Lezione 13 3/12/2024

Concrete Architecture

L'output saranno le funzionalità del sistema. Ho come obiettivo assegnare le responsabilità ai componenti logici

L'architettura concreta ha come obiettivo definire:

- **come i flussi informativi e di controllo sono concretizzati** in termini di **stili di interazione**
  - meccanismi di comunicazione (control flows)
  - trasferimento dell'informazione (information flows)
- **il design interno dei componenti**
  - Attivi vs passivi
  - Stateless vs stateful (se ha stato deve tenere traccia della sua evoluzione)

L'**architettura concreta** specifica l'insieme dei componenti concreti mostrando il design interno dei componenti e le interazioni fra questi.

Stili di interazione

Sappiamo che ci sono 2 tipi di comunicazione:

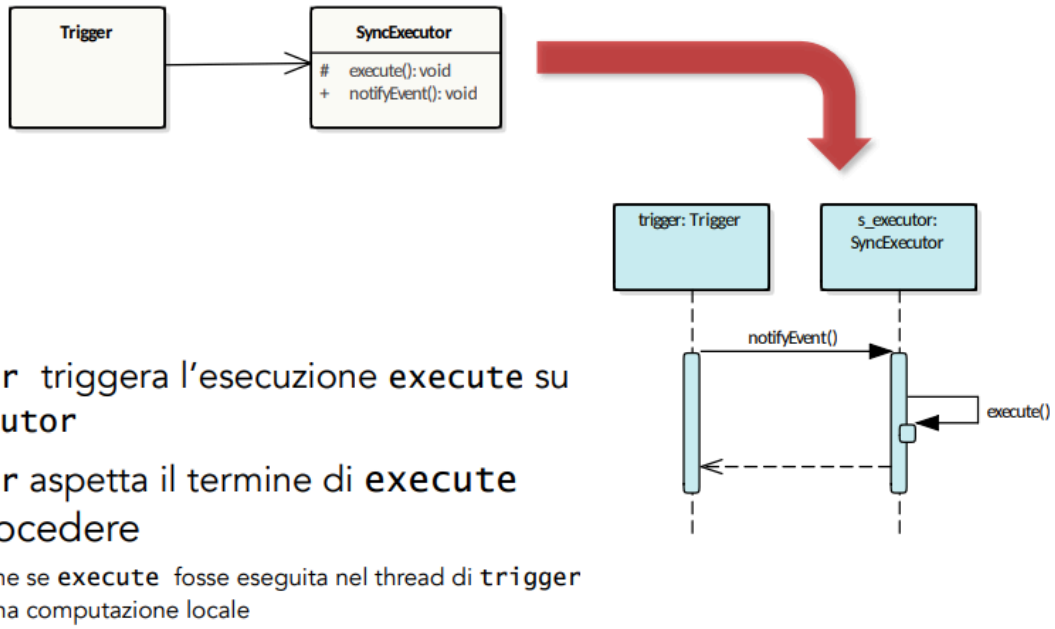
- **sincrona**: le parti della comunicazione ascoltano in maniera continuativa, e reagiscono in base alla risposta che ricevono dalla controparte. Un componente rimane **inattivo** finché non riceve una chiamata o una risposta.
- **asincrona**: le parti non ascoltano attivamente, ma "lascio un messaggio, e mi ritiro, non aspetto una risposta". Permette al componente di **continuare a funzionare** dopo aver generato una chiamata o una risposta.

Control flow

Questi sono segnali di controllo (non di dati), come nei diagrammi di attività avevamo separato il flusso dati e quello di controllo, che non per forza coincidono.

Due stili di base:

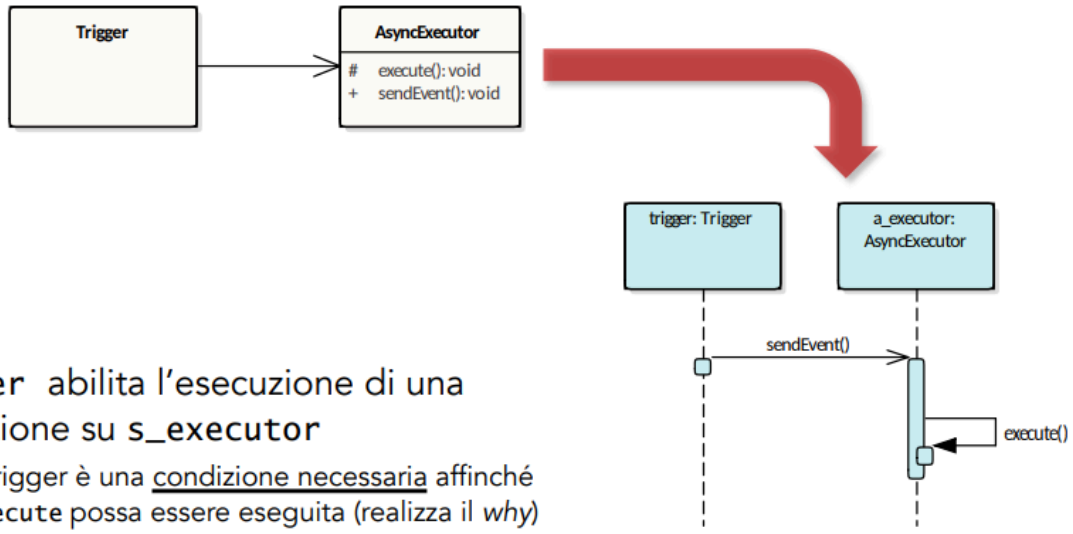
- Sincrono: le parti sono in attesa di ricevere eventi e reagiscono di conseguenza
- Asincrono: le parti non sono in attesa di eventi



L'execute incapsula tutta la logica di dominio. Ci interessa definire le interfacce, non quello che c'è all'interno. è "protetta" perchè è interna, poteva anche essere messa privata, perchè al resto del mondo non si vede.

Nell'immagine sopra è un sincrono. Abbiamo una freccia piena in notifyEvent() perchè ad un certo punto c'è un trigger dell'evento. è sincrono perchè manda un segnale, ma deve aspettare una risposta (quindi devo ritornare). Dopo aver ricevuto la notifica, ad un certo punto (per politiche proprie) eseguirà la responsabilità di quel componente.

La notify va a specificare il momento in cui sei abilitato a fare l'azione (why), però il when viene controllato dal componente.

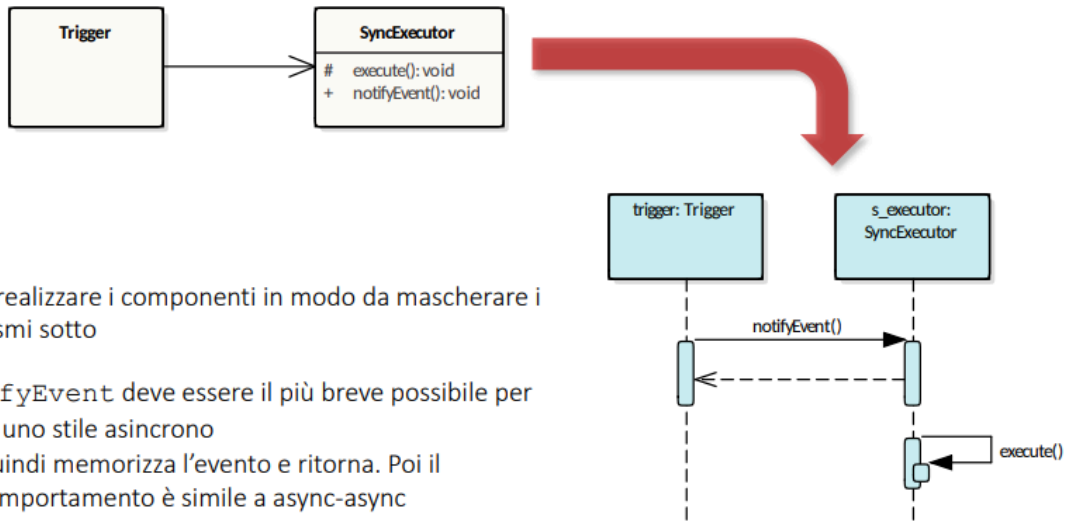


- `trigger` abilita l'esecuzione di una operazione su `s_executor`
  - Il trigger è una condizione necessaria affinché `execute` possa essere eseguita (realizza il *why*)
  - Non è sufficiente: il *when* specifica il quando
- `trigger` e `s_executor` hanno thread indipendenti

Nel caso asincrono invece abbiamo il sendEvent(), che non ha la freccia piena. Il ciclo di vita di questo componente finisce qua. Lancia la notifica, triggera il componente, ma non aspetta.

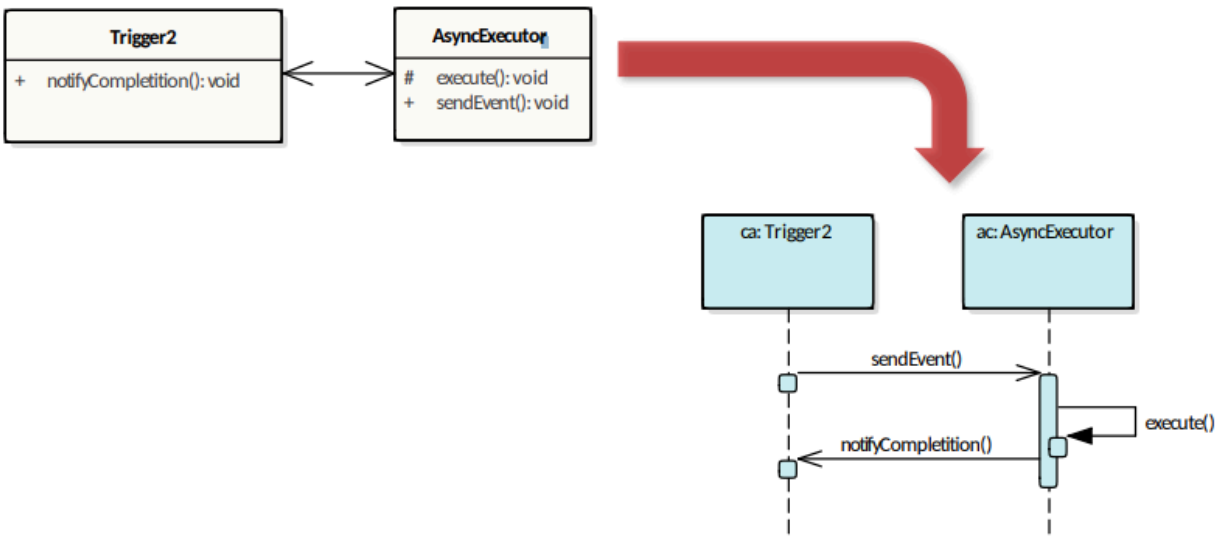
Non tutte le piattaforme supportano il meccanismo di comunicazione relativo al control flow che voglio andare a realizzare. Quindi magari devo costruire un asincrono su una piattaforma sincrona, o il contrario.

