

INGEGNERIA DEL SOFTWARE

CICLO DI VITA DEL SOFTWARE (prevede 3 stadi, 6 fasi)

Stadio1 = sviluppo, con fasi:

1. Requisiti
2. Specifiche (analisi dei requisiti)
3. Pianificazione (il modulo 1 di ISW si ferma qui)
4. Progetto
5. Codifica
6. Integrazione

Stadio2 = manutenzione, copre circa il 60% dei costi del ciclo di vita

Stadio3 = dismissione

+TESTING, che non è una fase separata ma ha luogo durante l'intero sviluppo. Ci sono due modi di fare testing:

- 1) Verifica (alla fine di ogni fase)
Risponde alla domanda "la fase è stata ben svolta?"
- 2) Validazione (alla fine dello sviluppo)
Risponde alla domanda "il prodotto finale è buono?"

DRE = Defect Removal Efficiency, si riferisce alla percentuale di difetti trovati prima del rilascio del prodotto software.

ASPETTI DI COSTO DEL PRODOTTO SW

Ci si riferisce ai tre costi seguenti:

- Costo verso dimensione, si intende la relazione tra costo e dimensione del sw: il costo è proporzionale al quadrato della dimensione: $c=aS^2$
- Costo verso repliche, produrre una replica non costa niente
- Costo verso ampiezza di mercato
Vendere un prodotto di size doppio per il mercato richiede:
un prezzo 4 volte superiore a parità di ampiezza di mercato
un mercato di ampiezza 4 volte maggiore a parità di prezzo

DEFINIZIONI

- 1) Prodotto software = codice + documentazione
- 2) Artefatto = prodotto software intermedio
- 3) Codice = prodotto software finale
- 4) Sistema software = insieme organizzato di prodotti software (o insieme di hardware software di un prodotto come secondo significato)
- 5) Cliente = soggetto che ordina il prodotto Sw
- 6) Sviluppatore = soggetto che lo produce
- 7) Utente = soggetto che lo usa
- 8) Sw interno = cliente e sviluppatore coincidono
- 9) Sw a contratto = cliente e sviluppatore sono soggetti differenti

ASPETTI DI AFFIDABILITA' (SW REABILITY)

Informalmente: credibilità del prodotto software

Formalmente: probabilità che il prodotto software lavori "correttamente" in un determinato intervallo temporale (mission time)

DIFETTO , GUASTO , ERRORE

Difetto (defect) = anomalia presente in un prodotto Sw

Guasto (failure) = comportamento anomalo del prodotto Sw dovuto alla presenza di un difetto

Errore = azione errata di chi (per ignoranza, distrazione, etc) introduce un difetto nel prodotto Sw

AFFIDABILITA' SOFTWARE

→ La regola 10-90

Esperimenti condotti su programmi di notevoli dimensioni mostrano che:

- Il 90% del tempo di esecuzione totale è speso eseguendo il solo 10% delle istruzioni

Detto 10% è chiamato:

- core (nucleo) del programma.

L'affidabilità osservata dipende da:

- come è usato il prodotto

- in termini tecnici, dal suo profilo operativo (operational profile)

L'affidabilità di un prodotto Sw dipende dall'utente

→ Confronto tra affidabilità hardware e affidabilità software

I guasti Sw:

- sono dovuti alla presenza di difetti nei programmi

- il software non si consuma

I guasti Hw sono quasi sempre dovuti a:

- consumo/deterioramento dei componenti

- qualche componente non si comporta più come specificato

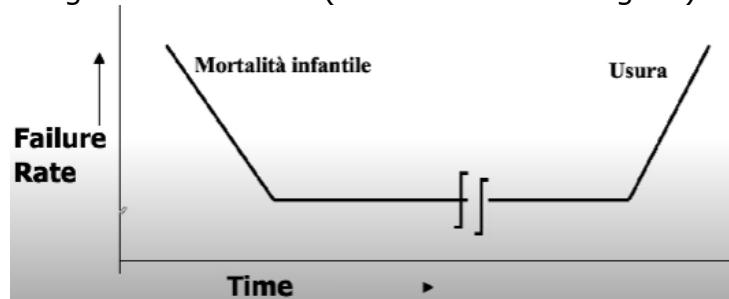
- qualche componente si rompe

Conclusione: a causa della differenza negli effetti dei difetti le metriche usate per l'affidabilità hw non sono estendibili al sw.

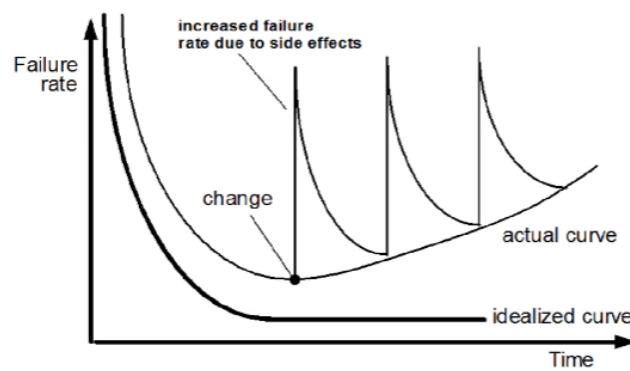
Obiettivo affidabilità hw = stabilità (cioè tenere la frequenza di guasto costante)

Obiettivo affidabilità sw = crescita di affidabilità (cioè far decrescere la frequenza del guasto).

Andamento frequenza di guasto hardware ("curva vasca da bagno"):



Andamento frequenza di guasto software:



DISPONIBILITA' (SOFTWARE AVAILABILITY)

% del tempo che il Sw è risultato usabile nel corso della sua vita.

Dipende

- dal numero di guasti che si verificano

- dal tempo necessario a ripararli

NASCITA ISW

→ Nel corso degli anni la produzione del software ha seguito varie fasi:

- fase di abilità, nella quale prevalgono gli aspetti di lavoro individuale e creativo
- fase artigianale, nella quale il software viene prodotto da piccoli gruppi specializzati, spesso di alto livello di professionalità
- fase industriale, nella quale l'attività di sviluppo e manutenzione del software viene pianificata e coordinata, ed il lavoro del progettista viene sempre più supportato da strumenti automatici.

→ Il termine «ingegneria del software» viene coniato per la prima volta nel 1968(Germania)
Lo standard IEEE Std. 610.12 (1990) ha formulato una definizione completa:

1. Applicazione di un approccio sistematico, disciplinato e misurabile allo sviluppo, esercizio e manutenzione del software, cioè applicazione di principi ingegneristici al software
2. Studio degli approcci di cui al punto 1.

→ Il software può essere considerato come un insieme di elementi che formano una "configurazione" che include:

- programmi
- documenti
- dati multimediali

Caratteristiche del software:

- il software va "ingegnerizzato"
- il software non si consuma
- il software è complesso, invisibile, si conforma, si cambia

→ I metodi e le tecniche di ingegneria del sw hanno lo scopo di rispondere alle domande seguenti:

- i. Come assicurare la qualità del software che si produce?
- ii. Come bilanciare la "domanda" crescente pur mantenendo il controllo del budget a disposizione?
- iii. Come aggiornare applicazioni vecchie (legacy) ma ancora necessarie?
- iv. Come evitare tempi di consegna più lunghi di quelli pianificati?
- v. Come applicare con successo le nuove tecnologie software?

→ I miti da sfatare del software:

- i. In caso di ritardo, basta aumentare il numero di programmatore
- ii. Una descrizione generica è sufficiente a scrivere i programmi. Eventuali modifiche si possono facilmente effettuare in seguito
- iii. Una volta messo in opera il programma, il lavoro è finito
- iv. Non c'è modo di valutare la qualità fino a quando non si ha a disposizione il prodotto finale
- v. L'ingegneria del software è costosa e rallenta la produzione

LEZ4 - 12/10/2023

PROCESSO SOFTWARE

→ Definizione: Per processo software si intende una serie di attività necessarie alla realizzazione del prodotto software nei tempi, con i costi e con le desiderate caratteristiche di qualità.

