- ***Lezione 24***

Al livello di astrazione di progettazione preliminare, scendendo più in dettaglio, per vedere pattern/protocolli che si usano in applicazioni service oriented per garantire certe proprietà.
Un aspetto fondamentale che deve essere garantito anche dalle applicazioni service oriented è quello legato alle proprietà di una transazione (studieremo quindi i software architectural transaction patterns).
Una Transazione rappresenta una richiesta effettuata da un client che contenga due o più operazioni, che però svolgono una singola funzione logica e devono essere completate interamente o per nulla.
(esempio: si vogliono trasferire i soldi al conto di un amico → le operazioni sono togliere i soldi dal mio e inviarli al conto dell'amico, entrambe costituiscono la stessa transazione in quanto hanno lo stesso scopo logico di trasferire i soldi e o entrambe riescono o entrambe falliscono, non posso svolgerne una e lasciare l'altra fallita).
Le proprietà di una transazione sono racchiuse nell'acronimo ACID (Atomicity, Consistency, Isolation e Durability).

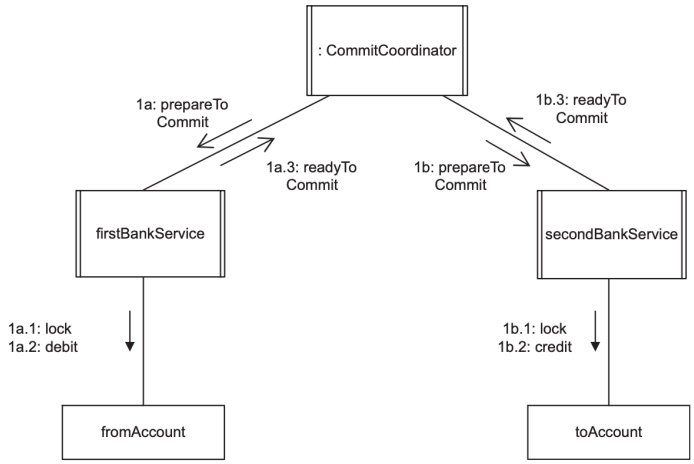
- Atomicità → la transazione pur essendo insieme di operazioni è vista come unità indivisibile di lavoro, o tutto (committed) o niente (rolled back)
- Consistenza → quando si esegue una transazione (sia di successo che non) il sistema deve essere in uno stato consistente
- Isolation → ogni transazione deve essere eseguita in modo isolato e quindi non essere compromessa da altre transazioni
- Durability → gli effetti prodotti dalla transazione sono permanenti e devono quindi sopravvivere anche ad eventuali system failures.

Come garantire queste proprietà in un ambiente distribuito basato su servizi?
Usiamo dei casi di studio per capire come e dove applicare questi pattern.
L'esempio tipico di una transazione è la transazione bancaria.
Quando avviene un trasferimento di denaro (es. bonifico da conto corrente a conto corrente) ciò che deve avvenire è che i soldi del conto corrente di origine vengano scalati e quelli di arrivo aggiornati. Ciò che non può e non deve succedere è che vengano scalati i soldi ma non aggiunti all'altro conto corrente e viceversa.
Nell'esempio specifico il primo conto è savings account (soldi per investimenti) e il secondo checking account (conto utilizzato normalmente per fare bonifici).
La transazione consisterà nelle due operazioni debit sul primo e credit sul secondo e deve essere o committed (i soldi sono trasferiti con successo) o aborted (nessuna delle due operazioni avviene mantenendo lo stato del sistema consistente).
Per garantire le proprietà di questa transazione si utilizza il protocollo (anche in basi di dati) del Two Phase Commit Protocol.

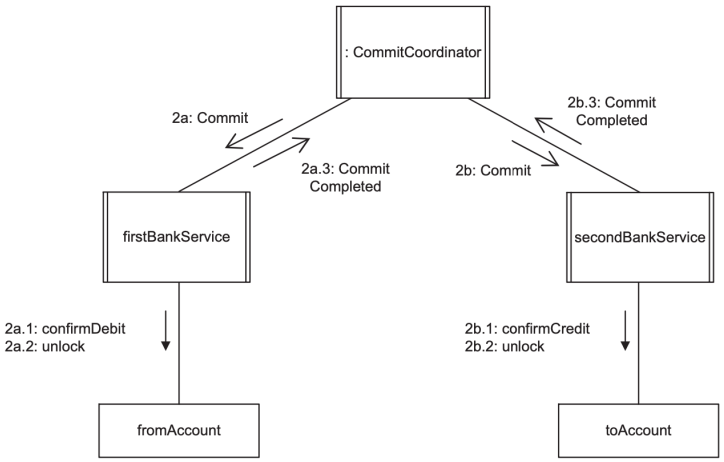
Vi sono due servizi nella transazione di trasferimento bancario ognuno su un'interfaccia di rete differente:

- firstBankService → per prelevare i soldi dal saving account
- secondBankService → per permettere il deposito del denaro sul secondo conto.

