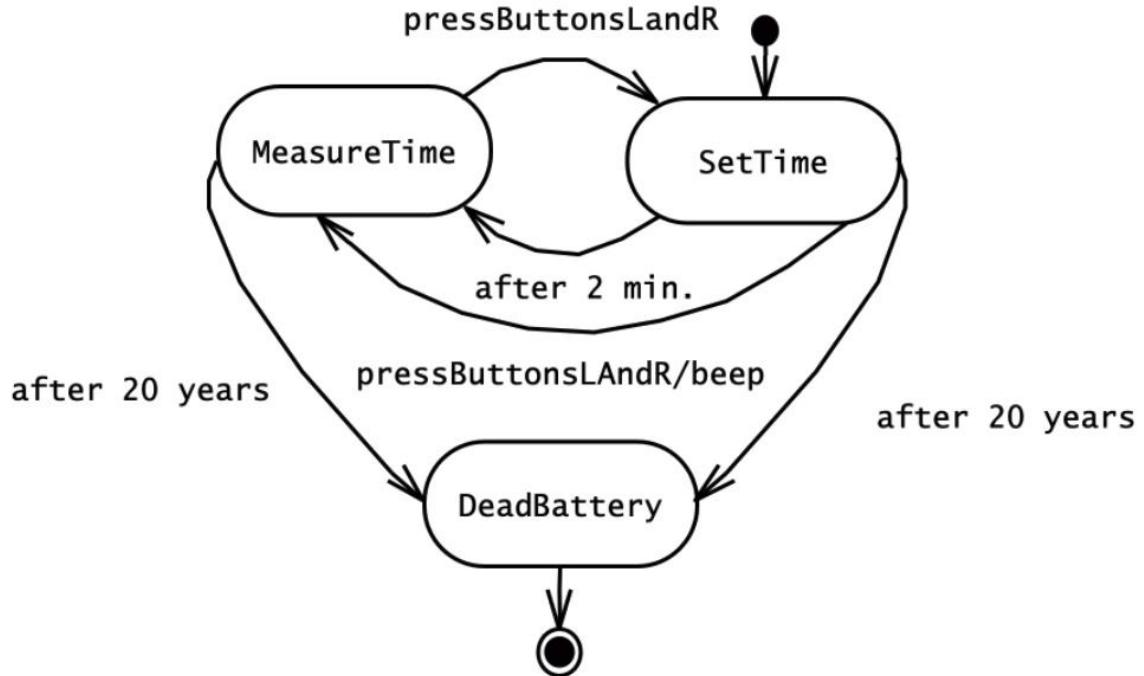
- Stato **Active**: situazione che richiede una risposta
- Stato **Inactive**: situazione gestita ma i cui rapporti non sono stati ancora scritti
- Stato **Closed**: situazione gestita e documentata
- Stato **Archived**: incidente in stato Closed la cui documentazione è stata archiviata



- I quattro stati possono essere rappresentati con un singolo attributo, **status**, nella classe Incident che può assumere uno dei quattro valori {*Active*, *Inactive*, *Closed*, *Archived*}
- In generale, uno stato può essere determinato dai valori di più attributi

ESEMPIO 2Bwatch:

- Al più alto livello di astrazione, 2Bwatch ha due stati, *MeasureTime* e *SetTime*
- 2Bwatch cambia stato quando l'utente preme e rilascia entrambi i pulsanti simultaneamente
- Durante la transizione dallo stato *SetTime* allo stato *MeasureTime*, 2Bwatch emette un beep
 - Indicato dall'azione **/beep** sulla transizione
- Quando 2Bwatch è acceso per la prima volta si trova nello stato *SetTime*
 - Indicato dal cerchietto nero che indica lo stato iniziale
- Quando la batteria si esaurisce, l'orologio non funziona
 - Cerchietto nero circondato da un altro cerchietto, lo stato finale

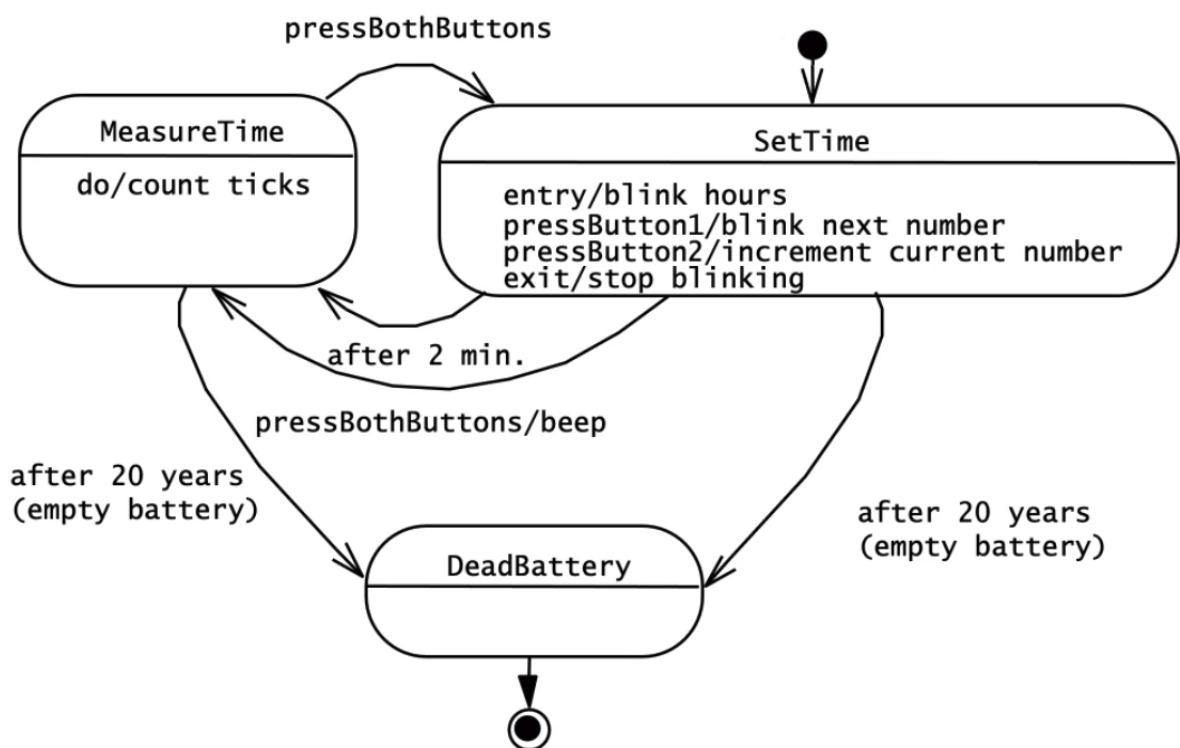


AZIONI

Le azioni sono piccoli comportamenti atomici eseguiti in punti specifici nella macchina a stati

TRANSIZIONI INTERNE

È un'azione che non comporta un cambiamento di stato



Lezione 5 → DIAGRAMMI DELLE ATTIVITA'

Forniscono la sequenza di operazioni che definiscono un'attività più complessa.

Essi permettono di rappresentare processi paralleli e la loro sincronizzazione.

Ogni stato è un'azione.

- Un diagramma delle attività può essere associato:
 - A una classe
 - A un caso d'uso
 - All'implementazione di un'operazione
-

ESEMPIO (class Incident)

- Un'azione è un'attività atomica: è rappresentata nel diagramma con un rettangolo arrotondato ed è identificata con una voce verbale che descrive l'azione stessa:
 - *HandleIncident*: il dispatcher riceve i rapporti e alloca risorse
 - *DocumentIncident*: tutti i FO e i Dispatcher documentano l'incidente dopo che è stato chiuso
 - *ArchiveIncident*: archiviazione delle info su dispositivi di memorizzazione.



Gli archi tra le attività rappresentano il flusso di controllo. Un’attività può essere eseguita solo dopo che le attività precedenti sono state già completate.

Ogni diagramma delle attività ha due nodi particolari:

- Inizio
- Fine
- Inizio: indica la prima azione da eseguire ed è rappresentato da un cerchio con solo archi in uscita
- Fine.

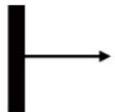
Un arco collega tra loro i nodi: rappresenta il flusso di esecuzione delle attività, è rappresentato con una freccia →

Una funzionalità complessa non è costituita da una semplice successione di azioni in sequenza: può presentare azioni da eseguire contemporaneamente o sotto particolari condizioni

NODI DI CONTROLLO

• Fork

- Descrive l'esecuzione in parallelo di più azioni: quelle puntate dal nodo fork sono avviate contemporaneamente ed in parallelo
 - rappresentato da una barra nera puntata da un solo arco e dalla quale partono due o più archi verso le azioni da eseguire in parallelo

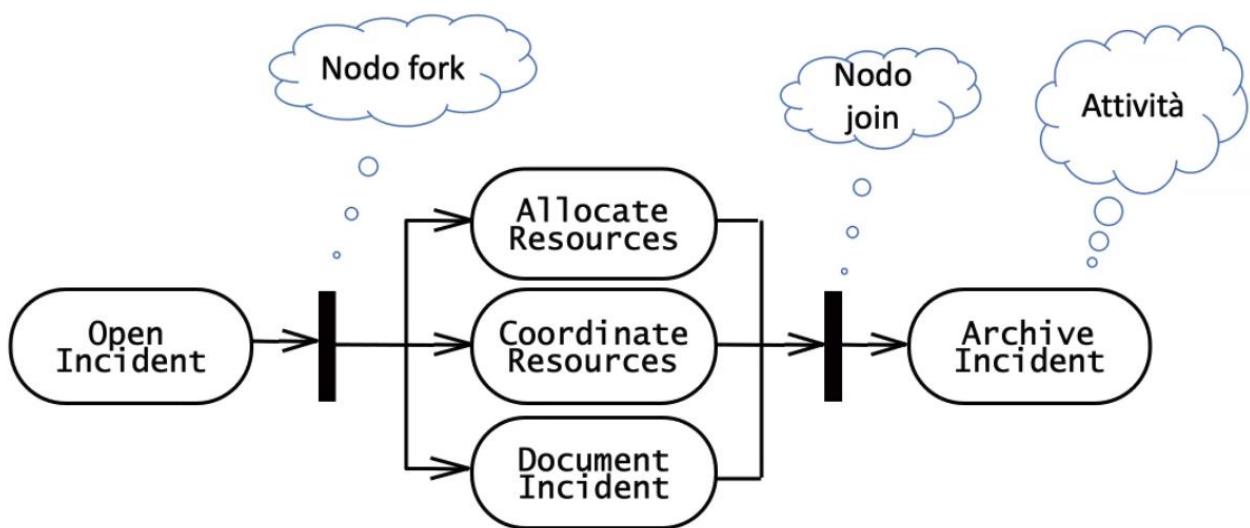


• Join

- E' il duale del fork. Specifica che un'azione è eseguita solo nel momento in cui le azioni precedenti hanno terminato la propria esecuzione. E' definito anche sincronizzazione perché sincronizza due rami svolti parallelamente, generando un unico flusso di esecuzione.
 - E' rappresentato da una barra nera →



Esempio di fork a join in *OpenIncident*:



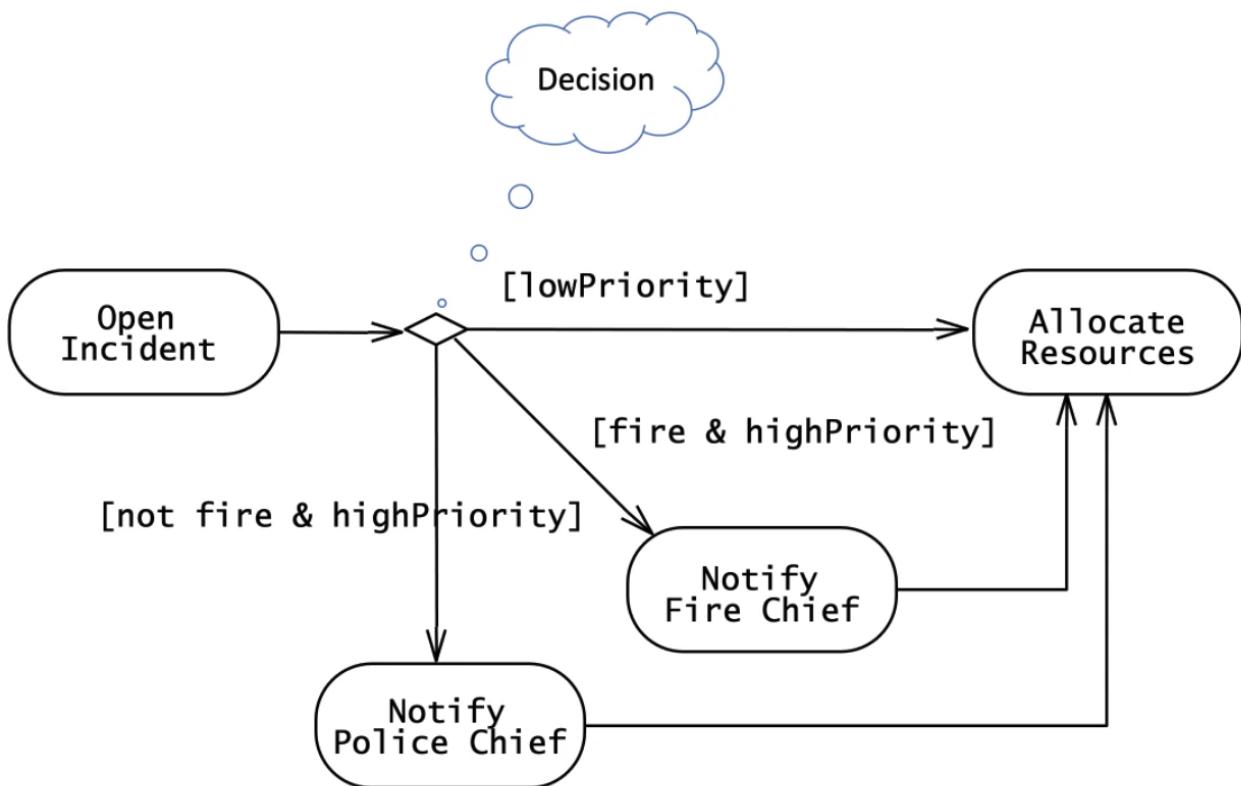
- Dopo aver aperto la *openIncident* possono accadere 3 eventi parallelemente. La notazione prevede che dopo aver completato OpenIncident possono avviare in parallelo le 3 attività. Quando

sono completate vanno in un nodo join che permette di iniziare l'attività *ArchiveIncident*

ALTRI NODI DI CONTROLLO

- Decision: indica che l'esecuzione di un'azione dipende dal verifarsi di una determinata condizione chiamata **guard**. Descrive il fatto che al termine di un'azione, lo scenario può proseguire in modo diverso con azioni che dipendono da specifiche situazioni.
- Merge:

Esempio di decisioni in *OpenIncident*



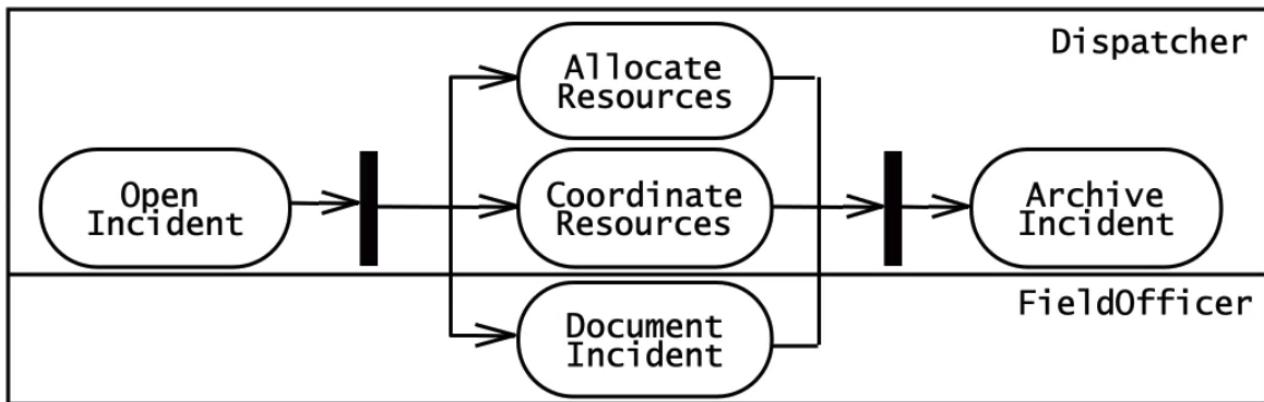
Quando si apre un incidente, a seconda della priorità posso scegliere se chiamare il capo della polizia (*NoIncendio e Alta priorità*) o dei pompieri (Incendio e AltaPriorità) o allocare direttamente le risorse (if *lowPriority*).

SWIMLANE

Le attività possono essere raggruppate in *swimlane* per denotare l'oggetto che le implementa:

- Sono rappresentate come rettangoli che racchiudono un gruppo di attività

Esempio:



La *swimlane* ci dice che determinate attività sono svolte dal *Dispatcher* mentre *Dlncident al FO*.

APPLICAZIONE DEI DIAGRAMMI DELLE ATTIVITA'

I diagrammi forniscono una visione task-centrica del comportamento di un insieme di oggetti.
Possono essere usati per descrivere vincoli di sequenza tra i casi d'uso, attività tra gruppi di oggetti o task di un progetto.

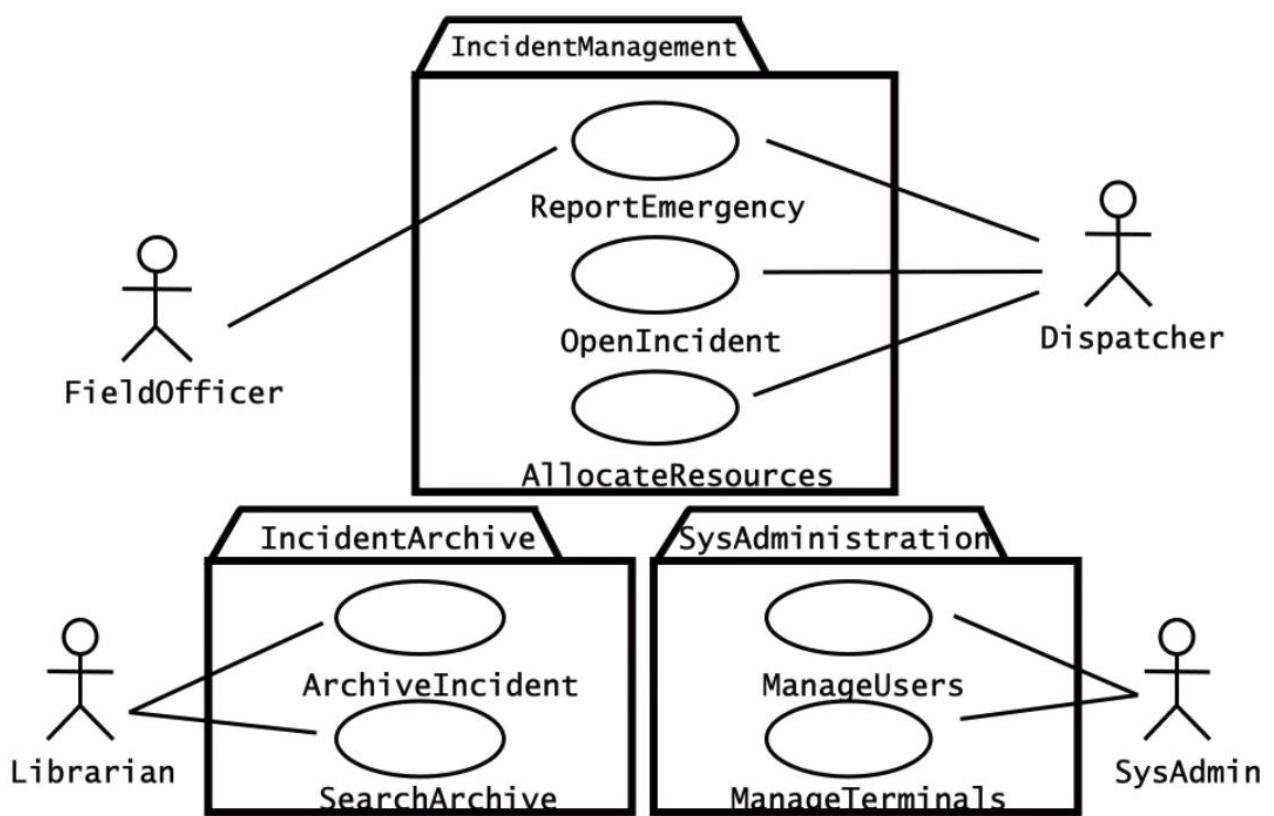
ORGANIZZAZIONE DEI DIAGRAMMI

I modelli di sistemi complessi diventano duri quando gli sviluppatori li rifiniscono.

La complessità può essere gestita raggruppando elementi in relazione tra loro in **package**.

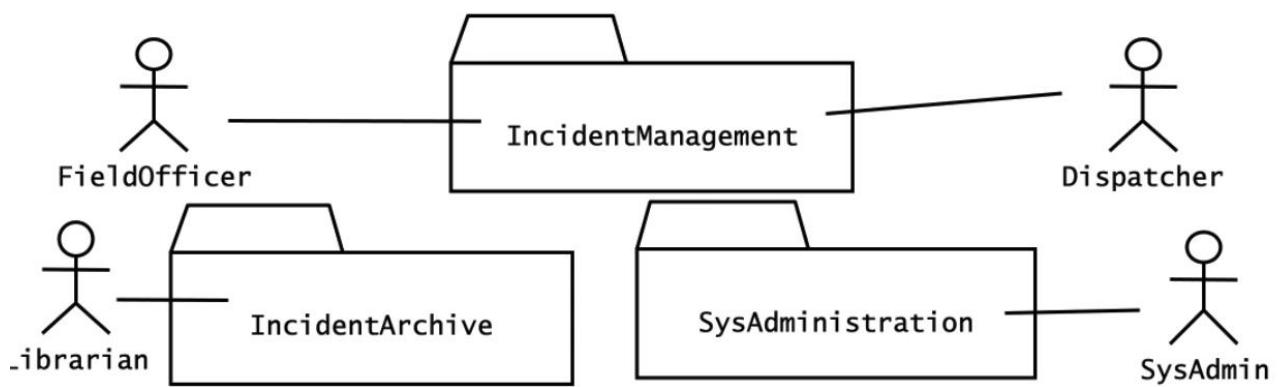
- Un package è un raggruppamento di elementi, come casi d'uso, classi, o attività che semplificano la comprensione

Esempio di package:



I package aiutano all'interpretazione e ci focalizzano sulle attività del sistema.

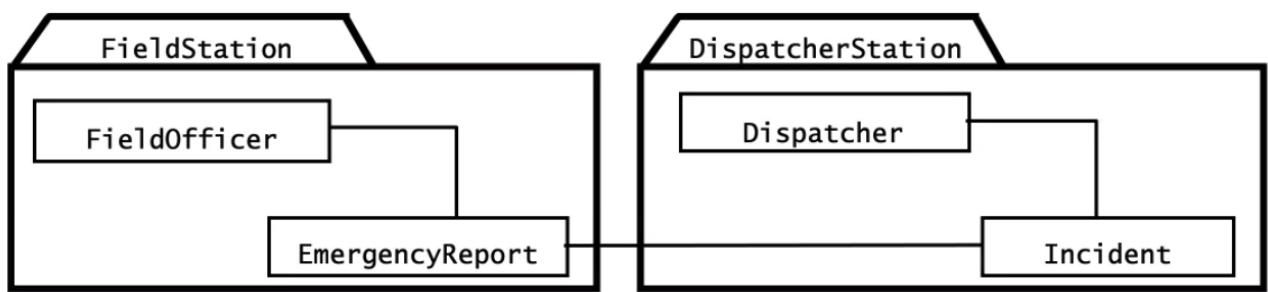
Possiamo usare i package anche in modo diverso: piuttosto che sulle funzionalità si mostrano i sottosistemi.



PACKAGE NEI DIAGRAMMI DELLE CLASSI

Anche i diagrammi delle classi possono essere organizzati in package.

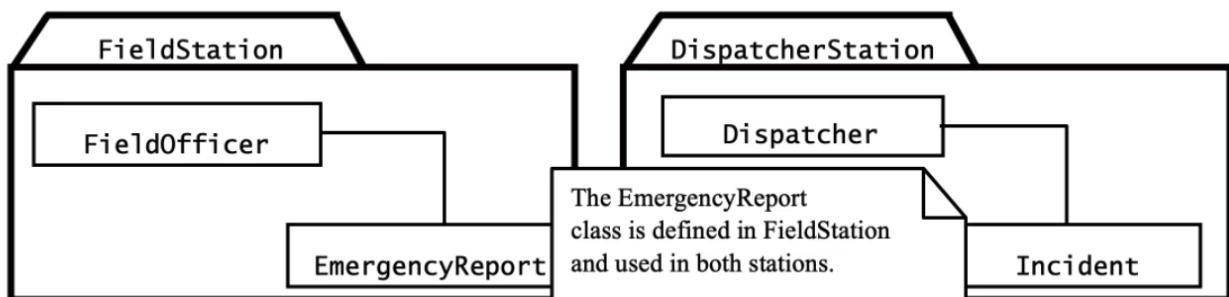
- Le classi del caso d'uso *ReportEmergency* sono organizzate rispetto al punto in cui gli oggetti sono creati
 - *FieldOfficer* e *EmergencyReport* sono parte del package *FieldStation*
 - *Dispatcher* e *Incident* sono parte di *DispatcherStation*



NOTE

Una nota è un commento allegato al diagramma.
Sono usate per allegare info ai modelli e agli elementi del modello.

Esempio di nota:



ESTENSIONE DEI DIAGRAMMI

L'obiettivo di UML consiste nel fornire un insieme di notazioni per modellare un'ampia classe di sistemi software.

Tuttavia, un insieme prefissato di notazioni non potrebbe consentire di perseguire un tale scopo. UML fornisce quindi un numero di meccanismi che consentono di estendere il linguaggio. Si usano quindi gli stereotipi.

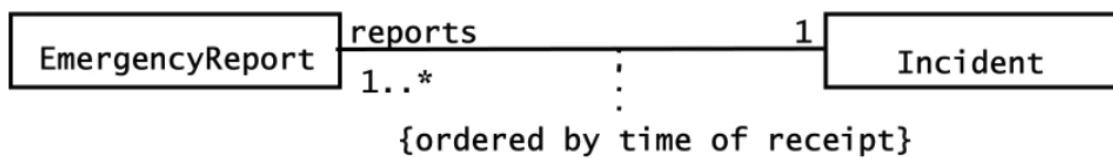
STEREOTIPI

Uno stereotipo è un meccanismo di estensione. È rappresentato da una stringa racchiusa tra << >> e allegata all'elemento a cui si applica, come classe o associazione.

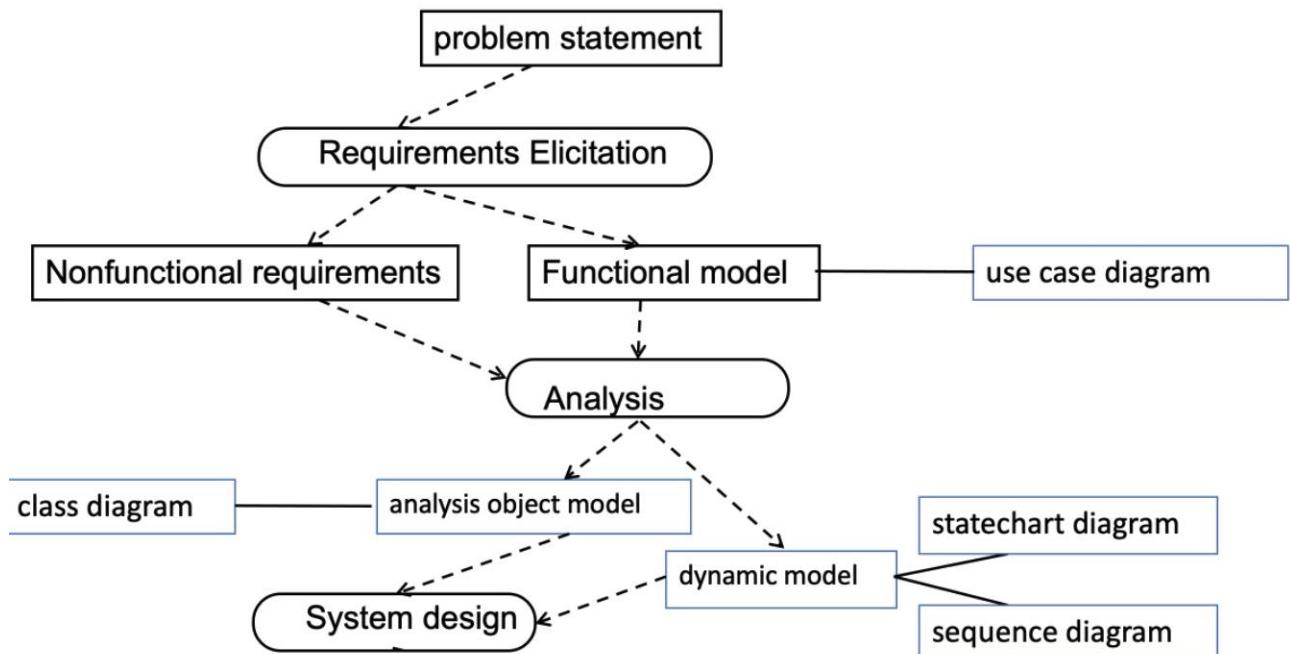
Formalmente, uno stereotipo corrisponde a creare una nuova classe nel metamodello UML.