→ Fasi del processo:

Il processo software segue un ciclo di vita che si articola in 3 stadi (sviluppo, manutenzione, dismissione).

Nel primo stadio si possono riconoscere due tipi di fasi:

- fasi di tipo definizione, si occupano di "cosa" il software deve fornire. Si definiscono i requisiti, si producono le specifiche

- fasi di tipo produzione, produzione definiscono "come" realizzare quanto ottenuto con le fasi di definizione. Si progetta il software, si codifica, si integra e si rilascia al cliente

Il secondo stadio, lo stadio di manutenzione, è a supporto del software realizzato e prevede fasi di definizione e/o produzione al suo intero.

Durante ogni fase si procede ad effettuare il testing di quanto prodotto, mediante opportune tecniche di verifica e validazione (V&V) applicate sia ai prodotti intermedi che al prodotto finale.

→ Ci sono diversi tipi di manutenzione:

1. Manutenzione correttiva, che ha lo scopo di eliminare i difetti (fault) che producono guasti (failure) del software
2. Manutenzione adattativa, che ha lo scopo di adattare il software ad eventuali cambiamenti a cui è sottoposto l'ambiente operativo per cui è stato sviluppato
3. Manutenzione perfettiva, che ha lo scopo di estendere il software per accomodare funzionalità aggiuntive
4. Manutenzione preventiva (o software reengineering), che consiste nell'effettuare modifiche che rendano più semplici le correzioni, gli adattamenti e le migliorie

CICLO DI VITA

Intervallo di tempo che intercorre tra l'istante in cui nasce l'esigenza di costruire un prodotto software e l'istante in cui il prodotto viene dismesso; include le fasi di definizione dei requisiti, specifica, pianificazione, progetto preliminare, progetto dettagliato, codifica, integrazione, testing, uso, manutenzione e dismissione.

Nota: tali fasi possono sovrapporsi o essere eseguite in modo iterativo

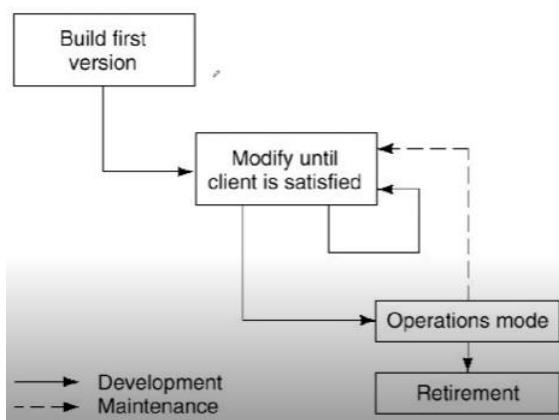
MODELLI DI CICLO DI VITA

Il modello del ciclo di vita del software specifica la serie di fasi attraverso cui il prodotto software progredisce e l'ordine con cui vanno eseguite, dalla definizione dei requisiti alla dismissione.

La scelta del modello dipende dalla natura dell'applicazione, dalla maturità dell'organizzazione, da metodi e tecnologie usate e da eventuali vincoli dettati dal cliente. L'assenza di un modello del ciclo di vita corrisponde ad una modalità di sviluppo detta "build & fix" (o "fix-it-later"), in cui il prodotto software viene sviluppato e successivamente rilavorato fino a soddisfare le necessità del cliente.

Alcuni modelli di ciclo di vita sono: Build&Fix, modello waterfall, Rapid prototyping model, ...

MODELLO BUILD&FIX



MODELLO WATERFALL

In ingegneria del software, il modello a cascata è il più tradizionale modello di ciclo di vita del software. Secondo questo modello, il processo di realizzazione del software è strutturato in una sequenza lineare di fasi, che comprende:

- analisi dei requisiti
- progettazione
- sviluppo
- collaudo
- manutenzione

I suoi capisaldi sono i seguenti: il processo di sviluppo è diviso in fasi sequenziali; ogni fase produce un output che è usato come input per la fase successiva; ogni fase del processo viene documentata.

MODELLO A PROTOTIPAZIONE RAPIDA

In questo modello cambia rispetto al waterfall solo il blocchetto iniziale denominato Rapid prototype. Ha lo scopo di migliorare la raccolta dei requisiti e convalidare più facilmente i requisiti definiti dall'utente. Nel caso del sw il prototipo è una versione prototipale del sw che di solito contiene soltanto l'interfaccia del prodotto senza implementare il resto, utile per capire cosa vuole e cosa si aspetta l'utente. Il beneficio principale si trova nell'evitare incomprensioni tra utenti e sviluppatori, ma sta anche nel fatto che gli utenti possono allenarsi fin da subito a utilizzare il prodotto.

Processo che porta allo sviluppo di un prototipo: stabilire obiettivi del prototipo, definire le sue funzionalità, svilupparlo, valutarlo. Tutto questo va fatto nell'arco di 10 giorni circa.

THROW-AWAY PROTOTYPING (PROTOTIPO USA E GETTA)

Svantaggi: aspettative da parte del cliente con pressione nei confronti degli sviluppatori (rilascio prototipo dopo pochi giorni ma poi passano mesi/anni per il prodotto finale); il prototipo non ha documentazione; la struttura potrebbe cambiare di molto rispetto al prototipo.

LEZ5 – 16/10/2023

VISUAL PROGRAMMING

Programmazione in cui il prototipo viene sviluppato creando un'interfaccia utente da elementi standard. Problemi con il visual programming possono essere:

- difficile da coordinare in team
- nessuna architettura sw esplicita
- il programma può causare problemi di manutenibilità

PROCESS ITERATION (ITERAZIONE DEL PROCESO)

È una caratteristica utilizzata da approcci non monolitici come Waterfall ma iterativi.

I requisiti si evolvono sempre nel corso di un progetto quindi l'iterazione in cui vengono rielaborate le fasi precedenti fa sempre parte del processo per i prodotti di grandi dimensioni.

L'iterazione può essere applicata a uno qualsiasi dei modelli di processo generici.

→ Due approcci (correlati) sono:

- 1) Sviluppo incrementale
- 2) Sviluppo a spirale

SVILUPPO INCREMENTALE (INCREMENTAL DEVELOPMENT)

Il prodotto viene sviluppato e consegnato in incrementi dopo aver stabilito un'architettura generale. Gli utenti possono sperimentare con gli incrementi consegnati mentre altri sono in fase di sviluppo. Pertanto, questi fungono da "forma del prototipo".

Lo sviluppo incrementale è destinato a combinare alcuni dei vantaggi di prototipazione ma con un processo più gestibile e una struttura migliore.

➔ **MODELLO INCREMENTALE:**

Il prodotto software viene sviluppato e rilasciato per incrementi (build) successivi.

Il modello incrementale include aspetti tipici del modello basato su rapid prototyping.

Si rivela efficace quando il cliente vuole continuamente verificare i progressi nello sviluppo del prodotto e quando i requisiti subiscono modifiche.

Può essere realizzato in due versioni alternative:

- versione con overall architecture
- versione senza overall architecture (più rischiosa)

→ IMPATTO SUI COSTI DEL SOFTWARE: all'aumentare del numero di build il costo dei build diminuisce ma il costo della loro integrazione aumenta. Al diminuire dei build il loro costo aumenta ma il costo della loro integrazione diminuisce. La regione di costo minimo si trova col giusto bilanciamento del numero di build.

→ CONFRONTO TRA MODELLO A CASCATA E MODELLO INCREMENTALE

Modello a cascata

- Requisiti "congelati" al termine della fase di specifica
- Feedback del cliente solo una volta terminato lo sviluppo
- Fasi condotte in rigida sequenza (l'output di una costituisce input per la successiva)
- Prevede fasi di progetto dettagliato e codifica dell'intero prodotto
- Team di sviluppo costituito da un numero elevato di persone

Modello incrementale

- Requisiti suddivisi in classi di priorità e facilmente modificabili
- Continuo feedback da parte del cliente durante lo sviluppo
- Fasi che possono essere condotte in parallelo
- Progetto dettagliato e codifica vengono effettuate sul singolo *build*
- Differenti team di sviluppo, ciascuno di piccole dimensioni

MODELLO A SPIRALE

Scomponete il processo di sviluppo in quattro fasi multiple, ciascuna ripetuta più volte.