Per coordinare corettamente le attività svolte dai due servizi dobbiamo coinvolgere anche un altro oggetto che svolga il ruolo di coordinatore, il CommitCoordinator.
Vediamo come funzionano le due fasi.



La transazione è gestita in modo centralizzato per cui sarà il commitcoordinator a gestire l'ordine della transazione.
In figura si vede il comunication diagram (ossia il collaboration diagram come è stato rinominato in UML 2).
Il commitcoordinator invia un primo messaggio 1a e 1b ai due servizi coinvolti in cui gli comunica di prepararsi al commit. A fronte di questa richiesta i due servizi effettuano 1a.1 e 1b.1 un lock per garantire l'Isolation e quindi la Consistency (non si vuole che altre transazioni operino mentre i due servizi effettuano l'operazione sui due conti). A questo punto il firstBankService farà l'operazione di debito e l'altro di accredito. Se le operazioni vanno a buon fine (quindi il lock ha successo) i servizi inviano al cooridnator un readyToCommit, in assenza di questo messaggio la transazione è abortita.
Se invece a buon fine si passa alla seconda fase.



Viene inviato commit dal coordinatore, confermata l'operazione di addebito e di accredito (2a.1, 2b.1), si libera il lock e viene completata la transazione.

Vediamo ora esempi di transazioni più complesse, partendo dalla Compound Transaction per poi descriverne il Pattern da adottare.
La compound transaction è una transazione composita, ossia costituita da un insieme di sottotransazioni. Mentre una transazione singola (flat) come quella della banca si basava sulla regola "o tutto o niente", nel caso di una compound si parla di una transazione costituita da singole sottotransazioni → in caso qualcosa vada storto si cerca di salvare il salvabile, ossia se la seconda transazione non va a buon fine ma la prima si si farà un rollback "pariziale", ossia rollback solo sulla seconda transazione.
Per illustrare questo tipo di transazione facciamo un esempio concreto: un agente di viaggio deve pianificare il viaggio per un cliente: è necessario prenotare il biglietto aereo, poi prenotare l'albergo e la macchina a noleggio. Piuttosto che vedere questa compound transaction come operazione indivisibile per cui o tutto o niente posso vederla come costituita di tre sottotransazioni in modo che se riesco a fare la prima ma non la seconda mi salvo la prima.

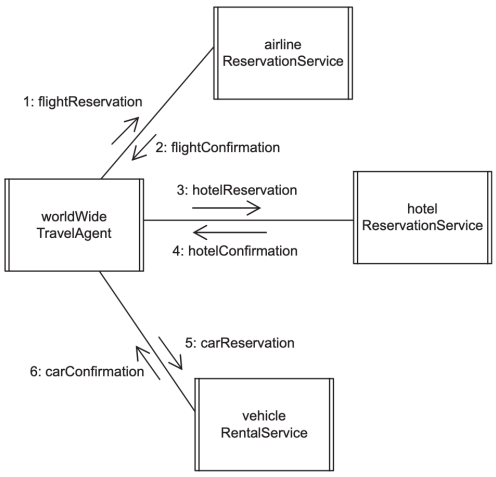
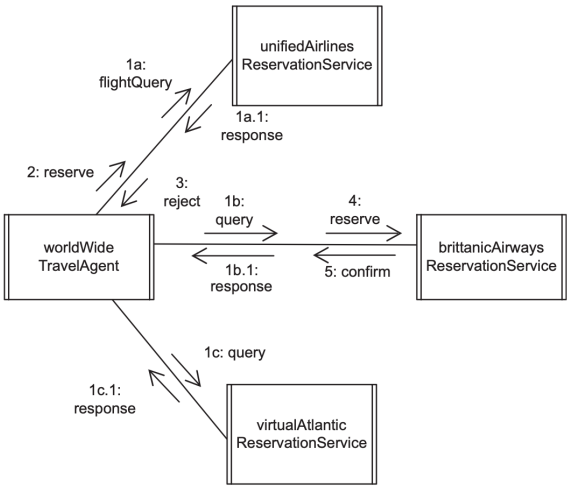
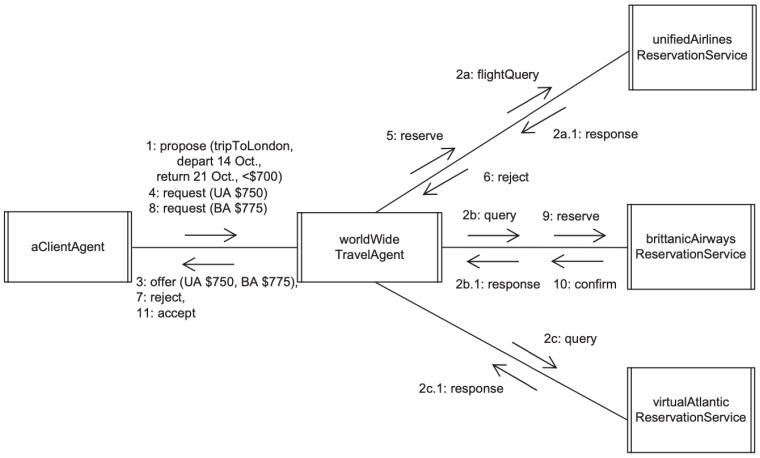