VINCOLI

Un vincolo è una regola che viene allegata ad un elemento restringendone la semantica



ATTIVITA' DI SVILUPPO E RELATIVI PRODOTTI



Lez 7

GESTIONE DEI PROGETTI

I progetti progrediscono velocemente fino al 90% del completamento, dopo restano ferme per sempre. Se si consente di cambiare il contenuto di un progetto liberamente, il tasso dei cambiamenti supera il tasso di progresso.

DEFINIZIONE DI PROGETTO

Un progetto è un'attività, limitata nel tempo, finalizzata al raggiungimento di una serie di obiettivi che richiedono uno sforzo coordinato tra tutte le risorse partecipanti.

Un progetto include:

- Un insieme di deliverable ad un cliente
- Un programma di lavoro
- Risorse per le attività
- Attività tecniche e di gestione richieste per produrre e consegnare i deliverable

Il manager, un cliente, un membro del team, l'achitetto SW sono coinvolti nel capire l'architettura iniziale del sistema, il dominio e poi il progetto. Si centra il problema e si inizia a capire come gettare le fondamenta per il progetto.

Inizia il progetto: il manager imposta l'infrastruttura del progetto, assume i partecipanti e li organizza in team, definisce i milestone principali e avvia il progetto.

Quando tutti i team sono formati e hanno le proprie task, l'attività di sviluppo raggiunge il suo stato stabile.

I partecipanti sviluppano il sistema. Fanno i rapporti per i team leader.

I team leader rapportano lo stato dei loro team al manager che valuta lo stato del progetto.

I team leader rispondono per le deviazioni dal piano, riallocando i compiti agli sviluppatori e ottenendo nuove risorse dal manager.

Il manager è responsabile dell'interazione col cliente, per ottenere un accordo formale e rinegoziare risorse e deadline.

Quando il sistema viene completato c'è la fase finale con la consegna del sistema: il risultato del progetto è

consegnato al cliente ed è raccolta la storia del progetto.

ORGANIZZAZIONE DI UN PROGETTO

Un'organizzazione definisce le relazioni tra risorse in un progetto.

Un'organizzazione dovrebbe definire:

- Chi decide
 - Chi rapporta il suo stato a chi
 - Chi comunica a chi
-

RUOLO

Un ruolo definisce un insieme di responsabilità.

Esempi:

- Ruolo: Tester
 - Scrivere test
 - Riportare fallimenti
- Ruolo: Architetto di sistema
 - Assicura le decisioni di progettazione e definisce le interfacce dei sottosistemi.
 - Formula una strategia di integrazione del sistema.
- Ruolo: Liaison

- Facilita la comunicazione tra due team.

POSSIBILI CORRISPONDENZE RUOLI-PERSONE

- Uno a uno
 - Multi-poche: ogni membro ha più compiti.
Pericolo di impegno eccessivo.
 - Multi-a-molti: alcune persone non hanno ruoli significativi. Mancanza di responsabilità e perdita di contatto col progetto.
-

TASK

Un task descrive la più piccola quantità di lavoro monitorata dalla direzione, tipicamente impiegano dai 3 a 10 giorni di lavoro. Un insieme di task forma un'attività.

Un task è specificato da un work package:

- Descrizione del lavoro da fare
- Condizioni di ingresso e uscita per avviare il task
- Durata
- Risorse richieste

Un task deve avere dei criteri di completamento.

Un work product è un risultato visibile di un task, come ad esempio:

- Un documento
- La revisione di un documento
- Un pezzo di codice
- Una presentazione
- Un rapporto di un test

I wp consegnati al cliente sono detti *deliverable*.

DIMENSIONE DEI TASK

I task sono composti in dimensioni che ne consentono il monitoraggio.

Individuare la dimensione giusta è cruciale.

ATTIVITA'

È l'insieme dei task.

Culmina in un milestone di progetto maggiore:

- Evento programmato usato per misurare il progresso.
- I checkpoint interni non dovrebbero essere visibili all'esterno.

Le attività sono raggruppate ancora in attività di più alto livello con nomi diversi (fase1, fase2, passo 1, passo2, ...).

Possono esserci relazionio di precedenza tra le attività: ES:<< A1 deve essere eseguito prima di A2>>.

Lez 8

COMUNICAZIONE

Un esempio di cattiva comunicazione

- Da un rapporto di un incidente aereo:
 - *“Due scatole prodotte da diversi appaltatori erano unite da una coppia di fili”*



È successo che, grazie a un controllo prima del volo, i fili delle due scatole erano stati invertiti e bisognava risolvere il problema.

Dopo l'incidente: l'analisi post-volo ho rivelato che gli appaltatori avevano corretto i fili invertiti secondo le istruzioni. In effetti, lo avevano fatto entrambi!

La comunicazione è *essenziale*.

Durante gli sforzi di sviluppo di grandi sistemi, si impiega più tempo nella comunicazione che nella codifica.

Un ingegnere SW ha bisogno di avere le seguenti soft skills:

- Collaborazione: negoziare i requisiti col clienti e membri del team.
 - Presetnazione: presentare una parte importante del sistema durante una revisione.
 - Gestione: facilitare le riunione dei team.
 - Scrittura tecnica: scrivere parte della documentazione del progetto.
-

EVENTO DI COMUNICAZIONE VS MECCANISMO

Evento di comunicazione:

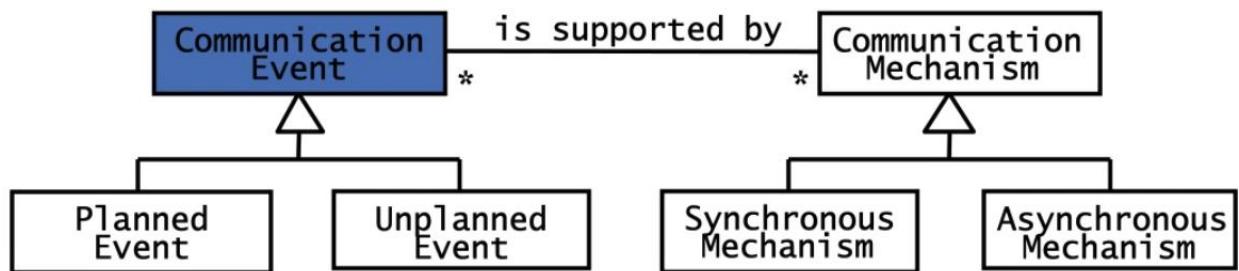
Scambio di info con obiettivi e ambito definiti.

- Programmato: comunicazione pianificata
 - Revisioni, riunioni coi teams...
- Non programmato: comunicazione guidata da eventi
 - Rapporto di un problema, modifiche...

Meccanismo di comunicazione:
tool o procedura che può essere usata per trasmettere informazioni.

- Sincrono: mittente e destinatario comunicano allo stesso momento.
 - Asincrono: mittente e destinatario non comunicano allo stesso momento.
-

MODELLARE LA COMUNICAZIONE



EVENTI DI COMUNICAZIONE PIANIFICATI

- Definizione del problema
 - Finalità: obiettivi correnti, requisiti e vincoli.
 - Esempio: presentazione del cliente.
 - Solitamente programmata all'inizio del prog.

- Revisione del progetto: attenzione sui modelli del sistema.
 - Finalità: valutare lo stato e revisione del sistema
 - Pianificato in prossimità di milestone.
- Revisione del cliente: attenzione sui requisiti
 - Finalità: informare il cliente e concordare la modifica dei requisiti.
 - La prima revisione del cliente è programmata dopo la fase di analisi.
- Peer review:
 - Walkthrough (informale)
 - Finalità: aumentare la qualità di un sottosistema.
 - Esempio: lo sviluppatore presenta il sottosistema ai membri del team.
 - Programmato da ogni team.
 - Ispezione (formale)
 - Finalità: conformità dei requisiti.
 - Esempio: dimostrazione del Sistema finale al cliente
 - Programmato dai gestori del progetto.

- Status review:
 - Finalità: trovare gli scostamenti rispetto a quanto pianificato e correggerli o identificare nuovi problemi
 - Esempio: sezione dello stato nella riunione settimanale periodica del team.
- Brainstorming:
 - Finalità: generare e valutare soluzioni a un problema
- Rilascio:
 - Finalità: fissare un prodotto di riferimento per il risultato di ogni attività di sviluppo del sw
 - Esempi:
 - Piano di gestione del progetto
 - Documento di analisi dei requisiti
 - Documento di progettazione del sistema
 - Versione beta sw
 - Versione finale sw
 - Manuale utente
- Revisione PostMortem:
 - Finalità: descrive la lezione imparata.
 - Programmata alla fine del progetto.

EVENTI DI COMUNICAZIONE NON PIANIFICATI

- Richiesta di chiarimento: la maggior parte avviene tra sviluppatore, clienti e utenti.

Esempio: uno sviluppatore può richiedere un chiarimento su una fase ambigua nella definizione del problema.

```
From: Alice  
Newsgroups: vso.discuss  
Subject: SDD  
Date: Wed, 2 Nov 9:32:48 -0400
```

When exactly would you like the System Design Document? There is some confusion over the actual deadline: the schedule claims it to be October 22, while the template says we have until November 7.

Thanks,-Alice

- Richiesta di modifica: un partecipante riporta un problema e propone una soluzione.

```
Report number: 1291 Date: 5/3 Author: Dave  
Synopsis: The STARS form should have a galaxy field.  
Subsystem: Universe classification  
Version: 3.4.1  
Classification: missing functionality  
Severity: severe  
Proposed solution: ...
```

- Risoluzione problema: seleziona una soluzione a un problema per il quale sono state proposte più soluzioni.

The screenshot shows a software interface for managing discussions or issues. At the top, there's a menu bar with icons for 'New Topic', 'New Issue', 'New Agenda', 'Edit Profile', and several other document-related functions. Below the menu, the word 'discussion' is prominently displayed. To the left, there's a sidebar with navigation links: 'By Thread', 'By Author', 'By Category', 'By Date', 'By Unread', and 'Archiving'. The main content area is organized by date, showing a list of topics. The first topic listed is '(Open) I: Can a dispatcher see other dispatchers' TrackSections? (Alice Parker)'. This topic has several replies, with the first few being: 'P: TrackSection has access list. (Dave Smith 28.06)', 'P: TrackSection has subscription operations. (Alice Parker 28.06)', 'pro: Extensibility. (Alice Parker 28.06)', 'pro: Centralize all protected operations. (Dave Smith 28.06)', 'P: NotificationService is not part of access (Ed Jones 28.06)', and 'pro: Dispatchers can see all TrackSections (Ed Jones 28.06)'.

MECCANISMI DI COMUNICAZIONE SINCRONI

- Conversazioni di corridoio: conversazioni non programmate, chiarimenti, modifiche. È più economico ed efficace per risolvere semplici problemi. I contro sono perdita di info e incomprensioni.
- Riunioni: conversazioni programmate, revisione cliente e del progetto, brainstorming ecc... Più efficace per la risoluzione di problemi e costruzione del consenso ma ha un costo elevato.

MECCANISMI DI COMUNICAZIONE ASINCRONI

- Email: ideale per annunci ma possibilità di fraintendimento o perdita.
 - Newsgroup: adatto per discussione tra persone che condividono interesse comune.
 - WWW (portale): fornisce l'utente di una metafora dell'ipertesto: i documenti contengono collegamenti ad altri documenti ma non supporta facilmente documenti che si evolvono in modo rapido.
-

Meccanismi per eventi pianificati

	Definizione problema/ Brainstorm	Revisione Progetto/ cliente	Revisione stato	Ispezione/ Walkthrough	Rilascio
Corridoio	■		■		
Riunione	■	■	■	■	
Email					
Newsgroup	■				
WWW				■	■

Meccanismi per eventi non pianificati

	Richiesta chiarimento	Richiesta modifica	Risoluzione problematica
Corridoio	☒		☒
Riunione	☒		☒
Email	☒	☒	
Newsgroup	☒	☒	
WWW		☒	

ATTIVITA' DI COMUNICAZIONE IN UN PROGETTO SW

1. Comprensione della definizione del problema.
2. Adesione a un team.
3. Programmazione e partecipazione alle riunioni dello stato del team.
4. Adesione all'infrastruttura di comunicazione.

Adesione a un team:

- Durante la fase di definizione, il responsabile forma un team per ogni sottosistema.
- Ogni team ha un leader.
- Altri ruoli includono:
 - o Colegamento API
 - o Web master
 - o Scrittore tecnico

Adesione all'infrastruttura di comunicazione:

- Una buona comunicazione è la spina dorsale di un progetto sw.

Lez 9

MODELLI DEL CICLO DI VITA DEL SOFTWARE

Problemi connessi allo sviluppo SW:

- I requisiti cambiano costantemente
 - Le continue modifiche sono difficili da gestire
 - C'è più di un sistema SW e il nuovo sistema spesso deve essere compatibile con sistemi già esistenti (*legacy system*).
-

CICLO DI VITA DEL SW

È l'insieme di attività e loro relazioni reciproche per supportare lo sviluppo di un sistema.

- Modello di ciclo di vita: un'astrazione che rappresenta un ciclo di vita del SW con lo scopo di capire, controllare o monitorare lo sviluppo di un sistema.

Secondo lo standard IEEE 610.12-1990, il processo prevede la traduzione:

- delle **necessità dell'utente** nelle
- **richieste del software**, trasformare le richieste del software nel
- **progetto**, implementare il progetto in forma di
- **codice, testing del codice** e talvolta,
- **installare e controllare** il software per un uso operativo
- Nota: queste attività possono sovrapporsi o essere eseguite iterativamente

CICLO DI VITA DEL SW: INTRODUZIONE

Non esiste un ciclo di vita del SW ideale.

Esistono diverse possibilità che dipendono dai fattori che entrano in gioco.

Ad esempio, per i sistemi critici, un processo è più strutturato ed adeguato, mentre per i sistemi aziendali i requisiti variano spesso e potrebbe essere più efficace un processo agile e flessibile.

IDENTIFICAZIONE DELLE ATTIVITA' DI SVILUPPO SW

Domande nello sviluppo sw:

- Qual è il problema?
- Come dividere il problema?
- Qual è la soluzione?
- Sono necessarie estensioni?
- La soluzione è efficace?
- Il cliente sa usare la soluzione?
- ...

Queste domande creano una serie di attività:

Analisi requisiti

Qual è il problema?

**Dominio dell'
Applicazione**

Progettazione sistema

Qual è la soluzione?

Progettazione dettagliata

**Quali sono i migliori meccanismi
per implementare la soluzione?**

Implementazione

**Come è costruita la
soluzione?**

**Dominio della
Soluzione**

Testing

Problema risolto?

Consegna

Il cliente in grado di usare la soluzione?

Manutenzione

Sono necessarie estensioni?

In sintesi:

Analisi requisiti

Qual è il problema?

**Dominio dell'
Applicazione**

Progettazione sistema

Qual è la soluzione?

Progettazione oggetti

**Quali sono i migliori meccanismi
Per implementare la soluzione?**

Implementazione

**Come è costruita
la soluzione?**

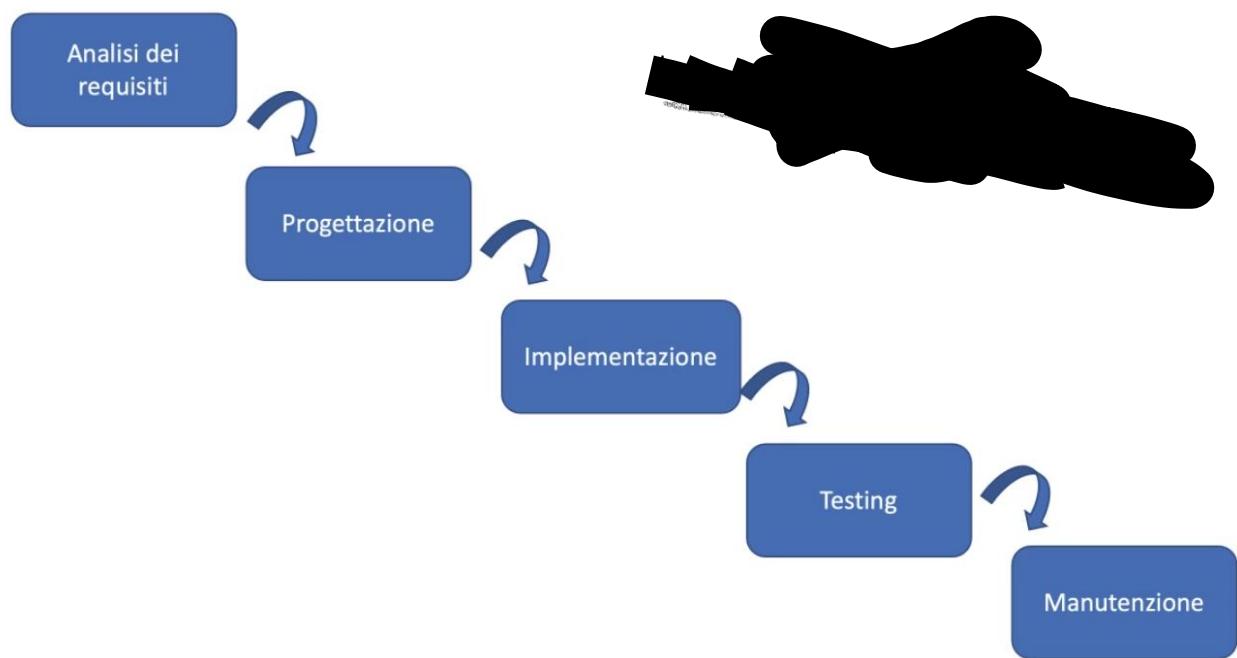
**Dominio della
Soluzione**

ATTIVITA' DEL PROCESSO SW

Le attività del processo sw comuni a tutti i processi sono:

- Specifiche del sw: si definiscono le funzionalità del sw e sono fissati i vincoli operativi.
- Progettazione e implementazione: si sviluppa il sw in modo da realizzare i requisiti.
- Convalida: si verifica che il sw garantisca le funzioni richieste.
- Evoluzione: il sw deve poter essere modificato in modo da soddisfare i cambiamenti del sistema.

Ciclo di vita del sw: una possibile organizzazione



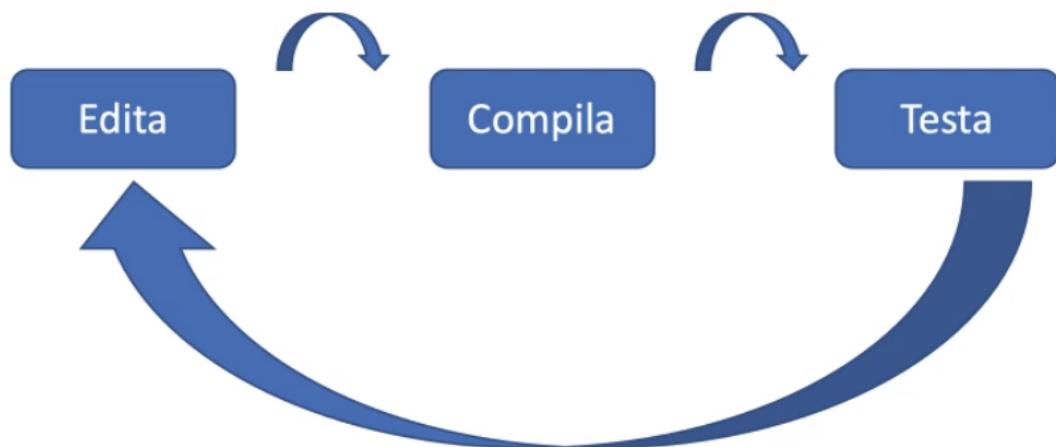
MODELLI DI PROCESSO: CARATTERISTICHE

Una strutturazione dell'organizzazione del lavoro nelle fabbriche del sw:

- Fasi della produzione
- Attività
- Controllo e misura dei progressi
- Collegamento

- Definizione dei deliverables e milestones.
-

Il più semplice (e peggior) modello di processo:



1. Molto veloce, feedback rapido.
2. Molti strumenti disponibili.
3. Specializzato per coding.
4. Non incoraggia la documentazione e ingestibile durante la manutenzione.

PROCESSI GUIDATAI DA UN PIANO E PROCESSI AGILI

I processi guidati da un piano sono processi in cui tutte le attività del processo sono pianificate in anticipo.

Nei processi agili la pianificazione è incrementale ed è più facile cambiare il processo per riflettere la modifica dei requisiti del cliente.

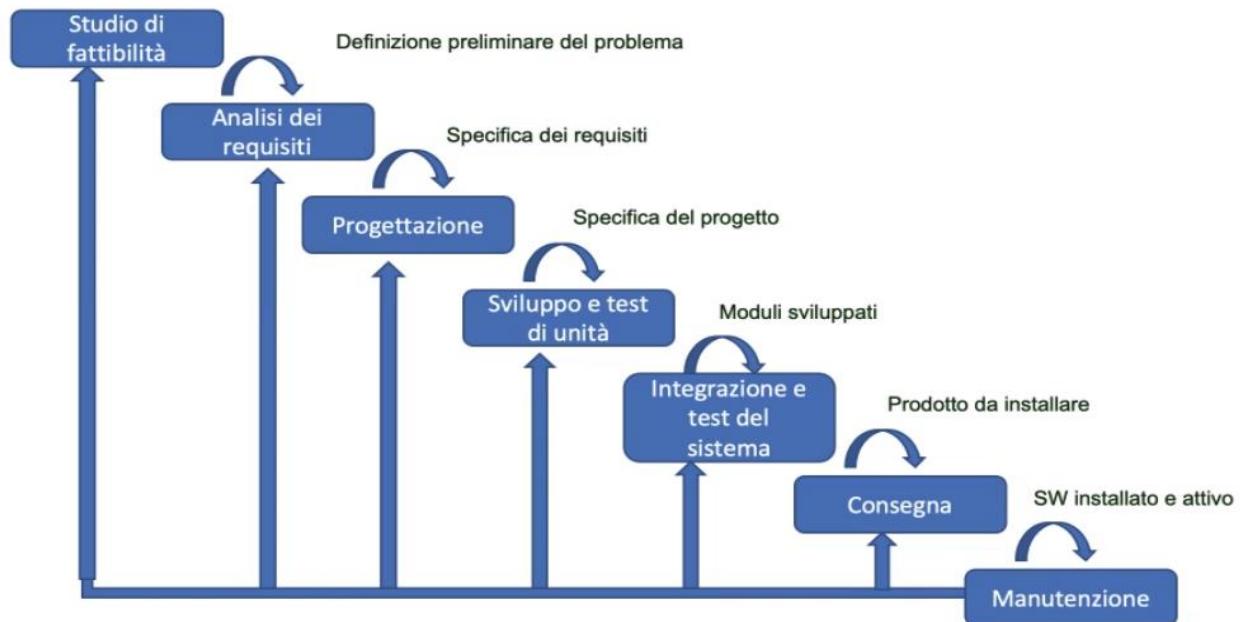
In pratica, molti processi concreti includono entrambi gli approcci, non ci sono processi sw corretti o meno.

MODELLI DEL PROCESSO SW

- Modello a cascata: modello guidato da un piano.
Fasi separate.
- Sviluppo incrementale: specifica, sviluppo e validazione sono intrecciate.
- Integrazione e configurazione (riuso): il sistema è assemblato da componenti esistenti configurabili.
- ...

Molti sistemi di grandi dimensioni usano processi che incorporano elementi da tutti questi modelli.

FASI DEL MODELLO A CASCATA



Vantaggi:

- Posso documentare e strutturare il processo di sviluppo.
- Facilmente applicabile.

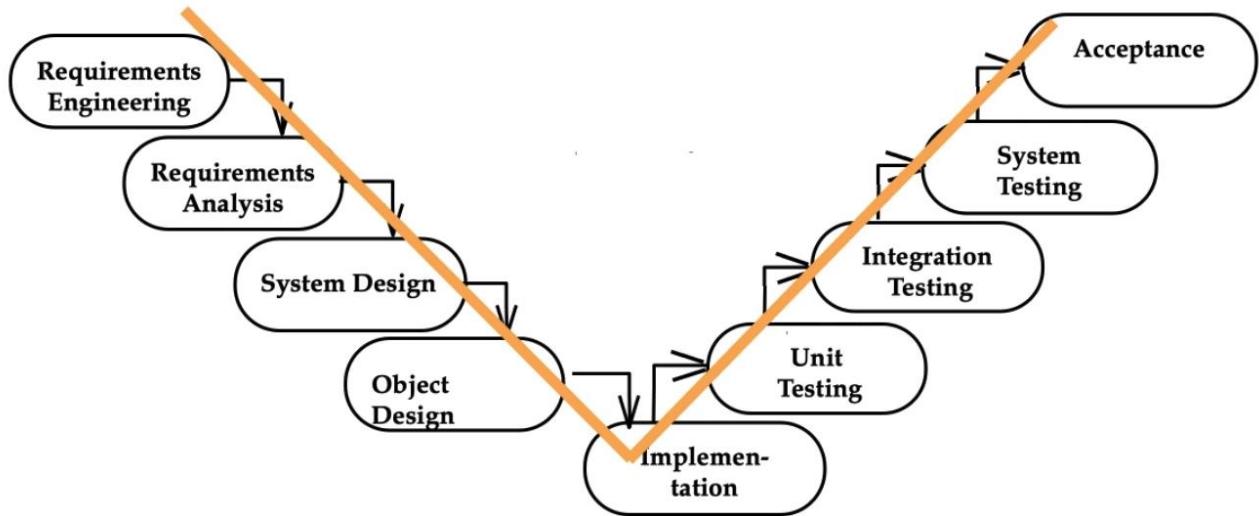
Svantaggi:

- Poco flessibile.
- Interazione col cliente solo all'inizio e alla fine.
- Il sistema diventa installabile solo quando è totalmente completato.

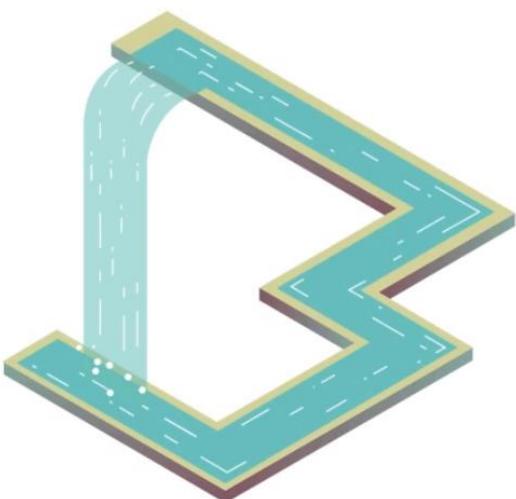
VARIANTE DEL MODELLO A CASCATA: V MODEL

Vado a separare le fasi del processo considerate di modellazione rispetto a quelle di convalida e di

verifica. È solo una variante del modello a cascata dato che le fasi sono le stesse.