Asincrono su sincrono: gli do un ritorno subito, e poi continuo ad eseguire.



- Occorre realizzare i componenti in modo da mascherare i meccanismi sotto
- La `notifyEvent` deve essere il più breve possibile per simulare uno stile asincrono
  - Quindi memorizza l'evento e ritorna. Poi il comportamento è simile a async-async

Sincrono su asincrono: il sendEvent è asincrono, il componente manderà una notifica poi.



Per esempio l'HTTP non supporta l'asincronismo di base, quindi ce lo costruiamo sopra.

Information flows

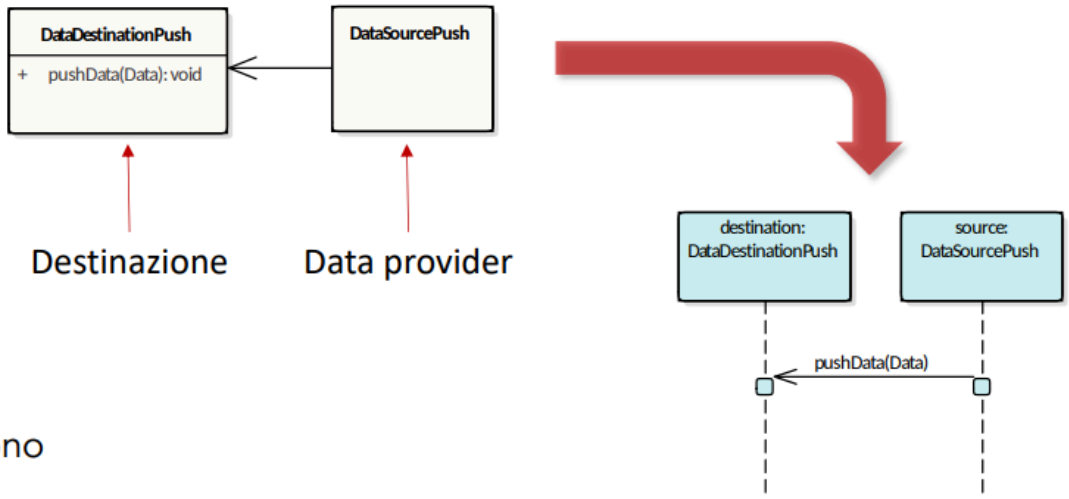
Abbiamo 2 meccanismi:

- **Push**: basato su una comunicazione asincrona

- **Pull**: basato su una comunicazione sincrona

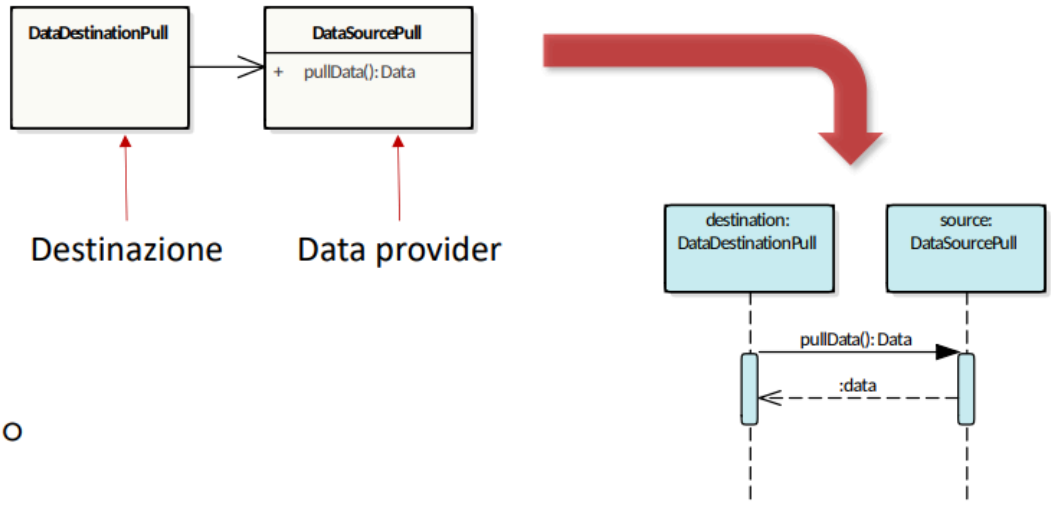
Qui abbiamo il data provider, che può essere il sensore per esempio. è stato chiamato DataSourcePush perchè pusha il dato. La destinazione è il componente interessato al dato. La strategia la sceglie il datasource, che quindi deve conoscere chi sono i componenti interessati al dato. Il componente interessato deve esportare un metodo che permette al datasource di inviargli i dati.

Questo è semplicemente l'invio di un dato. Se io voglio triggerare un componente anche mandandogli un messaggio, metto insieme control flow e data flow.



- Asincrono

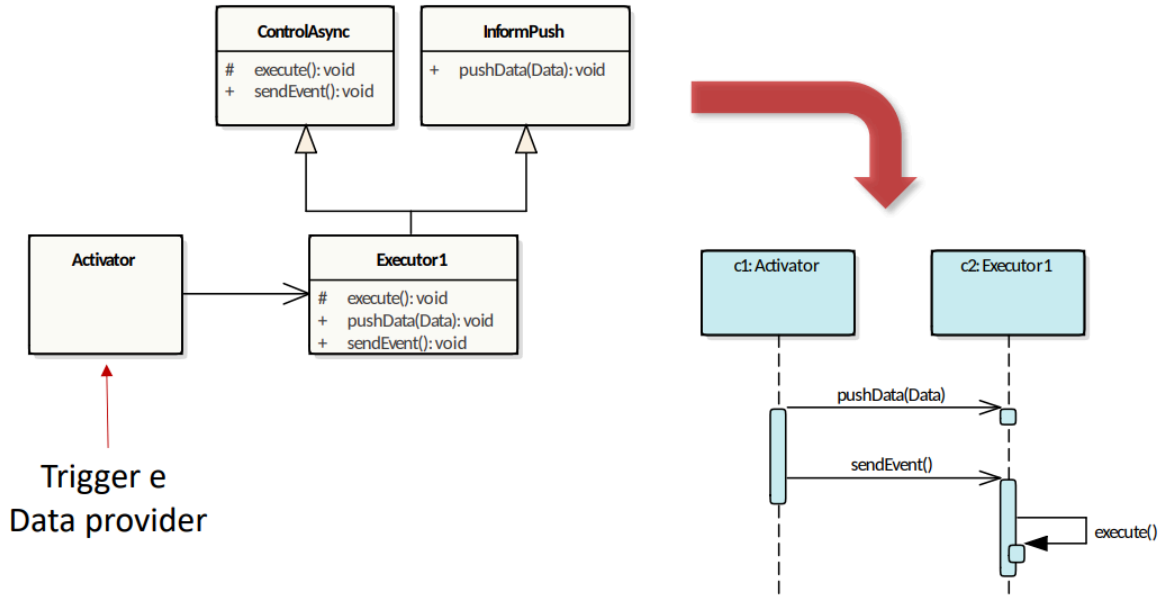
Questo invece è il pull. Il componente è in modalità sincrona, perchè recupera il dato e te lo restituisce, se non fosse così allora sarebbe complicato perchè un data source che magari è un sensore fornisce informazione a molti client, ad ogni richiesta che arriva dovrebbe memorizzarsi chi era e poi mandare indietro il dato quando l'ha elaborato.



- Sincrono

Esempio 1

Trasferimento dati push asincrono e controllo asincrono:



Qui c'è un activator, che è un qualsiasi componente, per esempio che si interfaccia con il sensore.

Executor1 ha la sua funzione execute(), ha un control flow asincrono ed è un information pusher perchè ha entrambi i metodi sendEvent() e pushData().

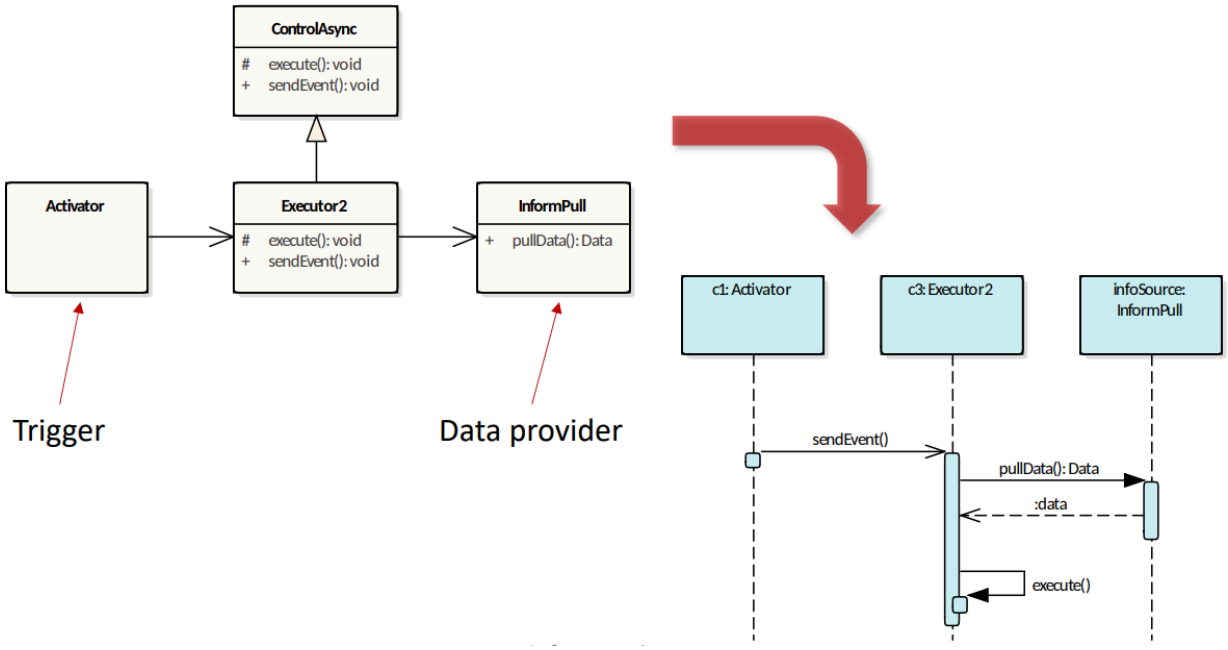
L’activator triggera e dà il dato (pushData(Data)), e questo lo può fare ad ogni dato letto.

