Lezione 4 15/10/2024

Problem architecture

Introduzione

Una **descrizione architetturale** deve:

- cogliere gli aspetti funzionali e le proprietà di qualità del sistema
- essere comprensibile e di valore a tutte le parti interessate (stakeholder)

L'architettura di un sistema complesso può essere descritta in modo molto più efficace tramite un insieme di **viste** separate (ma correlate) e dove ciascuna delle quali descrive un aspetto diverso dell'architettura.

Useremo alcuni formalismi UML (possiamo usarne uno qualsiasi, anche uno inventato da noi, finchè siamo coerenti). Strumento: Visual Paradigm o qualcosa di free

- https://www.visual-paradigm.com/download/community.jsp

o draw.io

è importante mantenere una **consistenza** tra le viste.

Noi vedremo 3 punti di vista diversi:

- **Problem architecture**: è il punto di vista dell'**utente finale**. Dobbiamo capire quali sono i **dati** che il sistema deve trattare, quali sono i **flussi informativi** (ovvero come l'informazione viene trasformata per raggiungere quello che l'utente finale si aspetta) e quali sono i **flussi di controllo** (perchè certe elaborazioni devono avere luogo). Ci sarà anche l'identificazione dei requisiti funzionali e non funzionali (qualità).
- **Logical architecture**: l'obiettivo è quello di individuare il modo migliore per partizionare le attività del sistema in **componenti**. Ovvero è l'assegnazione delle responsabilità ai **moduli**. è importante considerare la **modificabilità**, suddividendo per bene in moduli. Quindi vado ad individuare i componenti, i

moduli, le responsabilità, e vado a fare una **valutazione comparativa** tra le scelte.

- **Concrete architecture**: una volta che ho deciso i componenti, vado a definire in modo chiaro le loro **interfacce**. Questo diventa il modo standard per assicurare il fatto che i componenti siano ben separati, non si sa come facciano le cose, ma offrono ed usano servizi.
- **Deployment architecture**: (non la facciamo) si va a decidere ciascun componente (dal punto di vista della loro esecuzione) su che hardware girano.

Problem architecture

L'**architettura del problema** deve definire:

1. le **funzionalità** che il sistema deve offrire agli **attori esterni** (fruitori di funzionalità del sistema e/o entità che supportano funzionalità del sistema, bisogna identificare anche questi). Si può anche andare a definire quali attori ci forniranno servizi (per esempio firebase per autenticazione).
    a. Per le viste, useremo i **casi d'uso UML**.
2. i **tipi di informazioni** che il **sistema deve gestire** per la realizzazione delle funzionalità (viene considerato anche quanto un'informazione deve essere accurata e immediata)
    a. Per le viste, useremo i **diagrammi delle classi UML**. Questa è solo una rappresentazione, la semantica gliela diamo noi. Ci sono solo dei costrutti, dei building block, che vanno messi in relazione. Le relazioni hanno una semantica. Però il cosa rappresenta una classe è una cosa scelgo io.
3. i **flussi informativi** che il sistema deve supportare per realizzare le funzionalità definite al punto 1 (attraverso use case) operando sui dati definiti al punto 2 (in/out)
    a. Per le viste, useremo i **diagrammi delle attività UML**.
4. I **flussi di controllo** che il sistema deve supportare per garantire le funzionalità.
    a. Per le viste, useremo i **diagrammi delle attività UML**.

Qui va catturata la situazione dove tutto va come dovrebbe andare, non bisogna definire il "l'utente inserisce la password sbagliata, allora..." ci saranno altre viste

che modellano cosa fare nel momento in cui il sistema riscontra anomalie.

Ci stiamo mettendo nel punto di vista dell'utente finale, devo rispondere a domande del tipo:

- **Who**: chi sono gli **attori** che fanno parte del sistema
- **What:** quali sono i **dati** (le entità) e le **funzionalità** del sistema
- **Where:** dove sono i gli attori e le informazioni (per esempio se un attore è seduto al terminale o si muove con un tablet).

Quindi quali sono gli attori, le funzionalità e le informazioni (diagramma delle classi e diagramma degli use case).

- **How**: come si fa a manipolare l'informazione affinché il sistema faccia quello che è richiesto faccia. Quali **attività** deve svolgere il sistema per trasformare le informazioni.
- **Why:** perché le **attività** vengono attivate. Quali sono i meccanismi affinché certe attività vengono innescate.
- **When:** quando le **attività** vengono eseguite (frequenza).