ITERAZIONE NEL MODELLO A CASCATA



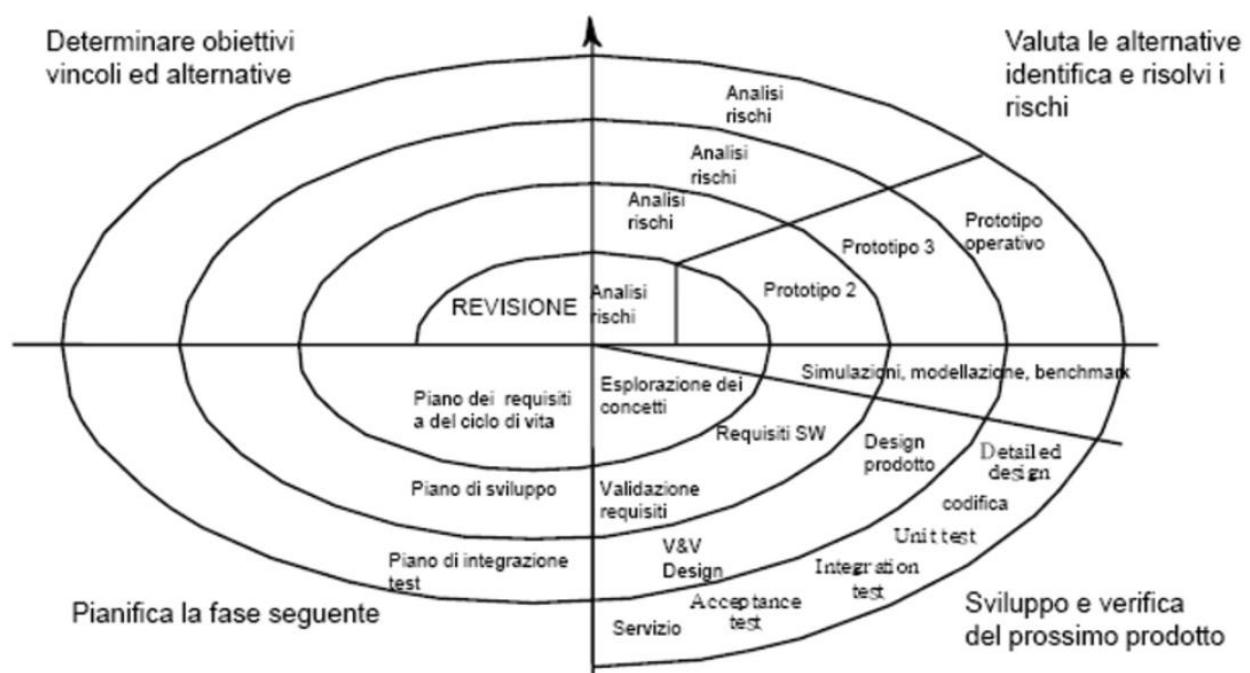
Crea paradossi. Non si usa

MODELLO A SPIRALE

Il modello a spirale prevede il seguente insieme di attività:

- Determinare obiettivi e vincoli
 - Valutare le alternative
 - Identificare i rischi
 - Risolvere i rischi assegnando priorità a essi
 - Sviluppare una serie di prototipi per i rischi
 - Usare un modello a cascata per lo sviluppo di un prototipo
 - Se un rischio è stato risolto con successo, valutare i risultati del ciclo e pianificare il prossimo ciclo.
 - Se un certo rischio non può essere risolto, terminare il progetto immediatamente.

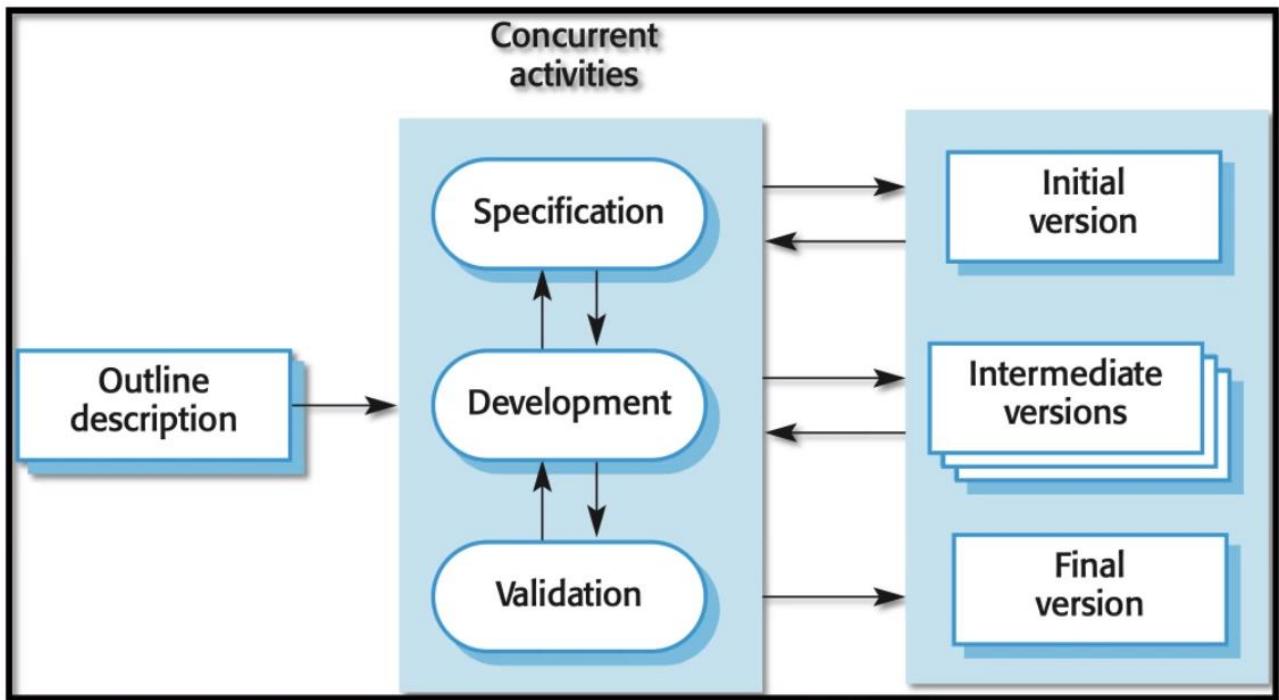
Questo insieme di attività è applicati a più cicli o round.



LIMITI DEL MODELLO A SPIRALE

Nessuno dei due gestisce i cambiamenti frequenti e non è flessibile.

SVILUPPO INCREMENTALE



Si adotta un paradigma basato sullo sviluppo incrementale.

L'idea è che vado a definire il problema e le attività di specifica, sviluppo e validazione vengono portate in parallelo. Si pianifica i vari incrementi da produrre prima di arrivare alla versione finale: parto da un insieme di requisiti essenziali, lasciando per ultimo i meno importanti, e ogni volta si gestisce meglio un'eventuale modifica sia grazie al lavoro

concorrente, sia grazie al fatto che le attività sono parallele e posso modificare in qualsiasi fase.

PROBLEMI DEL MODELLO INCREMENTALE

Il processo di sviluppo non è visibile: i responsabili hanno bisogno di deliverable consegnati con regolarità per misurare il progresso. Se i sistemi sono sviluppati velocemente, non è efficiente questa soluzione.

Inoltre la struttura del sistema tende a degradare quando sono aggiunti nuovi incrementi:

- A meno che non sono investiti tempo e denaro di refactoring per migliorare il sw, le modifiche regolari corrompono la struttura. Incorporare modifiche nel sw è difficile e costoso.
-

SVILUPPO BASATO SU COMPONENTI

Basato sul riuso del sw dove i sistemi sono integrati da componenti o sistemi esistenti.

Gli elementi riusati possono essere configuarti epr adattare il loro comportamento ai requisiti dell'utente.

TIPI DI SW RIUSABILE

- Sistemi applicativi stand-alone.
- Package
- Web services

Vantaggi e svantaggi

- Costi e rischi ridotti poiché è sviluppato meno software da zero
- Consegnata e distribuzione del sistema più veloce
- Compromessi sui requisiti inevitabili
 - il sistema potrebbe non soddisfare le reali esigenze degli utenti
- Perdita di controllo sull'evoluzione degli elementi del sistema riusati

GESTIONE DEI CAMBIAMENTI

AFFRONTARE LE MODIFICHE

Le modifiche sono inevitabili in tutti i grandi progetti SW:

- I cambiamenti aziendali portano a nuovi requisiti o modifiche
- Le nuove tecnologie aprono nuove possibilità
- Cambiare le piattaforme comporta modifiche

Le modifiche comportano una rilavorazione e quindi i costi delle modifiche sommano la rilavorazione e i costi di implementazione della nuova funzionalità.

RIDURRE I COSTI DI RILAVORAZIONE

- È necessario prevedere le modifiche: si sviluppa un prototipo per simulare il nostro sistema.
- Sviluppare un sistema tollerante alle modifiche.

GESTIRE I CAMBIAMENTI DEI REQUISITI

- Prototipizzazione del sistema
- Consegna incrementale

PROTOTIPAZIONE DEL SW

Un prototipo è una versione iniziale del sistema usato per dimostrare i concetti e determinare opzioni di progettazione.

Un prototipo è usato:

- a. Nel processo di ingegneria dei requisiti per aiutare la scoperta e la validazione di essi

- b. Nei processi di progettazione per esplorare le opzioni e sviluppare un progetto di interfaccia utente
- c. Nel processo di testing

DOMANDA DI ESAME

Se devo mettere appunto un sistema con requisiti non chiaro, che metodo adotto? Il prototipo.

BENEFICI DELLA PROTOTIPIZZAZIONE

- Migliore usabilità del sistema.
- Maggiore corrispondenza con necessità del cliente.
- Migliore qualità di progettazione.
- Migliore mantenibilità.
- Sforzo di sviluppo inferiore.

SVILUPPO DEL PROTOTIPO

Può essere basato su linguaggi o strumenti di prototipizzazioen rapida. Può trascurare alcune funzionalità poiché dovrebbe focalizzarsi su parti del prodotto che non sono ben comprese.

Il controllo di errori non può essere incluso in esso.
Il focus è sui requisiti funzionali e non su quelli non funzionali.

PROTOTIPI THOW-AWAY

I prototipi sono scartati dopo il loro uso perché se volessi usarlo come originale avrebbe uno scheletro instabile.

CONSEGNA INCREMENTALE

Piuttosto che consegnare il sistema in una volta, lo sviluppo e la consegna è suddivisa in incrementi, attraverso i quali si consegna parte della funzionalità richiesta.

VANTAGGI CONSEGNA INCREMENTALE

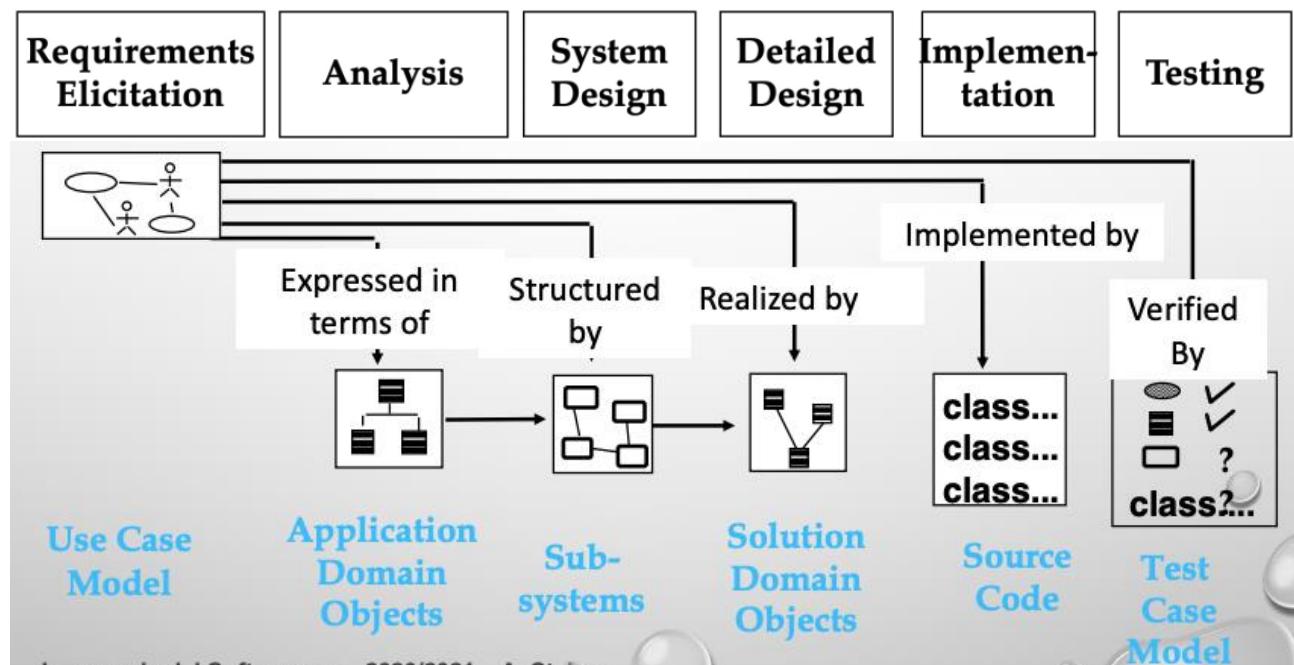
I primi incrementi fungono come un prototipo per aiutare nella scoperta dei requisiti per gli incrementi successivi.

SVANTAGGI

Molti sistemi reali si compongono di funzionalità che dipendono da altre parti del sistema.

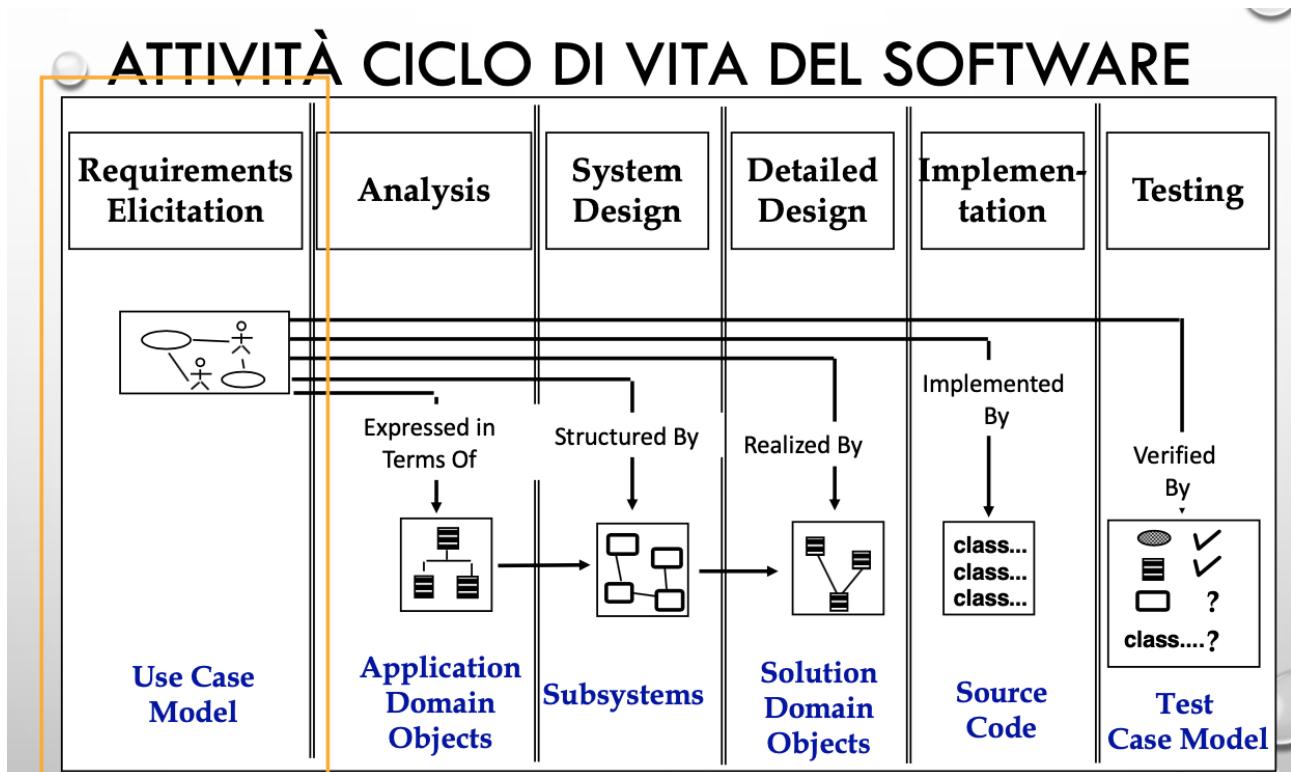
Quando definisco i requisiti nella consegna incrementale non sono ben definiti ed è difficile stabilire i servizi comuni a più entità.

ATTIVITA' DEL PROCESSO SW E I LORO MODELLI



Lez 10 → SCOPERTA DEI REQUISITI

Ricordare:



La scoperta dei requisiti pone gli sviluppatori a confrontarsi coi clienti per determinare le caratteristiche del sistema. È una fase delicata perché si trovano a confrontarsi due mondi diversi: quello dei clienti, esperto del dominio dell'applicazione, e quello degli sviluppatori, esperti di sviluppo SW. I due mondi rischiano, se non ben integrati, di gettare basi inconsistenti per lo sviluppo sw.

OBIETTIVO SCOPERTA DEI REQUISITI

Focalizzata sulla descrizione dello scopo del sistema.

Il cliente, gli sviluppatori e gli users identificano un'area del problema e definiscono un sistema. La suddetta definizione è detta specifica dei requisiti.

REQUISITI

I requisiti sono descrizioni testuali delle funzioni desiderate del sistema e delle loro proprietà. Sono divisi in due categorie secondo l'acronimo *FURPS* (*Functionality, Usability, Reliability, Performance, Supportability*).

Requisiti funzionali (F):

- Una singola funzione del sistema
- <<uno studente può vedere tutti i corsi del semestre corrente nella propria lista di esami>>

Requisiti non funzionali (URPS):

- Aspetti del sistema non legati al suo comportamento funzionale
 - Aspetto dell'interfaccia, tempo di risposta, sicurezza, linguaggio di programmazione.

Primo PASSO: IDENTIFICAZIONE DEL SISTEMA

È necessario rispondere a due domande:

1. Come identifichiamo lo scopo del sistema?
 - a. Quali sono i requisiti e i vincoli?
2. Cosa si trova nel sistema e cosa si trova fuori?

La risposta è fornita durante la scoperta dei requisiti e l'analisi.

- Scoperta dei requisiti: definizione del sistema in termini compresi dal cliente
- Analisi: definizione del sistema nella “terminologia” dello sviluppatore. Specifica tecnica

Processo di requisiti = scoperta dei requisiti + analisi.

TECNICHE PER SCOPRIRE I REQUISITI

Colmare il gap tra cliente e sviluppatori tramite:

- Questionari: chiedere all’user una lista di domande.
- Analisi dei compiti: osservare gli utenti nel loro ambiente di lavoro

- Scenari: descrivono l'uso del sistema come una serie di interazioni tra un utente e il sistema
 - Casi d'uso: astrazioni che descrivono una classe di scenari.
-

SCENARI

Sono una descrizione sintetica di un evento o una serie di azioni.

È una descrizione testuale dell'uso del sistema. La descrizione è scritta dal punto di vista dell'utente.

Uno scenario può includere testo, video, immagini...

SCENARI NEL PROCESSO SW

Gli scenari possono essere usati in modi diversi durante il ciclo di vita del SW:

- Scoperta dei requisiti: scenario “*as-is*”, visionario
- Test di accettazione del cliente
- Distribuzione del sistema

L'uso degli scenari dovrebbe essere iterativo: ogni scenario dovrebbe essere aggiornato e riorganizzato quando i criteri e le situazioni cambiano.

TIPI DI SCENARI

- Scenari *as-is*: descrive una situazione corrente. Usato in progetti di reingegnerizzazione. L'user descrive il sistema.
 - Scenari visionari: descrive un sistema futuro, spesso usato nei progetti di ingegneria.
 - Scenario di valutazione: descrizione di compito dell'utente rispetto al quale deve essere valutato il sistema.
 - Scenario di training: una descrizione delle istruzioni passo passo che guidano l'user novizio nell'uso del sistema.
-

COME TROVIAMO GLI SCENARI?

Il cliente potrebbe dar per scontato molte cose, soprattutto se il sistema non esiste.

Bisogna aiutare il cliente a formulare i requisiti.

Il cliente invece deve aiutare lo sviluppatore a capire i requisiti che si evolvono mentre si sviluppano gli scenari.

EURISTICHE PER TROVARE GLI SCENARI

- Quali sono i compiti principali del sistema?
- Che dati l'attore dovrà creare, memorizzare nel sistema?
- Quali modifiche esterne l'attore deve informare il sistema?
- Quali cambiamenti dovrà essere informato l'attore?

Inoltre bisogna osservare bene ogni circostanza e analizzare bene le situazioni.

DOPO CHE GLI SCENARI SONO FORMULATI

Bisogna trovare il caso d'uso che specifica le istanze del problema.

Descrivere ciascun caso d'uso con dettaglio:

- Attori partecipanti
- Descrivere le condizioni di entrata
- Descrivere il flusso degli eventi
- Descrivere le condizioni di uscita
- Descrivere le eccezioni
- Descrivere requisiti non funzionali

SCOPERTA DEI REQUISITI: DIFFICOLTA' E SFIDE

C'è un missmatch tra clienti e sviluppatori:

- Il cliente conosce il dominio dell'applicazione
 - Lo sviluppatore sa il dominio della soluzione
-

SPECIFICA DEI REQUISITI VS MODELLO DI ANALISI

La specifica dei requisiti usa un linguaggio naturale.

Il modello di analisi usa una notazione formale.

TIPOLOGIE DI REQUISITI

- Funzionali.
 - Non funzionali.
 - Vincoli: impostati dal cliente o dall'ambiente.
Sono detti anche “*pseudo-requisiti*”.
-

REQUISITI FUNZIONALI VS NON FUNZIONALI

Requisiti funzionali

- Descrivono compiti utente che il sistema deve supportare
- Espressi come azioni
 - «Pubblica una nuova legge»
 - «Programma un calendario»
 - «Notifica un nuovo gruppo di interesse»

Requisiti non funzionali

- Descrivono le proprietà del sistema o del dominio
- Espressi come vincoli o affermazioni negative
 - «tutti gli input dell'utente devono essere fornire la conferma di ricezione entro un secondo»
 - «Un crash del sistema non dovrebbe comportare perdita di dati»

REQUISITI NON FUNZIONALI

- Usabilità
- Affidabilità
 - Robustezza
 - Sicurezza
- Prestazioni
 - Tempo di risposta
 - Scalabilità
 - Throughput
 - Disponibilità
- Supportabilità
 - Adattabilità
 - Manutenibilità
- Implementazione
- Interfaccia
- Funzionamento
- Packaging
- Legale
 - Licenzia (GPL, LGPL)
 - Certificazione
 - Regolamentazione

Requisiti di qualità

Vincoli o pseudo requisiti

COSA NON DEVE ANDARE NEI REQUISITI FUNZIONALI?

Tutti gli aspetti di natura tecnica:

- Struttura
- Metodo di sviluppo
- Ambiente di sviluppo
- ...

VALIDAZIONE DEI REQUISITI

i requisiti devono essere corretti dal punto di vista dell'utente, completi e consistenti.

Devono essere inoltre:

- Chiari
- Reali
- Tracciabili

Il problema che sorge è che i requisiti cambiano velocemente durante la scoperta e sono necessari tool di supporto.

DIFFERENTI TIPI DI SCOPERTA DEI REQUISITI

- Greenfield engineering: lo sviluppo parte da zero, i requisiti provengono da utenti e clienti.
 - Re-engineering
 - Interface engineering
-

PRIORITA' AI REQUISITI

- Alta priorità: affrontato durante l'analisi, la progettazione e l'implementazione. Una caratteristica ad alta priorità deve essere dimostrata.

- Media priorità: affrontato durante l'analisi e la progettazione.
- Bassa priorità: affrontato solo durante l'analisi. Illustra come il sistema sarà usato in futuro con una tecnologia non disponibile al momento.

Lez 11 SCOPERTA DEI REQUISITI PT2

ATTIVITA' DI SCOPERTA DEI REQUISITI

Le attività di scoperta dei requisiti fanno corrispondere la definizione di un problema a una specifica dei requisiti, che può essere presentata come un insieme di:

- Attori
 - Scenari
 - Casi d'uso
-
- Le attività di scoperta dei requisiti includono
 - Identificazione degli attori
 - Identificazione degli scenari
 - Identificazione dei casi d'uso
 - Identificazione delle relazioni tra attori e casi d'uso
 - Identificazione degli oggetti iniziali dell'analisi
 - Identificazione dei requisiti non funzionali
-

IDENTIFICAZIONE DEGLI ATTORI

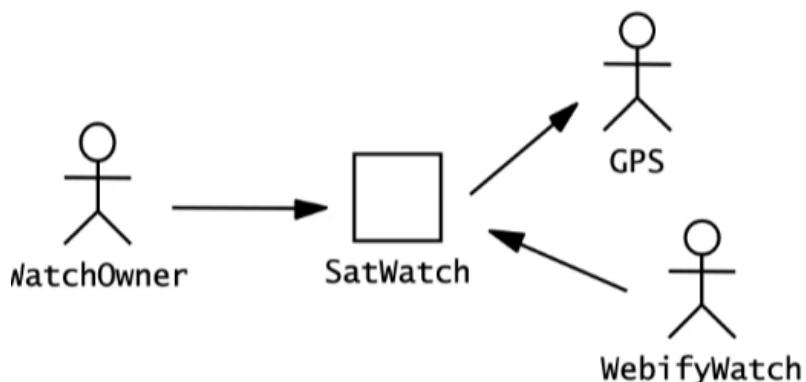
Gli attori rappresentano entità esterne che interagiscono col sistema:

- Può essere un utente o un sistema esterno.

Nell'esempio di SatWatch il proprietario dell'orologio, il GPS e il Webify Watch sono attori:

- Tutti scambiano informazioni con SatWatch.
- Tutti hanno interazioni specifiche col sistema:
 - o Il proprietario guarda e indossa
 - o L'orologio monitora il GPS
 - o WW scarica nuovi dati nell'orologio

Dopo aver identificato gli attori vado a rappresentare il sistema e gli attori.



Attori e ruoli sono astrazioni e non necessariamente corrispondono a persone:

- Per quanto riguarda FRIEND, la stessa persona può essere FO o Dispatcher in tempi diversi, quindi i due ruoli sono modellati con due attori differenti poiché una stessa persona può ricoprire più ruoli.

SCOPERTA DEI REQUISITI E IDENTIFICAZIONE ATTORI

Il primo passo nella scoperta dei requisiti è l'identificazione degli attori: serve per definire i confini del sistema.

Quando il sistema è inserito in un'organizzazione esistente, esistono molti attori prima che il sistema sia sviluppato, essi corrispondono a ruoli nell'organizzazione.

Durante la fase iniziale dell'identificazione del problema, è difficile distinguere gli attori dagli oggetti.

Una volta definito il confine, non ci sono problemi di distinzione:

- Gli attori sono fuori dal confine del sistema, sono esterni.
- I sottosistemi e gli oggetti sono interni.
- Ogni sistema SW esterno che usa il sistema è un attore.

DOMANDE PER IDENTIFICARE GLI ATTORI

- 1.Quali gruppi di utenti sono supportati dal sistema per eseguire il proprio lavoro?
- 2.Quali gruppi di utenti eseguono le funzioni principali del sistema?

- 3.Quali utenti eseguono le funzioni secondarie, come manutenzione e amministrazione?
 - 4.Con quali sistemi SW e HW esterni interagirà il sistema?
-

IDENTIFICAZIONE DEGLI SCENARI

Uno scenario è una descrizione concreta di una funzionalità del sistema.

Gli scenari non devono sostituire i casi d'uso. Essi potenziano la scoperta dei requisiti fornendo uno strumento comprensibile a users e clienti.

DOMANDE PER IDENTIFICARE GLI SCENARI

- Quali sono i task che l'attore vuole che il sistema esegua?
- A quali informazioni l'attore accede?
 - Chi crea i dati?
 - Possono essere modificati o rimossi?
 - Da chi?
- Quali modifiche esterne l'attore deve notificare al sistema?
 - Quanto spesso?
 - Quando?
- Quali eventi il sistema deve notificare l'attore?
 - Con quale latenza?

Gli sviluppatori usano documenti esistenti sul dominio applicativo per rispondere a tali domande. Questi documenti includono manuali utente precedenti, manuali di procedure, note, interviste... Gli sviluppatori dovrebbero sempre scrivere gli scenari usando la terminologia del dominio applicativo.

IDENTIFICAZIONE DEI CASI D'USO

Uno scenario è un'istanza del caso d'uso:

- Un caso d'uso specifica i possibili scenari per una data parte di funzionalità.

Un caso d'uso è iniziato da un attore.

Rappresenta un flusso di eventi completo attraverso il sistema:

- Descrive una serie di interazioni collegate, che sono il risultato dell'avvio.

ORGANIZZARE I CASI D'USO

Gli sviluppatori danno nomi ai casi d'uso, li allegano agli attori e forniscono una descrizione ad alto livello. Il nome del caso d'uso deve essere una frase verbale che denota l'attore cosa sta cercando di realizzare.

DESCRIZIONE DEI CASI D'USO

Allegare i casi agli attori che li iniziano consente agli sviluppatori di chiarire i ruoli degli utenti.

GUIDA ALLA SCRITTURA DEI CASI D'USO

Scrivere i casi d'uso è un'arte.