Di base nel momento in cui faccio un pushData e poi subito dopo il sendEvent faccio una cosa stupida, perchè a sto punto potrei incastrare nella semantica del pushData il fatto che lui triggera anche un evento.

Però può essere che ci sia un buffer, e quindi Executor1 registra i dati, e il sendEvent avviene solo dopo l’invio di un tot di dati.

Esempio 2

Controllo asincrono e trasferimento dati pull sincrono:



Ho sempre un activator, il componente che notifica, poi ho l’executor che è un control async e utilizza una fonte dati che lavora in pull.

L’activator non è altro che un componente che manda “l’allarme”. L’executor applica la logica di dominio.

In questo caso abbiamo che la strategia di attivazione del componente (il why) è separato dal componente, come è giusto che sia. Il componente (executor) è applicativo, e applica una regola di dominio (tipo alzo il termostato), l’activator semplicemente ogni tot dice a executor di fare il suo lavoro. InformPull va a recuperare i dati (per esempio delle registrazioni video).

Sintesi

Io posso fare flusso di controllo e flusso informativo sia in maniera sincrona che asincrona. Di base il push funziona in modalità asincrona e il pull in modalità sincrona. Però questi sono gli unici due meccanismi per scambiare informazioni tra componenti.

Il flusso di controllo non trasporta informazioni ed ha l'obiettivo di realizzare il why. Il flusso informativo dati trasporta informazioni.

Design interno dei componenti

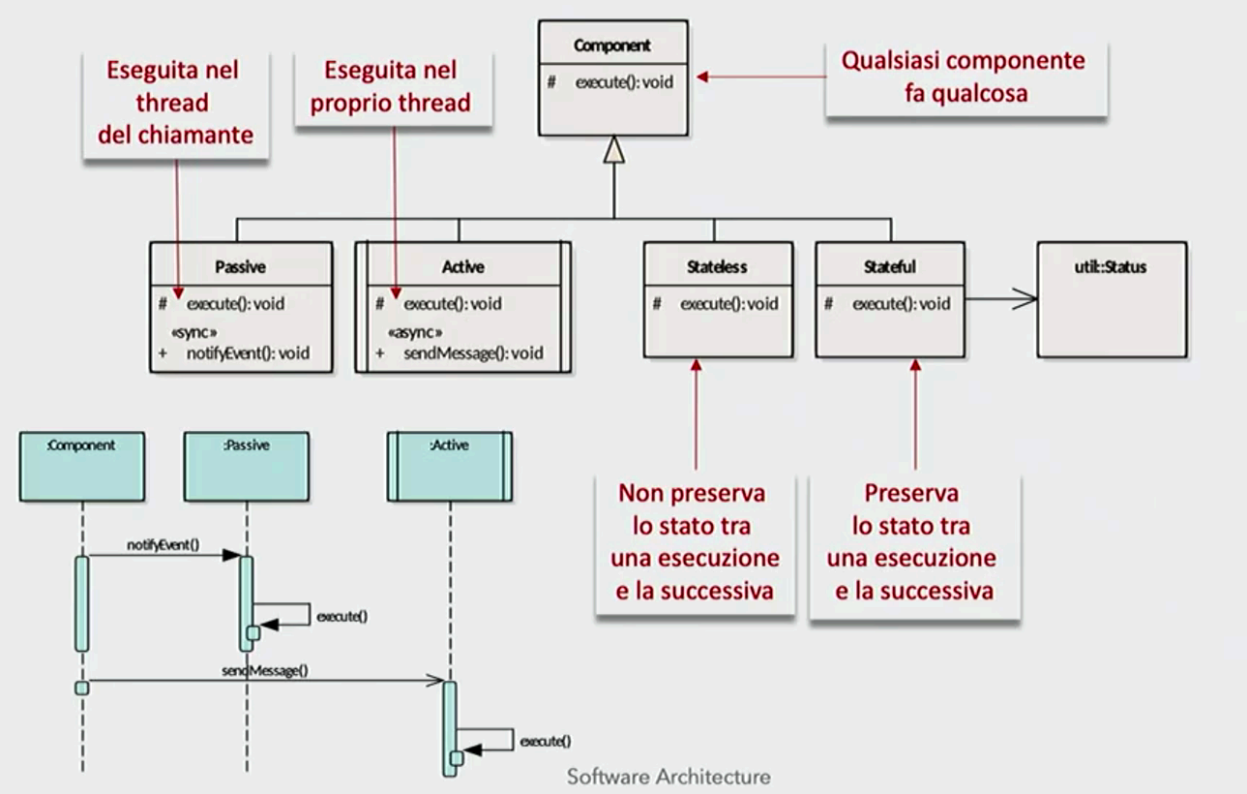
Il design interno del componente è strettamente legato agli stili di interazione con gli altri componenti. Occorre specificare componenti possono essere:

- Passivi
- Attivi

E possono essere

- Stateless
- Stateful (si comporta come macchina a stati)

Tipologie di componenti

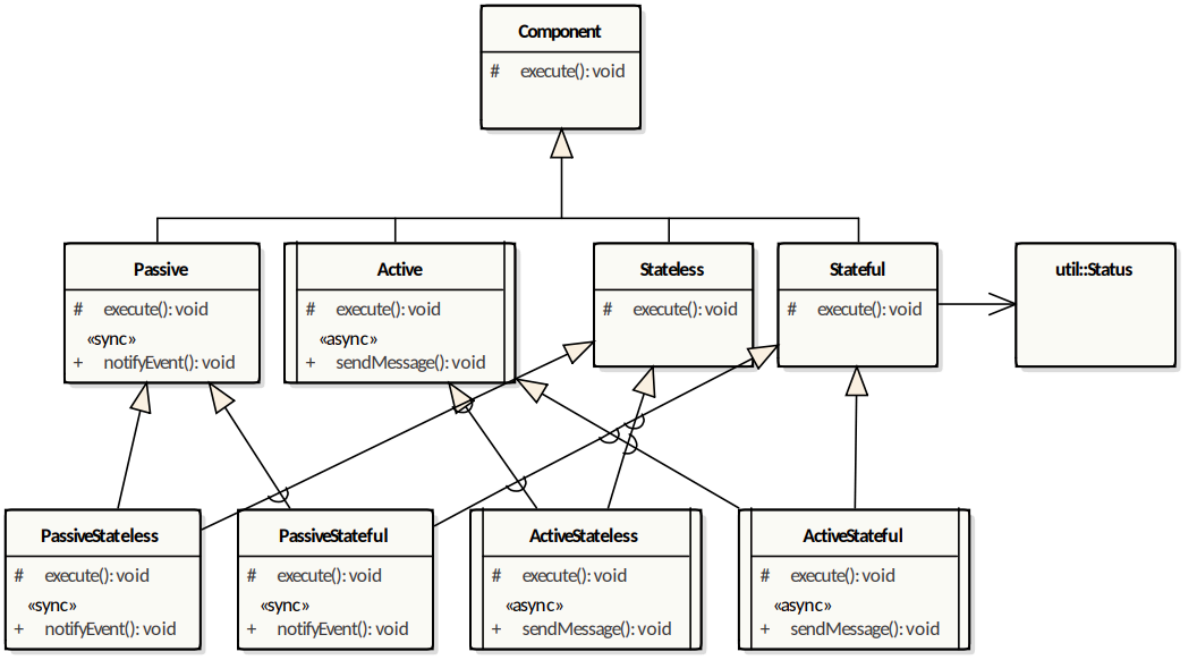


Tutti i componenti hanno un'execute.

Poi ho la linea di sinistra, che sono quelle che possono essere attive o passive. La doppia barra in UML significa attivo.

sendMessage() è marcato come async, va specificato. invece la notifyEvent() è sincrona perchè il componente è passivo, quindi deve essere bloccante per riuscire a girare sul thread del chiamante.

Stateless ha un'execute normale, mentre stateful è rappresentato il fatto che ha uno stato, qualunque esso sia (può essere lo stato di una transazione, il numero di elementi nel carrello...).



Si può fare di tutto e di più nei vari componenti, PassiveStateless, PassiveStateful... etc

I passivi non possiamo pensare di farli remotamente, perchè non posso girare nel thread del chiamante, lo vedremo in un esempio particolare.