Quindi come si manipola l'informazione, perché e quando (diagramma delle attività).

Esempio

Catena di supermercati

- Diverse sedi sparse nel territorio
- Ogni sede ha un magazzino locale
- Sede centrale con magazzino
- Esiste un servizio che trasferisce merci dal magazzino centrale a quello locale

Obiettivo: gestire la catena di approvvigionamento alle sedi locali per fare in modo che localmente gli scaffali non si svuotino.

Per soddisfare l'obiettivo occorre che il sistema sia in grado di:

- Avvertire l’addetto della sede locale che deve riempire lo scaffale con la merce nel magazzino locale.
- Avvertire il responsabile della sede locale di quali sono le esigenze quotidiane di approvvigionamento
- Avvertire la sede centrale che occorre fare una pianificazione degli acquisti

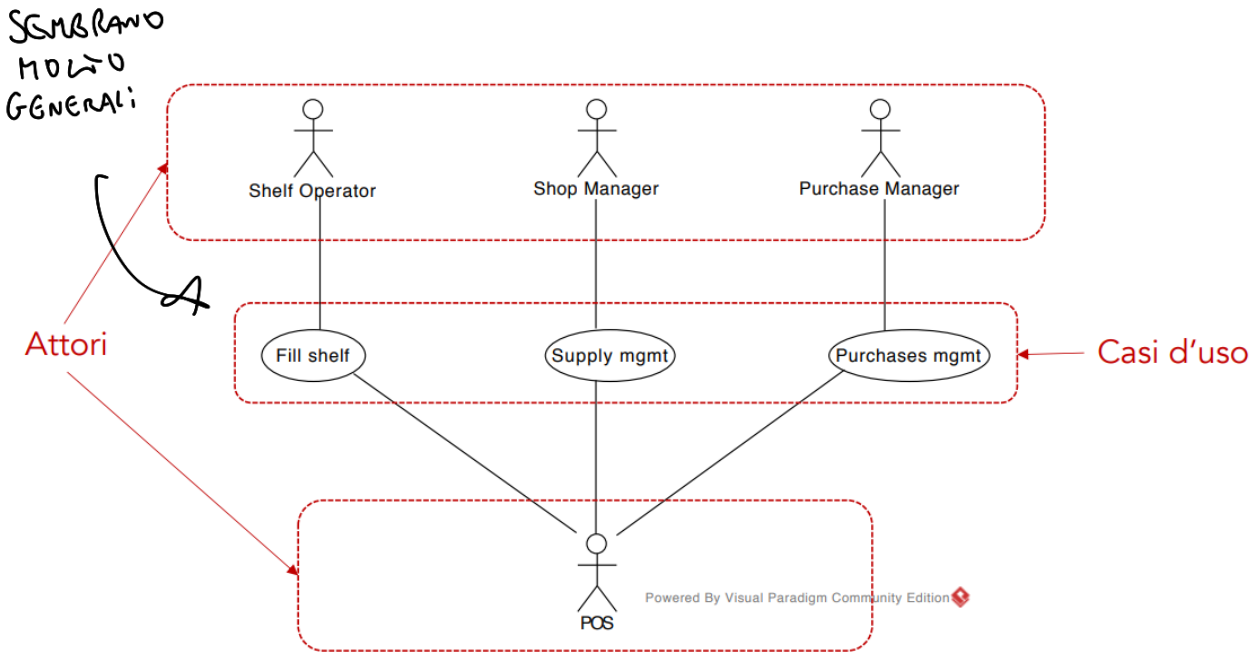
Bisogna quindi fare delle **assunzioni**:

- Esiste una sorgente di informazione (POS) che rileva cosa si sta vendendo
- Ci sono 10 Shop
- Ogni Shop ha 10 POS
- Esistono fino a 1000 Prodotti diversi venduti
- Ogni POS genera una vendita ogni 2 secondi
- Ogni scaffale contiene un solo tipo di prodotto (semplificazione)
- Ogni scaffale per 5 ore ha merce sufficiente
- I magazzini locali sono riforniti ogni giorno
- Ogni magazzino ha prodotti sufficienti per 5 giorni di attività
- Le richieste di rifornimento sono concentrate nei 30 minuti dopo la chiusura

Who

abbiamo 3 attività:

- riempi lo scaffale, assegnato ad un addetto
- gestisci l’approvvigionamento locale
- gestisci l’approvvigionamento centrale



Il POS è un'entità **esterna al sistema** (come tutti gli **attori**, è qualcosa già realizzato da qualcun altro). I boundaries tra sistema e attori sono importanti per capire cosa devo realizzare. Quindi non esiste l'attore sistema. Può esistere un caso d'uso che non si collega ad un attore, per esempio perché è triggerato in automatico, ma invece si collegherà con una freccia all'attore su cui ha effetto.