Risulta difficile produrre una specifica dei requisiti consistente ma è possibile usare una guida alla stesura dei casi d'uso.

- I casi d'uso dovrebbero essere definiti con frasi verbali. Il nome del caso d'uso dovrebbe indicare cosa sta cercando l'utente (ReportEmergency, OpenIncident, ecc.)
- Gli attori dovrebbero essere indicati con dei sostantivi (FieldOfficer, Dispatcher, ecc.)
- Il confine del sistema dovrebbe essere chiaro. I passi realizzati dall'attore ed i passi realizzati dal sistema dovrebbero essere chiaramente distinguibili
- I passi dei casi d'uso nel flusso di eventi dovrebbero essere descritti con voce attiva. Ciò rende esplicito chi esegue il passo
- La relazione di causalità tra i passi successivi dovrebbe essere chiara
- Le eccezioni dovrebbero essere descritte separatamente
- Un caso d'uso non dovrebbe descrivere l'interfaccia utente del sistema. Ciò distoglie l'attenzione dai passi effettivi realizzati dall'utente
- Un caso d'uso non dovrebbe superare una lunghezza di due o tre pagine. Altrimenti, si devono usare le relazioni **include** e **extend** per decomporlo in casi d'uso più piccoli

EURISTICHE PER SCRIVERE SCENARI E CASI D'USO

- Usare gli scenari per comunicare con gli utenti e per validare le funzionalità
 - Rifinire un singolo scenario per capire le assunzioni sul sistema dell'utente
 - Definire scenari non molto dettagliati per definire lo scopo del sistema. Validare con l'utente
 - Usare mock-up solo come supporto visuale; il progetto dell'interfaccia utente dovrebbe avvenire come task separato dopo che la funzionalità è sufficientemente stabile
 - Presentare all'utente più alternative diverse. Valutare diverse alternative allarga l'orizzonte dell'utente. Generare alternative differenti forza gli sviluppatori a "pensare oltre l'ordinario"
 - Dettagliare una parte più ampia quando lo scopo del sistema e le preferenze dell'utente sono ben capiti. Validare con l'utente
-

RIFINIRE I CASI D'USO

L'enfasi di questa attività è completezza e correttezza
Gli sviluppatori identificano le funzionalità non coperte dagli scenari e le documentano rifinendone i casi d'uso.

Gli sviluppatori raramente descrivono casi e gestioni delle eccezioni così visti dal lato degli attori.

Gli aspetti dei casi d'uso che sono inizialmente ignorati e poi dettagliati sono:

- Gli elementi manipolati dal sistema.
- La sequenza a basso livello delle interazioni tra attore e sistema.

- Permessi di accesso (quale attore invoca un caso).
 - Eccezioni mancanti e la loro gestione.
 - Funzionalità comuni tra casi d'uso.
-

ASSOCIAZIONI NEI CASI D'USO

- Un modello dei casi d'uso consiste di casi d'uso ed associazioni di casi d'uso
- Un'associazione è una relazione tra casi d'uso
- Le associazioni più importanti sono: *Communication, Include, Extend, Generalization*

IDENTIFICAZIONE DI RELAZIONI TRA ATTORI E CASI:

- Anche sistemi di medie dimensioni hanno numerosi casi d'uso
- Le relazioni tra gli attori ed i casi d'uso permettono a sviluppatori ed utenti di ridurre la complessità del modello ed aumentarne la comprensione
 - Si usano **relazioni di comunicazione** tra attori e casi d'uso per descrivere il sistema in livelli di funzionalità
 - Le **relazioni di estensione** sono usate per separare flussi di eventi eccezionali da flussi di eventi comuni
 - Le **relazioni di inclusione** sono usate per ridurre la ridondanza tra casi d'uso
 - Le **generalizzazioni** specializzano casi d'uso astratti

- RELAZIONI DI COMUNICAZIONE:
 - Bisogna distinguere chi inizia il caso d'uso dagli altri attori e chi non lo può iniziare.
 - Specifica quali attori comunicano con un caso, chi accede a info specifiche e chi no.
- RELAZIONE EXTEND: sono introdotte quando descrivo un caso ma si possono verificare azioni eccezionali e bisogna estenderlo.
 - Un'associazione *extend* dal caso d'uso A al caso d'uso B indica che il caso d'uso B è un'estensione del caso d'uso A
- RELAZIONE DI INCLUSIONE: una funzione nella definizione del problema originale è troppo complessa per risolverla immediatamente.
 - Soluzione: descrivere la funzione come un'aggregazione di un sistema di funzioni più semplici. Il caso d'uso associato è decomposto in casi più piccoli.

EXTEND VS INCLUDE

- Entrambe le relazioni sono simili
 - Inizialmente potrebbe non essere chiaro allo sviluppatore quando usarle
- La principale distinzione tra le due relazioni sta nella direzione della relazione
 - Per le relazioni *include*, l'evento che innesca il caso d'uso target (quello incluso) è descritto nel flusso di eventi del caso d'uso sorgente
 - Per le relazioni *extend*, l'evento che innesca il caso d'uso sorgente (quello che estende) è descritto nel caso d'uso sorgente come precondizione
 - In altre parole, per le relazioni *include*, ogni caso d'uso che include deve specificare dove il caso d'uso incluso dovrebbe essere invocato
 - Per le relazioni *extend*, solo il caso d'uso che estende specifica quali casi d'uso sono estesi

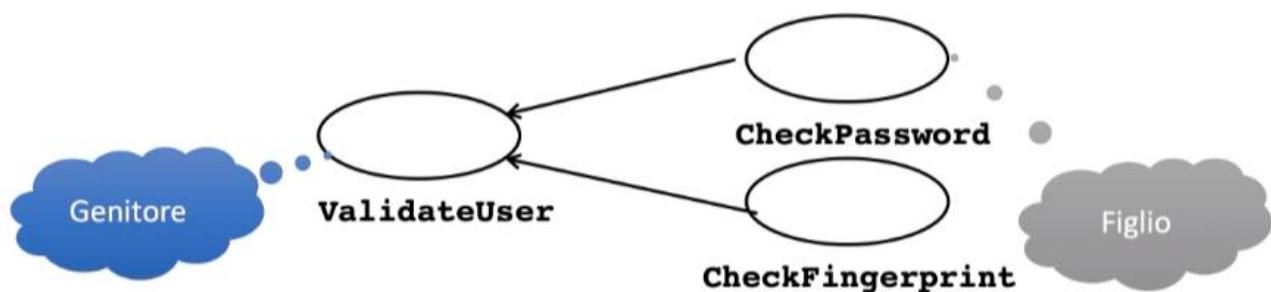
ASSOCIAZIONE DI GENERALIZZAZIONE TRA CASI

Problema: c'è un comportamento comune ma più specifico tra casi d'uso.

Soluzione: l'associazione di generalizzazione tra casi esplicita comportamenti comuni. I casi d'uso figli ereditano il comportamento del genitore ed aggiungono altri comportamenti (*ereditarità*).

- Esempio

- Consideriamo il caso d'uso **ValidateUser**, responsabile della verifica dell'identità di un utente. Il cliente potrebbe richiedere due realizzazioni: **CheckPassword** e **CheckFingerprint**



IDENTIFICAZIONE DEGLI OGGETTI DI ANALISI INIZIALI

- Uno dei primi ostacoli che sviluppatori ed utenti incontrano quando cominciano a collaborare è la terminologia differente
 - Anche se gli sviluppatori imparano la terminologia degli utenti, il problema si pone ancora quando si aggiungono nuovi sviluppatori al progetto
 - Le incomprensioni si verificano per l'uso di stessi termini usati in contesti differenti con significati diversi

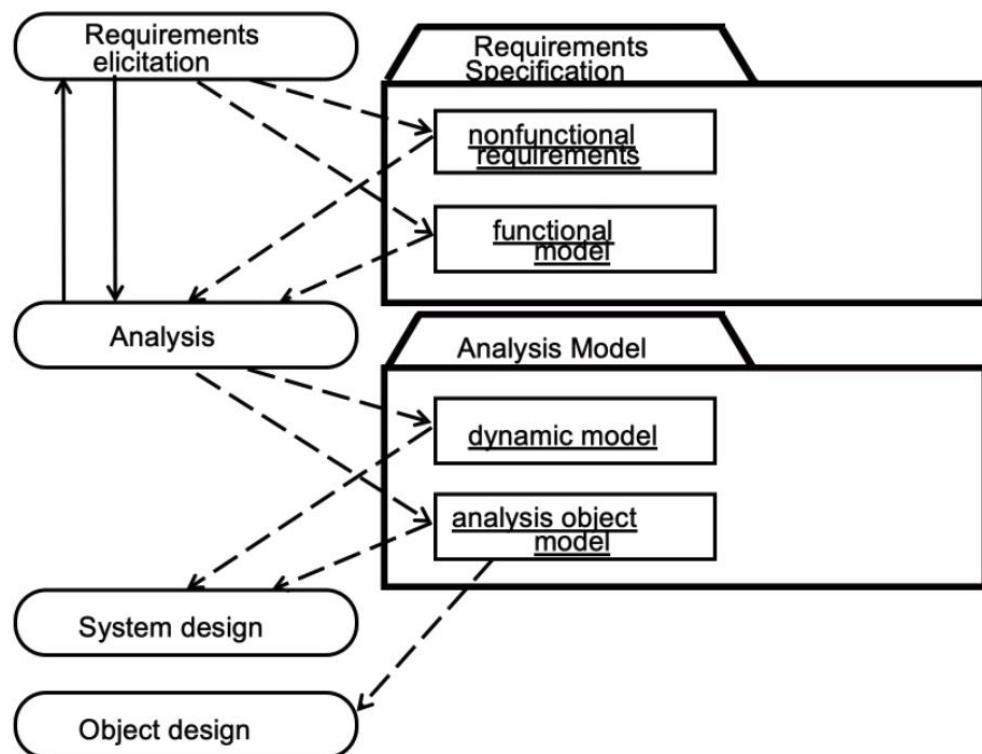
- Le eccezioni dovrebbero essere descritte separatamente
 - Un caso d'uso non dovrebbe descrivere l'interfaccia utente del sistema. Ciò distoglie l'attenzione dai passi effettivi realizzati dall'utente
 - Un caso d'uso non dovrebbe superare una lunghezza di due o tre pagine. Altrimenti, si devono usare le relazioni **include** e **extend** per decomporlo in casi d'uso più piccoli
-

Glossario

- Il glossario è incluso nella specifica dei requisiti e successivamente nei manuali utente
- Gli sviluppatori mantengono il glossario aggiornato durante l'evoluzione della specifica dei requisiti
- I benefici del glossario sono diversi
 - I nuovi sviluppatori hanno a disposizione un insieme consistente di definizioni
 - E' usato un termine singolo per ogni concetto
 - Ogni termine ha un preciso e chiaro significato ufficiale

Lez 13 → Analisi, modellazione degli oggetti.

Analisi e suoi prodotti



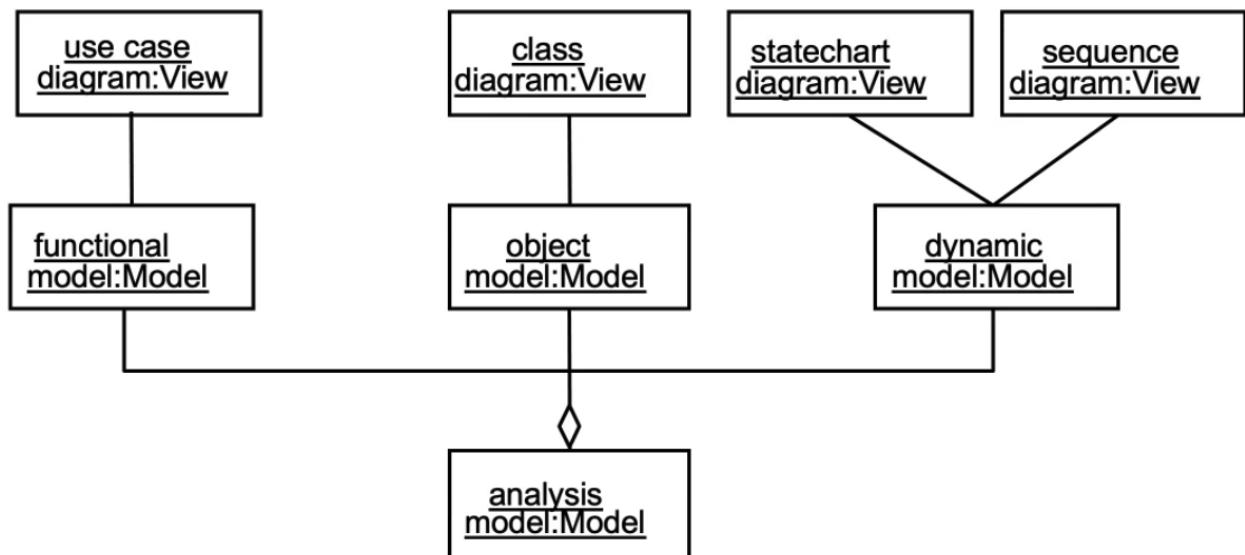
L'analisi consiste nel trovare il diagramma delle classi.

MODELLO DI ANALISI

È composto da 3 singoli modelli:

1. Modello funzionale: rappresentato da casi d'uso e scenari.
2. Modello ad oggetto: rappresentato da diagrammi delle classi e oggetti
3. Modello dinamico: rappresentato dal digramma di stato e delle sequenze.

Il modello di analisi



ATTIVITA' DI ANALISI DEI REQUISITI

Modellazione degli oggetti:

- Attività durante la modellazione degli oggetti
- Identificazione degli oggetti
- Tipi di oggetti:
 - *Entità (entity)*
 - *Confine (boundary)*
 - *Controllo (control)*
- Stereotipi
- Tenica di Abbot:
 - Aiuta nell'identificazione degli oggetti.

ATTIVITA' DURANTE LA MODELLAZIONE DEGLI OGGETTI

Obiettivo principale: trovare le astrazioni importanti:

Passi:

1. Identificare le classi
2. Trovare gli attributi
3. Trovare i metodi
4. Trovare le associazioni tra le classi

L'ordine delle operazioni è un'euristica solamente, non è fondamentale.

- Cosa accade se le astrazioni sono errate?
 - Iteriamo e revisioniamo il modello.
-

IDENTIFICAZIONE DELLE CLASSI

Passo cruciale: aiuta a identificare le entità importanti del sistema.

Le assunzioni di base sono:

- Possiamo trovare le classi in un nuovo sistema SW.
- Possiamo identificare le classi in un sistema esistente.

Come possiamo procedere?

Ci sono diversi **approcci**:

- Approccio del dominio applicativo: chiedere agli esperti di identificare le astrazioni.
- Approccio sintattico: iniziare coi casi d'uso, analizzare il testo per identificare gli oggetti, estrarre gli oggetti partecipanti dal flusso di eventi.
- Approccio con design pattern: usare design pattern riusabili.
- Approccio basato su componenti: i dentificare classe di soluzioni esistenti.

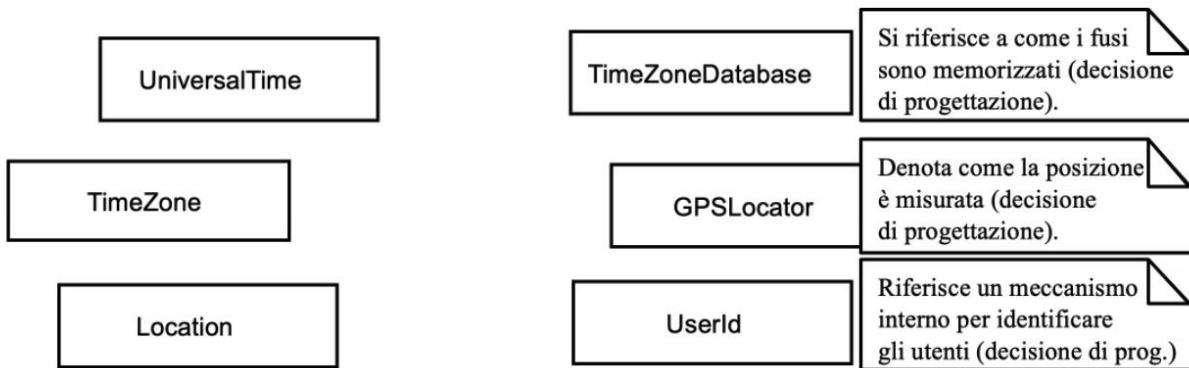
IDENTIFICARE OGGETTI E' UN PROBLEMA DIFFICILE

- Un problema: definizione del confine del sistema
 - Quali astrazioni sono interne e esterne?
 - Gli attori sono interni
 - Classi e oggetti sono esterni
-

Esempi e contro esempi di classi nel modello ad oggetti di analisi di *SatWatch*

- Il modello ad oggetti di analisi rappresentano concetti a livello utente e non le classi o componenti effettivi del software

Concetti del dominio che dovrebbero essere rappresentati nel modello ad oggetti di analisi Classi software che non dovrebbero essere rappresentate nel modello ad oggetti di analisi



TIPI DIVERSI DI OGGETTI

- OGGETTI *entity*:
 - Rappresentano le info persistenti mantenute dal sistema.
- Oggetti *boundary*:
 - Rappresentano l'interazione tra utente e sistema.
- Oggetti *control*:
 - Rappresentano i compiti di controllo eseguiti dal sistema.

Per denotarli in uml si usano gli stereotipi.

Esempio: modellazione 2BWatch

Per distinguere i diversi tipi di oggetti in un modello possiamo usare il meccanismo UML dello Stereotipo

<<Entity>>
Year

<<Entity>>
Month

<<Entity>>
Day

<<Boundary>>
Button

<<Control>>
ChangeDate

<<Boundary>>
LCDDisplay

Oggetti Entity

Oggetti Control

Oggetti Boundary

TIPI DI OGGETTI

Le tre categorie di oggetti consentono ai modelli di essere più resilienti alle modifiche:

- L'interfaccia di un sistema cambia con maggior probabilità rispetto al controllo
- Il mondo in cui il sistema è controllato cambia con maggior possibilità rispetto alle entità di un dominio applicativo.
 - I tipi degli oggetti sono stati introdotti in Smalltalk
 - Model, View, Controller (MVC)
 - Model <-> Oggetto Entity
 - View <-> Oggetto Boundary
 - Controller <-> Oggetto Control

TROVARE OGGETTI PARTECIPANTI AL CASO D'USO

Prendere un caso d'uso e leggere il flusso degli eventi

Fare un'analisi testuale (anali sostantivo-verbo):

- I sostantivi sono candidati per oggetti/classi
 - I verbi per le operazioni
 - Questa procedura è nota come tecnica di *Abbot*.
-
- Dopo che oggetti/classi sono stati trovati, identificare i tipi:
 - Identificare le entità del mondo reale.
 - Identificare le procedure del mondo reale.
 - Identificare artefatti di interfaccia.

VEDIAMO UN ESEMPIO:

Flusso di eventi:

- The customer enters the store to buy a toy.
- It has to be a toy that his daughter likes and it must cost less than 50 Euro.
- He tries a videogame, which uses a data glove and a head-mounted display. He likes it.
- An assistant helps him.
- The suitability of the game depends on the age of the child.
- His daughter is only 5 years old.
- The assistant recommends another type of toy, namely the boardgame "Monopoly".

(Tecnica di Abbott)

<i>Example</i>	<i>Part of speech</i>	<i>UML model component</i>
“Monopoly”	Proper noun	object
Toy	Improper noun	class
Buy, recommend	Doing verb	operation
is-a	being verb	inheritance
has an	having verb	aggregation
must be	modal verb	constraint
dangerous	adjective	attribute
enter	transitive verb	operation
depends on	intransitive verb	Constraint, class, association



MODI PER TROVARE OGGETTI

Oltre a queste tecniche, usare altre sorgenti di conoscenza:

- Conoscenza dell'applicazione: user e esperti conoscono le astrazioni del dominio.
 - Conoscenza della soluzione: astrazioni nel dominio della soluzione
 - Conoscenza generale del mondo: conoscenze personali e intuizione.
-

ORDINE DELLE ATTIVITA' PER IDENTIFICAZIONE DEGLI OGGETTI

1. Formulare pochi scenari con l'aiuto di un utente o esperto del dominio applicativo.
2. Estrarre i casi d'uso dagli scenari.
3. Procedere in parallelo con:
 - a. Analizzare il flusso degli eventi con Abbot.
 - b. Generare il diagramma delle classi.

PASSI DELLA GENERAZIONE DEL DIAGRAMMA DELLE CLASSI

1. Identificazione della classe (analisi testuale, esperto del dominio)
2. Identificazione di attributi e operazioni (talvolta prima che le classi siano trovate)
3. Identificazione di associazioni tra classi
4. Identificazione delle molteplicità
5. Identificazione dei ruoli
6. Identificazione dell'ereditarietà

IDENTIFICARE GLI OGGETTI BOUNDARY

Gli oggetti boundary rappresentano l'interfaccia del sistema con gli attori.

In ciascun caso d'uso, ogni attore interagisce con almeno un oggetto boundary.

L'oggetto boundary raccoglie le info dagli attori e scambiano.

Euristiche per identificare gli oggetti boundary

- Identificare gli elementi dell'interfaccia utente di cui l'utente necessita per iniziare il caso d'uso (es., ReportEmergencyButton)
 - Identificare le form con le quali l'utente immette dati nel sistema (es., EmergencyReportForm)
 - Identificare avvisi e messaggi che il sistema usa per rispondere all'utente (AcknowledgmentNotice)
 - Quando multipli attori sono coinvolti in un caso d'uso, identificare i terminali degli attori (es., DispatcherStation) per riferirsi all'interfaccia utente in questione
 - Non modellare aspetti visuali dell'interfaccia con oggetti boundary
 - Usare sempre i termini dell'utente finale per descrivere le interfacce; non usare termini dai domini implementativi
-

IDENTIFICARE GLI OGGETTI CONTROL

Gli oggetti control sono responsabili per coordinare boundary ed entity:

- Solitamente non hanno una controparte concreta nel mondo reale

- Esiste una relazione stretta tra un caso d'uso e un oggetto control:
 - o Un oggetto control di solito è creato all'inizio di un caso d'uso e cessa di esistere alla terminazione dello stesso.
- È responsabile di raccogliere info dagli obiettivi boundary agli oggetti entity.

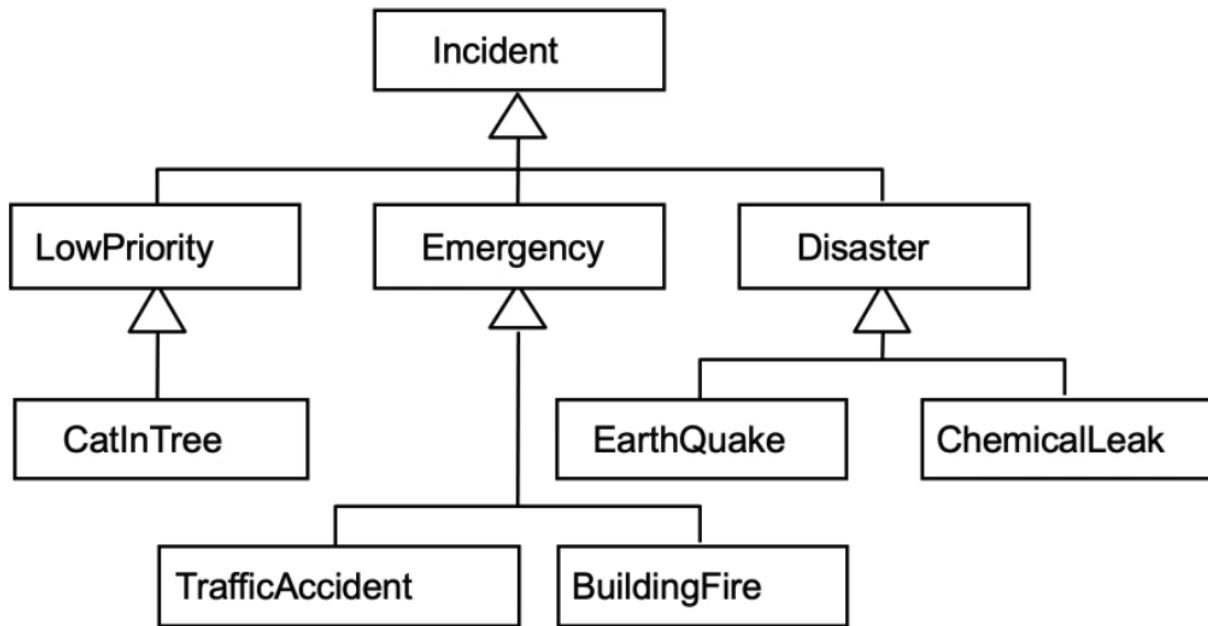
Euristiche per identificare gli oggetti control

- Identificare un oggetto control per caso d'uso
 - Identificare un oggetto control per attore nel caso d'uso
 - La durata di vita di un oggetto control dovrebbe coprire l'estensione del caso d'uso o l'estensione di una sessione utente. E' difficile identificare l'inizio e la fine dell'attivazione di un oggetto control, il caso d'uso corrispondente probabilmente non ha condizioni di entrata e di uscita ben definite
-

GENERALIZZAZIONE E SPECIALIZZAZIONE

L'ereditarietà consente di organizzare i concetti in gerarchie:

- In cima c'è il concetto generale
- In fondo i concetti più specializzati
- Nel mezzo ci sono livelli intermedi con concetti più o meno specializzati.



La specializzazione è l'attività che identifica concetti più specializzati da quelli ad alto livello:

- Esempio: stiamo costruendo un sistema di gestione emergenze cominciando da zero e stiamo discutendo le funzionalità con il cliente
 - Il cliente ci introduce il concetto di incidente e poi descrive tre tipi di incidenti (*Incidents*)
 - Disastri (*Disasters*), che richiede la collaborazione di diverse agenzie
 - Emergenze (*Emergencies*), che richiede una risposta immediata ma può essere gestita da una singola agenzia
 - Incidenti con priorità bassa (*LowPriorityIncidents*), che non deve essere necessariamente gestita se le risorse sono richieste da altri incidenti con maggiore priorità

CHI NON USA I DIAGRAMMI DELLE CLASSI?

Il cliente e l'utente non sono interessati ai diagrammi delle classi di solito.

I clienti si focalizzano su problemi relativi alla gestione del progetto, gli user alle funzioni.

Lez 14

ANALISI, MODELLAZIONE DINAMICA

COME TROVIAMO LE CLASSI?

Abbiamo già esaminato diverse sorgenti per identificare le classi:

- Analisi del dominio applicativo: troviamo le classi parlando col cliente e identificando le astrazioni osservando l'utente.
- Conoscenza generale
- Analisi testuale dei flussi di eventi nei casi d'uso (*Abbot*).

Oggi identifichiamo le classi dai modelli dinamici.

Ci sono 2 euristiche buone:

- 1-Azioni e attività nei diagrammi degli stati sono candidate per le operazioni nelle classi.
- 2-Le linee di attività nei diagrammi delle sequenze sono candidate per gli oggetti.

MODELLAZIONE DINAMICA CON UML

Ci sono 2 tipi di diagrammi per la modellaz. Dinamica:

- Diagrammi delle interazioni: descrivono il comportamento dinamico tra oggetti.

- Diagrammi degli stati: descrivono il comportamento dinamico di un singolo oggetto.
-

COME DETERMINIAMO LE OPERAZIONI?

Cerchiamo oggetti che stanno interagendo ed estraiamo il loro “protocollo”.

Cerchiamo gli oggetti che individualmente hanno un comportamento interessante.

- Un buon *punto di partenza*: flusso di eventi nella descrizione dei casi d’uso: dal flusso procediamo col diagramma delle sequenze per trovare gli oggetti partecipanti.
-

COSA E’ UN EVENTO?

Qualcosa che si verifica in un istante.

Un evento comporta l’invio di info da un oggetto all’altro.

Gli eventi possono avere associazioni tra loro:

- Relazione causale: un evento si verifica sempre prima o dopo un altro evento.
- Relazioni non causale: gli eventi sono concorrenti.

Gli eventi possono essere raggruppati in classi con una struttura gerarchica (*tassonomia degli eventi*).

DIAGRAMMA DELLE SEQUENZE

Descrizione grafica degli oggetti che partecipano in un caso d'uso usando la notazione DAG.