Queste sono le situazioni classiche che dobbiamo andare a realizzare quando abbiamo il nostro sistema:

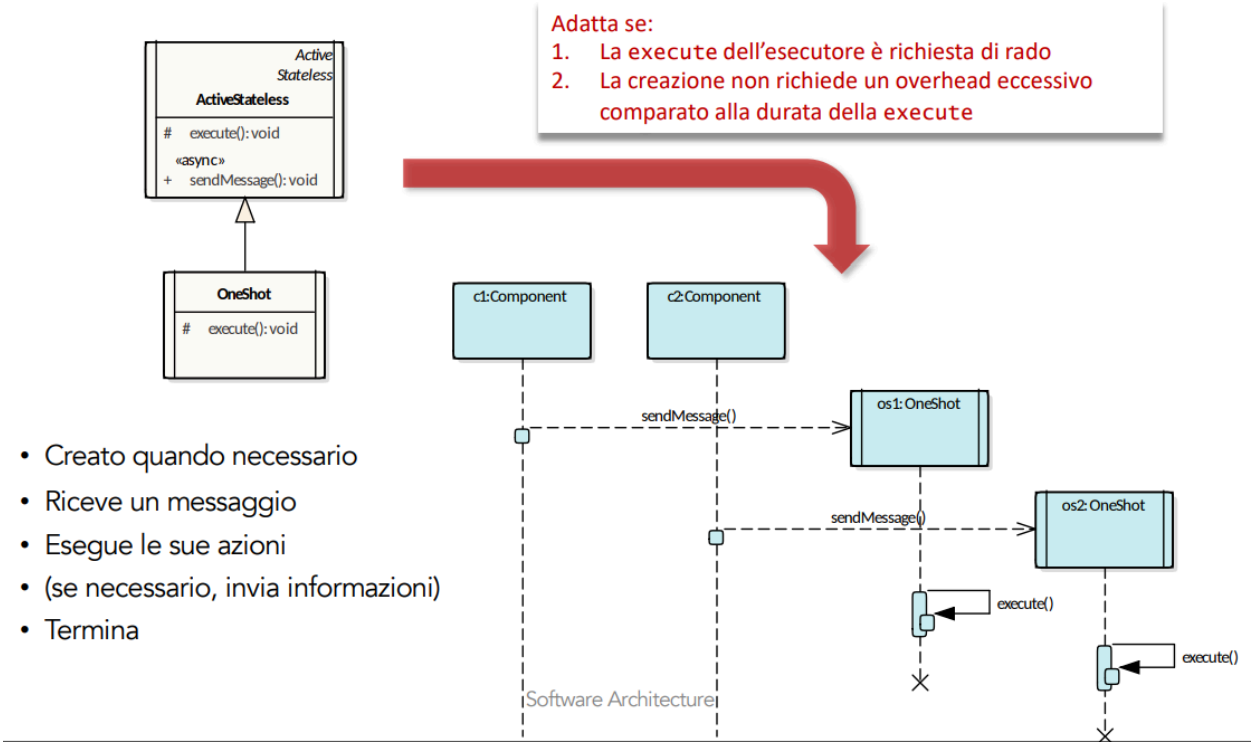
- Componenti attivi e senza stato
  - One-shot
  - Sequenziale
  - Concorrente
  - Farm manager
- Componenti con stato
  - Farm manager stateful
  - Shop chart (tengo traccia dei prodotti già inseriti nel carrello)

- Pagamenti elettronici (lo stato corrente dell'esecutore determina quali messaggi possono essere accettati e cosa deve essere fatto)

Esecutore one-shot

Ho un componente che è ActiveStateless, quindi ha un sendMessage() (mi immagino che un componente isolato non faccia nulla, quindi questo ha un'interfaccia).

Di base viene costruito un oggetto one-shot, che è quel componente che viene creato nel momento in cui c'è necessità, fa quello che deve fare e poi termina (viene eliminato il processo che lo ospita)



Questo è l'esempio che avevamo fatto nell'esempio del supermercato, la prima ripartizione good-driven dove c'era un componente che si occupava di tutto.

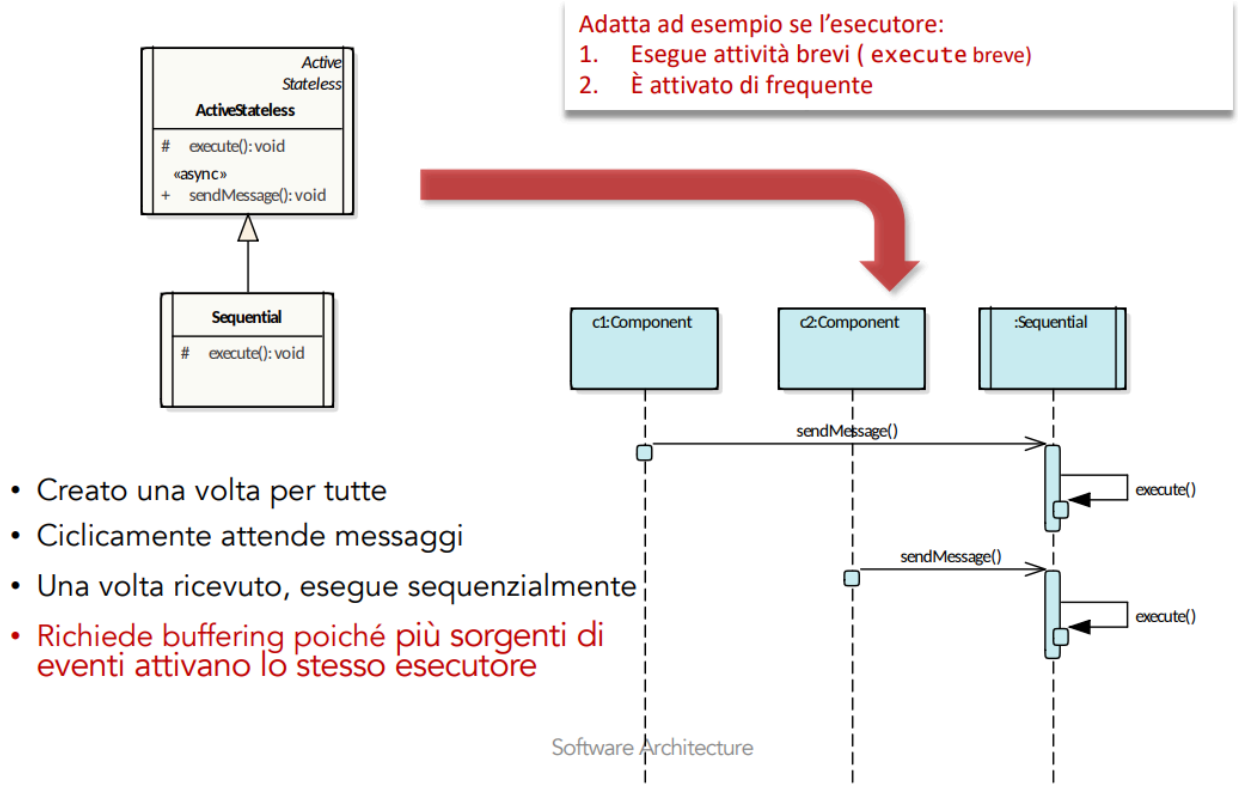
Questo tipo di soluzione va bene, ma di norma se il processo di istanziazione richiede effort, allora vale la pena adottare questa soluzione se l'execute è eseguito di rado.

Nel caso del supermercato questa soluzione non va bene, le sold hanno una frequenza troppo alta.

Esecutore sequenziale

In questo caso ho Sequential che è un ActiveStateless anche lui, quindi ha execute e sendMessage, anche qua ho componenti che triggerano un evento, però questi arrivano al componente che non fa altro che registrare la richiesta, ovvero il fatto che è stato generato un evento e che lui l'ha ricevuto, e poi elabora.

Lui ciclicamente attende messaggi, una volta ricevuto, esegue sequenzialmente. Quindi richiede del buffering, perchè posso avere richieste a frequenza alta, più alta del delay impiegato per gestire un evento.

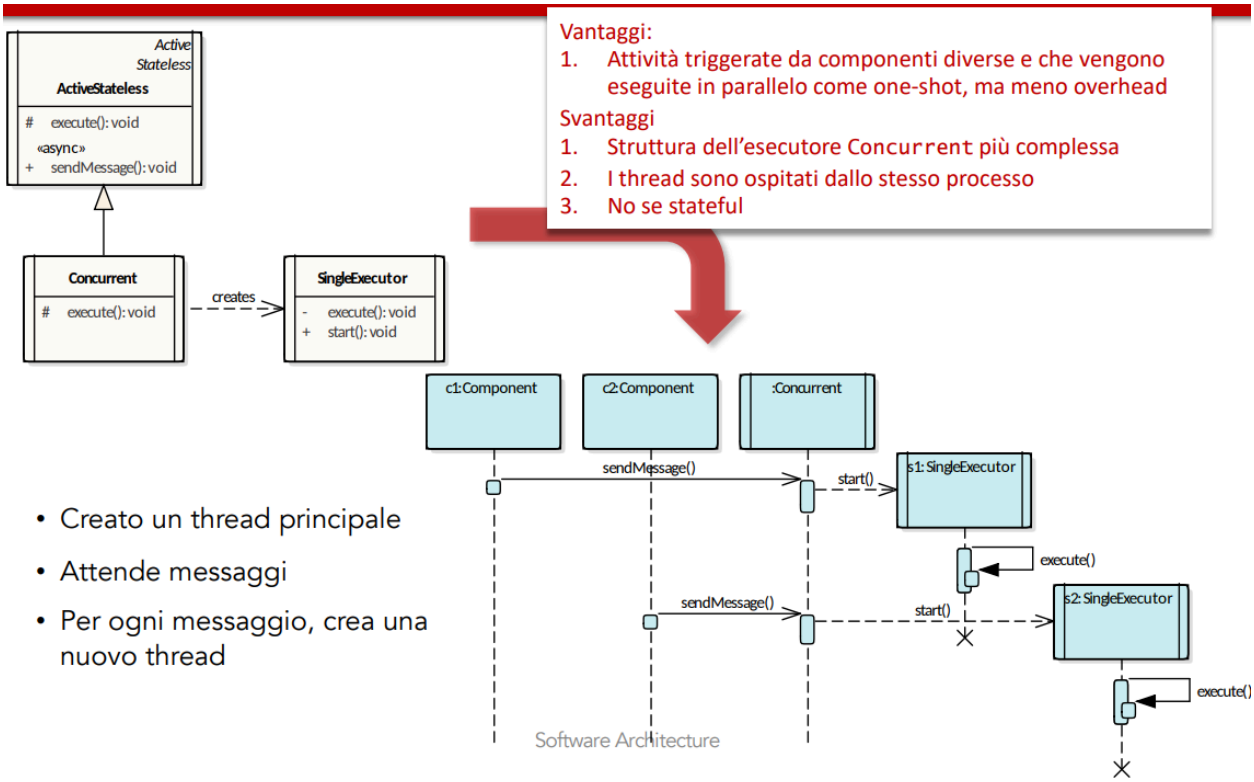


Questa situazione va bene quando è attivato frequentemente, però devo avere delle execute che sono abbastanza veloci, ho un trade off tra quante richieste arrivano e quanto ci metto ad evaderle. Se tutto sta in piedi, allora questa è una soluzione molto semplice che funziona.