Le 4 fasi sono: Pianificazione, Analisi dei rischi, Sviluppo, Verifica(Testing)

Nel modello iterativo sono quindi presenti le stesse fasi del modello a cascata, ma i tempi sono più ristretti e dalla fase di testing si torna poi a quella di pianificazione per applicare eventuali correzioni al risultato dello sviluppo.

RISK MANAGEMENT (GESTIONE DEL RISCHIO)

Va sempre fatto. La gestione del rischio si occupa di identificare rischi e l'elaborazione di piani per ridurne al minimo l'effetto su un progetto.

Categorie di rischio:

- I rischi del progetto influiscono sulla pianificazione o sulle risorse
- I rischi del prodotto incidono sulla qualità o sulle prestazioni software in fase di sviluppo
- I rischi aziendali riguardano l'organizzazione in via di sviluppo o acquisto del software

Il processo di risk management prevede: Identificazione del rischio, Analisi e valutazione del rischio, Pianificazione del rischio (elaborare piani per evitare o minimizzare gli effetti del rischio), Monitoraggio del rischio.

LEZ6 – 19/10/2023

IDENTIFICAZIONE DEI RISCHI

Possiamo avere diversi tipi di rischi:

- 1) Rischi tecnologici, ad esempio componenti sw difettose
- 2) Rischi legati alle persone(risorse umane), ad esempio mancanza di competenza
- 3) Rischi organizzativi, ad esempio cambio del budget
- 4) Rischi sui tool, ad esempio strumenti inefficienti
- 5) Rischi sui requisiti, ad esempio i requisiti cambiano in modo significativo
- 6) Rischi su stime effettuate, ad esempio sottostima della dimensione del sw

ANALISI DEI RISCHI (RISK ANALYSIS)

La probabilità di rischio può essere: Molto bassa (<10%)

Bassa (10-25%)

Moderata (25-50%)

Alta (50-75%)

Molto alta (75%<)

Gli effetti dei rischi possono essere catastrofici, seri, tollerabili o insignificanti.
Ad esempio: problemi finanziari che ricadono sul progetto hanno probabilità bassa ma effetti catastrofici.

→ A partire dai rischi catastrofici/più seri si passa ad attività di risk planning (rischi "top ten").

RISK PLANNING

Considerare ogni rischio e sviluppare una strategia per gestire tale rischio.

→ Strategie di evitamento (avoidance strategies):

- cerco di diminuire la probabilità dei rischi per provare ad evutarli

→ Strategie di minimizzazione (minimization strategies):

- cerco di ridurre l'impatto del rischio qualora dovesse verificarsi

→ Piani di emergenza (contingency plane):

- In caso di rischio, i piani di emergenza sono strategie per far fronte a tale rischio

RISK MONITORING

Valutare regolarmente ogni rischio per vedere se la probabilità che accada sta aumentando, diminuendo o è stabile. Per eseguire la valutazione, esaminare i fattori di rischio.

Valutare inoltre se gli effetti del rischio sono cambiati (in tal caso tornare a analisi).

Ciascun rischio principale dovrebbe essere discusso in riunioni sullo stato di avanzamento.

ALTRI MODELLI SONO:

1) Modello di ingegneria simultanea (o concorrente)

Ha come obiettivo la riduzione di tempi e costi di sviluppo, mediante un approccio concorrente per la creazione del progetto, le fasi di sviluppo coesistono invece di essere eseguite in sequenza.

2) Modello basato su metodi formali

Comprende una serie di attività che conducono alla specifica formale matematica del software, al fine di eliminare ambiguità, incompletezze ed inconsistenze e facilitare la verifica dei programmi mediante l'applicazione di tecniche matematiche, viene enfatizzata la possibilità di rilevare i difetti del software in modo più tempestivo rispetto ai modelli tradizionali.

IL MODELLO MICROSOFT

La Microsoft ha dovuto affrontare problemi di incremento della qualità dei prodotti software, problemi di riduzione di tempi e costi di sviluppo.

Per cercare di risolverli si è adottato un processo che è al tempo stesso iterativo, incrementale e concorrente e che permette di esaltare le doti di creatività delle persone coinvolte nello sviluppo di prodotti software. L'approccio usato attualmente da Microsoft è noto come "**synchronize-and-stabilize**" ed è basato su:

- **sincronizzazione** quotidiana delle attività svolte da persone che lavorano sia individualmente che all'interno di piccoli team (da 3 a 8 persone), mediante
- **assemblaggio** dei componenti software sviluppati (anche parzialmente) in un prodotto (daily build) che viene testato e corretto
- **stabilizzazione** periodica del prodotto in incrementi(milestone) successivi durante l'avanzamento del progetto, piuttosto che un'unica volta alla fine.

LEZ7 – 23/10/2023

CICLO DI SVILUPPO A 3 FASI

Fase1: Planning phase (pianificazione), serve a definire la visione del prodotto, le specifiche e la pianificazione.

Fase2: Development phase (fase di sviluppo), sviluppo di feature in 3/4 sottoprogetti in cui i program managers coordinano l'evoluzione delle specifiche e in cui gli sviluppatori e tester lavorano in coppia facendo test continuamente.

Fase3: Stabilization phase (fase di stabilizzazione), test interni(alpha test) ed esterni completi(beta test), prodotto finale, stabilizzazione, rilascio. Più nello specifico i program manager coordinano OEM(realizzatori a proprio marchio del prodotto) e ISV(fornitori di software indipendenti) e monitorano il feedback del cliente. Gli

sviluppatori eseguono il debug finale e la stabilizzazione del codice. I tester ricreano e isolano gli errori. Per test interni si intendono gli alpha test approfonditi del prodotto completo all'interno dell'azienda. Per test esterni si intendono i beta test approfonditi del prodotto completo esterni all'azienda da siti "beta", come OEM, ISV, e gli utenti finali.

STRATEGIE E PRINCIPI:

1. Strategia per definire prodotto e processo: "considerare la creatività come elemento essenziale".

Principi di realizzazione:

- a) Dividere il progetto in milestone (da 3 a 4)
- b) Definire una "product vision" e produrre una specifica funzionale che evolverà durante il progetto
- c) Selezionare le funzionalità e le relative proprietà in base alle necessità utente
- d) Definire un'architettura modulare per replicare nel progetto la struttura del prodotto (si lavora con tanti piccoli team, quindi l'idea di allocare specifiche parti di prodotto ai vari team)
- e) Assegnare task elementari e limitare le risorse

2. Strategia per lo sviluppo e la consegna dei prodotti: "lavorare in parallelo con frequenti sincronizzazioni".

Principi di realizzazione:

- a) Definire team paralleli ed utilizzare daily build per la sincronizzazione
- b) Avere sempre un prodotto da consegnare, con versioni per ogni piattaforma e mercato
- c) Usare lo stesso linguaggio di programmazione all'interno dello stesso sito di sviluppo
- d) Testare continuamente il prodotto durante il suo sviluppo
- e) Usare metriche per il supporto alle decisioni

MILESTONES

Milestone è un termine inglese che letteralmente significa pietra miliare. Indica importanti traguardi intermedi nello svolgimento del progetto.

MODELLO DEL CICLO DI SVILUPPO SYNCH-AND-STABILIZE

Synchronize&stabilize è un modello di sviluppo del ciclo di vita del software che consente ai team di lavorare in modo efficiente in parallelo su diversi moduli applicativi individuali. Ciò viene fatto sincronizzando frequentemente il lavoro come individui e come membri di team paralleli e stabilizzando e/o effettuando periodicamente il debug del codice durante l'intero processo di sviluppo.

Questo approccio suddivide i progetti di grandi dimensioni in piccoli segmenti, che i team con competenze correlate possono gestire e completare(rilasciare) in modo efficiente.