Euristiche per trovare gli oggetti partecipanti:

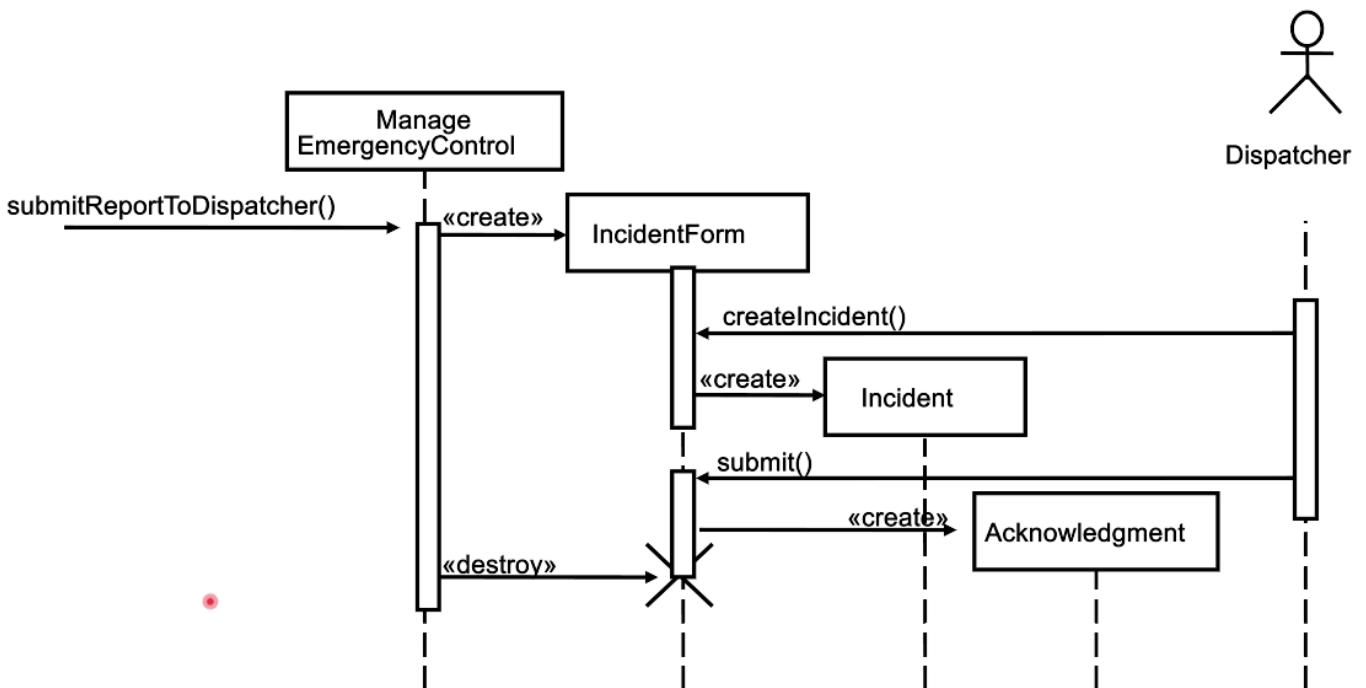
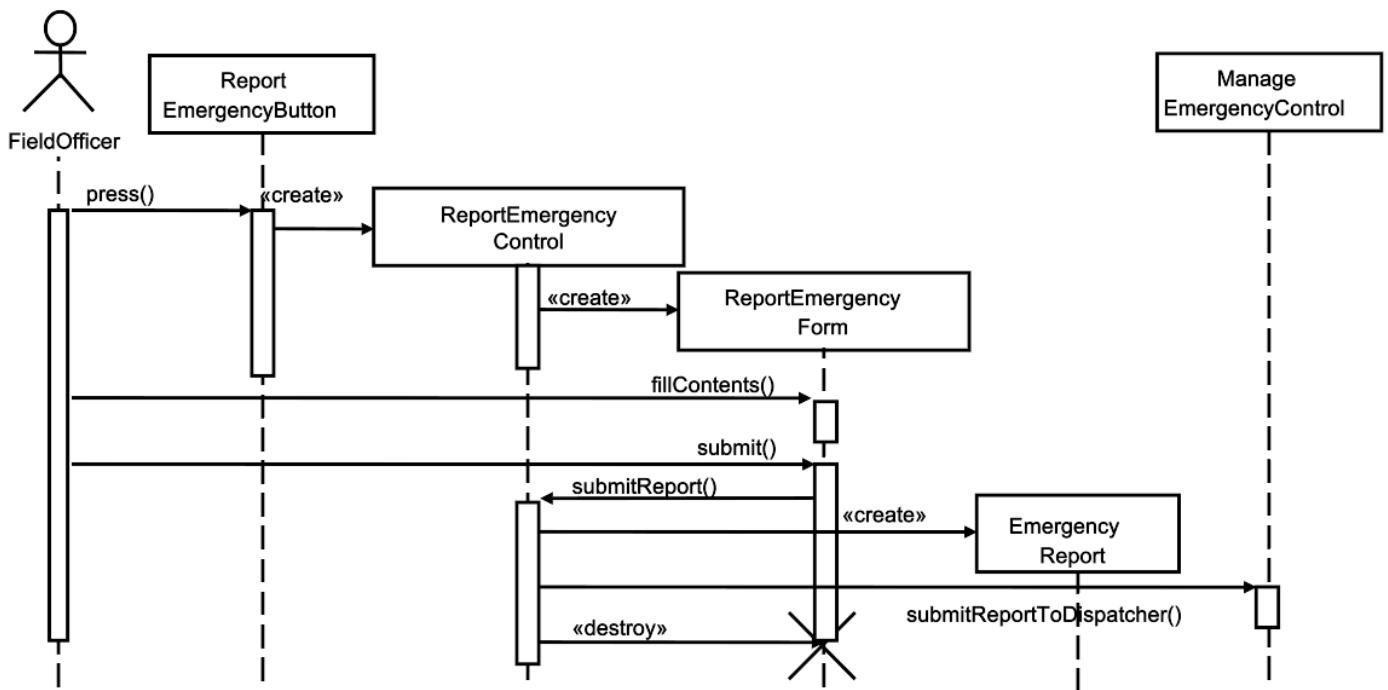
- Un evento ha sempre mittente e destinatario.
 - Trovare *mittenti e destinatari* per ciascun oggetto → questi sono gli oggetti che partecipano in un caso d'uso
-

MAPPIND CASI D'USO-OGGETTI CON DIAGRAMMI DI SEQUENZE

Un diagramma di seq lega i casi d'uso con gli oggetti. Mostra come il comportamento di un caso d'uso o scenario sia distribuito tra oggetti partecipanti.

ESEMPIO REPORTEMERGENCY

- Vediamo i diagrammi di sequenze associati con il caso d'uso *ReportEmergency*
 - Le colonne rappresentano gli oggetti che partecipano al caso d'uso
 - La colonna più a sinistra rappresenta l'attore che inizia il caso d'uso
 - Le frecce orizzontali attraverso le colonne rappresentano messaggi o stimoli inviati da un oggetto all'altro
 - Il tempo trascorre verticalmente dall'alto in basso



DIAGRAMMI DI SEQUENZE: DISTRIBUZIONE DEL COMPORTAMENTO

Cou i diagrammi delle sequenze non solo si modella l'ordine delle interazioni fra oggetti ma si distribuisce anche il comportamento del caso d'uso:

- Si attribuiscono le responsabilità ad ogni oggetto sotto forma di insieme di operazioni
 - Queste operazioni possono essere condivise da qualsiasi caso d'uso in cui un dato oggetto partecipa
 - È da osservare che la definizione di un oggett condiviso attraverso 2 o più casi d'uso dovrebbe essere identica:
 - Se un'operazione appare in più diagrammi delle seq, il suo comportamento dovrebbe essere identico.
-

DIAGRAMMI DELLE SEQ: CONDIVISIONE OPERAZIONI

Condividere le operazioni attraverso i casi d'uso consente di eliminare ridondanze nella specifica dei requisiti. Alla chiarezza bisognerebbe sempre dare la precedenza rispetto all'eliminazione della ridondanza. Frammentare il comportamento attraverso molte operazioni complica inutilmente la specifica dei requisiti.

DIAGRAMMA DI SEQ: USO NELL'ANALISI

Nell'analisi, i diagrammi delle seq sono usati per aiutare ad identificare nuovi oggetti partecipanti e comportamenti mancanti.

Questioni legate all'implementazioni non sono affrontate in questo punto.

I diagrammi delle seq richiedono molto tempo per cui gli sviluppatori dovrebbero focalizzarsi prima su funzionalità problematiche o non specificate:

- Disegnare diagrammi per parti semplici o ben definite del sistema non rappresenta un buon investimento di risorse.
-

EURISTICHE PER DIAGRAMMI DELLE SEQUENZE

- Layout:
 - Colonna 1 :attore
 - Colonna 2: oggetto *boundary*
 - *Colonna 3: oggetto control*
- Creazioni degli oggetti
 - Creare oggetti *control* all'inizio del flusso
 - Gli oggetti *control* creano gli *boundary*

- Accesso agli oggetti
 - Gli oggetti *entity* sono acceduti da oggetti *control* e *boundary*
 - Gli *entity* non dovrebbero accedere a oggetti *control* e *boundary*
-

COSA POSSIAMO AGGIUNGERE SUI DIAGRAMMI DELLE SEQUENZE?

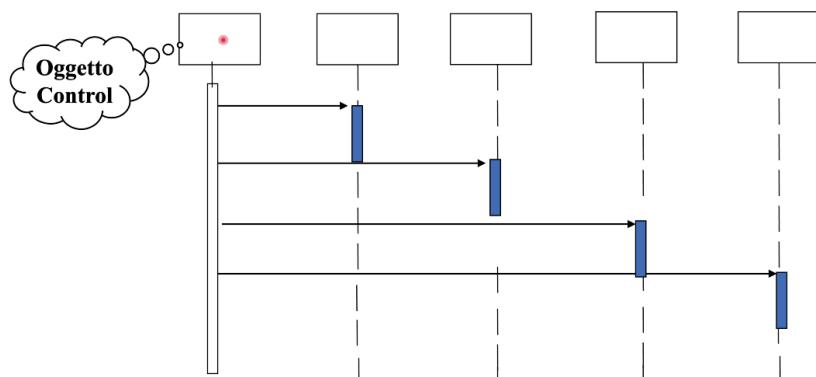
I diagrammi delle seq sono derivati dai casi d'uso.
La struttura del diagramma delle seq ci aiuta a determinare quanto è decentralizzato il sistema.

Distinguiamo 2 strutture per i diagrammi delle seq:

- ***Fork and Stair Diagrams***
-

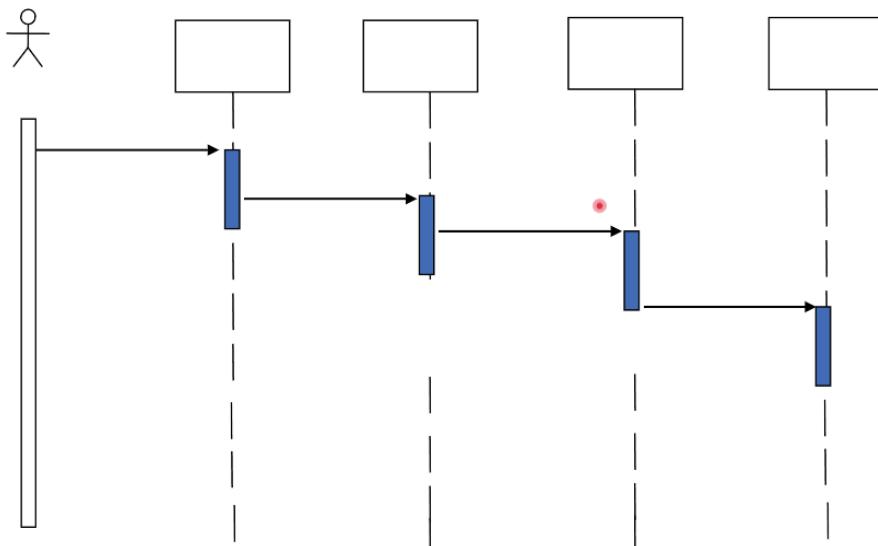
FORK DIAGRAM

Il comportamento dinamico è posto in un singolo oggetto, solitamente l'oggetto *control*: conosce tutti gli oggetti e li usa per domande e comandi diretti.



STAIR DIAGRAM

Il comportamento dinamico è distribuito. Ogni oggetto delega la responsabilità ad altri oggetti.



FORK O STAIR?

I fan “OO” sostengono che la stair sia la migliore.
Consiglio di modellazione:

- **Scegliere stair** - una struttura di controllo decentralizzata - se
 - Le operazioni hanno una forte connessione
 - Le operazioni saranno sempre eseguite nello stesso ordine
- **Scegliere fork** - una struttura di controllo centralizzata - se
 - Le operazioni possono cambiare ordine
 - Ci si aspetta che possano essere aggiunte nuove operazioni a seguito di nuovi requisiti

QUALE MODELLO E' DOMINANTE?

- Modello a oggetti: il sistema ha classi con stati non banali e molte relazioni tra le classi.

- Modello dinamico: molti tipi di eventi diversi
- Modello funzionale: il modello esegue trasformazioni complicate

Il modello dominante dipende dal tipo di sistema che sviluppiamo.

Esempi di modelli dominanti

- Compilatore:
 - Il modello funzionale è più importante
 - Il modello dinamico è banale poiché c'è solo un tipo di input solo pochi output
 - E' vero anche per gli IDE?
- Database:
 - Il modello ad oggetti è il più importante
 - Il modello funzionale è banale, poiché lo scopo delle funzioni è memorizzare, organizzare e recuperare i dati
- Spreadsheet:
 - Il modello funzionale è il più importante
 - Il modello dinamico è interessante se il programma consente di fare calcoli sulle celle
 - Il modello ad oggetti è banale

MODELLARE IL COMPORTAMENTO DIPENDENTE DALLO STATO DEGLI OGGETTI

I diagrammi delle seq. Sono usati per distribuire il comportamento tra oggetti e identificare le operazioni. I diagrammi degli stati rappresentano il

comportamento dal punto di vista di un singolo oggetto.

Vedere il comportamento dalla prospettiva di ogni oggetto consente di costruire una descrizione più formale del comportamento dell'oggetto.

Focalizzandosi sugli stati gli sviluppatori possono identificare nuovi comportamenti.

È da osservare che non è necessario costruire diagrammi degli stati per ogni classe del sistema:

- Si considerano oggetti con arco di vita esteso e con comportamento dipendente dallo stato.
 - Questa è quasi sempre la regola per oggetti *control*, meno per *entity* e quasi mai per *boundary*.
-

RIVEDERE IL MODELLO DI ANALISI

Il modello di analisi è costruito incrementalmente ed iterativamente:

- Raramente si ottiene un modello completo al primo passo.
- Sono necessarie molte iterazioni con clienti e utenti.

Una volta che il modello è stabile, viene rivisto dagli sviluppatori e poi coi clienti. L'obiettivo della revisione è assicurare che la specifica dei requisiti sia corretta, completa, consistente e non ambigua.

DOMANDE DA PORSI PERCHE' IL MODELLO SIA CORRETTO

- Il glossario degli oggetti entità è comprensibile all'utente?
- Le classi astratte corrispondono ai concetti a livello utente?
- Le descrizioni sono tutte in accordo con le definizioni dell'utente?
- Tutti gli oggetti entity e boundary hanno predicati nominali significativi come nomi?
- I casi d'uso e gli oggetti control hanno tutti i predicati verbali significativi come nomi?
- I casi d'errore sono tutti descritti e gestiti?

DOMANDE SE E' CONSISTENTE

- Ci sono classi o casi d'uso multipli con lo stesso nome?
- Entità (casi d'uso, classi, attributi) con lo stesso nome denotano concetti simili?
- Ci sono oggetti con attributi e associazioni simili che non sono nella stessa gerarchia di generalizzazione?

DOMANDE SE E' REALISTICO

- Ci sono caratteristiche nuove nel sistema? Sono stati costruiti prototipi o eseguiti studi per assicurarne la fattibilità?
 - I requisiti delle prestazioni e dell'affidabilità possono essere soddisfatti? Questi requisiti sono stati verificati da qualche prototipo eseguito su hardware selezionato?
-

DOMANDE PER L'ANALISI DEI REQUISITI

1. Quali sono le trasformazioni?



Modellazione funzionale

Creare scenari e diagrammi dei casi d'uso

- Parlare con il cliente, osservare, acquisire informazioni da archivi

2. Qual è la struttura del sistema?



Modellazione degli oggetti

Creare i diagrammi delle classi

- Identificare gli oggetti
- Quali sono le associazioni tra loro?
- Qual è la loro molteplicità?
- Quali sono gli attributi degli oggetti?
- Quali operazioni sono definite sugli oggetti?

3. Qual è il comportamento?

Creare i diagrammi delle sequenze

- Identificare mittenti e destinatari
- Mostrare sequenze di eventi scambiati tra gli oggetti
- Identificare le dipendenze tra eventi e concorrenza di eventi

Creare i diagrammi degli stati

- Solo per oggetti dinamicamente interessanti

Analisi in pratica

1. Analizzare la definizione del problema

- Identificare i requisiti funzionali
- Identificare i requisiti non funzionali
- Identificare i vincoli (pseudo requisiti)

2. Costruire il modello funzionale:

- Sviluppare i casi d'uso per illustrare i requisiti delle funzionalità

3. Costruire il modello dinamico:

- Sviluppare i diagrammi delle sequenze per illustrare le interazioni tra gli oggetti
- Sviluppare i diagrammi di stato per gli oggetti con comportamento interessante

4. Costruire il modello ad oggetti:

- Sviluppare i diagrammi delle classi che mostrano la struttura del sistema
-

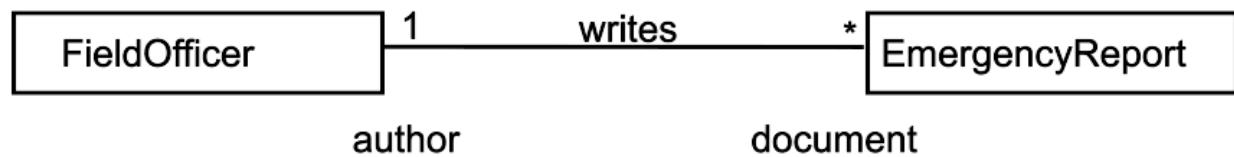
IDENTIFICARE LE ASSOCIAZIONI

Un'associazione mostra una relazione tra 2 o più classi.

Identificare le associazioni comporta 2 vantaggi:

1. Chiarisce il modello di analisi rendendo esplicite le relazioni tra gli oggetti
2. Consente agli sviluppatori di scoprire casi limite associati ai collegamenti
 - a. I casi limite sono eccezioni che devono essere chiare nel modello.

Esempio di associazione tra le classi *EmergencyReport* e *FieldOfficer*



IDENTIFICARE LE ASSOCIAZIONI

- Nome
- Ruolo
- Molteplicità

IDENTIFICAZIONE DELLE ASSOCIAZIONI

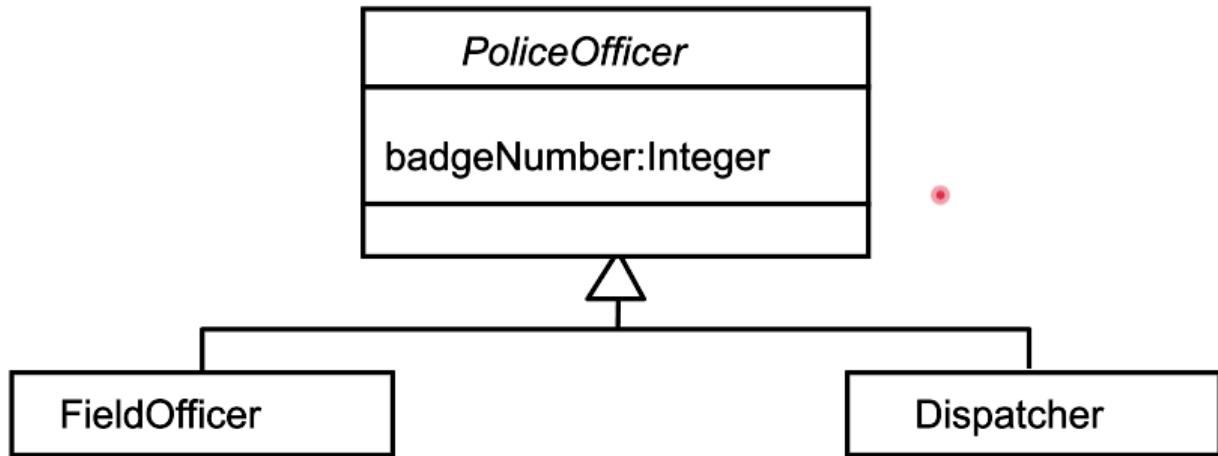
Le associazioni fra oggetti sono le più importanti.
In accordo con Abbot, le associazioni possono essere identificate esaminando verbi e predicati.

ELIMINARE ASSOCIAZIONI RIDONDANTI

Il modello ad oggetti include, all'inizio, troppe associazioni. In una fase di revisione si elimina qualche associazione inutile.

MODELLARE RELAZIONI DI EREDITARIETA' TRA OGGETTI

La generalizzazione è usata per eliminare ridondanza.
Ecco un esempio:



SOMMARIO DELL'ANALISI

L'attività di scoperta dei requisiti è iterativa e incrementale.

Inizialmente la scoperta dei requisiti somiglia a un *brainstorming*: non appena la descrizione del sistema cresce e i requisiti sono più concreti, gli sviluppatori devono estendere e modificare il modello di analisi in modo più ordinato.

Lez 15 → Progettazione del sistema: decomposizione in sottosistemi

“ La progettazione è difficile. ”

Questo perché mentre l’analisi è un’attività che prevede step consolidati, la progettazione è caotica poiché comporta unione di aspetti relativi il dominio dell’applicazione e la parte relativa all’implementazione. Quando si parla dell’implementazione si parla anche dello stato della tecnologia odiera. Bisogna capire ciò che si sviluppa dove deve girare in termini di HW e altro... C’è una varietà in un contesto in cui le modifiche sono all’ordine del giorno a causa della modifica dei requisiti frequentemente.

SCOPO DELLA PROGETTAZIONE DEL SISTEMA

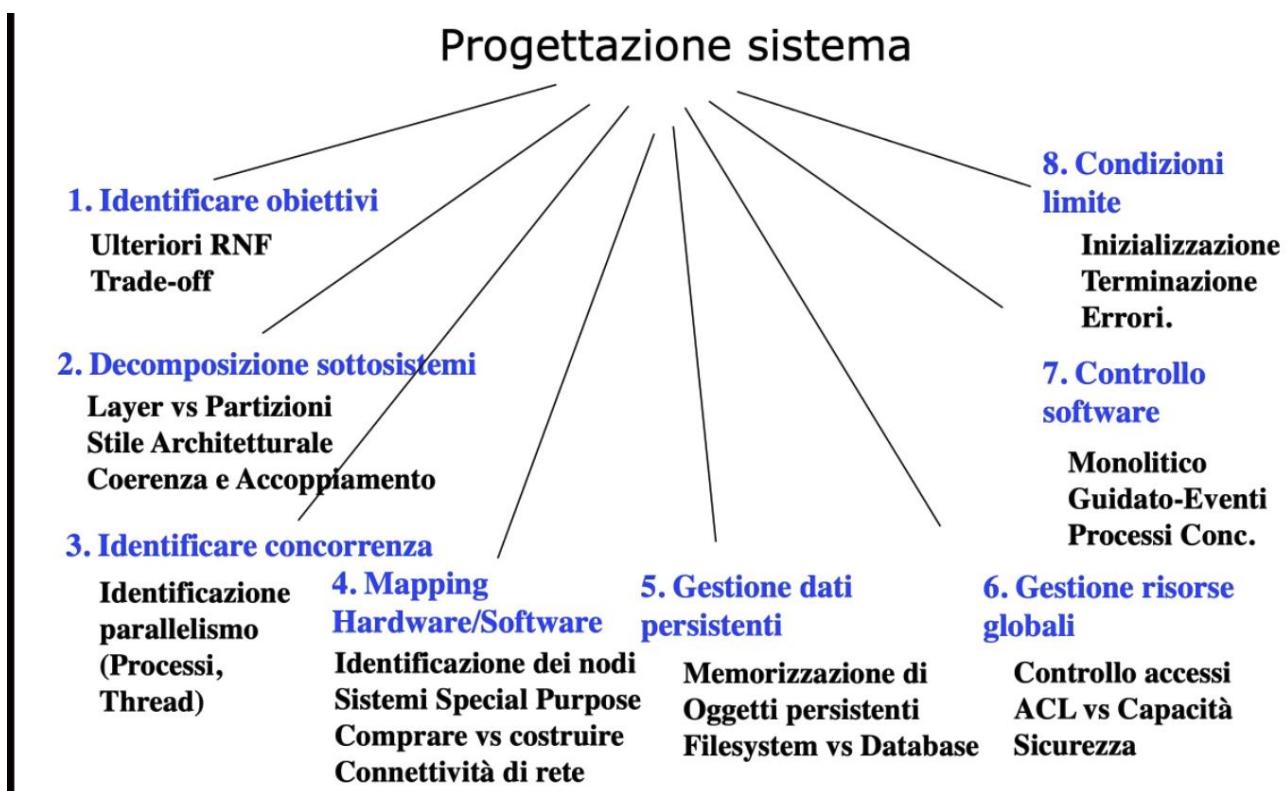


Colmare il gap tra problema e sistema da consegnare al cliente.

Come?

- Con la tecnica *Divide et Impera* dividento problemi in sottoproblemi affrontando i principali obiettivi.

PROGETTAZIONE SISTEMA: 8 PROBLEMATICHE



COME I MODELLI DI ANALISI INFLUENZANO LA PROGETTAZIONE

- Requisiti non funzionali: definizione degli obiettivi di progettazione
- Modello funzionale: decomposizione in sottosistemi
- Modello oggetto: mapping HW/SW

- Modello dinamico: gestione risorse globali, controllo SW, concorrenza
 - Decomposizioni in sottosistemi: condizioni limite.
-

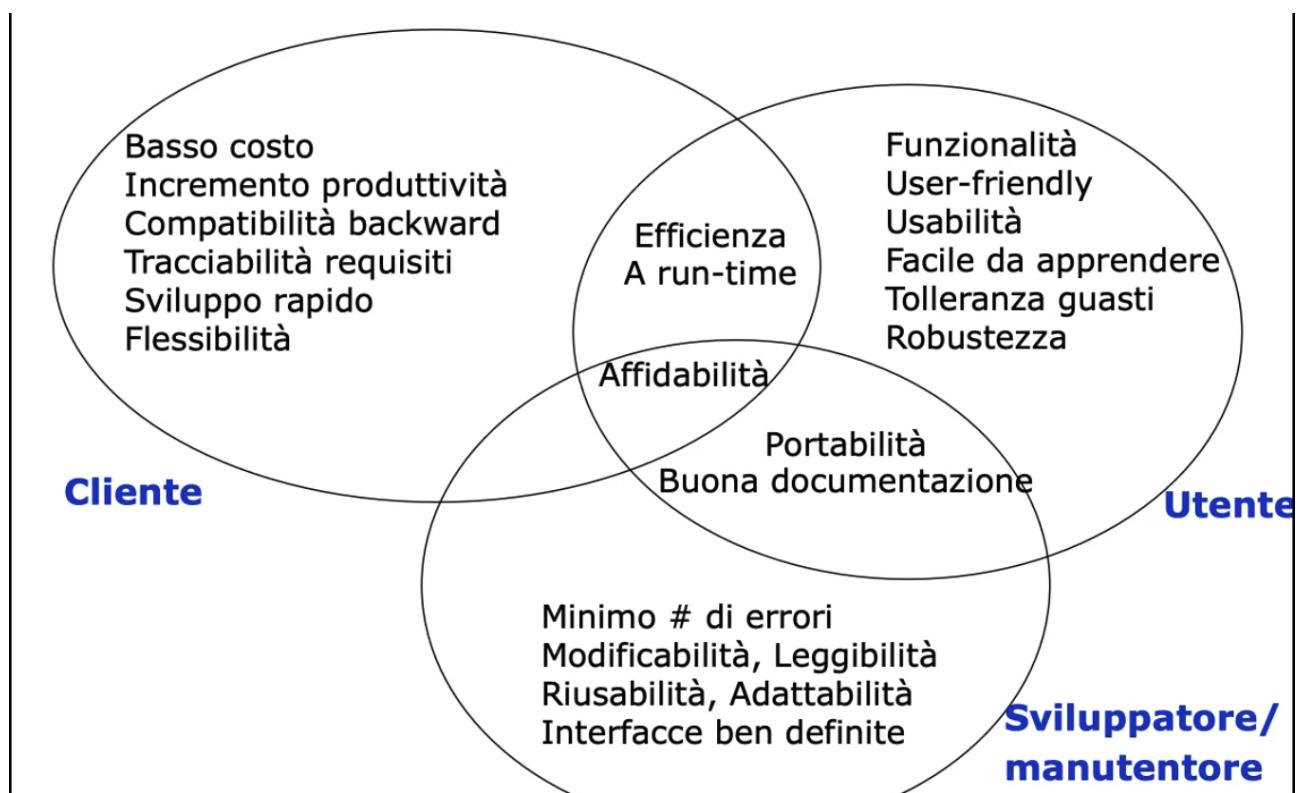
OBIETTIVI DI PROGETTAZIONE

Governan le attività di progettazione.