Esecutore concorrente

Potrebbe non andarmi bene il meccanismo di bufferizzazione. In questo caso, se magari tutto gira su una macchina con tanti core, e posso programmarla di modo

che a ogni thread che eseguo posso dedicare un core (costa molto di infrastruttura però), lo faccio se è un'esigenza. In questo modo le execute vengono fatte in maniera più o meno concorrente rispetto a quando arriva la richiesta.



Questa soluzione, fatta in questa maniera, se i singoli executors devono avere uno stato, non funziona bene.

La differenza rispetto al one-shot è che nel one-shot genero proprio un processo che gira nel suo sandbox, qua invece siamo nello stesso processo dell'applicazione di concurrent.

Osservazioni

One-shot e concurrent sono da preferire se le azioni da eseguire hanno una durata impredicibile e/o lunga. Sequenziale invece se le operazioni sono di brave durata.

Drawback:

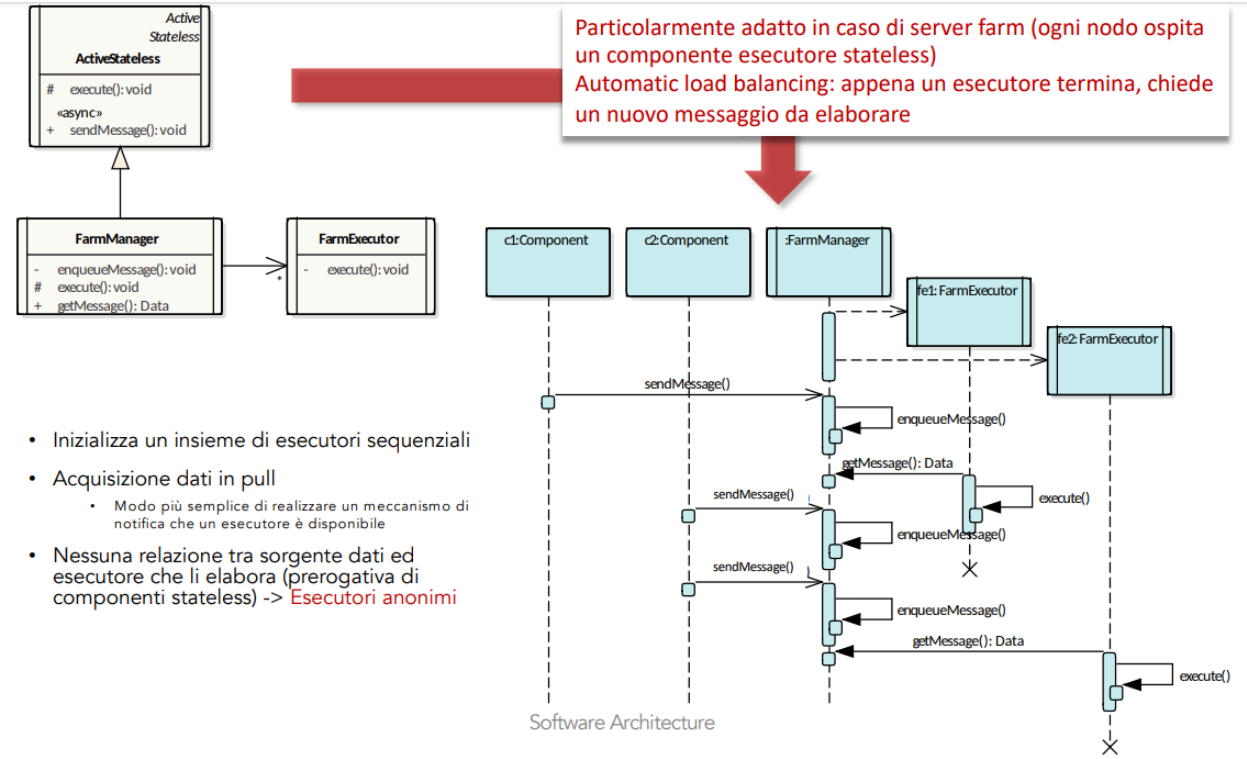
- One-shot: overhead per la creazione di un nuovo processo

- Concurrent executor: più thread sullo stesso processo, nessuna concorrenza reale

Esecutore farm manager stateless

Possibile soluzione: farm manager (quindi scaling orizzontale, dove nel mezzo ho un load balancer): istanzia un insieme di esecutori sequenziali (possibilmente su nodi diversi) e smista un messaggio in ingresso ad uno degli esecutori disponibili.

è stateless, ho un FarmManager, ho il mio componente che non gira in nessun thread, ma quando lancio l'esecuzione sta in un processo (può essere distribuito, non deve essere per forza nello stesso processo del concurrent).



In questo esempio ho una serie di componenti, una istanza di FarmManager che inizializza una serie di esecutori seriali (in questo caso ne crea 2 di FarmExecutor).

I componenti invocano una sendMessage() sul FarmManager, che è asincrono perchè è un componente attivo. Lui ogni volta che riceve un messaggio, ha il metodo di enqueue che semplicemente accoda il messaggio e poi i FarmExecutor quando vogliono (di base quando sono liberi) si appendono e aspettano fino a

quando c'è un nuovo messaggio che è stato messo in coda. Quando arriva un messaggio, loro ottengono il messaggio, fanno l'execute e nel caso notificano.

La cosa importante è che ho degli esecutori anonimi, non ho nessuna informazione sul fatto che la richiesta per esempio fatta dal componente c1 sia per esempio fatta dal FarmExecutor fe2. Anche per questo devono essere stateless, il fatto di renderli anomali mi rende tutto più semplice.

Se il componente ha uno stato (per esempio sta facendo acquisti online), allora in quel caso devo tenere traccia di che FarmExecutor sta gestendo il componente.

Esecutore 'farm manager' stateful

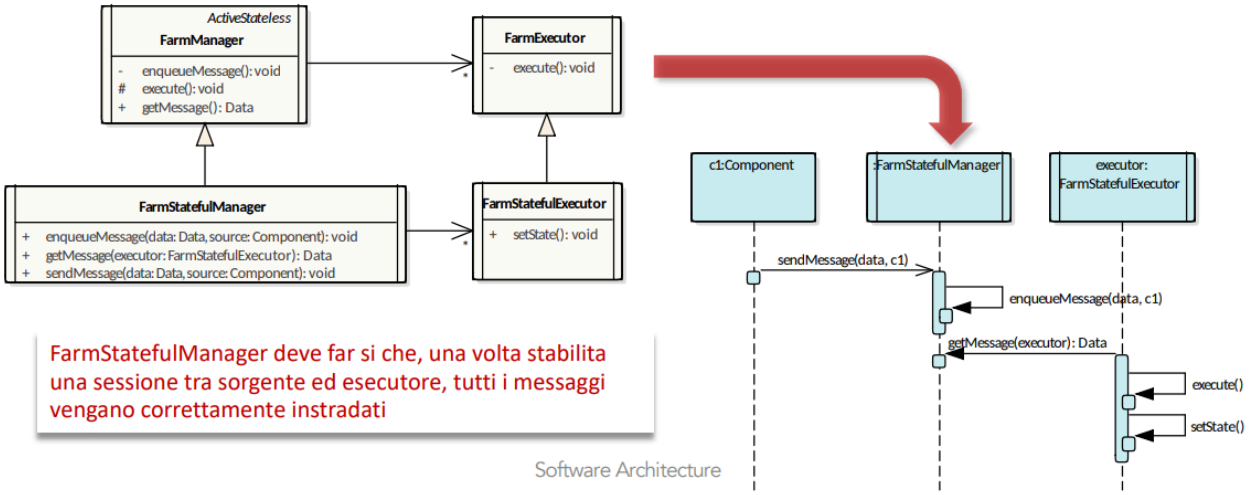
Nel caso di stateful, dobbiamo gestire lo stato. Lo possiamo fare o nel servizio (nel Farm) o mantengo nel client, o metto tutti in un database.

La soluzione più semplice è la seguente, dove questo è il servizio a mantenere lo stato.

Ho il solito FarmManager, come caso particolare gli vado a mettere questo FarmStatefulManager, che è quello che mantiene la memoria.

Utilizzo il solito sendMessage() che è nel FarmStatefulManager, ma passo anche chi è il componente che ne ha chiesto l'esecuzione. A questo punto lui fa la solita enqueue, e si terrà una sorta di mapping tra componente ed esecutore associato.

- Esecutori non sono anonimi
  - Tutti i messaggi dalla stessa sorgente devono essere elaborati dallo stesso esecutore
    - Messaggi devono specificare chi è la sorgente/esecutore designato in modo da instradare correttamente
      - Informazione nota. Questo esempio vuole enfatizzare la necessità



FarmStatefulManager deve far si che, una volta stabilita una sessione tra sorgente ed esecutore, tutti i messaggi vengano correttamente instradati

Sintesi

- Un documento di specifica dell'architettura concrete include:
  - Il disegno interno dei componenti che include
    - Stateless vs stateful
    - Active vs passive
    - Le interazioni con gli altri componenti, gli attori e i datastore (magari con gli stereotopi **push**, **pull**, **sync**, **async**)

    ➡ Strumento di modellazione: diagramma delle classi in cui devono comparire come classi esclusivamente i componenti identificati dall'architettura logica
  - Le interazioni fra componenti, attori e datastore in casi esemplificativi che includono:
    - Flussi di controllo (asyn e sync)
    - Flussi dati, sia transienti sia datastore (push/pull)

    NON devono includere le interazioni interne dei componenti

    ➡ Strumento di modellazione: diagrammi di sequenza con istanze di componenti, attori e datastore

Esempio di riferimento

Good driven

Questo è il componente one-shot.

Ci vuole comunque un componente di mezzo, che è quello che istanzia, qua è chiamato delegator ma può essere chiamato come si vuole. è quello che si interfaccia con il pos.