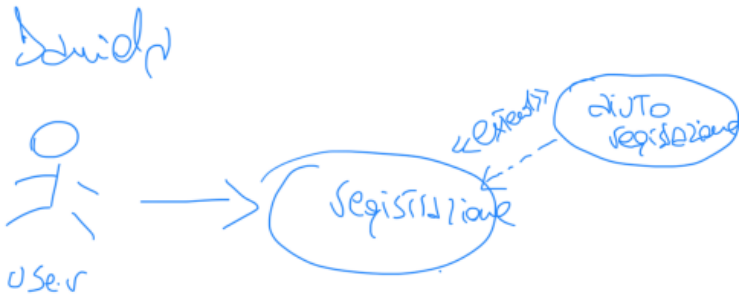
Se metto una freccia, con navigabilità, da user a registrazione, sto dicendo che l'user usa la registrazione.

Se non metto la freccia ma lascio solo la linea, non so se è push o pull. Mettendo la freccia lo rendo esplicito. Se la freccia è verso il sistema è **push**, se è verso l'attore è **pull** (per esempio attore accelerometro e use case acquisizione, va ad acquisire un dato quando gli serve).

Potrei avere anche l'attore "utente registrato" può fare la registrazione. L'utente autenticato può fare l'acquisto. La prof non vuole che facciamo "utente registrato" fa "acquista" che include (freccia tratteggiata) il collegamento ad autenticazione. Perchè vuol dire che ogni volta che ogni volta che acquista, gli viene chiesto l'autenticazione, che è sbagliato. La soluzione migliore è quella di avere l'utente registrato che si autentica, e poi l'utente autenticato che acquista, vede il suo saldo punti...

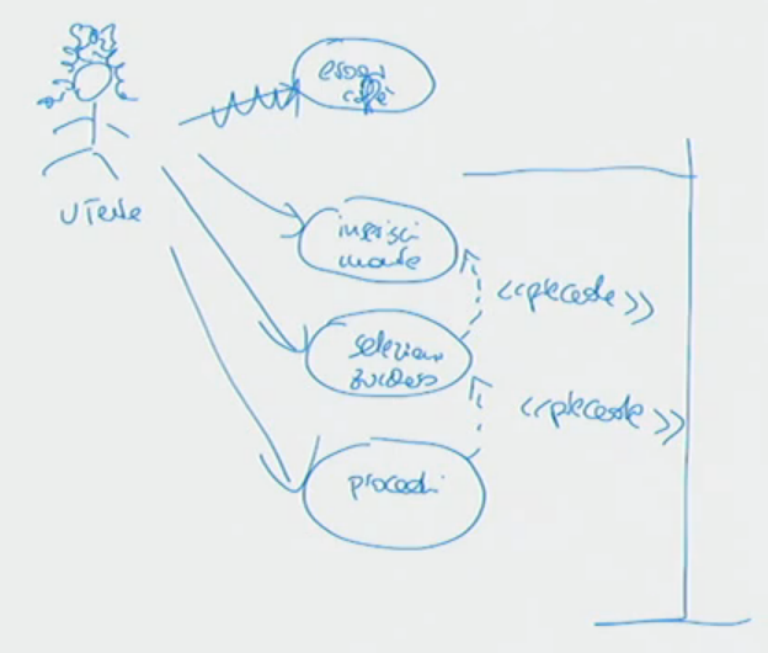
Extension point:

Potrei avere l'utente che vuole registrarsi, e l'estensione è che l'utente richiede l'aiuto registrazione.



ha senso farlo per esempio l'estensione a password dimenticata nello use case registrazione.