CONFRONTO TRA MODELLI SYNCH-AND-STABILIZE E WATERFALL

Synch-and-Stabilize	Sequential Development
Product development and testing done in parallel	Separate phases done in sequence
Vision statement and evolving specification	Complete "frozen" specification and detailed design before building the product
Features prioritized and built in 3 or 4 milestone subprojects	Trying to build all pieces of a product simultaneously
Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)	One late and large integration and system test phase at the project's end
"Fixed" release and ship dates and multiple release cycles	Aiming for feature and product "perfection" in each project cycle
Customer feedback continuous in the development process	Feedback primarily after development as inputs for future projects
Product and process design so large teams work like small teams	Working primarily as a large group of individuals in a separate functional department

IL MODELLO NETSCAPE

Anche alla Netscape si è adottato un modello di tipo synchronize-and-stabilize, con opportuni adattamenti allo sviluppo di applicazioni Internet, si è puntato su:

- dimensione dello staff, in media 1 tester ogni 3 sviluppatori
- processo, che comprende
 - scarso effort di pianificazione (tranne che su prodotti server)
 - documentazione incompleta
 - scarso controllo sullo stato di avanzamento del progetto (lasciato all'esperienza e all'influenza dei project manager)
 - scarso controllo su attività di ispezione del codice (code review)
 - pochi dati storici per il supporto alle decisioni

Processo di sviluppo Netscape:

Step1= Analisi requisiti del prodotto e proposta di progetto.

- Meeting per il brainstorming di idee (marketing, sviluppo, executive)
- Generata visione del prodotto da ingegneri senior
- Un po' di progettazione e codifica da parte di ingegneri
- Compilazione documento sui requisiti del prodotto da parte del product manager, con l'aiuto dagli sviluppatori
- Revisione informale di questa specifica preliminare da parte degli ingegneri
- Specifiche funzionali avviate dagli ingegneri, a volte con l'aiuto di responsabili di prodotto
- Programmazione del budget plane compilato da addetti al marketing e ingegneri e discussione in modo informale con i dirigenti

Step2= Prima revisione esecutiva.

- I dirigenti esaminano il documento sui requisiti del prodotto, il programma e il budget; se necessario si aggiusta il piano

Step3= Inizio fase di sviluppo.

- Progettazione e codifica delle features, lavorazione architettura se necessario.
- Integrazione di build giornaliera
- Elenchi bug generati e correzioni avviate.

Step4= Revisione esecutiva provvisoria (se necessario).

- Le specifiche funzionali dovrebbero essere complete a questo punto
- Correzioni a metà corso nelle specifiche o nelle risorse del progetto, se necessarie
- Coordinamento con altri prodotti o progetti discussi, se necessario.
- Lo sviluppo continua...

Step5= Primo rilascio interno (alpha)

- Circa 6 settimane
- Lo sviluppo si interrompe temporaneamente
- Debug intensivo e testing codice esistente

Step6= Beta pubblica 1 o test sul campo 1

- Circa 6 settimane
- Ripetere fasi di sviluppo e test nel passaggio 5

Step7= Beta pubblica 2 e 3

- Circa 6 settimane per ogni beta

Step8= Codice completo

- Nessun altro codice aggiunto se non per risolvere bug, features complete

Step9= Test e rilascio finale

- Debugging finale e stabilizzazione della release candidata
- Riunione dirigenti
- Rilascio alla produzione e rilascio commerciale

AGILE METHODS

All'inizio degli anni 2000 si è notato che processi software attentamente pianificati sono troppo restrittivi per gli sviluppatori. Il termine metodo agile è stato introdotto per estendere il

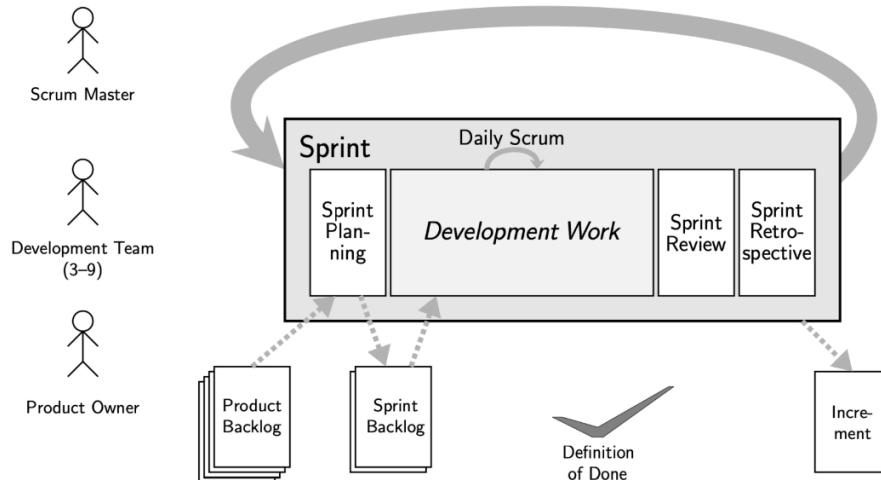
concepto originale di "iterativo" e "incrementale" a concetti come comunicazione intensiva all'interno del progetto, feedback rapido, poche regole esterne per il modo di lavorare, ecc. I valori e i principi comuni dei metodi agili sono riassunti nel Manifesto Agile.

Il Manifesto Agile (2001) definisce i valori agili e punta a valorizzare:

- Individui e interazioni rispetto a processi e strumenti
- Software funzionante rispetto a documentazione completa
- Collaborazione con il cliente rispetto a negoziazione del contratto
- Rispondere al cambiamento rispetto a seguire un piano

"Mentre c'è valore negli elementi a destra, diamo più valore agli elementi a sinistra".

SCRUM (metodo agile)



Scrum master = garantisce che la metodologia sia compresa e correttamente implementata dal team di sviluppo e dal proprietario del prodotto. Inoltre supporta il team aiutando tutti gli altri ad interagire con la squadra secondo le regole di Scrum.

Product owner = gestisce e aiuta a dare priorità ai requisiti da implementare come documentato nel backlog del prodotto

Development team = è responsabile dello sviluppo del prodotto e di tutti gli aspetti ad esso pertinenti (ad esempio progettazione, codifica e test)

Sprint: alla base della metodologia SCRUM c'è il concetto di sprint:

- uno sprint viene eseguito per fornire un nuovo incremento del software di lavoro e in genere richiede da 2 a 4 settimane.
- Si inizia con la riunione di pianificazione dello sprint in cui il team scrum trasferisce gli elementi da sviluppare dal backlog del prodotto nello sprint backlog.
- Durante lo sprint, il team di sviluppo lavora sull'incremento, con brevi riunioni scrum giornaliere (ovvero stand-up meeting) del team di sviluppo per sincronizzare il lavoro e affrontare eventuali problemi.
- Alla fine di uno sprint, l'incremento viene presentato al proprietario del prodotto
- Infine, viene eseguita la riunione retrospettiva dello sprint per identificare e pianificare eventuali miglioramenti per il prossimo sprint.

Definition of done: scrum richiede una "definition of done" in cui il team di sviluppo definisce da solo cosa significa per un item work essere "done" (prima che possa essere integrato nel ramo principale). Requisiti minimi tipici inclusi nella definition of done includono un numero adeguato di casi di test, oltre a controllare l'integrazione del nuovo codice per garantire che non rompa il ramo di sviluppo principale. La definition of done richiede anche che il codice sia stato documentato adeguatamente, dove il team definisce da solo cosa significa adeguatamente.

LEZ8 – 26/10/2023

USER STORIES

È una pratica comune, utilizzata nello sviluppo agile, spesso in combinazione con Scrum (ma non definito nella Guida Scrum).

Una user storie è un formato per descrivere i requisiti dell'utente come "storia". Dovrebbe essere breve, in genere solo una frase, e descritto dal punto di vista dell'utente che utilizza un modello comune, ad esempio:

Come <ruolo> voglio <goal> in modo che <benefit>

Esempio: "In qualità di ingegnere di processo, voglio vedere le dipendenze tra le diverse fasi del processo in modo che io possa facilmente verificare e convalidare".

User stories di grandi dimensioni vengono suddivise in più piccole chiamate "epics".