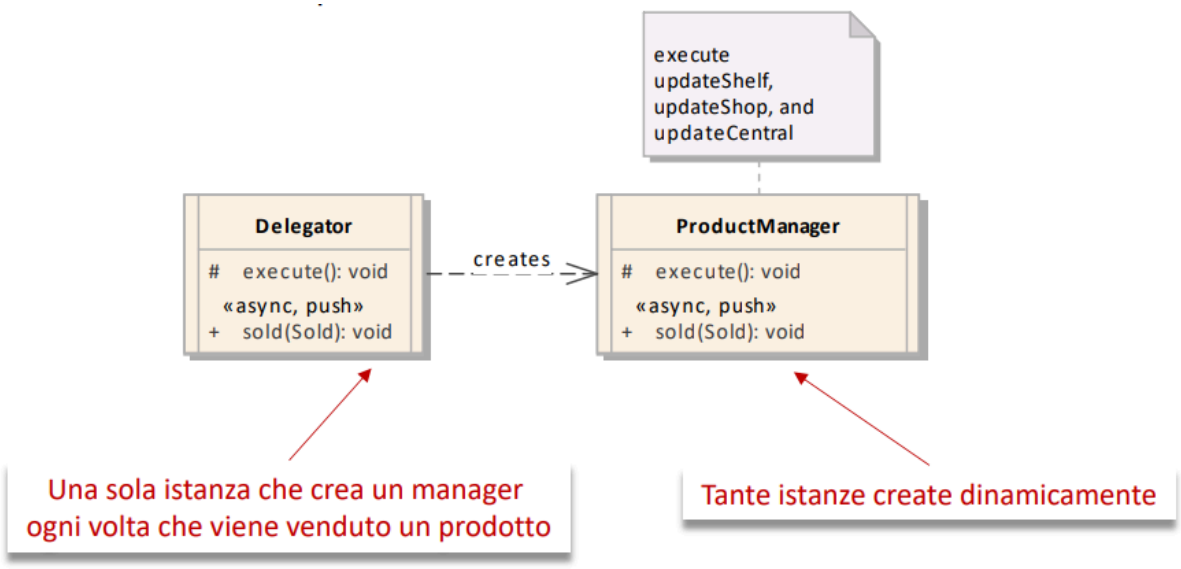
Il pos fa una vendita, arriva a questo componente di mezzo che provvede ad istanziare il componente one-shot.

Questa è una vista diversa del componente logico, quindi manteniamo gli stessi nomi (come ProductManager).

Il ProductManager ha la sua execute, possiamo mettere una nota dicendo cosa fa, ma non ci interessa nel dettaglio le operazioni che fa. Il diagramma delle attività diceva che queste operazioni venivano fatte in sequenza.

La cosa che si può vedere qua è che questo è un componente attivo (ProductManager) e la sua interfaccia è dotata di una sola risorsa, che si chiama Sold, che è una risorsa asincrona e push, il che vuol dire che questo fa sia flusso di controllo che flusso dati.

Il fatto che è push significa che la fonte dati gli manda un dato, che è il sold. Anche qua è asincrona perchè il delegator manda il dato e va a gestire altre richieste.



Per modellare i componenti concreti in termini di disegno interno e delle interazioni... quando devo rappresentare nella documentazione il disegno concreto, io devo modellare i componenti concreti, che sono appunto la vista concreta dei componenti logici. Quindi per ciascun componente logico che ho identificato nell'architettura logica ho un componente concreto, vado a rappresentare se il componente è attivo o passivo, vado a dire se ha stato o meno. Questo per la struttura interna. Poi devo dire se ha un'interfaccia, qundi devo andare a fare l'elenco delle risorse, che se vediamo il diagramma sono i metodi. Per ogni metodo (ovvero per ogni risorsa), vado a specificare se è un control flow: se è asincrono o sincrono. Se è un data flow, se è push o pull. Oppure posso fare controllo del flusso e data flow insieme, e quindi devo andare a specificare asincrono o sincrono per il data flow, e push o pull per la modalità in cui il dato è spedito.

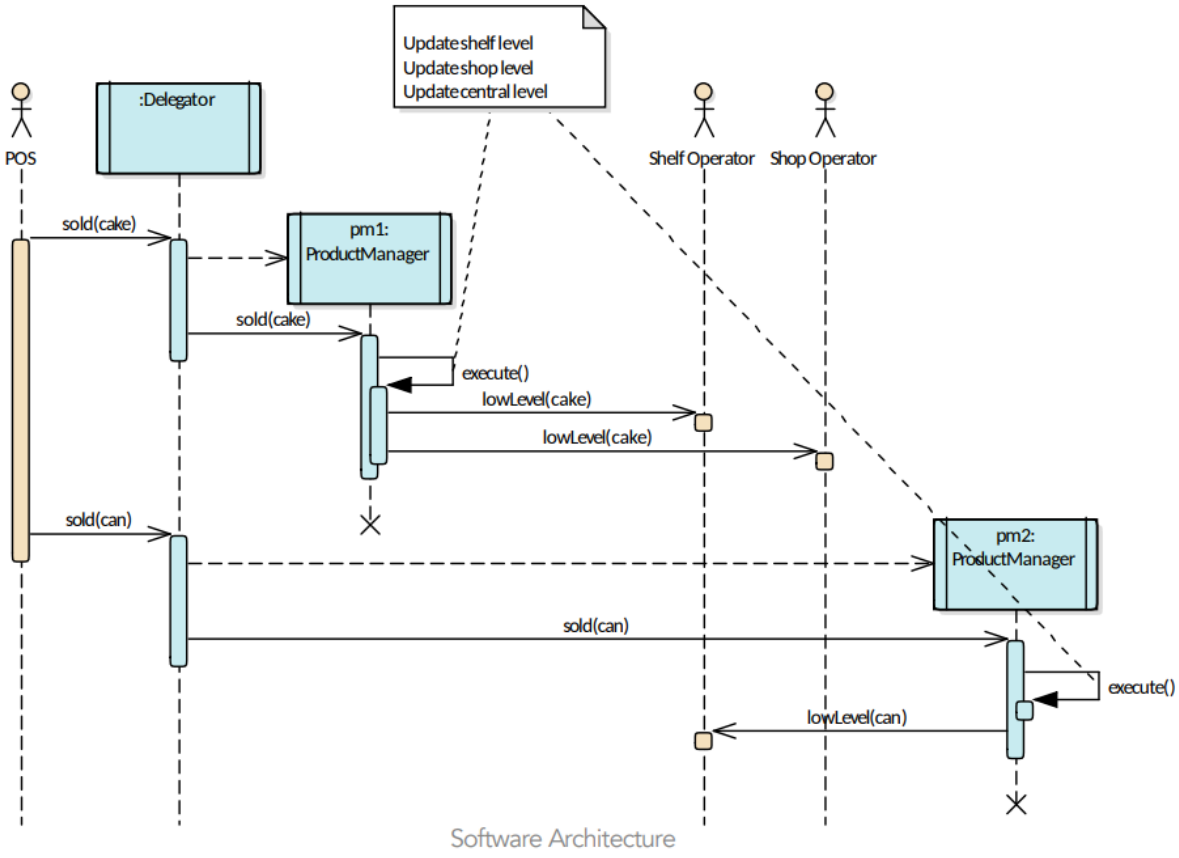
Questo si traduce nell'esempio, dove abbiamo visto che esistono due componenti, tutti e due attivi, non c'è il disegno di uno stato e quindi sono stateless.

A questo punto devo andare a definire le interfacce, e quindi nell'esempio ci sono per ciascuno una risorsa, e siccome questa risorsa è sia flusso di controllo che flusso dati, specifico che è async per flusso di controllo e push per flusso dati.

Noi dobbiamo fare vedere un qualche esempio di esecuzione, 2/3 diagrammi di sequenza che fanno vedere le componenti concrete come interagiscono tra di loro.

In questo esempio ho un pos (potrei metterne tanti), ho l'attore che è il pos, un'istanza di delegator a cui arrivano tutte le richieste (sold), che sono asincroni e in modalità push del dato, che è quello che abbiamo specificato sopra.

Il delegator, ad ogni evento di tipo sold che riceve, non fa altro che istanziare un'istanza di ProductManager (perchè è one-shot) e gli chiama il sold suo (cake). Gli manda come trigger l'evento, e poi gli passa il dato. Ciascun ProductManager ha la sua execute, al massimo dico che deve fare l'update dello shelf, dello shop e dell'update.

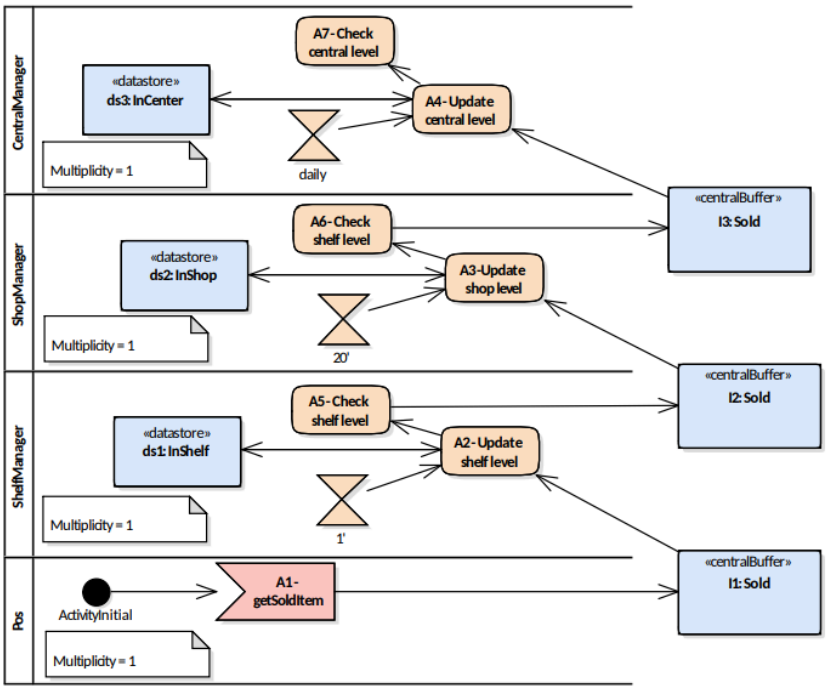


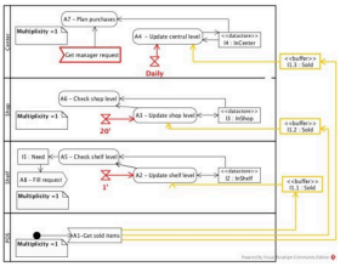
Il fatto che viene contattato lo shelf operator e/o lo shop operator diversamente nei 2 casi, è un esempio dato dall'esecuzione.