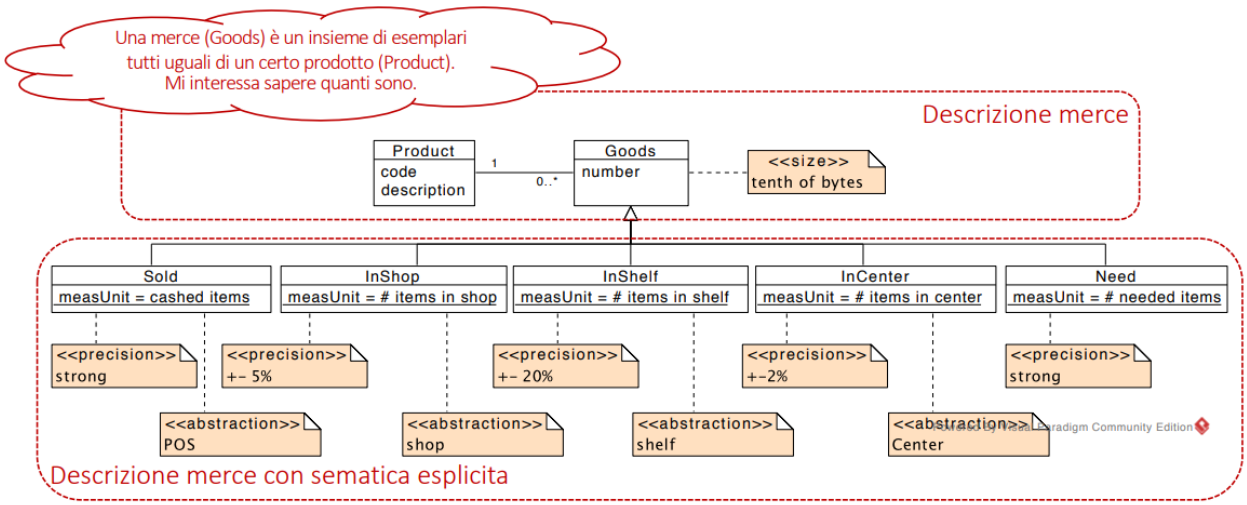
Lo use case NON deve descrivere come il caso d'uso deve essere eseguito, quindi questa sequenza di "precede" nella procedura di fare un caffè non ha senso qui. Ha senso farla poi in un information flow. Le precedenze non devono comparire nel diagramma use case.



Tornando all'immagine con i collegamenti in rosso: avrebbe senso fare una freccia degli use case verso i 3 attori sopra, visto che gli attori vengono risvegliati.

What

Il pezzo cerchiato di rosso mi dice i dati che il sistema deve gestire.



Decido di rappresentare la merce avendo un concetto di prodotto che ha un codice e una descrizione, con collegata la quantità di questo prodotto come numero.

Con questo tipo di modellazione mi perdo un'informazione, ovvero qual è la specifica istanza del prodotto che abbiamo comprato (dipende se ci interessa). Per esempio se è un prodotto elettronico, ognuno ha un numero seriale univoco, quindi mi interessa esattamente quale è stato venduto, per esempio per attivare la garanzia. Se invece sto vendendo bottigliette d'acqua non mi interessa.

Nella sezione rossa sotto: ho una modellazione che va molto nel dettaglio. La notazione UML dice che questi sono casi particolari, sono sottoclassi. Ovvero, rispetto a questa vista che sto trattando, mi interessa andare nel dettaglio e dire che ci sono 5 tipi di goods. A livello implementativo avrò sempre il concetto di goods, ma il tipo diverso dipenderà da dove è salvato e quindi non avrò questa differenziazione dei dati. Però a livello di analisi qui è importante.

Può essere importante andare a specificare l'unità di misura dei dati.

Mi può interessare sapere anche se devo trasportare l'informazione, quanto mi pesa (quel "tenth of bytes").

La precision descrive quanto deve essere preciso questo dato. Per esempio l'inshelf ha precisione più bassa perchè tra quando un prodotto è stato preso dallo

scaffale, e quando viene acquistato, può passare del tempo, quindi l'aggiornamento è più lento.

aggiunte dall'inizio della prossima lezione:

L'immagine è un formalismo usato per rappresentare un'informazione. Ciascuna classe modella un dato, non è un grafico implementativo, quindi non ha alcun tipo di informazione relativa alla visibilità (privato/pubblico). Qui serve soltanto cercare di descrivere l'informazione, il dato. Qui non serve neanche l'ID. Qui si parla di "datastore" in generale, non di "database".