CAPABILITY MATURITY MODEL (CMM)

È un modello predisposto dal SEI (software engineering institute), che serve a determinare il livello di maturità del processo software di un'organizzazione (ovvero una misura dell'efficacia globale dell'applicazione di tecniche di ingegneria del sw).

Questo modello è basato su un questionario ed uno schema valutativo a cinque livelli, ogni livello comprende tutte le caratteristiche definite per il livello precedente.

La maggior parte delle aziende sono certificate a livello 3.

I 5 livelli sono:

livello 1 = livello iniziale, ci si affida molto alle persone

livello 2 = Gestione della configurazione software, Garanzia della qualità del software,
Gestione del subappalto di software, Monitoraggio e supervisione del
progetto software, Progettazione software, Gestione dei requisiti

livello 3 = Peer review, Coordinamento tra gruppi, Ingegneria dei prodotti software,
Gestione software integrata, Programma di formazione, Definizione del
processo organizzativo, Focus sul processo organizzativo

livello 4 = Gestione della qualità del software, Gestione quantitativa dei processi

livello 5 = Gestione del cambiamento di processo, Gestione del cambiamento
tecnologico, Prevenzione dei difetti

Il CMM associa a ogni livello di maturità alcune KPA (Key Process Area), tra le 18 definite, che descrivono le funzioni che devono essere presenti per garantire l'appartenenza ad un certo livello.

Ogni KPA è descritta rispetto a:

- obiettivi
- impegni e responsabilità da assumere
- capacità e risorse necessarie per la realizzazione
- attività da realizzare
- metodi di "monitoring" della realizzazione
- metodi di verifica della realizzazione.

REQUISITI SOFTWARE (software requirements)

Descrizione dei servizi che un sistema software deve fornire, insieme ai vincoli da rispettare sia in fase di sviluppo che durante la fase di operatività del software.

Definizione rigorosa definita dallo standard IEEE Std 610.12 (1990):

- (A) Una condizione o una capacità che è necessaria a un utente per risolvere un problema o per raggiungere un certo obiettivo
- (B) Una condizione che deve essere posseduta o da un sistema o da un componente del sistema per soddisfare un contratto, una norma, una specifica o un altro documento imposto
- (C) Una rappresentazione documentata di una condizione o di una capacità come nella definizione (A) o (B)

I requisiti vengono generati applicando un processo di ingegneria dei requisiti (requirements engineering).

Tipi di requisiti: requisiti utente e requisiti di sistema.

- ➔ Requisiti utente (user requirements) = descrizione in linguaggio naturale, con eventuale aggiunta di diagrammi, dei servizi che il sistema deve fornire e dei vincoli operativi; sono scritti per (e con) il cliente.
- ➔ Requisiti di sistema (system requirements) = specificati mediante la stesura di un documento strutturato che descrive in modo dettagliato i servizi che il sistema software deve fornire. Il documento risultante costituisce un "contratto" tra cliente e fornitore

NOTA BENE: Quando si parla di requisiti utente si parla di "definizione" dei requisiti utente, per quanto riguarda i requisiti di sistema invece si usa il termine "specifiche" dei requisiti.

DEFINIZIONE DEI TERMINI(da non dimenticare):

cliente (customer, client) = la persona od organizzazione che paga per la fornitura di un prodotto software
fornitore (supplier, contractor) = la persona od organizzazione che produce software per il cliente
utente finale (end-user) = la persona che interagisce direttamente con il prodotto software.
Non corrisponde necessariamente al cliente.

ESEMPI DI REQUISITI

Requisito utente:

- 1.1 Il sistema software deve fornire un mezzo per rappresentare e visualizzare file esterni generati da altri tool.

Requisito di sistema (più specifico):

- 1.1 L'utente deve avere la possibilità di definire il tipo dei file esterni
- 1.3 Ogni tipo di file esterno deve essere rappresentato mediante una specifica icona sullo schermo
- 1.5 Quando l'utente seleziona un'icona che rappresenta un file esterno, deve poter essere eseguito il tool in grado di visualizzare il file

LEZ9 – 30/10/2023

CHI LEGGE I REQUISITI?

Gli user requirements (requisiti utente) vengono letti da: manager del cliente, manager degli sviluppatori, clienti finali che utilizzeranno il prodotto, ingegneri del cliente, gli architetti di sistema.

I system requirements (requisiti di sistema) vengono letti da clienti finali che utilizzeranno il prodotto, ingegneri del cliente, architetti di sistema, software developers

CATEGORIE DI REQUISITI

Perché si parla di categorie dei requisiti(preambolo): la classificazione dei requisiti avviene lungo due dimensioni: la prima basata su livello di astrazione (uno meno dettagliato, l'altro più dettagliato), la seconda basata sul concetto di categoria dei requisiti.

I requisiti si dividono in funzionali, non funzionali e di dominio:

- ➔ Requisiti funzionali: descrivono le funzionalità del sistema software, in termini di servizi che il sistema software deve fornire, di come il sistema software reagisce a specifici tipi di input e di come si comporta in situazioni particolari.

Es.1 Il sistema software deve fornire un appropriato visualizzatore per i documenti memorizzati

Es.2 L'utente deve essere in grado di effettuare ricerche sia sull'intero insieme di basi di dati che su un loro sottoinsieme

Es.3 Ad ogni nuovo ordine deve essere associato un identificatore unico (Order ID)
OSS: in tutti e tre gli esempi i requisiti sono requisiti utente, in quanto astratti e discorsivi.

NOTA: un requisito funzionale può essere un requisito utente o un requisito di sistema.

Ogni volta che definiamo un requisito dobbiamo capire se si tratta di un requisito utente o di un requisito di sistema e se è un requisito funzionale o non funzionale.

- **Requisiti non funzionali**: descrivono le proprietà del sistema software in relazione a determinati servizi o funzioni e possono anche essere relativi al processo:
- caratteristiche di efficienza, affidabilità, safety, ecc.
 - caratteristiche del processo di sviluppo (standard di processo, linguaggi di programmazione, metodi di sviluppo, ecc.)
 - caratteristiche esterne (vincoli legislativi, ecc.)
- Es.1 Il tempo di risposta del sistema all'inserimento della password utente deve essere inferiore a 10 sec
- Es.2 I documenti di progetto (deliverable) devono essere conformi allo standard XYZ-ABC-12345
- Es.3 Il sistema software non deve rilasciare ai suoi operatori nessuna informazione personale relativa ai clienti, tranne nominativo e identificatore
- OSS: in tutti e tre gli esempi i requisiti sono requisiti utente, in quanto astratti e discorsivi.

- **Requisiti di dominio**:
- requisiti derivati dal dominio applicativo del sistema software piuttosto che da necessità dettate dagli utenti
- requisiti funzionali, nuovi o adattati, relativi al particolare dominio applicativo
 - requisiti non funzionali, nuovi o adattati, relativi a standard esistenti o a procedure e regolamenti da applicare
- Es.1 I documenti di rendiconto contabile, secondo la normativa XYZ.03, devono essere stampati alla ricezione e cancellati immediatamente
- Es.2 L'interfaccia utente per l'accesso al database magazzino deve essere conforme allo standard ZX.01

PROBLEMI CON I REQUISITI SOFTWARE:

- 1) *Ambiguità*, si intendono requisiti interpretabili in modo differente.
Esempio 1: specificare un tempo senza fornire il riferimento al fuso orario (in un'applicazione che gestisce chiamate intercontinentali)
- 2) *Incompletezza*, i requisiti non includono la descrizione di tutte le caratteristiche richieste
- 3) *Inconsistenza*, quando ci sono conflitti o contraddizioni nella descrizione delle caratteristiche del sistema.
Esempio: Req 1: ogni form di input non deve contenere più di 5 campi editabili dall'utente
Req 2: nella form di input relativa all'inserimento dei dati anagrafici l'utente deve introdurre i seguenti dati: nome, cognome, anno di nascita, luogo di nascita, indirizzo, telefono, fax, e-mail

VERIFICABILITA' DEI REQUISITI

I requisiti non funzionali espressi in modo generico dall'utente (es. il sistema software deve essere easy-to-use) possono risultare non quantificabili e difficili da verificare.
È quindi necessario esprimere i requisiti non funzionali usando una misura determinata che permetta di verificare quantitativamente se il requisito verrà soddisfatto dal sistema software.