Abstraction-driven: rivista

In alto a destra c'era quella versione che aveva un flusso dati enorme.

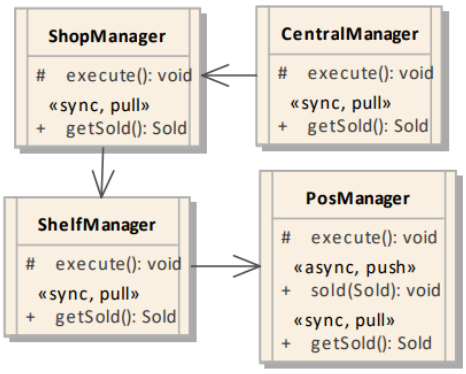
Sotto c'è una versione rivisitata, i centralbuffer dei vari livelli vengono riempiti solo nel momento in cui il componente è attivato. Quindi, arriva un sold ad ogni vendita, però visto che A2 parte ogni minuto, faccio la connessione con il buffer e li prendo solo in quel momento.





Fatta così, ho 4 componenti e questa è l'architettura concreta corrispondente.

Ho componenti sequenziali che recuperano le informazioni sulle merci vendute ed effettuano i controlli.



Una sola istanza per ciascun componente

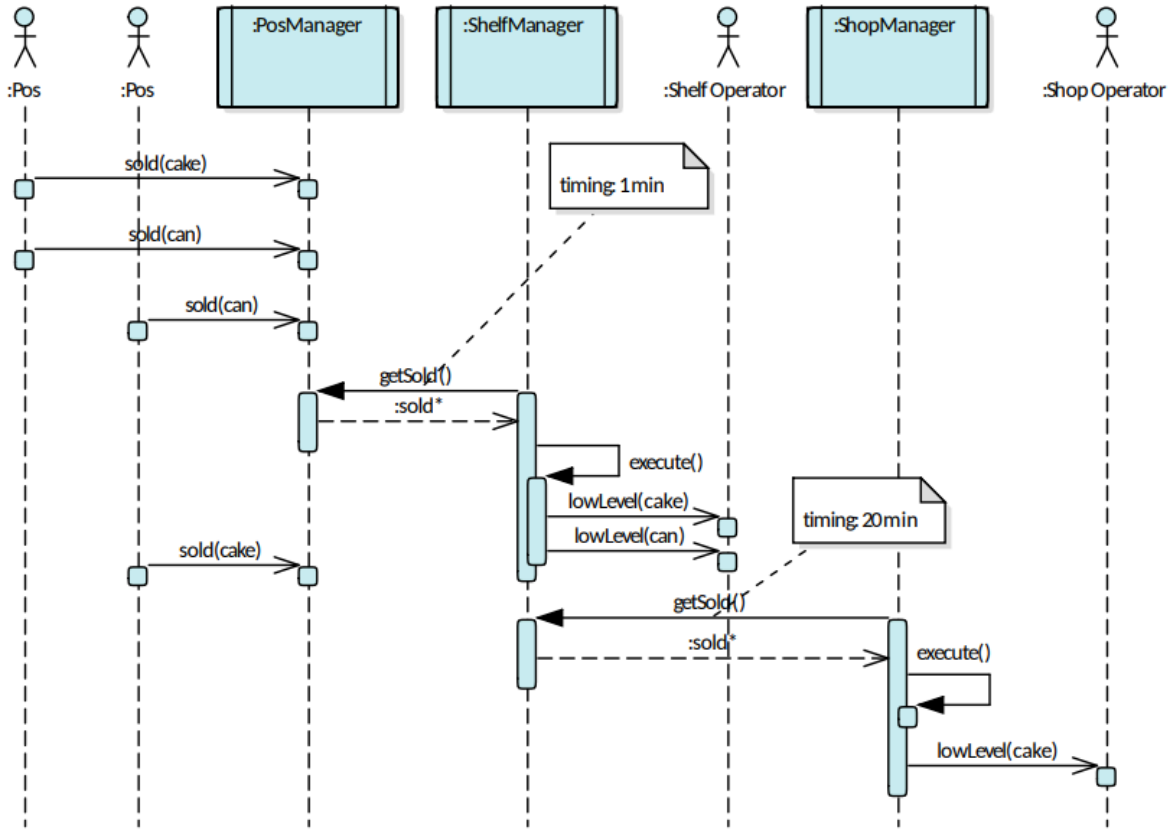
Il PosManager ha la sua execute, ha asincrono il sold del push, questo è quello che arriva dal pos fisico, che triggera e manda il dato. Poi ho una getsold che è

sincrona e pull, che viene usata dallo shelfManager, per chiedere gli ultimi x sold. E quindi c'è una async pull del getSold che usa lo shopmanager, è a ritroso.

Nel diagramma abbiamo i 2 Pos, che mandano i prodotti venduti tutti al PosManager, che non fa niente. Semplicemente registra i dati. Lo ShelfManager, ogni minuto fa un get del sold, quindi fa una pull del dato, gli viene restituito sold* perchè sono più di uno, e a questo punto fa la sua execute e poi comunica con lo shelf operator.

Lo ShopManager ogni 20 minuti fa lo stesso comportamento, chiedendo allo ShelfManager perchè lui tiene i sold pronti. In questa maniera, lo ShelfManager comunica solo con lo ShelfOperator, e lo ShopManager comunica solo con lo ShopOperator.

In questa slide manca il CentralManager, ma è analogo agli altri 2. Anche noi dobbiamo fare così, non dobbiamo mostrare tutto, qualcosa.



Deployment architetture

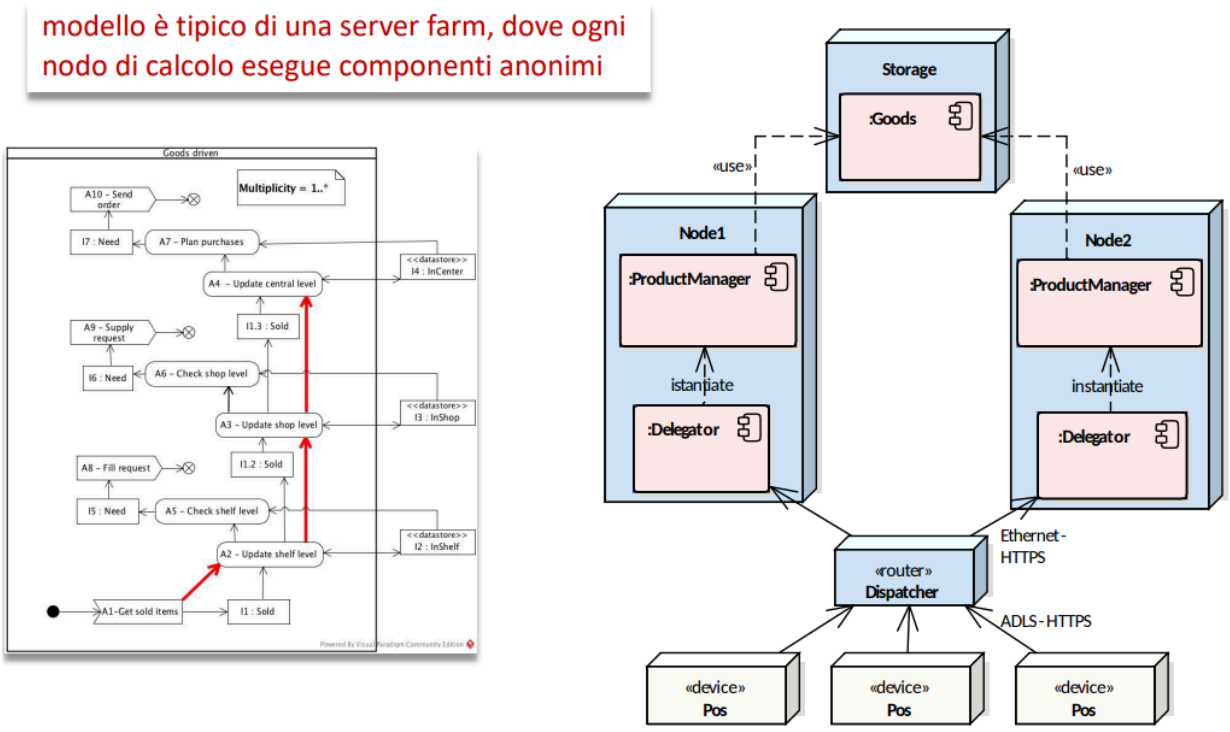
Architettura di deployment ha come obiettivo definire:

- Identificare i nodi computazionali necessari
  - Stime di costi (hw e di persone)
- Allocare istanze di componenti nei nodi computazionali