Ogni requisito non funzionale è un obiettivo di progettazione.

Gli obiettivi di progettazione sono spesso in conflitto.

Gli stakeholder hanno diversi obiettivi di progettazione.



TRADE OFF DI PROGETTAZIONE

- Costo vs Robustezza: se voglio un sistema a basso costo e “robusto” è impossibile.
- Efficienza vs Portabilità: se vogliamo sviluppare un gioco real-time efficiente e portabile significa che vogliamo usare librerie grafiche specifiche che sfruttano HW dedicati e quindi la portabilità non è applicabile dato che si usano determinati HW.
- Sviluppo veloce vs Funzionalità.
- Compatibilità vs Leggibilità.
- Funzionalità vs Usabilità.
- Costo vs Riusabilità.

Esempio: funzionalità vs usabilità

Un sistema con 100 funzioni è usabile?

Alcuni sistemi non sono usabili neanche con due funzioni

Foursquare for searching venues

Foursquare è un **social network** basato sulla geolocalizzazione disponibile tramite web e applicazioni per **dispositivi mobili**

Foursquare ha due casi d'uso base:

1. checking-in in posizioni e condividere la propria posizione con i propri amici
2. un caso d'uso per trovare un luogo ben votato nelle vicinanze



COSTO VS RIUSABILITA'

Supponiamo di aver modellato l'associazione tra due classi con molteplicità 1 a 1: facile da implementare, bassi costi per il test ma poco riusabile.

Modifichiamo da 1 a 1 con M a N: costo maggiore.

In breve la figura spiega tutto:



DECOMPOSIZIONE IN SOTTOSISTEMI: SOTTOSISTEMI E SERVIZI

- Sottosistema:
 - Insieme di classi, associazioni, operazioni ed eventi che sono strettamente correlati.
 - Le classi nel modello ad oggetti costituiscono la base per i sottosistemi.

- Servizio
 - Gruppo di operazioni collegati che condividono uno scopo comune.
 - I casi d'uso sono la base per individuare i servizi.

SERVIZI E INTERFACCE DEI SOTTOSISTEMI

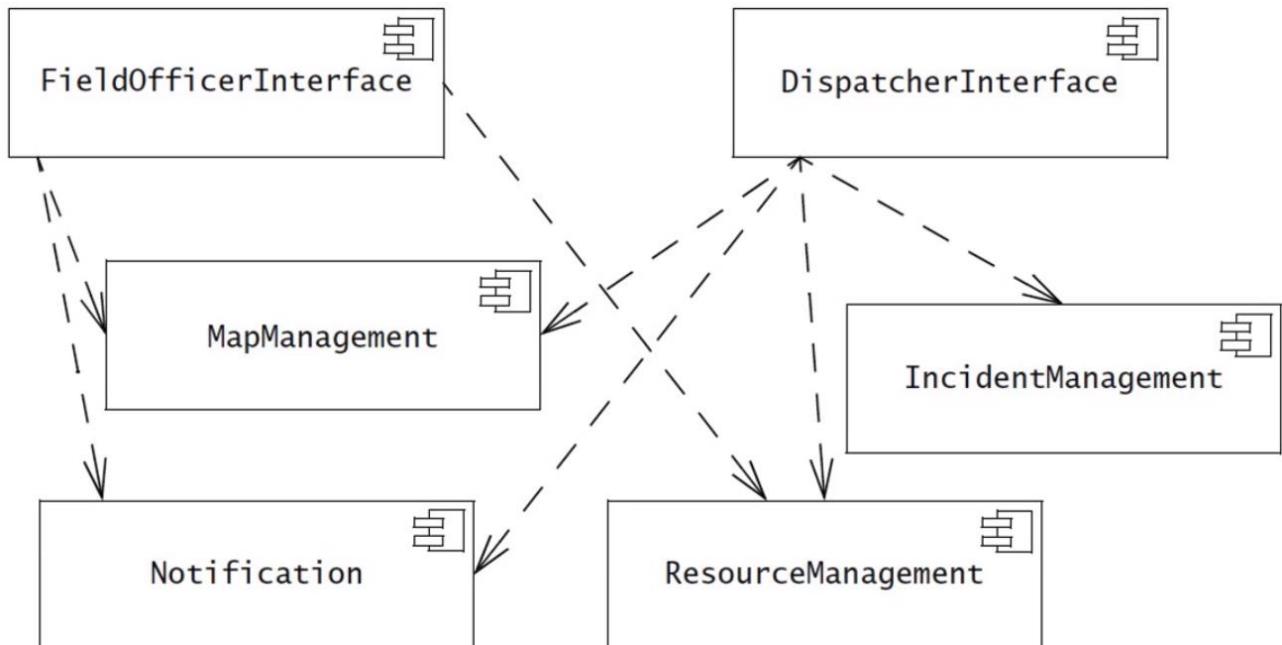
L'interfaccia del sottosistema include il nome delle operazioni, parametri e tipi e valori di ritorno.
La progettazione del sistema si focalizza sulla definizione dei servizi forniti da ogni sottosistema.

La progettazione degli oggetti si focalizza sulle API, che rifiiniscono ed estendono le interfacce dei sottosistemi.

Esempio

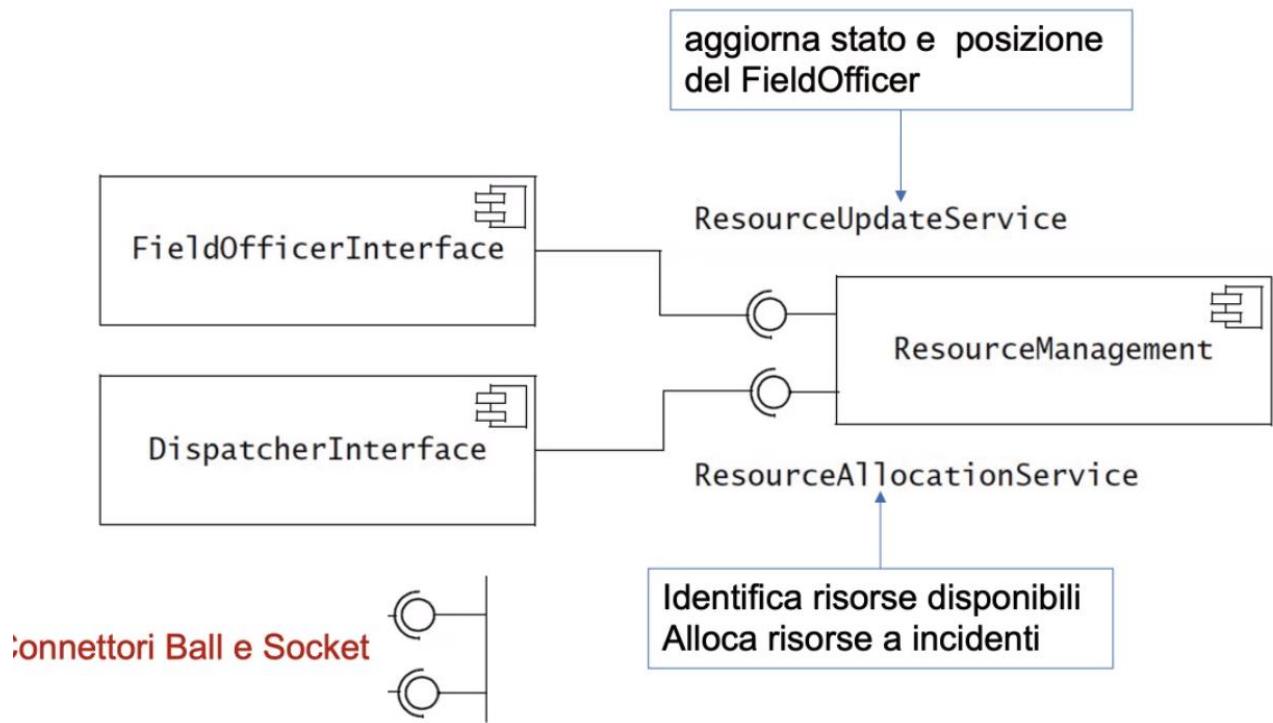
- Il sistema di gestione degli incidenti può essere decomposto in
 - sottosistema *DispatcherInterface*, che realizza l'interfaccia utente per il *Dispatcher*
 - sottosistema *FieldOfficerInterface*, che realizza l'interfaccia utente per il *FieldOfficer*
 - sottosistema *IncidentManagement*, responsabile della creazione, modifica e memorizzazione degli incidenti (*Incidents*)
 - sottosistema *ResourceManagement*, responsabile di tener traccia delle risorse (*Resources*) disponibili (Mezzi dei vigili del fuoco, ambulanze, etc.)
 - sottosistema *MapManagement*, per illustrare mappe (*Maps*) e posizioni (*Locations*)
 - sottosistema *Notification*, che implementa la comunicazione tra i terminali dei *FieldOfficer* e le stazioni dei *Dispatcher*

DIAGRAMMI DELLE COMPONENTI



Ci sono i sottosistemi e le frecce sono le dipendenze.

Interfacce richieste e fornite



- Ball servizi offerti.
- Socket servizi richiesti.

ACCOPPIAMENTO E COESIONE DEI SOTTOSISTEMI

Obiettivi: ridurre la complessità del sottosistema al contempo permettendo la modificabilità.

- Coesione (misura la dipendenza tra le classi):
 - Alta coesione: le classi nel sottosistema eseguono compiti simili e sono collegate con diverse associazioni (è OK).
 - Bassa coesione: molti classi miste e ausiliarie.

- Accoppiamento (misura la dipendenza tra sottosistemi)
 - Alto accoppiamento: le modifiche ad un sottosistema avranno un forte impatto sugli altri.
 - Basso accoppiamento: una modifica a sottosistema non influenza altri (è OK).
-

COME AVERE ALTA COESIONE

Domande da porsi:

- Un sottosistema chiama sempre un altro sottosistema per uno specifico servizio?
 - Si: prendere in considerazione di spostarli nello stesso sottosistema
- Quali sottosistemi invocano i rispettivi servizi?
 - Può essere evitato ristrutturando i sottosistemi o cambiando l'interfaccia del sottosistema?
- I sottosistemi possono anche essere ordinati gerarchicamente (a strati)?

COME AVERE BASSO ACCOPPIAMENTO

Si ha se una classe invocante non ha bisogno di conoscere dettagli interni della classe invocata.

Domande da porsi:

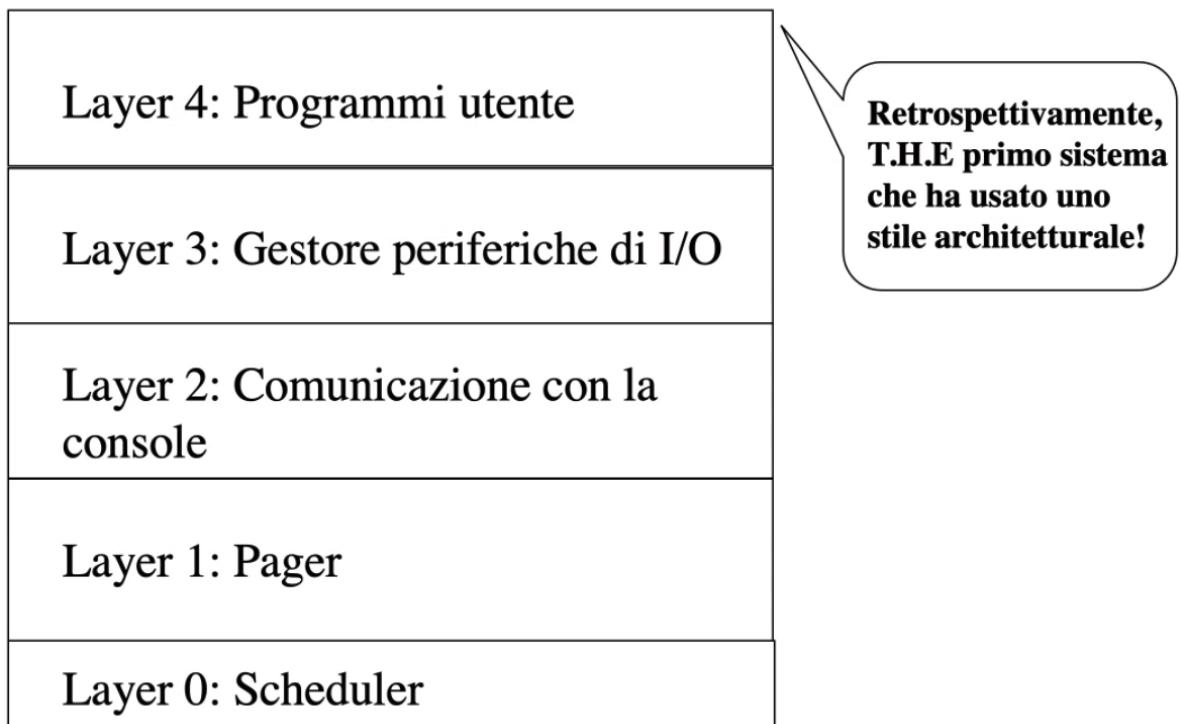
- La classe invocanti deve conoscere qualche attributo delle classi nei livelli inferiori?
- E' possibile che la classe invocante chiami solo le operazioni delle classi del livello inferiore?

Per una buona progettazione bisogna evitare lo *Spaghetti Design*.

Dijkstra sviluppò il sistema T.H.E.

Gli strati del Sistema T.H.E.

“Un sistema operativo è una gerarchia di strati , con ogni strato che usa i servizi offerti dai livelli inferiori”



STRATI E PARTIZIONI

Uno strato è un sottosistema che fornisce un servizio ad un altro sottosistema con le seguenti restrizioni:

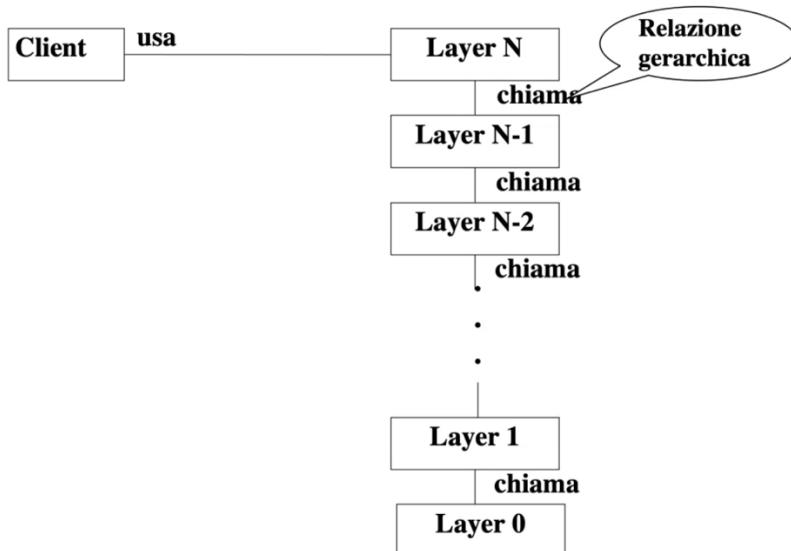
- Uno strato dipende solo dai servizi degli strati inferiori.
- Uno strato non ha conoscenza dei superiori.

Uno strato può essere diviso orizzontalmente in più sottosistemi indipendenti chiamati partizioni:

- Le partizioni forniscono i servizi ad altri partizioni sullo stesso strato.

ESEMPIO

Stile architetturale a strati



RELAZIONE GERARCHICA TRA SOTTOSISTEMI

Ci sono 2 tipi di gerarchia:

- Uno strato A **dipende** da B (dipendenza a tempo di compilazione)
- Lo strato A **chiama** lo strato B (dip a run time).
 - Esempio: Un web browser chiama un web server
 - Gli strati client e server possono essere eseguiti sulla stessa macchina?
 - Sì, sono strati, non nodi di elaborazione
 - Il mapping degli strati ai processori è deciso durante il mapping Software/hardware!
- Convenzioni UML:
 - Relazioni a runtime sono associazioni con line tratteggiate
 - Relazioni a tempo di compilazione sono associazioni con line continue

ARCHITETTURA CHIUSA (stratificazione opaca)

ARCHITETTURA APERTA (" trasparente")

Ogni strato può chiamare operazioni da qualsiasi strato inferiore.

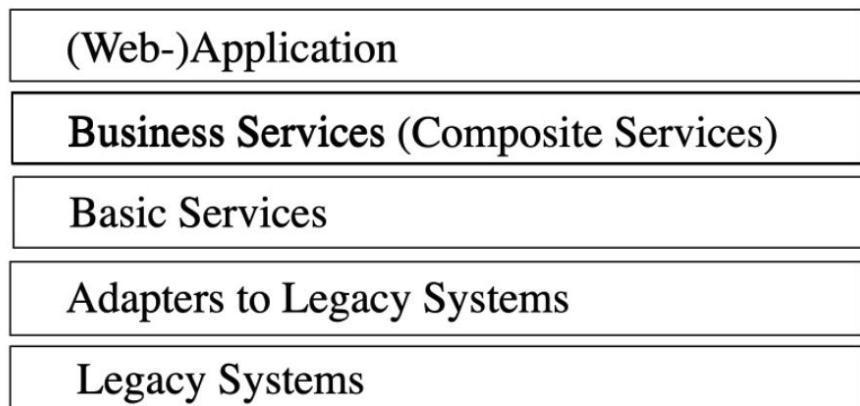
Obiettivo di progettazione: efficienza a runtime.

È più flessibile e migliora le prestazioni.

SOA è uno stile architetturale a strati

Service Oriented Architecture (SOA)

- Idea base: Un **service provider** (“business”) offre servizi (“business processes”) a un **service consumer** (applicazione, “customer”)
 - I servizi sono scoperti in modo dinamico, di solito offerti in applicazioni web-based
 - I servizi sono creati componendoli dai servizi dei livelli inferiori (servizi base)
- I servizi base solitamente sono basati su sistemi legacy
- Sono usati degli adattatori per fare da collante tra servizi base e sistemi legacy



PROPRIETA' DEI SISTEMI A STRATI

Sono gerarchici. È una progettazione desiderabile poiché riduce la complessità.

Le architetture chiuse sono più portabili mentre le aperte sono efficienti.

I sistemi a strati spesso hanno il problema dell'*uovo e della gallina (quale è nato prima?)*.

Un altro esempio dell'architettura a strati è il modello ISO/OSI.

ARCHITETTURA CLIENT/SERVER

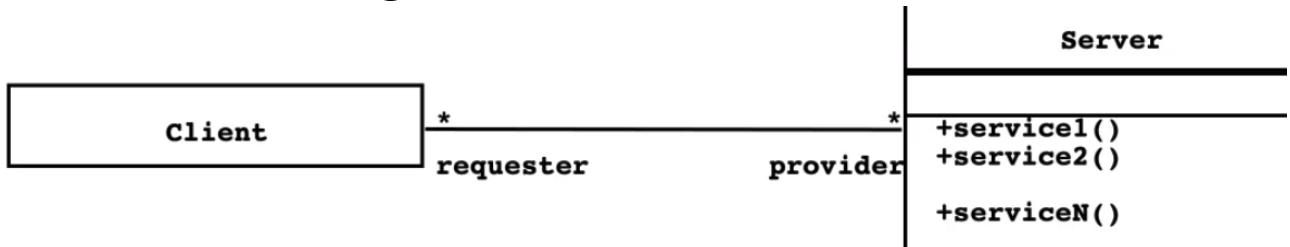
Usate spesso nella progettazione dei database.

- Funzioni eseguite dal client:
 - Input dell'user.
 - Elaborazione front-end dei dati di input.

- Funzioni eseguite dal server di database:
 - Gestione dei dati.
 - Integrità dei dati.
 - Sicurezza del database.

È un caso speciale di stile architetturale stratificato perché il client è in uno strato e il server in un altro.

Ogni client invoca il server che esegue servizi e ritorna i risultati. La risposta è di solito immediata.
Gli utenti interagiscono solo col client.



OBIETTIVI DI PROGETTAZIONI PER LE ARCHITETTURE CLIENT/SERVER

- Portabilità del servizio
- Trasparenza posizione
- Elevate prestazioni
- Scalabilità
- Flessibilità
- Affidabilità

Problemi con architetture Client/Server.
I C/S non forniscono comunicazioni *peer-to-peer*.
Le comunicazioni p2p sono spesso necessarie.

Abbiamo quindi le architetture p2p.

STILE ARCHITETTURALE P2P

Generalizzazione dello stile C/S.

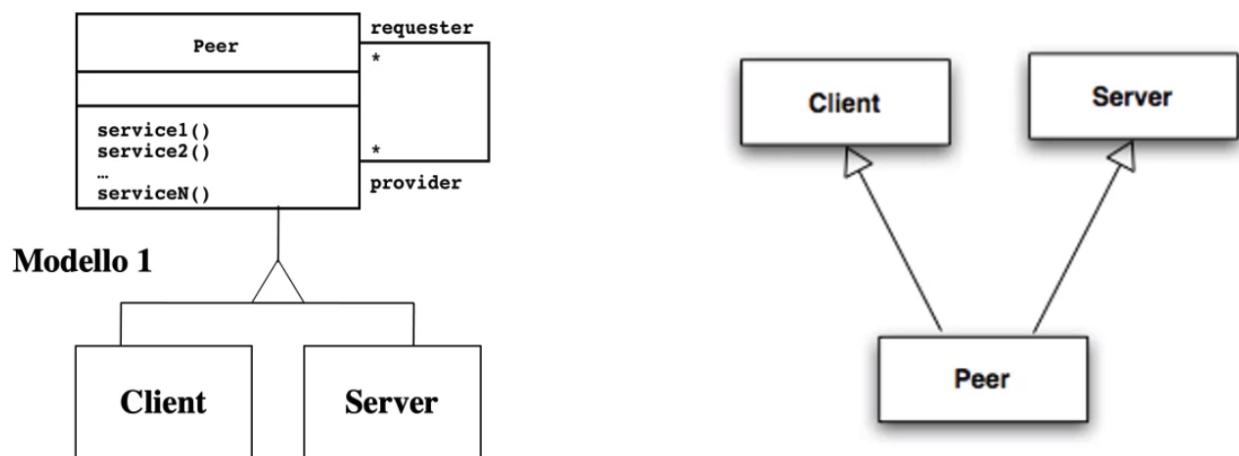
I client e i server possono essere entrambi *peer*.

Come modelliamo questa affermazione?

- Proposta 1: un peer può essere un client o un server
- Proposta 2: un peer può essere un client e un server

Definizione del problema: i client possono essere server e viceversa.

- Modello 1: un peer può essere un client o un server
- Modello 2: un peer può essere un client e un server



Continuo lezione 15

STILE ARCHITETTURALE A 3-STRATI e ARCHITETTURA 3-TIER

- Definizione: a 3-STRATI:
 - Uno stile architetturale in cui un'applicazione consiste di 3 sottosistemi ordinati gerarchicamente
 - Interfaccia utente, *middleware* e un database.
 - Il sottosistema *middleware* serve la richiesta di dati tra l'interfaccia utente e il database.
- Definizione: Architettura 3-TIER
 - Un'architettura SW in cui i 3 strati sono allocati su tre *nodi HW* separati.

NB: *Layer* è un tipo (classe, sottosistema) e *Tier* è un'istanza (oggetto, nodo hw). *Layer* e *Tier* sono spesso usati in modo intercambiabile.

ESEMPIO A 3-STRATI

Usato spesso per lo sviluppo di siti web:

1. Il **Web Browser** implementa l'interfaccia utente
2. Il **Web Server** serve le richieste dal web browser
3. Il **Database** gestisce e fornisce accesso ai dati persistenti

VARIANTE: ARCHITETTURA A 4-STRATI

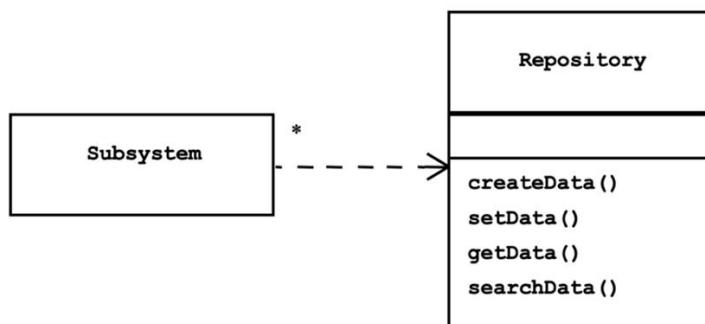
Viene usata per lo sviluppo di siti di commercio elettronico:

1. Il **Web Browser**, che fornisce l'interfaccia utente
2. Un **Web Server**, per servire le richieste HTML statiche
3. Un **Application Server**, che fornisce la gestione della sessione (es., i contenuti di un carrello elettronico) e l'elaborazione delle richieste HTML dinamiche
4. Un **Database back end**, che gestisce e fornisce accesso a dati persistenti
 - Nelle architetture 4-tier commerciali, è un database relazionale (RDBMS)

STILE ARCHITETTURALE REPOSITORY

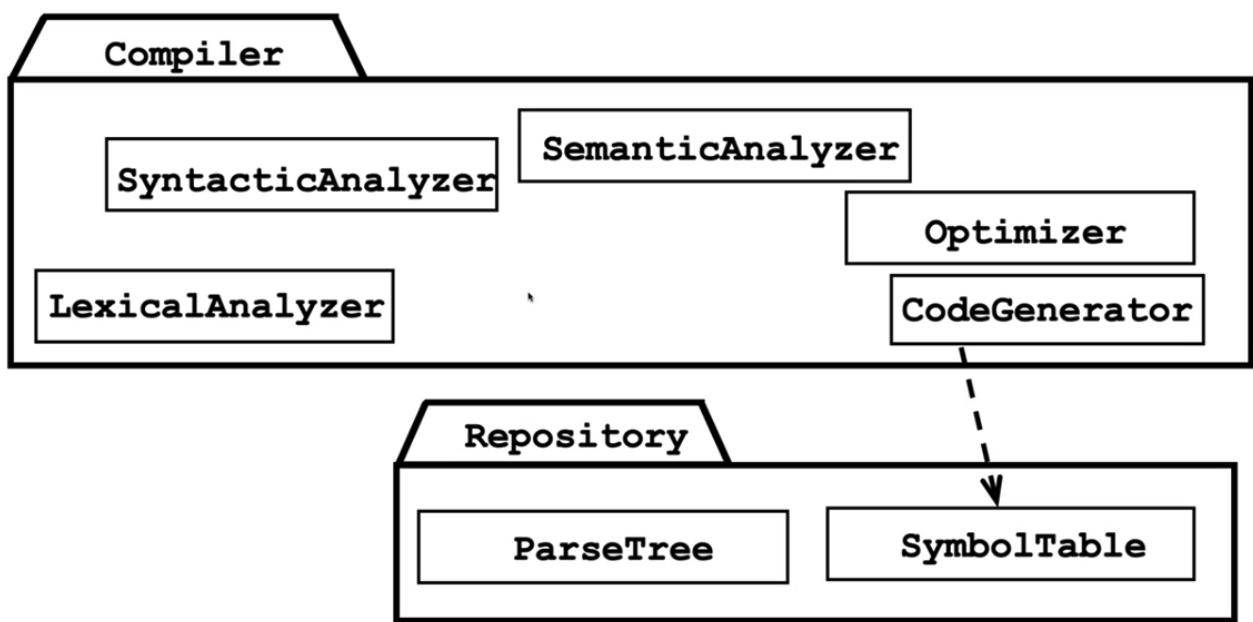
L'idea base dietro questo stile è supportare una collezione di programmi indipendenti che lavorano cooperativamente su una struttura dati comune chiamata *repository*.

I sottosistemi accedono e modificano i dati dal repository.



ESEMPIO ARCHITETTURA REPOSITORY: GLI IDE

La parte che riguarda il compilatore è costituito da una serie di sottosistemi: analizzatore, ottimizzatore, controllo del lessico... queste componenti accedono all'albero di *parsing* e alla *symbol table*, si sposta il parse e la symbol dalla componente del compilatore e viene inserita nella repository.



STILE MODEL-VIEW-CONTROLLER

Problema: nei sistemi con alto accoppiamento modificate all'interfaccia utente spesso forzano modificate agli oggetti *entity*:

- L'interfaccia utente non può essere reimplementata senza modificare la rappresentazione degli *entity*.