Un esempio di misura per il requisito di "affidabilità" di un prodotto hardware può essere il "mean time to failure", ossia il tempo medio al primo malfunzionamento del prodotto.

REQUISITI UTENTE

Descrivono requisiti funzionali e non funzionali, espressi in modo da risultare comprensibili agli utenti del sistema sprovvisti di conoscenze tecniche. I requisiti utenti sono generalmente espressi in linguaggio naturale, tenendo in considerazione alcune linee guida:

- usare un formato standard per tutti i requisiti
- usare il linguaggio naturale in modo consistente (es. uso di "deve" per requisiti necessari e "dovrebbe" per requisiti desiderabili)
- evidenziare le parti fondamentali di un requisito
- evitare l'uso di termini tecnici

REQUISITI DI SISTEMA (SPECIFICHE)

Specifiche più dettagliate dei requisiti utente. Sono usati come base per il progetto software. Possono essere espressi facendo uso di notazioni differenti. Notazioni possibili:

1. linguaggio naturale strutturato
2. linguaggio di descrizione del programma (PDL, Program Description Language)
3. notazioni grafiche (usando linguaggio di modellazione)
4. specifiche matematiche

LEZ10 – 2/11/2023

DOCUMENTO DI ANALISI DEI REQUISITI (DOCUMENTO DI SPECIFICA)

Documento ufficiale che descrive in dettaglio le caratteristiche del sistema da sviluppare. Include sia la definizione (utente) dei requisiti che la loro specifica (sistema).

Describe COSA il sistema deve fornire (dominio del problema) e non COME il sistema deve essere sviluppato (dominio della soluzione).

Gli utenti del documento di analisi dei requisiti sono:

- System customers (il cliente), che contribuisce alla definizione specifica dei requisiti e alle modifiche a tali requisiti
- Managers, tipicamente sprovvisti di conoscenze di sviluppo software che usano questo documento per capire quanto investire, se conveniente o meno per l'organizzazione, che gestiscono fase contrattuale ecc...
- System engineers, che usano questo documento per capire nei dettagli cosa deve essere sviluppato
- System test engineers, usano i requisiti per la validazione e il testing del sistema
- System maintenance engineers, usano i requisiti per aiutare a capire il sistema e le relazioni tra le parti

La struttura del documento di specifica si basa sullo standard IEEE 830-1998:

Prefazione : Lettori previsti, cronologia delle versioni, riepilogo delle modifiche

Introduzione : Breve descrizione del sistema, l'interazione con altri sistemi, l'ambito di applicazione nel contesto aziendale

Glossario : Definizione dei termini tecnici utilizzati nel documento

Definizione dei requisiti dell'utente : Requisiti funzionali e non funzionali dell'utente

Architettura del sistema : Panoramica di alto livello dei componenti del sistema

Specifiche dei requisiti di sistema : Requisiti di sistema funzionali e non funzionali

Modelli di sistema : Descrizione delle relazioni tra i componenti del sistema e il sistema e il suo ambiente

Evoluzione del sistema : presupposti su cui si basa il sistema e le modifiche previste
(hardware evoluzione, cambiamenti delle esigenze dell'utente,...)

Appendici : informazioni specifiche relative all'applicazione che si sta sviluppando
(es. HW e le descrizioni del DB)

Indice : sommario, indice alfabetico, elenco dei diagrammi, ecc.

IL PROCESSO DI INGEGNERIA DEI REQUISITI

Il processo di ingegneria dei requisiti (requirements engineering) varia in base al dominio applicativo, alle persone coinvolte ed all'organizzazione che sviluppa il sistema software.

Si può però individuare un insieme di attività generiche comuni a tutti i processi:

- 1) studio di fattibilità: fase iniziale del processo di ingegneria dei requisiti che si basa su una descrizione sommaria del sistema sw, informazioni raccolte da colloqui con client manager, ingegneri del sw con esperienza nello specifico campo applicativo, esperti delle tecnologie da utilizzare e utenti finali del sistema. Lo studio di fattibilità produce come risultato un report che stabilisce se procedere o meno allo sviluppo del sistema software. Domande tipiche dello studio di fattibilità sono: "In che termini il sistema software contribuisce al raggiungimento degli obiettivi strategici del cliente?", "Può il sistema software essere sviluppato usando le tecnologie correnti e rispettando i vincoli di durata e costo complessivo?", "Può il sistema software essere integrato con altri sistemi già in uso?"

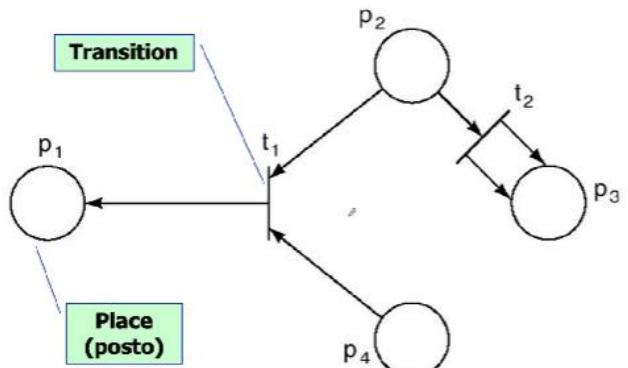
- 2) identificazione e analisi dei requisiti: Il team di sviluppo incontra il cliente e gli utenti finali al fine di identificare l'insieme dei requisiti utente, dalla cui analisi si generano i requisiti di sistema (specifiche). In questa fase vengono coinvolte diverse figure, si usa il termine STAKEHOLDER per identificare tutti coloro che hanno un interesse diretto o indiretto sui requisiti del sistema software da sviluppare.
- Task per questa fase di identificazione e analisi dei requisiti sono la comprensione del dominio, la raccolta dei requisiti mediante l'interazione con gli stakeholder, classificazione dei requisiti, risoluzione dei conflitti, assegnazione delle priorità e verifica di compattezza e consistenza dei requisiti individuati.
- 3) specifica dei requisiti (req. specification): le tecniche di analisi e specifica dei requisiti possono essere formali(basate su Petri Net, FSM, Z, etc) o semi-formali.
- 4) convalida dei requisiti (req. validation): questa attività è finalizzata ad accertare se il documento dei requisiti, ottenuto come risultato della fase di analisi, descrive realmente il sistema software che il cliente si aspetta. I controlli da effettuare includono validità, consistenza, completezza, realizzabilità, e verificabilità.
Le tecniche di convalida dei requisiti includono revisioni informali(peer review), revisioni formali, prototipizzazione, generazione dei test-case, e analisi di consistenza automatizzata.
- 5) gestione dei requisiti (req. management): processo di identificazione e controllo delle modifiche subite dai requisiti di un sistema software lungo il ciclo di vita. I requisiti di un sistema software possono essere classificati in termini della loro evoluzione come requisiti stabili, requisiti volatili (mutabili, emergenti, consequenziali, di compatibilità). Se necessarie modifiche dei requisiti, quest'ultime vanno pianificate mediante identificazione univoca dei requisiti, gestione delle modifiche (con analisi di costi, impatto e realizzazione), politiche di tracciabilità, e uso di tool CASE per il supporto alle modifiche.

SPECIFICHE FORMALI CON PETRI NET

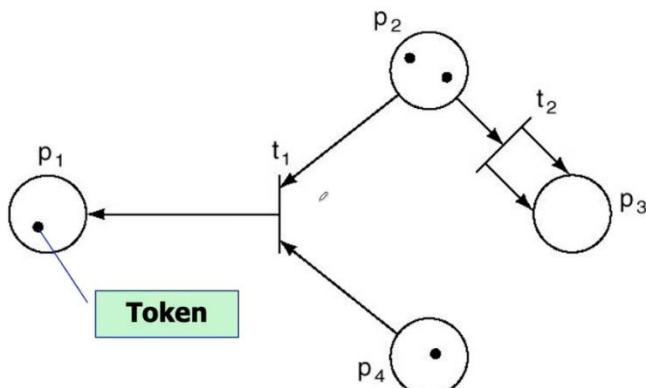
Introdotte per problemi di temporizzazione e sincronizzazione, sono molto usate nell'ambito delle telecomunicazioni.