Esempio di riferimento

Good driven

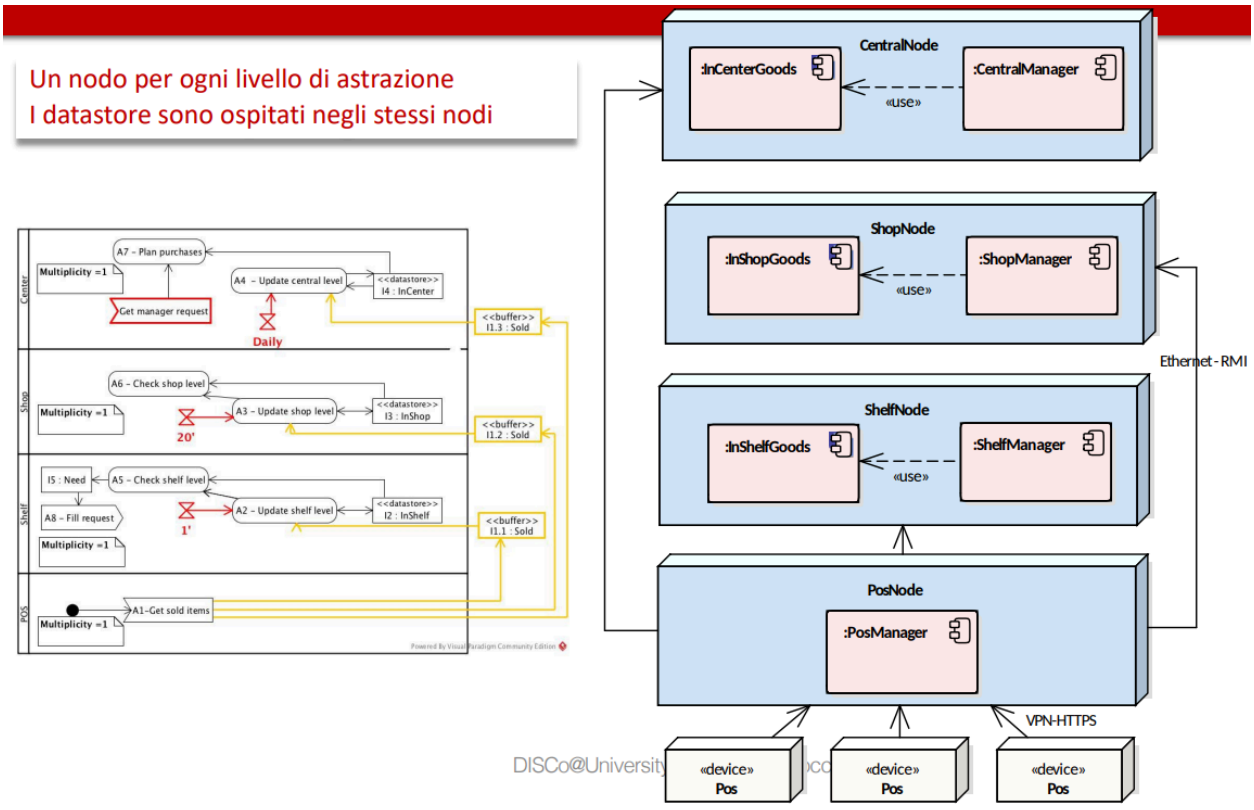
Vado a prendere dei diagrammi UML (di deployment in questo caso) dove vado a vedere che ho i miei vari pos, e poi c'è un dispatcher che in questo caso fa il lavoro di una sorta di load balancer, e invia i dati ad un delegator o all'altro, che istanzia il ProductManager...



I Pos sono geograficamente collocati in punti diversi, quindi devo utilizzare la rete per raggiungere il dispatcher, ma la precisione dei sold deve essere massima, e quindi dovrò pagare molto l'infrastruttura per non perdere dati, e quindi non vale la pena farlo così.

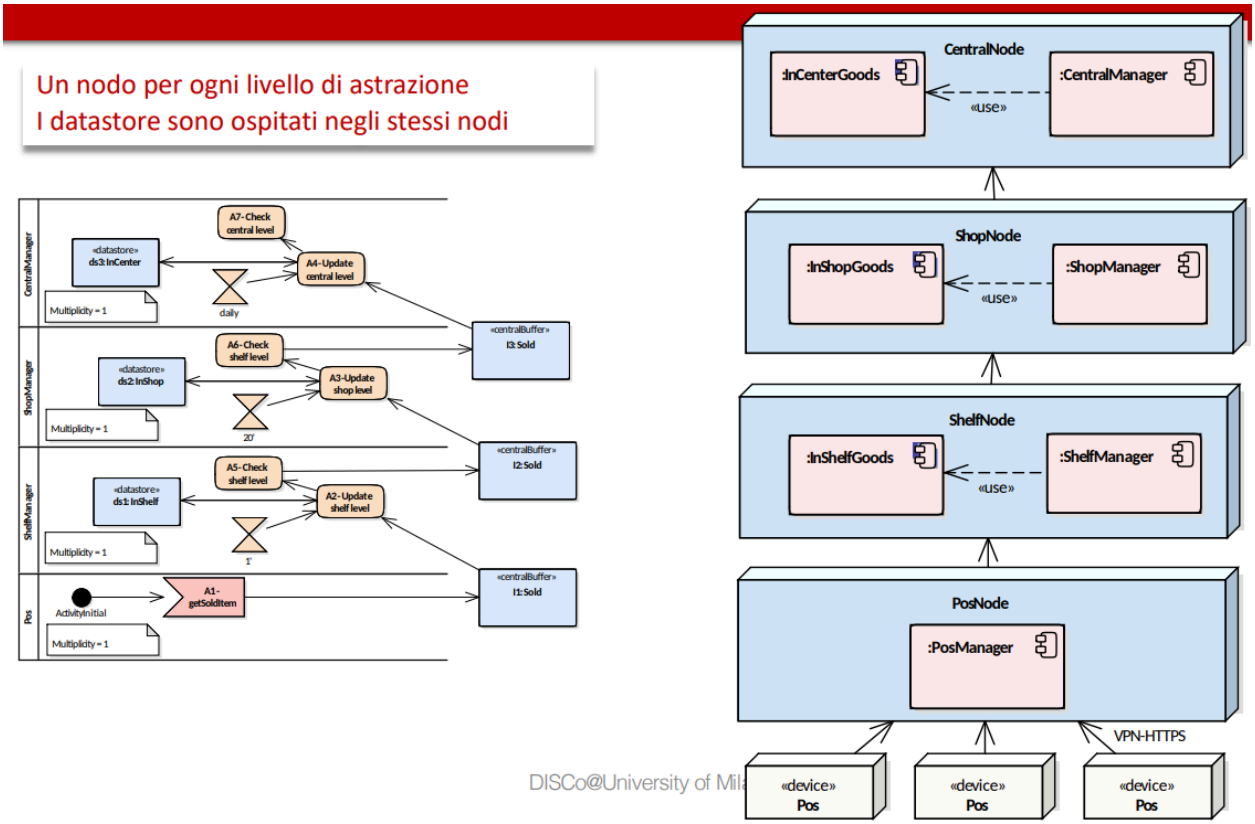
- Pros:
  - Alta scalabilità poiché possono essere istanziati tanti ProductManager e/o usati nodi quanti necessari
  - Configurazione di sistema e amministrazione semplificata (tutti i nodi ospitano le stesse componenti)
- Cons:
  - Componenti complessi poiché devono elaborare Goods a tutti i livelli di astrazione
  - Accesso ad un unico storage può essere un collo di bottiglia
    - Si può suddividere per prodotti
  - Il dispatcher e la rete possono essere un collo di bottiglia

Abstraction driven



- Pros:
  - Chiara separtion of concerns
  - Layer con vincoli temporali su nodi diversi
    - Si sceglie la tecnologia appropriata per ottenere una soluzione cost-effetctive
  - Nodi autonomi anche in caso di mancanza degli altri (ad eccezione del POS) poiché hanno tempi diversi e lavorano su dati privati diversi
- Cons:
  - Larghezza di banda e disponibilità della stessa critico per la comunicazione Pos e PosNode e tra PosNode e gli altri componenti (tanta informazione da trasportare)
  - Supporto limitato alla decentralizzazione (un solo componente per layer, la molteplicità era 1)

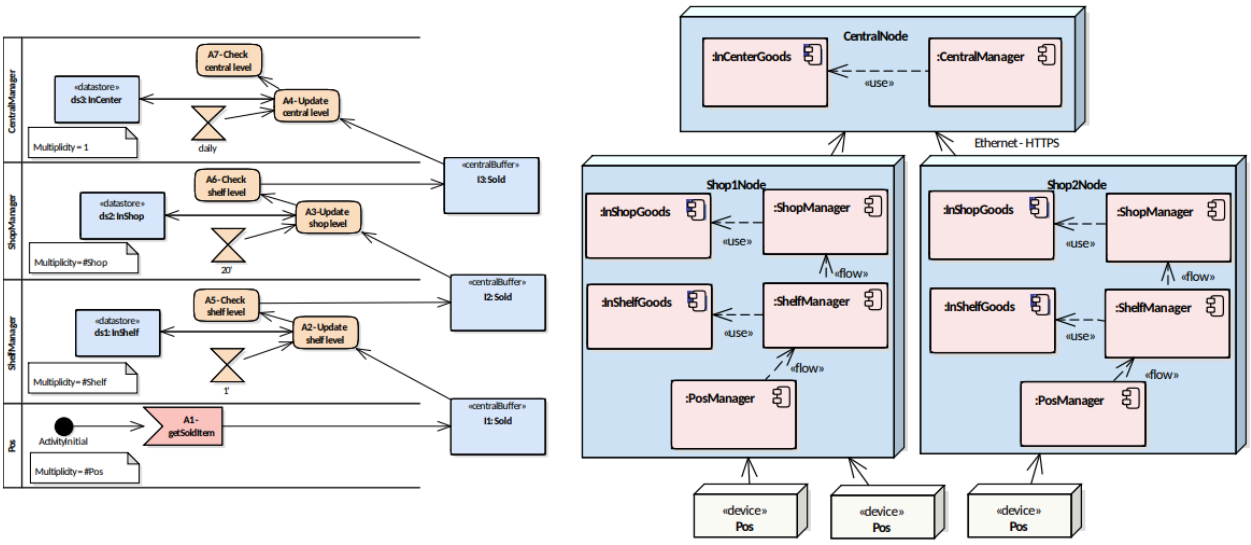
Abstraction driven (variante)



- Pros:
  - Chiara separation of concerns
  - Layer con vincoli temporali su nodi diversi
    - Si sceglie la tecnologia appropriata per ottenere una soluzione cost-effetctive
  - Nodi autonomi anche in caso di mancanza degli altri poiché hanno tempi diversi e lavorano su dati privati diversi
  - Larghezza di banda non critica poiché i layer hanno timing diversi
- Cons:
  - Supporto limitato alla decentralizzazione (un solo componente per layer, la molteplicità era 1)

Abstraction-location-driven

Un nodo per negozio e un nodo centrale



- Pros:
  - Chiara separation of concern
  - I manager locali e i datastore locali possono essere implementati con tecnologie differenti
  - Supporto alla decentralizzazione geografica (i nodi li sposto in periferia)
  - Pochi requisiti di banda e di disponibilità della stessa (solo tra nodo centrale manager locali e tra manager locali e pos)
  - Anche se ho perdita di banda, le attività del negozio continuano ad essere portate avanti
- Cons:
  - Configurazione e amministrazione più complesse

Riassumendo

Specificare i nodi (nodo di processing, tablet, smartwatch, device, etc.)

Allocare i componenti ai nodi

Specificare i protocolli di trasmissione fisici e applicativi

Fare una stima dei costi (hw e sw) e di manutenzione