- Gli oggetti *entity* non possono essere riorganizzati senza modificare l'interfaccia utente
- SOLUZIONE: disaccoppiare! Il *Model view controller* disaccoppia l'accesso ai dati e la presentazione dei dati:

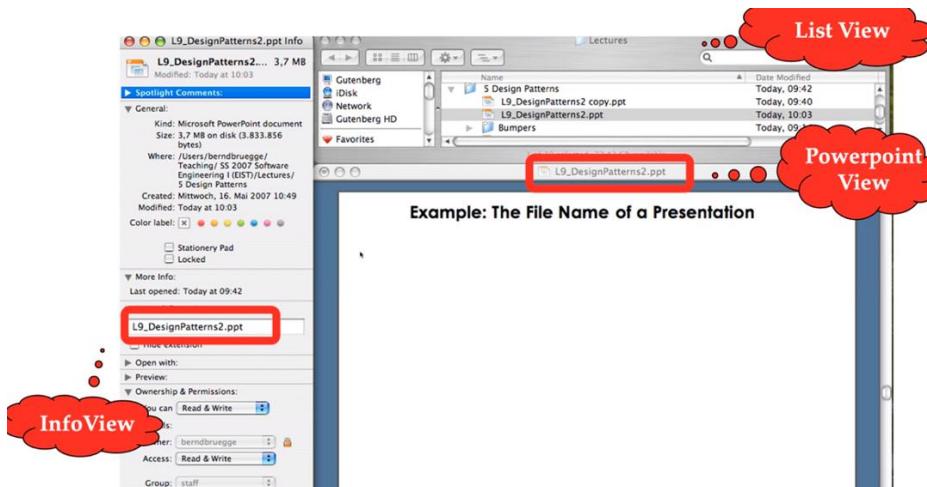
- View: sottosistemi che contengono oggetti boundary
- Model: sottosistema con oggetti entity
- Controller: sottosistema che media tra View (presentazione dati) e Modelli (accesso ai dati)

I sottosistemi sono classificati in 3 tipi diversi:

- Model: responsabile della conoscenza del dominio applicativo.
- View: responsabile della visualizzazione delle info all'utente
- Controller: responsabile dell'interazione con l'utente e della notifica alle view delle modifiche al modello.

Il MVC viene spesso usato in applicazioni per gestire le interfacce grafiche.

Abbiamo diverse view di un file:



Se cambiassi nome al file ho 3 possibilità, come funziona: in un'architettura MVC, il modulo *controller* effettua la modifica.

STILI MVC e 3-TIER A CONFRONTO

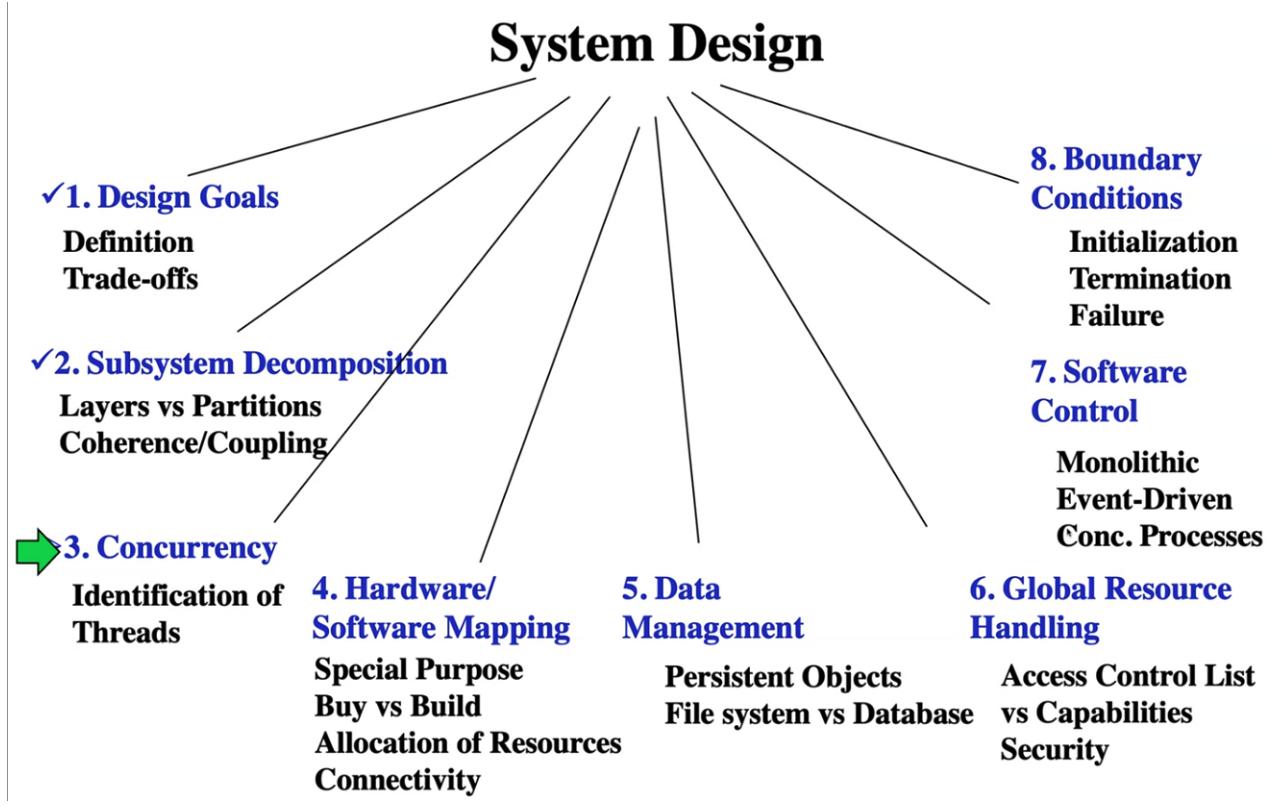
- MVC NON è gerarchico
 - Il 3-TIER invece si.
 - MVC è nato allo *xerox parc*, centro di ricerca in California.
 - 3-TIER è nato con l'introduzione delle applicazioni web, dove gli strati client, middleware e data sono eseguiti su piattaforme separate fisicamente.
-

PIPE E FILTRI

Una *pipeline* consiste di una catena di elementi di elaborazione organizzati in modo che l'output di un elemento sia l'input dell'elemento successivo, l'info che fluisce spesso è un flusso di record, byte e bit.

FINE SLIDE INIZIO LEZIONE 16

Lez 16 → OBIETTIVI DI PROGETTAZIONE



3. CONCORRENZA

Requisiti non funzionali da affrontare: prestazioni, tempo di risposta, latenza e disponibilità.

Due oggetti sono concorrenti se possono ricevere eventi nello stesso tempo senza interagire.

Oggetti concorrenti possono essere assegnati a differenti *thread di controllo*.

Oggetti con un'attività mutuamente esclusiva possono essere racchiusi in un singolo thread.

THREAD DI CONTROLLO

È un cammino attraverso un diagramma degli stati su cui è attivo un singolo oggetto alla volta.

I thread possono portare a *race condition*.

Esempio: problema coi thread → Bancomat.

Ho un cliente che va a prelevare. Il cliente vuole prelevare 50 euro. L'oggetto controllo riceve questo messaggio. Supponiamo che ci sia un altro bancomat con un altro cliente che accede con lo stesso bancomat contemporaneamente altrove. Anche egli ritira 50 euro (sul conto ce ne sono 200 in TOT).

Il cliente 1 chiede di vedere il saldo e riceve il messaggio 200 euro, lo stesso al cliente 2.

Poiché il cliente 1 preleva 50 il nuovo saldo sarà 150 e lo invia al conto corrente. Lo stesso avviene per cliente 2, il saldo sarà 150... sono scomparsi quindi 50 euro alla banca, questo è un errore gravissimo.

Una soluzione potrebbe essere accorpare i due flussi di controllo in un unico flusso di controllo sincronizzati.

IMPLEMENTARE LA CONCORRENZA

I sistemi concorrenti possono essere implementati su qualsiasi sistema che fornisce:

- Concorrenza fisica: thread forniti dall'hw

- Concorrenza

In tutti i casi dobbiamo provvedere alla schedulazione dei thread.

I SO moderni mettono a disposizione diversi meccanismi di scheduling.

A volte è necessario intervenire direttamente per schedulare l'attività (controllo del sw).

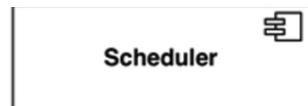
4 passo: MAPPING HW/SW

Quando abbiamo suddiviso il sistema in sottosistemi e ognuno di esso è costituito da classi e oggetti, questi sistemi vanno realizzati in HW o in SW?

Come corrispondiamo il modello a oggetti ai componenti hw e sw scelti?

- Mapping degli oggetti:
 - Processore, memoria, Input/Output
- Mapping delle associazioni
 - Connessioni di rete

DUE NUOVI TIPI DI DIAGRAMMA UML



- Diagramma dei componenti (**component**)
 - Mostra le dipendenze tra i componenti in fase di progettazione, compilazione e di esecuzione



- Diagramma di distribuzione (**deployment**)
 - Mostra la distribuzione dei componenti in fase di esecuzione
 - Usa nodi e connessioni per illustrare le risorse fisiche nel sistema

5- GESTIONE DEI DATI

- Alcuni oggetti nel modello del sistema sono **persistenti**
 - I valori dei loro attributi hanno un tempo di vita più lungo di una singola esecuzione
- Un oggetto persistente può essere realizzato con uno dei seguenti meccanismi
 - **Filesystem**
 - Se i dati sono usati da più lettori ed un solo scrittore
 - **Database**
 - Se i dati sono usati da lettori e scrittori concorrenti

Il filesystem è facilmente realizzabile ma è usabile solo se i dati sono acceduti da più lettori e un solo scrittore.

Se abbiamo più entità allora dobbiamo usare un database.

Domande per la gestione dei dati:

- Il database quanto spesso è acceduto?
- I dati devono essere archiviati?
- ---

MAPPING DEI MODELLI DEGLI OGGETTI

I modelli degli oggetti possono essere mappati a database relazionali.

Mapping:

- Ogni classe è una tabella

- Ogni attributo è una colonna della tabella
 - Un'istanza della classe rappresenta una riga
 - I metodi non hanno corrispondenze.
-

6. GESTIONE RISORSE GLOBALI

Riguarda il controllo degli accessi, descrive i permessi di accesso per diverse tipologie di attori e la modalità di protezione fornita da un oggetto rispetto ad accessi non autorizzati.

Nei sistemi multi utente, attori diversi hanno permessi di accesso diversi relativi a diverse funzioni e dati.

Come modelliamo tali accessi?

- Modelliamo l'accesso sulle classi con una *matrice degli accessi*:
 - Le righe rappresentano gli attori del sistema
 - Le colonne rappresentano le classi il cui accesso vogliamo controllare.

Il permesso d'accesso è un'entrata nella matrice degli accessi.

ESEMPIO: matrice degli accessi

		Classi	Permessi accesso			
		Attori	Arena	League	Tournament	Match
Attori	Operator	<<create>> createUser() view ()	<<create>> archive()			
LeageOwner	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()		
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()		
Spectator	view() applyForPlayer()	view() subscribe()	view()	view() replay()		

La matrice degli accessi non è ottimale. Si può migliorare usando una lista di controllo accessi o lista *capability*.

- **Lista controllo accessi**

- Associa una lista di coppie (attore, operazione) ad ogni classe da accedere
- Ogni volta che un'istanza di tale classe è acceduta, è controllata la lista degli accessi per il corrispondente attore e operazione

- **Capability**

- Associa una coppia (classe, operazione) con un attore
- Una capability attribuisce ad un attore il controllo accesso ad un oggetto della classe descritta dalla capability

Domande per le risorse globali

- Il sistema ha bisogno di autenticazione, se sì quale?
-

7. CONTROLLO DEL SW

Le scelte principali sono:

- Controllo implicito.
- Controllo esplicito (centralizzato o decentralizzato).
 - Controllo centralizzato
 - Guidato da procedura: il controllo risiede nel codice
 - Guidato da eventi: controllo in un dispatcher che invoca funzioni via callback
 - Controllo decentralizzato
 - Il controllo risiede in più oggetti indipendenti
 - Esempi: sistema basato su messaggi, RMI
 - Possibile speedup mappando gli oggetti su processori diversi, maggior overhead di comunicazione

CENTRALIZZAZIONE VS DECENTRALIZZAZIONE: DOMANDE

- Dovrebbe essere usata una progettazione centralizzata o decentralizzata?
 - Considerare il diagramma delle sequenze e gli oggetti di controllo dal modello di analisi.
 - Controllare la partecipazione degli oggetti di controllo nei diagrammi di sequenza
 - Se il diagramma appare come una biforcazione -> progettazione centralizzata
 - Se il diagramma sembra una scala -> progettazione decentralizzata
-

8. CONDIZIONI LIMITE

- Inizializzazione: il sistema è portato da uno stato non inizializzato a uno stato stabile.
- Terminazione: le risorse sono ripulite e sono notificati altri sistemi al momento della terminazione
- Guasto: possibili guasti (bug, errori...)

Una buona progettazione prevede i guasti fatali e fornisce meccanismi per affrontarli.

Domande per le condizioni limite

- Inizializzazione
 - Quali dati devono essere acceduti in fase di avvio?
 - Quali servizi devono essere registrati?
 - Cosa fa l'interfaccia utente all'avvio?
 - Come si presenta all'utente?
- Terminazione
 - E' permesso a singoli sottosistemi di terminare?
 - I sottosistemi sono notificati se un singolo sottosistema termina?
 - Come sono comunicati gli aggiornamenti ad un database?
- Guasto
 - Come si comporta il sistema quando un nodo o un collegamento si guasta?
 - Ci sono link di collegamento di backup?
 - Come recupera il sistema da un guasto?
 - E' diverso dall'inizializzazione?

MODELLARE LE CONDIZIONI LIMITE

Le condizioni limite sono meglio rappresentate come casi d'uso con attori e oggetti.

Sono chiamati casi d'uso limite o amministrativi.

Attore: spesso l'amministratore di sistema.

- Casi d'uso interessanti
 - Avvio di un sottosistema
 - Avvio dell'intero sistema
 - Terminazione di un sottosistema
 - Errore in un componente o un sottosistema, guasto di un sottosistema o di un componente

Lez 17 → PROGETTAZIONE DEL SISTEMA ESEMPIO

ATTIVITA' DI PROGETTAZIONE DEL SISTEMA: DAGLI OGGETTI AI SOTTOSISTEMI (chiede all'orale)

La progettazione del sistema consiste nel trasformare il modello di analisi nel modello di progetto prendendo in considerazione i requisiti non funzionali descritti nel documento dei requisiti.

Illustriamo l'attività con un esempio: *MyTrip*.

- Sistema di pianificazione di percorsi stradali per automobilisti
- Partiamo con un breve modello di analisi per *MyTrip*
 - Successivamente descriviamo l'identificazione degli obiettivi di progetto e il progetto della decomposizione iniziale in sottosistemi

MODELLO DI ANALISI PER MYTRIP

Usando mytip un automobilista può pianificare il proprio viaggio dal pc di casa attraverso un servizio di pianificazione (PlanTrip). Il viaggio è salvato sul server

Nome caso d'uso	PlanTrip
Flusso di eventi	<ol style="list-style-type: none">1. Il Driver attiva il proprio computer e si logga nel servizio Web di pianificazione viaggio2. Il Driver immette i vincoli per il viaggio come sequenza di destinazioni3. Sulla base di un db di mappe, il servizio di pianificazione calcola la via più breve per visitare le destinazioni nell'ordine specificato. Il risultato è una sequenza di segmenti che legano una serie di incroci ed una lista di direzioni4. Il Driver può revisionare il viaggio aggiungendo o rimuovendo destinazioni5. Il Driver salva il viaggio pianificato per nome nel db del servizio di pianificazione per successivi accessi

EXECUTE TRIP

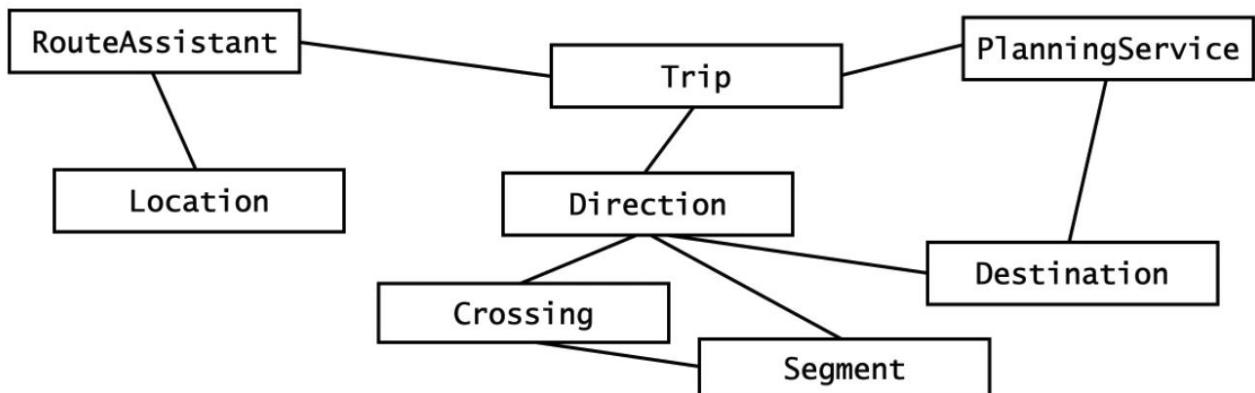
A questo punto l'automobilista inizia il viaggio mentre il pc di bordo fornisce le direzioni in base a:

- Posizione corrente
- Info di viaggio del servizio

Nome caso d'uso	ExecuteTrip
Flusso di eventi	<ol style="list-style-type: none">1. Il Driver avvia l'automobile e si logga nel sistema di bordo di assistenza al viaggio2. Avvenuto il log-in, il Driver specifica il servizio di pianificazione ed il nome del viaggio da eseguire3. L'assistente di bordo ottiene la lista delle destinazioni, le direzioni, i segmenti e gli incroci dal servizio di pianificazione4. Data la posizione attuale, l'assistente di bordo fornisce all'automobilista il successivo insieme di direzioni5. Il Driver arriva a destinazione e spegne l'assistente di bordo

MODELLO DI ANALISI PER MYTRIP

Abbiamo un diagramma delle classi dove mettiamo insieme i concetti del dominio applicativo.



Crossing	Punto geografico in cui si incontrano diversi segmenti
Destination	Rappresenta una località in cui il Driver desidera andare
Direction	Dato un Crossing ed un Segment adiacente, una Direction descrive in linguaggio naturale come guidare l'auto su un dato Segmento
Location	Posizione dell'auto nota a partire dal sistema GPS
PlanningService	Web server che suggerisce un viaggio, collegando un numero di destinazioni in forma di Crossing e Segment
RouteAssistant	Fornisce le direzioni all'automobilista, data la Location corrente e il prossimo Crossing
Segment	Rappresenta la strada tra due Crossing
Trip	Sequenza di Direction tra due Destination

REQUISITI NON FUNZIONALI PER MYTRIP

1. MyTrip è in contatto con il PlanningService via un modem wireless. Assumiamo che il modem funzioni correttamente alla destinazione iniziale
2. Una volta che il viaggio è iniziato, MyTrip dovrebbe dare le direzioni corrette anche se il modem fallisce nel mantenere una connessione con PlanningService
3. MyTrip dovrebbe minimizzare il tempo di connessione per ridurre i costi operativi
4. La ripianificazione è possibile solo se è possibile la connessione con PlanningService
5. Il PlanningService può supportare almeno 50 differenti automobilisti e 1000 viaggi

IDENTIFICARE GLI OBIETTIVI DI PROGETTO

Gli obiettivi di progetto identificano le qualità su cui il nostro sistema deve focalizzarsi.

Si possono inferire molti obiettivi dai requisiti non funzionali.

Devono essere definiti esplicitamente in modo che ogni decisione di progettazione possa avvenire in modo coerente.

GLI OBIETTIVI PER MYTRIP

Sulla base dei requisiti non funzionali, identifichiamo gli obiettivi di progetto *affidabilità e tolleranza agli errori per la perdita di connessione*.

Successivamente identifichiamo *sicurezza* poiché molti automobilisti accederanno allo stesso server.

Aggiungiamo *modificabilità* poiché vogliamo fornire la possibilità ai guidatori di selezionare un proprio servizio di pianificazione viaggi.

Ecco quindi schematicamente gli obiettivi di progetto:

- **Affidabilità**: MyTrip dovrebbe essere affidabile [**generalizzazione del requisito 2**]
- **Tolleranza agli errori**: MyTrip dovrebbe essere tollerante agli errori rispetto alla perdita della connessione con il servizio [**requisito non funzionale 2 riformulato**]
- **Sicurezza**: MyTrip dovrebbe essere sicuro, cioè, non deve consentire ad altri guidatori o utenti non autorizzati di accedere ai viaggi di un guidatore [**dedotto dal dominio applicativo**]
- **Modificabilità**: MyTrip dovrebbe essere modificabile per usare diversi servizi stradali [**anticipazione di modifiche da parte degli sviluppatori**]

OBIETTIVI DI PROGETTO

In generale, selezioniamo gli obiettivi di progetto dalla lunga lista di qualità desiderabili in cui i criteri di design sono organizzati in:

- Prestazioni
- Affidabilità
- Costi
- Manutenzione
- End user

- I criteri *Prestazioni*, *Affidabilità* ed *End user* solitamente sono **specificati nei requisiti o inferiti dal dominio applicativo**
- I criteri costi e manutenzione sono **dettati dal cliente e dal fornitore**

IDENTIFICARE I SOTTOSISTEMI

L'individuazione dei sottosistemi durante la progettazione è simile al modo di trovare gli oggetti durante l'analisi.

Ad esempio, alcune tecniche di identificazione degli oggetti di analisi come le euristiche di *Abbot* possono essere applicate all'identificazione dei sottosistemi.

La decomposizione in sottosistemi viene inoltre revisionata spesso laddove sono esaminate nuove questioni.

SOTTOSISTEMI PER MYTRIP

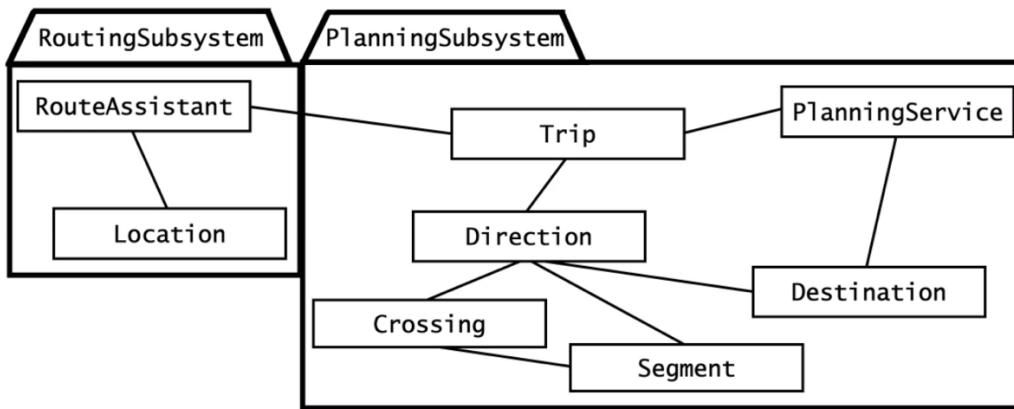
La decomposizione iniziale in sottosistemi dovrebbe essere derivata dai requisiti funzionali.

In MyTrip identifichiamo 2 gruppi principali di oggetti:

- Quelli coinvolti durante il caso d'uso PlanTrip
- ExecuteTrip.

- Le classi **Trip**, **Direction**, **Crossing**, **Segment** e **Destination** sono condivise tra i due casi d'uso. Tale insieme di classi è altamente accoppiato poiché sono usate come un tutt'uno per rappresentare un viaggio
 - Decidiamo allora di assegnarle con **PlanningService** a **PlanningSubsystem** e le restanti classi sono assegnate a **RoutingSubsystem**
 - Otteniamo così una sola associazione tra i due sottosistemi

Abbiamo quindi messo insieme le classi.



E' da osservare che tale decomposizione è un repository in cui **PlanningSubsystem** è responsabile per la struttura dati centrale

Abbiamo 2 sottosistemi che sono debolmente accoppiati perché collegati da una sola linea.

IDENTIFICAZIONE DEI SOTTOSISTEMI

Un'altra euristica per l'identificazione dei sottosistemi consiste nel mantenere insieme oggetti funzionalmente collegati.

- Un punto di partenza è assegnare gli oggetti partecipanti identificati in ciascun caso d'uso ai sottosistemi
 - Alcuni gruppi di oggetti, come il gruppo *Trip* in *MyTrip* sono condivisi e usati per comunicare le informazioni da un sottosistema ad un altro

Possiamo creare sia un nuovo sottosistema dove allocarli che assegnarli al sottosistema che li crea.

Per gestire la complessità del dominio minimizzando l'accoppiamento tra sottosistemi è possibile usare i *design pattern* che consentono di ridurre l'accoppiamento tra sottosistemi.

EURISTICHE

- Assegnare gli oggetti identificati in un caso d'uso nello stesso sottosistema.
 - Creare un sottosistema dedicato per gli oggetti usati per muovere i dati tra i sottosistemi
 - Gli oggetti nello stesso sottosistema devono esser funzionalmente legati.
-

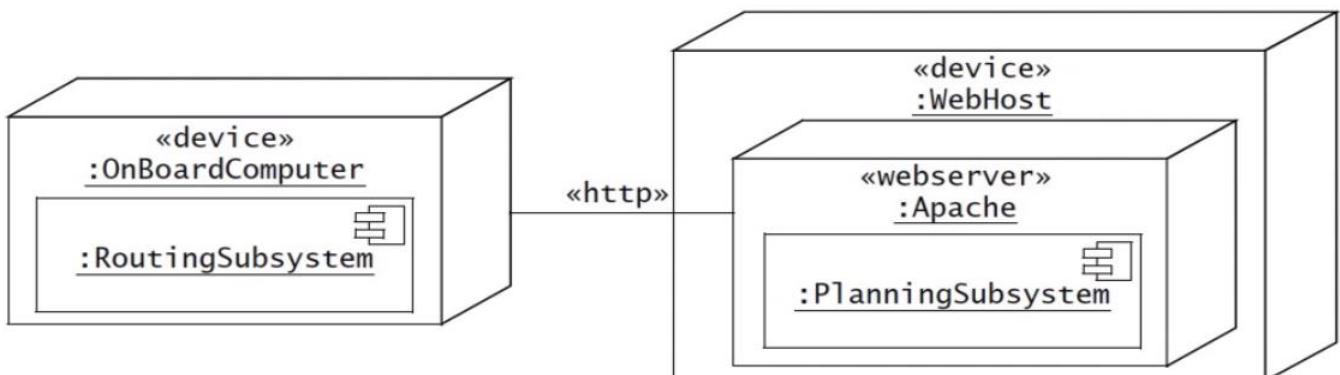
MAPPING DEI SOTTOSISTEMI A PROCESSO E COMPONENTI

L'attività di mapping dell'HW ha un impatto fondamentale su prestazioni e complessità del sistema.

In MyTrip, deduciamo che PlanningSubsystem e RoutingSubsystem sono eseguiti su due nodi diversi:

- Il primo è un servizio web su un host
- Il secondo è eseguito sul computer di bordo

La figura mostra l'allocazione dell'hw per Mytrip con due dispositivi e un ambiente di esecuzione “Apache”.



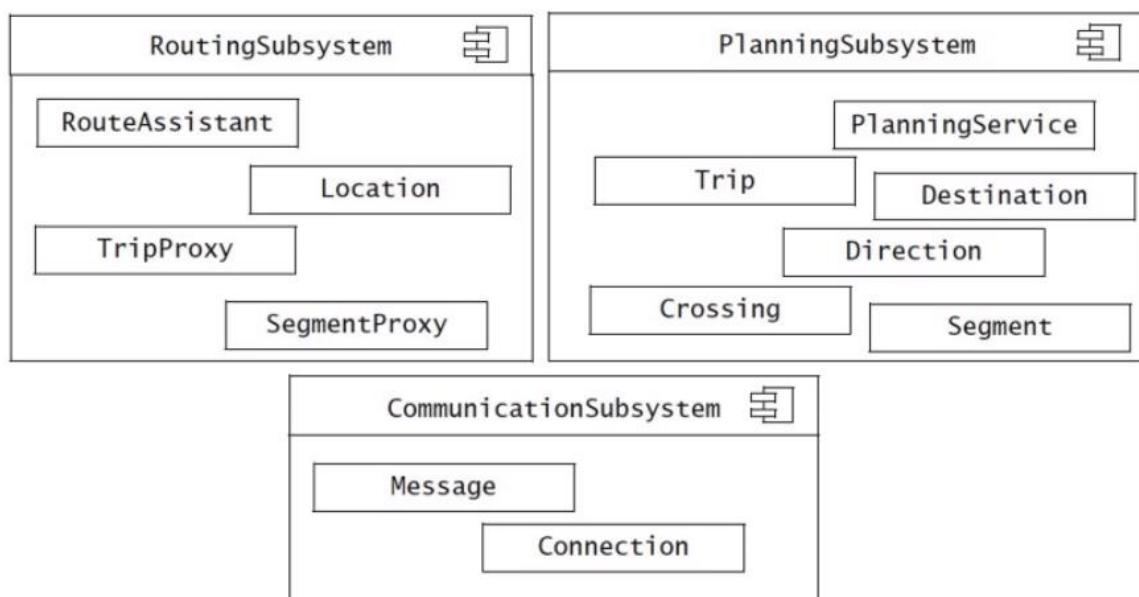
ALLOCAZIONE DI OGGETTI E SOTTOSISTEMI AI NODI

Tale fase spesso comporta l'identificazione di nuovi oggetti e sottosistemi per trasportare i dati tra i nodi.