Utilizza una sintassi di tipo visuale (3 costrutti fondamentali: cerchio, archi orientati, barre)

In generale non si applicano al software ma si possono adattare ad esso.



La rete di Petri serve a capire come un certo sistema evolve nel tempo in fase d'esecuzione. Per rappresentare questa dinamica si aggiunge un altro elemento: il token (gettone) raffigurato da un pallino nero inserito all'interno di un posto.



Quindi una rete di Petri evolve passando attraverso una serie di stati e il modo attraverso cui conferiamo uno stato a una rete di Petri è legato alla distribuzione/allocazione di questi token all'interno dei vari posti.

L'operazione con cui inserisco i token si chiama marking (marcatura della rete di Petri).

Regola di abilitazione delle transizioni: una transizione è abilitata se esiste almeno un token all'interno di ogni posto collegato in ingresso alla transizione. Quindi la transizione è abilitata se e solo se esiste almeno un token dentro ogni posto collegato in ingresso.

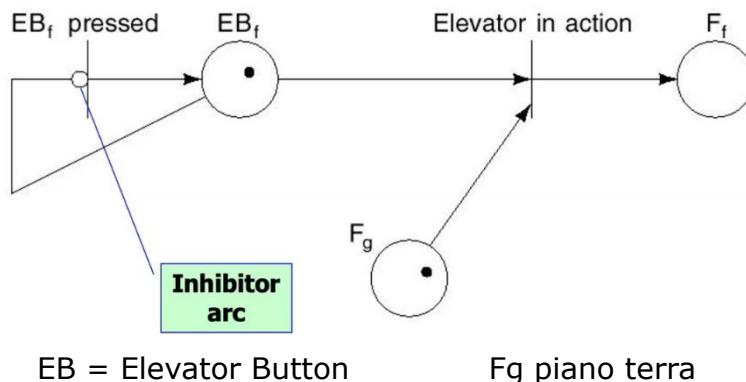
Il termine "firing"("scatto"): il firing di una transizione corrisponde all'esecuzione dell'operazione associata. Quando si effettua il firing si preleva un token per ogni arco in ingresso alla transizione e si consegnano tanti token quanti sono gli archi uscenti dalla transizione.

Come si descrive lo stato di una rete di Petri? Lo stato è definito dal numero di token presenti all'interno di ogni transizione.

L'obiettivo del formalismo Petri Net è rappresentare un sistema dal punto di vista dinamico cercando di capire come può evolvere durante l'esecuzione a partire da uno stato iniziale arrivando a uno stato finale cercando di analizzare tutti i possibili percorsi di esecuzione per capire se sono corretti o meno.

LEZ11 - 6/11/2023

Un esempio di rete di Petri(ascensore):



Una barra con il pallino davanti nel gergo delle reti di Petri si definisce come arco inibitore, questo dal punto di vista della semantica di questo costrutto sta a significare che quella transizione funziona esattamente al contrario di come funziona una transizione non inibita (quindi la sua regola di abilitazione è tale per cui quella transizione sarà abilitata se e soltanto se non c'è nessun token nel posto in ingresso).

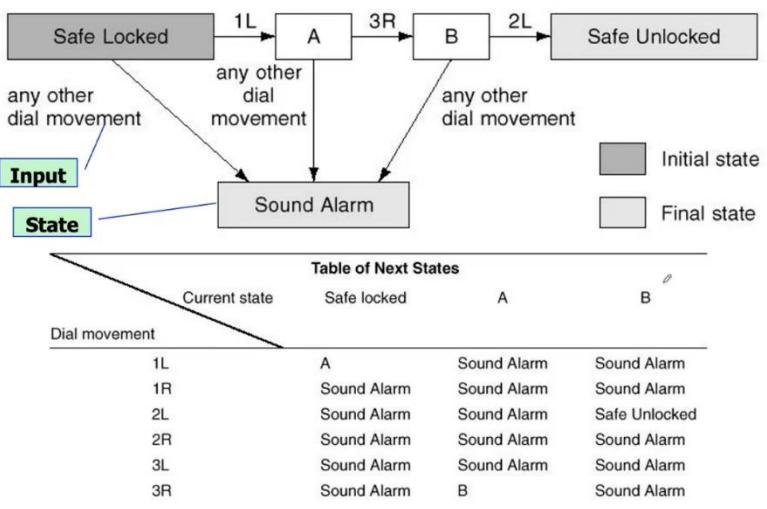
Nota: nell'immagine è presente una self transition, ossia una transizione il cui posto in ingresso e posto in uscita coincidono.

Una limitazione delle Petri Net è che ogni transizione è istantanea, ma ciò non è sempre vero. Per superare le diverse limitazioni sono stati creati diversi dialetti che partendo dalla Petri Net la estendono.

SPECIFICHE FORMALI CON MACCHINE A STATI FINITI (Finite State Machine , FSM)

Simile a rete di Petri, ma a differenza di quest'ultima nel caso di FSM la primitiva di base è quella che si utilizza per rappresentare direttamente un possibile stato del sistema(piuttosto che farlo indirettamente come in PetriNet).

Le sue primitive sono il rettangolo per rappresentare uno stato del sistema, l'arco orientato rappresenta un evento che porta ad un cambiamento di stato. Si possono identificare inoltre gli stati iniziali con un colore più scuro rispetto agli stati finali con colore più chiaro come vediamo anche nell'esempio seguente. Esempio(cassaforte):



SPECIFICHE FORMALI CON LINGUAGGIO Z

Consiste di un set di schemi, ogni schema Z ha il seguente formato(S è il nome dello schema):

S	
	<i>declarations</i>
	<i>predicates</i>

Il concetto di schema è l'unico costrutto del linguaggio Z.

Un esempio di specifica di stato è il seguente (esempio pulsante dell'ascensore):

<i>Button_State</i>	
floor_buttons, elevator_buttons	: P Button
buttons	: P Button
pushed	: P Button
$\text{floor_buttons} \cap \text{elevator buttons} = \emptyset$	
$\text{floor_buttons} \cup \text{elevator buttons} = \text{buttons}$	

Button è l'insieme di tutti i pulsanti
P denota l'insieme potenza (insieme di
tutti i possibili sottoinsiemi)
Sintassi declarations = nomeVar :

Abstract Initial State

```
Button_init := [Button_State' | pushed' = Ø]
```

<i>Push_Button</i>	
Δ Button_State	
button?: Button	
(button? \in buttons) \wedge	
(((button? \notin pushed) \wedge (pushed' = pushed \cup {button?})) \vee	
((button? \in pushed)' \wedge (pushed' = pushed))	

Il simbolo delta Δ si utilizza per dire qual è lo stato/quali sono gli stati sui quali agisce l'operazione Push_Button (si fa riferimento a Button_State dichiarato precedentemente).

In "button?:Button" il carattere "?" sta ad indicare che si tratta di un parametro di input, i parametri di output invece vendono caratterizzati dal punto esclamativo "!"

SPECIFICHE SEMI-FORMALI: MODELLI DEL SISTEMA

Per modello del sistema si intende una rappresentazione astratta del sistema che facilita la comprensione delle proprietà del sistema e delle sue caratteristiche di funzionamento, prima che il sistema venga costruito. L'uso di modelli dei sistemi software è formalizzato all'interno di metodi di analisi dei requisiti (specifiche) del software che fanno uso di tecniche semi-formali.