Figura su: *Example Compound Transaction Pattern*

Qui il caso in cui tutte e tre le sottotransazioni hanno esito positivo. Nel caso in cui una singola sottotransazione non abbia esito positivo, si salvano le prenotazioni che hanno avuto successo.
Questa transazione diventa ancora più complessa quando si mette in mezzo anche l'elemento umano, si parla di Long Living Transaction.
In queste transazioni si ha il così detto human in the loop per cui si deve tener conto di possibili ritardi dovute alla decisione di persone coinvolte nelle operazioni.
Il pattern in questo caso prevede di organizzare la transazione distinguendo in essa due o più sottotransazioni separate proprio dall'elemento umano.
Un esempio concreto può essere il considerare una prenotazione aerea in cui c'è il coinvolgimento di un essere umano, dove l'utente fa una ricerca per capire i posti disponibili in un certo volo, scegliere un posto disponibile e confermare la prenotazione. Per effettuare la scelta però l'utente umano impiega del tempo, e in questo periodo può succedere che il posto scelto dall'utente venga prenotato da qualcun altro. Quindi prima di riconfermare la transazione quindi è necessario un recheck



Nell'esempio in figura il servizio che funge da intermediario tra utente e compagnia aerea (TravelAgent) chiede a 3 compagnie aeree (1a, 1b, 1c) di mostrare i posti disponibili in un certo volo.
Ottenuta la risposta l'utente decide di prenotare il posto in unifiedAirlines, (2. reserve), il servizio fa un recheck che ha esito negativo e quindi invia 3. reject all'utente che dovrà rivolgersi quindi ad un'altra compagnia aerea (in questo caso britannic) prenotando e in questo caso il recheck ha esito positivo e quindi la transazione ha successo.
Ultimo caso che consideriamo è quello del Pattern di Negoziazione.
Anche chiamato Agent Based Negotiation in quanto subentra la figura di un servizio che lavora per conto dell'utente (l'utente si affida al servizio chiedendogli di far qualcosa al posto suo).
Es. riguardo la transazione per prenotazione posti vista prima si immagina un servizio che permetta, sapendo che devo andare a new york tot data, di trovare le offerte disponibili (senza che io utente vada a vedere il sito di ogni compagnia aerea).
Si ha quindi un client agent che lavora per conto dell'utente, egli interagirà con il service agent per soddisfare le esigenze del cliente.
Il service agent mostra una lista di offerte al client agent che si avvicinano a soddisfare la sua richiesta. A questo punto il client agent lavorando per conto dell'utente può decidere se rifiutare/accettare una richiesta etc… (da qui negoziazione)
In particolare il client agent svolge tre tipi di operazioni:
- Proposta di servizio (es. cerca volo roma new york a meno di 1000 euro)
- Richiesta servizio se a fronte dell'offerta (risposta alla proposta) del service agent è soddisfatto
- Rifiutare il servizio altrimenti
Invece dal punto di vista del service agent:
- Offrire un servizio di fronte alla proposta di servizio del client agent
- Rifiutare una richiesta/proposta del client agent in base alle disponibilità
- Accettare la richiesta/proposta del client agent.
La principale differenza tra richiesta e proposta è che la seconda è negoziabile (es. cerco il volo da roma a ny a meno di 1000 euro ma sono disposto a vedere anche offerte di prezzo lievemente superiori), mentre la richiesta è imperativa.