- MyTrip

- RoutingSubs e PlanningSubs condividono gli oggetti Trip, Destination, Crossing, Segment e Direction.
- Le istanze delle classi comunicano via modem mediante un protocollo:

■ Si introduce un nuovo sottosistema per supportare la comunicazione: **CommunicationSubsystem**, localizzato su ambo i nodi per gestirne la comunicazione



CommunicationSubsystem The CommunicationSubsystem is responsible for transporting objects from the PlanningSubsystem to the RoutingSubsystem.

Connection A Connection represents an active link between the PlanningSubsystem and the RoutingSubsystem. A Connection object handles exceptional cases associated with loss of network services.

Message A Message represents a Trip and its related Destinations, Segments, Crossings, and Directions, encoded for transport.

Solo i segmenti che formano il viaggio pianificato sono memorizzati nel routingSubs.

I segmenti adiacenti che non ne fanno parte sono memorizzati solo nel PlanningSubs.

Se il guidatore ripianifica il viaggio, deve poter richiedere al CommunicationSubs di recuperare le info associate.

Per tenerne conto, abbiamo bisogno di oggetti in RoutingSubs che possano fungere da surrogati di Segment e Trip nel PlanningSubs:

- In RoutingSubsystem si introducono gli oggetti **SegmentProxy** e **TripProxy**
 - I proxy sono esempi di del design pattern Proxy
 - Infine, il CommunicationSubsystem è usato per trasferire un viaggio completo da PlanningSubsystem al RouteAssistant
-

IDENTIFICAZIONE DATI PERSISTENTI

Un altro passo è identificare i dati persistenti.

Come li gestiamo? Filesystem, db...

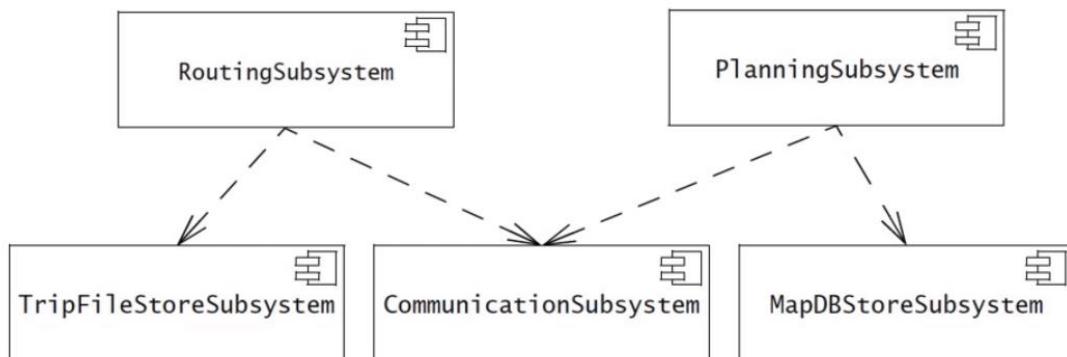
Come gestisco queste info impatta sul sistema perché se scegliamo un dbms, impatterà sulla gestione della concorrenza e del controllo dei dati persistenti: se uso un dbms non mi occupo di gestire un accesso concorrente alle info perché lo fa automaticamente, se non lo uso devo implementare la sincronizzazione.

Una decisione di questo tipo impatta ovviamente sul SW.

Nel caso di MyTrip si decidono di creare 2 categorie di dati persistenti:

- I viaggi correnti sono memorizzati in un filesystem
- I viaggi invece pianificati sono memorizzati in un db.

Vado ad aggiungere altri sottosistemi:



TripFileStoreSubsystem

The TripFileStoreSubsystem is responsible for storing trips in files on the onboard computer. Because this functionality is only used for storing trips when the car shuts down, this subsystem only supports the fast storage and loading of whole trips.

MapDBStoreSubsystem

The MapDBStoreSubsystem is responsible for storing maps and trips in a database for the PlanningSubsystem. This subsystem supports multiple concurrent Drivers and planning agents.

- Tripfilestore memorizza viaggi in file all'interno del computer di bordo.
- mapDBStore è responsabile di mappe e viaggi in un DB per il PlanningSubs.

IDENTIFICAZIONE OGGETTI PERSISTENTI

Gli oggetti entity identificati durante l'analisi sono candidati per la persistenza:

- MyTrip: Trip, Crossing, Destination, PlanningService e Segment
- N.B.: non tutti gli oggetti entity sono persistenti: Location e Direction sono ricalcolati costantemente al muoversi dell'auto

Gli oggetti persistenti non sono solo oggetti entity:

- in un sistema multiutente, le info sugli utenti sono persistenti (Driver) ma anche alcuni attributi di oggetti boundary.

In generale, gli oggetti persistenti si identificano esaminando tutte le classi che devono sopravvivere allo *shutdown* del sistema.

Che tipo di supporto di memorizzazione usare?

When should you choose flat files?

- Voluminous data (e.g., images)
- Temporary data (e.g., core file)
- Low information density (e.g., archival files, history logs)

When should you choose a relational or an object-oriented database?

- Concurrent accesses
- Access at finer levels of detail
- Multiple platforms or applications for the same data

When should you choose a relational database?

- Complex queries over attributes
- Large data set

When should you choose an object-oriented database?

- Extensive use of associations to retrieve data
- Medium-sized data set
- Irregular associations among objects

CONTROLLO ACCESSI

Durante la progettazione, modelliamo l'accesso determinando quali oggetti sono condivisi tra attori e definendo come gli attori possono controllare l'accesso.

A seconda dei requisiti del sistema possiamo definire:

- come autenticare attori nel sistema.
 - selezionare quali dati del sistema devono essere crittografati.
-

DECISIONI SUL FLUSSO DI CONTROLLO GLOBALE

- flusso di controllo: sequenza di azioni nel sistema.
- Nei sistemi OO, la sequenza delle azioni comporta decidere quali operazioni eseguire e in quale ordine.

Esempio di controllo guidato da procedura

Le operazioni attendono gli input se hanno bisogno di dati da un attore.

```
Stream in, out;
String userid, passwd;
/* Initialization omitted */
out.println("Login:");
in.readln(userid);
out.println("Password:");
in.readln(passwd);
if (!security.check(userid, passwd)) {
    out.println("Login failed.");
    system.exit(-1);
```

Esempio di controllo guidato da eventi

- Si usa un loop dove si intercettano gli eventi che si possono verificare e il codice deve distribuire l'evento all'oggetto associato.

```
Iterator subscribers, eventStream;  
Subscriber subscriber;  
Event event;  
EventStream eventStream;  
/* ... */  
while (eventStream.hasNext()) {  
    event = eventStream.next();  
    subscribers = dispatchInfo.getSubscribers(event);  
    while (subscribers.hasNext()) {  
        subscriber = subscribers.next();  
        subscriber.process(event);  
    }  
}
```

Un evento è preso da eventstream ed inviato agli oggetti interessati

DECISIONI SUL FLUSSO DI CONTROLLO GLOBALE

Fissato il meccanismo del flusso di controllo lo si realizza con un insieme di uno o più oggetti di controllo.

- Gli oggetti di controllo
 - registrano eventi esterni
 - memorizzano stati temporanei ad essi relativi e
 - determinano l'invio dell'ordine corretto di chiamate ad operazioni su oggetti boundary e entity associate con l'evento esterno
- Localizzare il flusso di controllo per un caso d'uso in un singolo oggetto
 - codice più leggibile
 - Sistema resiliente ai cambiamenti nelle implementazione del flusso di controllo

IDENTIFICAZIONE SERVIZI

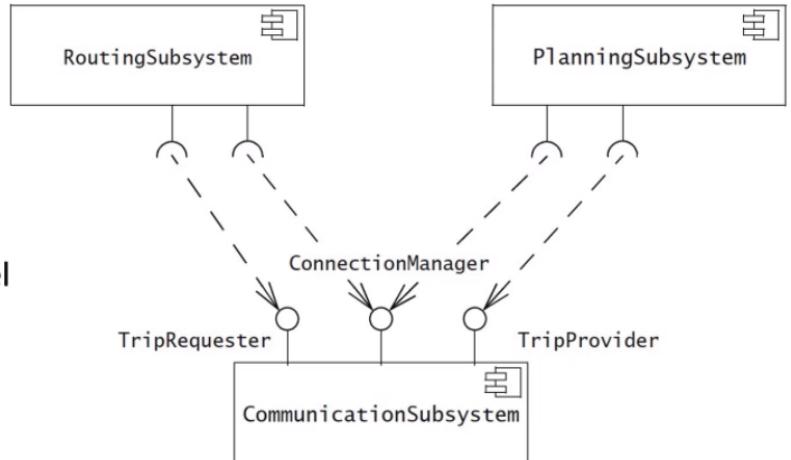
Finora abbiamo esaminato le decisioni di progettazione chiave che impattano sulla decomposizione.

Abbiamo identificato i sottosistemi.

Ora dobbiamo rifinire la decomposizione del sottosistema identificando i servizi forniti da ciascuno.

Esempio per mytrip:

- La responsabilità di CommunicationSubsystem è di trasportare i viaggi dal PlanningSubsystem a RoutingSubsystem
- RoutingSubsystem inizia la connessione poiché PlanningSubsystem è il server e sempre disponibile, mentre Routing solo quando la macchina è accesa
- **ConnectionManager**: permette ad un sottosistema di registrarsi con CommunicationSubsystem, autenticarsi, trovare altri nodi e iniziare e chiudere la connessione
- **TripRequester**: permette ad un sottosistema di richiedere una lista di viaggi disponibili e fare il download del viaggio selezionato
- **TripProvider**: permette ad un sottosistema di fornire una lista di viaggi disponibili per lo specifico guidatore e rispondere a specifiche richieste di viaggio



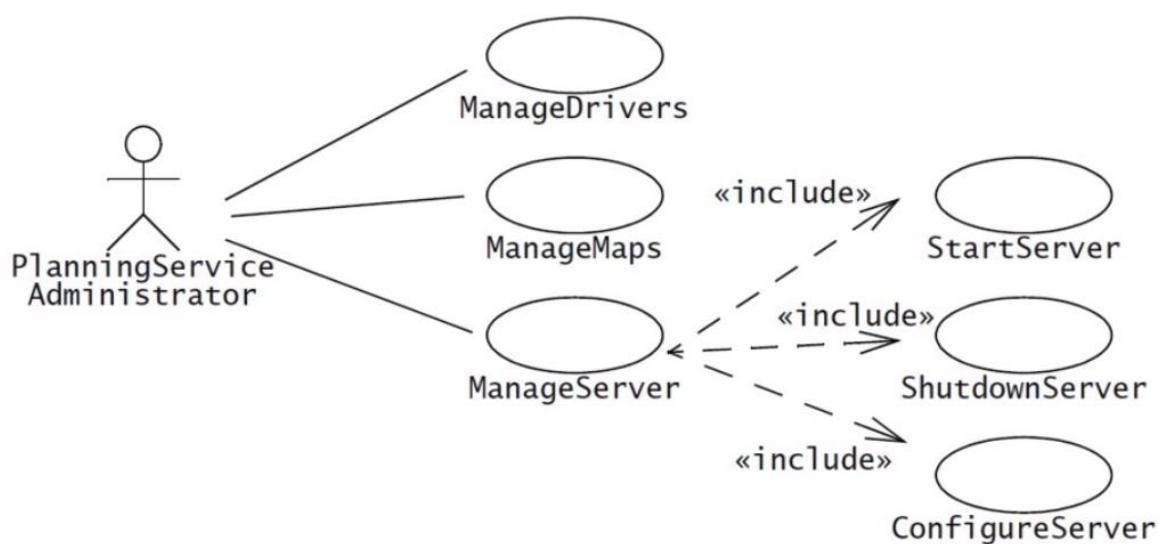
IDENTIFICAZIONE DELLE CONDIZIONI LIMITE

Le condizioni limite sono:

- Avvio
- Spegnimento
- Inizializzazione
- Gestione guasti

Si introducono creando dei *boundary use case*.

Per MyTrip aggiungiamo casi d'uso boundary:



- **ManageDrivers**: per aggiungere, rimuovere e modificare i guidatori
- **ManageMaps**: per aggiungere, rimuovere e modificare le mappe per generare i viaggi
- **ManageServer**: per eseguire routine di configurazione, avvio, spegnimento

Lez 18 → TESTING DEL SW

COSA STUDIEREMO DEL TESTING

- Terminologia: guasto, errore, difetto.
- Attività di Test:
 - Test dell’unità: testing di singole componenti.
 - Test di integrazione: unione delle componenti e test.
 - Test del sistema: test dell’intero sistema.

Il testing serve per valutare se il sistema rispetta le specifiche ed è ben funzionante.

I BUG

Nella storia dell’informatica ci sono molti casi di sistemi con comportamenti che deviano dalle specifiche:

- Primo caso di bug della storia:
 - Una falena (bug) entrò nel Pannello F del pc di Havard, creando un cortocircuito.

- La falena fu presa (de-bug) ed attaccata sul logbook del pc con dello scotch, con la scritta “primo caso di bug trovato”.
-

TERMINOLOGIA

- Guasto: qualsiasi differenza del comportamento osservato rispetto a quelli attesi.
 - Stato d'errore: il sistema è in uno stato tale che ulteriori elaborazioni fatte portano a un guasto.
 - Difetto: la causa di un errore a livello algoritmico (uscita da un loop o cose simili) o meccanico.
 - o Causato da problemi di comunicazione tra team.
 - o Implementazione errata di una specifica di uno dei team.
 - Validazione: attività di controllo delle differenze tra comportamento osservato di un sistema e la sua specifica.
-

ESEMPI DI DIFETTI ED ERRORI

- **Difetti nella specifica dell'interfaccia**
 - Differenza tra ciò che il client ha bisogno ed il server offre
 - Differenza tra requisiti ed implementazione
- **Difetti algoritmici**
 - Mancanza di inizializzazione
 - Condizione di branching sbagliata
 - Mancanza del test su null
- **Difetti meccanici**
 - Temperatura di funzionamento al di fuori delle specifiche dell'apparecchiatura
- **Errori**
 - Input utente sbagliato
 - Errori di riferimento a null
 - Errori di concorrenza
 - Eccezioni

COME GESTIAMO ERRORI, GUASTI E DIFETTI?

Ci sono diversi approcci:

- Ridondanza modulare: tollerare guasti ed errori (anche in presenza di errori, il sistema ha precauzioni che permettono di funzionare).
- Patching: modificare il sistema in modo che possa comportarsi correttamente lo stesso.
- Testing: individuare difetti ed errori e correggerli.

Il testing è un insieme di tecniche che individuano e coraggono un difetto.

OSSERVAZIONI

È impossibile testare completamente qualsiasi sistema:

- Limitazioni pratiche: testing completo proibitivo per tempi e costi.
- Limitazioni teoriche.

Un testing a buon fine vuol dire che ha trovato degli errori.

Il tesing non è gratis.

IL TESTING RICHIENDE CREATIVITA'

Per sviluppare un test efficace, bisogna avere:

- Una comprensione del sistema
- Conoscenza dei domini dell'applicazione
- Conoscenza delle tecniche di testing e abilità nell'applicarle.

Il testing è realizzato meglio da chi non ha sviluppato il sistema.

ATTIVITA' DI TEST

TIPI DI TEST

➤ Testing delle unità:

- Sono testati singoli componenti
- Realizzato dagli sviluppatori
- L'obiettivo è che la componente funziona.

➤ Test integrazione:

- Sono testati gruppi di sottosistemi e poi l'intero sistema.
- Realizzato da sviluppatori.
- L'obiettivo è test delle interfacce tra i sottosistemi.

➤ Testing del sistema

- Test dell'intero sistema.
- Realizzato da sviluppatori.
- Obiettivo è determinare se il sistema soddisfa i requisiti.

➤ Testing di accettazione

- Fatto dal cliente.
 - Valuta il prodotto consegnato al cliente.
 - Obiettivo è dimostrare che il sistema soddisfa i requisiti richiesti.
-

QUANDO SI PREPARA UN TEST?

- Tradizionalmente dopo la stesura del codice.
- Nella programmazione estrema (XP) si prevede che il test avvenga prima della stesura del codice. Lo sviluppo è guidato dai test.

TERMINOLOGIA

- Un componente di test è una parte del sistema che può essere isolato a scopo di test.
- Un caso di test è un insieme di input e risultati attesi con lo scopo di causare guasti.

- Uno stub di test è un'implementazione parziale dei componenti da cui dipende il componente testato.
 - Un driver di test è un'implementazione parziale di un componente che dipende dal componente di test.
 - Una correzione è una modifica a un componente per riparare un difetto.
-

TESTING DELLE UNITÀ'

Abbiamo due possibilità:

- Testing statico: il sistema non va in run e prevede analisi statiche e delle revisioni.
- Testing dinamico: a runtime, viene effettuato un testing *black-box o white-box*.

ANALISI STATICÀ

Viene effettuata una lettura del codice sorgente e viene ispezionato.

Viene poi effettuato un *walk-through*, cioè una presentazione informale agli altri.

Osservazioni: l'analisi statica determina errori ma alcuni non interessano. L'analisi statica riesce solo a determinare il 10-15% degli errori del codice.

ANALISI DINAMICA

- Testing black-box: test del comportamento input/output.
 - Testing white-box: test della logica interna del sottosistema
 - Test basato su strutture dati: i tipi di dati determinano i casi di test.
-

TESTING BLACK_BOX

- Focus: comportamento I/O
 - Se per un qualsiasi input dato, possiamo predire l'output, allora l'unità passa il test.
 - Richiede un *oracolo* di test.
 - È quasi impossibile generare tutti i possibili input.

L'obiettivo è ridurre il numero di casi di test con le partizioni di equivalenza:

- Significa suddividere gli input in classi di equivalenza, cioè classi che dovrebbero determinare un comportamento simile. Basta quindi selezionare solo alcuni input di quelle classi di equivalenza per evitare di inserire infiniti input simili.
-

Lez 19 → Testing del software pt.2

TESTING WITHE BOX

- Focus: completezza. Ogni istruzione del componente è eseguita almeno una volta.

Ci sono 4 tipi di testing w-b:

- Testing di istruzione.
- Testing del ciclo.
- Testing del percorso.
- Testing delle condizioni.

- **Testing di istruzione**

- Testa ogni istruzione

- **Testing di ciclo**

- Ciclo da eseguire esattamente una volta
 - Ciclo da eseguire più di una volta
 - Salto completo della condizione del ciclo

- **Testing di percorso**

- Assicura che tutti i percorsi del programma siano eseguiti

- **Testing condizionale**

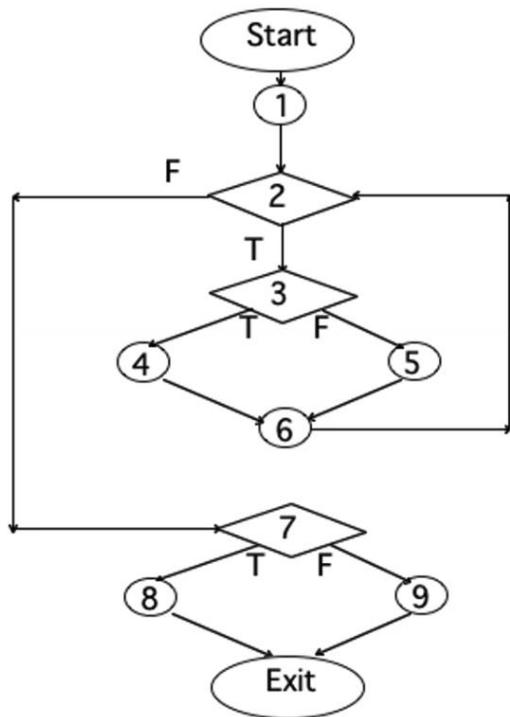
- Assicura che ogni esito in una condizione sia testato almeno una volta

ESEMPIO DI TESTING CONDIZIONALE

```
if ( i = TRUE) printf("Yes");else printf("No");
```

- Abbiamo bisogno di due casi di test con i seguenti dati di input
 1. i = TRUE
 2. i= FALSE

COSTRUIRE IL DIAGRAMMA DEL FLUSSO LOGICO



Tutti i percorsi che si possono avere nel caso in cui le condizioni nel blocco logico sono vere o false.

CASI DI TEST

- Caso di Test1 : do un valore col quale faccio una sola esecuzione.
- Test2: salto il ciclo.
- Test3: esegue il ciclo più volte.

Così facendo ho completato il test.

EURISTICHE PER TEST DELLE UNITÀ'

1. Creare i test di unità quando è completato il progetto degli oggetti
 - Test black-box: test del modello funzionale
 - Test white-box: test del modello dinamico
2. Sviluppare i casi di test
 - Obiettivo: trovare un numero efficace di casi di test
3. Cross-check dei casi di test per eliminare duplicati
 - Non sprecare tempo!
4. Controllare a tavolino il proprio codice
 - Talvolta riduce il tempo per il testing
5. Creare un impianto (fixture) di test
 - I driver del test e gli stub sono necessari per il testing di integrazione
6. Descrivere l'oracolo del test
 - Spesso il risultato del primo test eseguito con successo
7. Eseguire i casi di test
 - Rieseguire il test quando è fatta una modifica (**test di regressione**)
8. Confrontare i risultati del test con l'oracolo del test
 - Automatizzarlo se possibile

JUNIT

È un framework Java per la scrittura ed esecuzione di test di unità.

Organizza i casi di test sfruttando come impianto un design pattern chiamato *Composite*.

Confronto dei testing white e black-box

• Testing white-box

- Deve essere testato un numero potenzialmente infinito di percorsi
- Spesso testa cosa è fatto, invece di cosa dovrebbe essere fatto
- Non può individuare casi d'uso mancanti

• Testing black-box

- Potenziale esplosione combinatoria dei casi di test (dati validi e non)
- Spesso non chiaro se i casi di test selezionati scoprono un errore particolare

• Necessari ambo i tipi di testing

- Testing White e black-box sono gli estremi di uno stesso spettro continuo di testing
- Qualsiasi scelta di caso di test si trova nel mezzo e dipende da:
 - Numero di possibili percorsi logici
 - Natura dei dati di input
 - Quantità di computazione
 - Complessità di algoritmi e strutture dati

Dopo il test delle unità si passa al testing di *integrazione*.

TEST INTEGRAZIONE

L'intero sistema è visto come una collezione di sottosistemi (insiemi di classi) determinati durante la progettazione del sistema e degli oggetti

- Obiettivo: testare tutte le interfacce tra i sottosistemi e l'interazione dei sottosistemi
- La strategia di integrazione del testing determina l'ordine in cui i sottosistemi sono selezionati per il testing e l'integrazione

I test delle unità valutano la singola unità in isolamento.

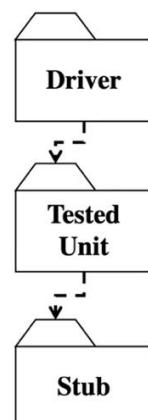
Molti guasti derivano da difetti nell'interazione tra sottosistemi.

Spesso sono usati molti componenti che non possono essere testati come unità e senza il testing d'integrazione, il test del sistema sarebbe troppo costoso in termini di tempo.

I guasti che non sono scoperti nel test di integrazione saranno scoperti dopo che il sistema è stato consegnato e ciò comporterebbe enormi costi.

STUB E DRIVER

- Driver: un componente che invoca la TestedUnit
 - Controlla i casi di test
- STUB: un componente da cui TestedUnit dipende, restituisce valori fintizi.



APPROCCIO BIG BANG

Prendo le componenti (sottosistemi), li testo singolarmente e li metto tutti insieme.

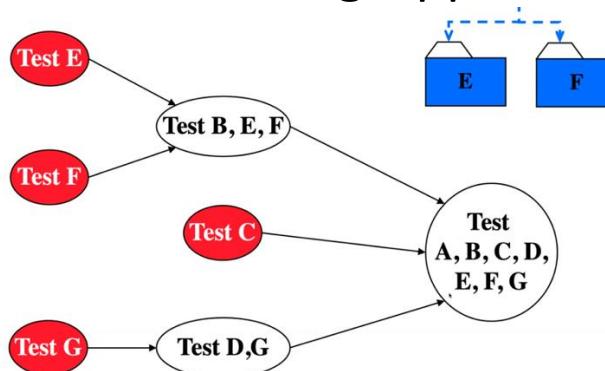
Un problema di questo approccio è che dal momento in cui unisco tutto e si presenta un errore, è complicato individuarne l'origine.

Il vantaggio è che *stub e driver* non servono.

STRATEGIA TESTING BOTTOM UP

- I sottosistemi nello strato più basso della gerarchia di chiamate sono testati individualmente
- Successivamente sono integrati i successivi sottosistemi che chiamano i sottosistemi testati precedentemente
- Ciò viene ripetuto fino a che sono inclusi tutti i sottosistemi
- Sono necessari i **driver**

Testo il singolo elemento del livello più basso, testo poi quello del livello superiore e testo in un insieme il padre coi figli, alla fine testo il gruppo completo.



Pro:

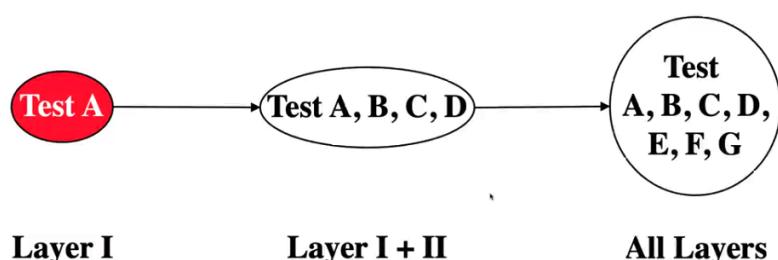
- Non sono necessari stub.
- Errori più facilmente rilevabili.
- Utili per sistemi OO, real-time e con requisiti stretti.

Conto:

- Necessari i driver
 - Testa per ultimo i sottosistemi più importanti.
-

STRATEGIA TOP DOWN

Parto dall'alto, testo poi i livelli minori in gruppo.



Pro:

- Non sono necessari driver

Contro:

- Necessari stub
-

STRATEGIA DI TEST A SANDWICH

Combina la strategia top-down e bottom-up.

Il sistema è visto come un sistema a 3 strati:

- Uno strato target nel mezzo.
- Uno sotto al di sopra del target.
- Uno strato al di sotto del target.

Non tutte le unità potrebbero essere testate.

Pro:

- Test degli strati fatti in parallelo
- Stub e driver non sono necessari per strati top e bottom

Contro:

- Non testa i singoli sottoinsiemi.
- Soluzione: sandwich modificata.

- Testing sandwich prevede una variante: prima della fase sandwich vengono testate tutte le unità singolarmente.
-

TESTING DEL SISTEMA

- Testing funzionale che valida i requisiti funzionali.
- Testing delle prestazioni valida requisiti non funzionali.
- Testing di accettazione eseguito dai clienti.

TESTING FUNZIONALE

- Obiettivo: testare le funzioni del sistema.
 - I casi di test derivano dai diagramma dei casi d'uso.
-

TESTING DELLE PRESTAZIONI

Sovraccarico il sistema per vedere il comportamento.

Cercare ordini di esecuzione insoliti.

Controlla la risposta del sistema rispetto a grandi quantità di dati.

TESTING DI ACCETTAZIONE

L'obiettivo è dimostrare che il sistema è pronto all'uso.

Il cliente fa il test con lo sviluppatore (alpha test) oppure da solo (beta test).

IL TESTING HA MOLTE ATTIVITA'

- Stabilire gli obiettivi del test
- Progettare i casi di test
- Scrivere i casi di test
- Testare i casi di test
- Eseguire i test
- Valutare i risultati del test
- Cambiare il sistema
- Fare il testing di regressione