In questo esempio 1. Il clientagent invia una proposta (negoziabile!) per viaggio a londra il 14 ottobre e ritorno il 21 che costi meno di 700 euro.
A fronte di questa richiesta il serviceagent si confronta con diversi servizi offerti da compagnie aeree inviando in modo concorrente le richieste (2a, 2b, 2c).
Il messaggio che torna al clientagent è una offer con le soluzioni migliori trovate dal serviceagent (unifiedairlines a 750 euro o britannic a 775).
Il client invia quindi un messaggio di richiesta (non negoziabile!) (4) per prenotare UA, ma nel frattempo biglietto già acquistato da qualcun altro quindi reject dalla compagnia al serviceagent e reject dal serviceagent al clientagent (7).
Allora il clientagent richiede quello a 775 di BA, in questo caso la reserve da parte del serviceagent ha successo → il serviceagent risponde con accettazione finale alla richiesta non negoziabile del clientagent.
I servizi sono quindi elementi che interagiscono tramite scambio di messaggi, ed è quindi fondamentale come visto saper gestire la logica di controllo delle applicazioni.
Un altro problema fondamentale è saper progettare correttamente l'interfaccia dei servizi (stesso problema anche per le classi nell'analisi dei requisiti object oriented, ma qua ancora più forte perché mentre la classe è un elemento ben identificato sappiamo come la granularità dei servizi sia molto variabile e non nota all'utente del servizio).
Possiamo sfruttare UML 2 e in particolare il concetto di classi strutturate per progettare un servizio come se fosse una classe avente un'interfaccia, che però non è necessariamente monolitica ma composita e quindi costituita eventualmente da ulteriori servizi al suo interno.
Quindi da una parte progettazione del servizio a livello strutturale, dall'altra a livello comportamentale (dinamica attraverso cui questi servizi interagiscono), dove una parte fondamentale è come coordinare questi servizi (infatti a differenza della architettura component based dove i framework permettevano di guidare anche l'integrazione tra le varie componenti, nelle service oriented i servizi sono indipendenti e se me ne servono diversi devo saperli coordinare).
In questo contesto di Service Coordination in SOA si utilizzano due termini per indicare due tipi di coordinamento tra servizi: orchestrazione e coreografia.

- Orchestrazione → si ha un elemento centralizzzato che coordina l'esecuzione dei servizi (simile a quanto visto nel caso dei two phase commit protocol)
- Coreografia → gli aspetti di coordinamento sono decentralizzati (distribuiti)

Nella progettazione di applicazioni service oriented tuttavia è difficile che venga usata piena orchestrazione o piena coreografia, tipicamente si usano entrambi gli approcci. Per questa ragione in generale con il termine coordinazione si vuole intendere il controllo e la sequenza delle azioni tra i servizi, indipendentemente dal fatto che siano centralizzati o distribuiti.
I pattern delle transazioni visti in precedenza possono essere usati per la coordinazione tra servizi.
Con ciò la sottofase di progettazione preliminare è conclusa.
Tornando all'inizio della fase di progettazione avevamo detto come sia costituita da due sottofasi: la prima in cui definire l'architettura software (di cui ci siamo già occupati) e la seconda di progettazione dettagliata (tutti gli elementi da progettare in modo dettagliato sia dal punto di vista strutturale che comportamentale).
Nel nostro caso avendo usato un approccio OO fin dall'analisi dei requisiti parleremo di una sottofase di progettazione dettagliata che farà uso di un approccio OO → Detailed OOD.
OOD rappresenta una diretta continuazione di OOA, dove secondo il principio di stepwise refinement raffineremo sempre più in dettaglio quanto trovato nella prima parte del corso, in particolare occupandoci di come far collaborare gli oggetti affinché vengano resi disponibili i servizi che l'applicazione dovrebbe fornire.
Mentre l'OOA era guidata dai casi d'uso quindi l'OOD è guidata dalla collaborazione degli oggetti.
La collaboration è primitiva UML (per questo in UML 2 il collaboration diagram rinominato in comunication diagram) ed è costituita da due parti fondamentali (qui torniamo a quanto visto nella fase di analisi orientata agli oggetti, dove il modello del sistema software era costituito da vari sottomodelli: dati, comportamentale e dinamico):

- una parte comportamentale: rappresenta la dinamica che mostra come gli elementi collaborano tra loro. Si definirà facendo uso dei comunication diagrams.
- una parte strutturale: rappresenta gli aspetti statici della collaborazione ed è definita facendo uso del class diagram, facendo però anche uso dell'estensione fornita da UML 2 che permette di progettare la struttura della classe in modo gerarchico (quindi useremo i composite structure diagram)

Uno degli aspetti più importanti di cui tener conto in fase di progettazione a livello comportamentale è la gestione del controllo (la logica che l'applicazione utilizza per eseguire le funzioni che mette a disposizione). Questa parte ricade nel Control Management.