I metodi di analisi dei requisiti software sono di due tipi:

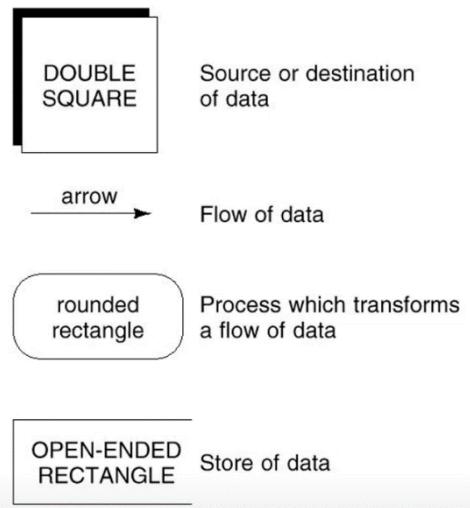
- metodi di analisi strutturata (o procedurale)
- metodi di analisi(dei requisiti) orientata agli oggetti

Per descrivere completamente un sistema è necessario costruire vari modelli che rappresentino il sistema da vari punti di vista (informazioni, funzioni e comportamento dinamico).

Per descrivere la specifica semi-formale di un sistema software si usano 3 tipi di modelli:

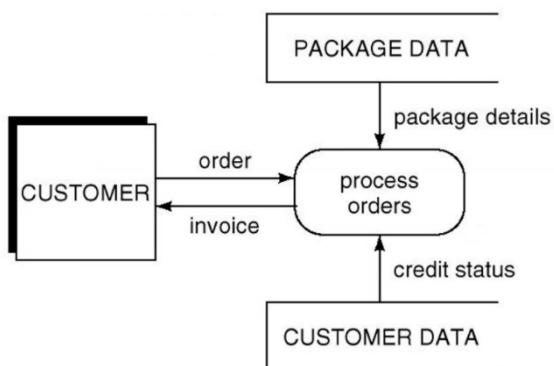
1. modello dei dati: rappresenta gli aspetti statici e strutturali relativi ai dati (data requirements) (class diagram (UML)).
2. modello comportamentale: rappresenta gli aspetti funzionali del sistema (functional requirements) (data flow diagram (not UML), use case diagram (UML), activity diagram (UML), interaction diagram (UML)).
3. modello dinamico: rappresenta gli aspetti di "controllo" e di come le funzioni del modello comportamentale modificano i dati introdotti nel modello dei dati (state diagram (UML))

DATA FLOW DIAGRAM (DFD)

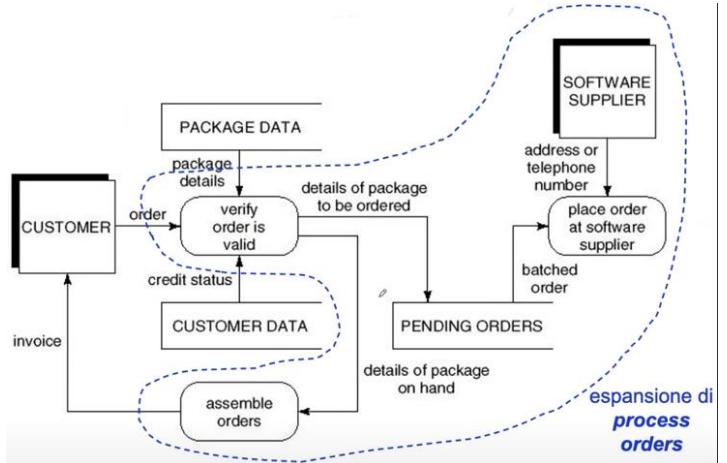


Esempio (software che elabora ordini):

(Primo raffinamento)



(Secondo raffinamento)



LEZ12 - 9/11/2023

STRUCTURED SYSTEM ANALYSIS (SSA) (METODO DI ANALISI STRUTTURATA)

È un metodo introdotto nel 1979 costituito da 9 step che si basa sul concetto di step-wise-refinement (raffinamento graduale). Altri metodi di analisi strutturata sono "DeMarco" e "Yourdon and Constantine".

I 9 step che lo costituiscono sono i seguenti:

1. disegnare il DFD utilizzando il documento dei requisiti (o il prototipo) e perfezionarlo
2. decidere quali sono le sezioni da computerizzare e quali no in base ai benefici attesi
3. determinare i dettagli del flusso di dati (data flow), decidere quale elemento di dati deve andare nei vari flussi di dati
4. definire la logica dei processi, ad esempio costruendo un decision tree
5. determinare i data stores, definendo i contenuti esatti di ogni archivio e la sua rappresentazione e definendo inoltre i livelli di accesso
6. definire le risorse fisiche, ad esempio se si usa un DBMS specificare le informazioni delle varie tabelle
7. determinare le specifiche di input/output
8. determinare la dimensione, computando il volume dell'input, la frequenza, le dimensioni dei record, la dimensione di ogni file
9. determinare i requisiti hardware, determinando i vari requisiti della memoria ecc..

Dopo la fase 9 e dopo l'approvazione del cliente, il documento di specifica passa alla fase di progettazione.

Svantaggi SSA: non può essere usato per determinare tempi di risposta, la dimensione della CPU e il timing non possono essere determinati con accuratezza.

OBJECT ORIENTED ANALYSIS - OOA

La fase di OOA definisce, secondo un approccio ad oggetti, COSA un prodotto software deve fare (mentre la fase di OOD definisce, sempre secondo un approccio ad oggetti, COME un prodotto software deve fare quanto specificato in fase di OOA).

OOA e OOD devono fornire, ciascuno dal proprio punto di vista, una rappresentazione corretta, completa e consistente:

- degli aspetti statici e strutturali relativi ai dati (modello dei dati)
- degli aspetti funzionali del sistema (modello comportamentale)
- degli aspetti di "controllo" e di come le funzioni del modello comportamentale modificano i dati introdotti nel modello dei dati (modello dinamico)

METODI DI OOA

Un metodo di OOA definisce l'insieme di procedure, tecniche e strumenti per un approccio sistematico alla gestione e allo sviluppo della fase di OOA.

L'input di un metodo di OOA è costituito dall'insieme dei requisiti utente (contenuti nel documento di analisi dei requisiti).

L'output di un metodo di OOA è costituito dall'insieme dei modelli del sistema che definiscono la specifica del prodotto software (e che sono anch'essi contenuti nel documento di analisi dei requisiti).

I metodi di OOA fanno principalmente uso di notazioni visuali (diagrammi), ma possono essere affiancati da metodi tradizionali per la definizione di requisiti di sistema di tipo testuale (in linguaggio naturale strutturato).

Lo sviluppo dei modelli di OOA non è un processo sequenziale (prima modello dei dati, poi modello comportamentale, infine modello dinamico).

La costruzione dei modelli avviene in parallelo, e ciascun modello fornisce informazioni utili per gli altri modelli.

I metodi di OOA fanno uso di un approccio iterativo, con aggiunta di dettagli per raffinamenti successivi (iterazioni).

Alcuni metodi di OOA (e OOD) sono:

- Catalysis: metodo OO particolarmente indicato per lo sviluppo di sistemi software a componenti distribuiti.
- Objectory: metodo ideato da I. Jacobson che fonda lo sviluppo di prodotti software ad oggetti sull'individuazione dei casi d'uso utente (use case driven).
- Shlaer/Mellor: metodo OO particolarmente indicato per lo sviluppo di sistemi software real-time.
- OMT (Object Modeling Technique): metodo sviluppato da J. Rumbaugh basato su tecniche di modellazione del software iterative. Pone in particolare risalto la fase di OOA.
- Booch: metodo basato su tecniche di modellazione del software iterative. Pone in particolare risalto la fase di OOD.
- Fusion: metodo sviluppato dalla HP a metà degli anni novanta. Rappresenta il primo tentativo di standardizzazione per lo sviluppo di software orientato agli oggetti. Si basa sulla fusione dei metodi OMT e Booch.

NOTAZIONI PER OOA (e OOD)

Ciascun metodo di OOA (e OOD) fa uso di una propria notazione per la rappresentazione dei modelli del sistema. Al fine di unificare le notazioni per i metodi di OOA e OOD è stato introdotto il linguaggio UML (Unified Modeling Language), adottato nel 1997 come standard OMG (ObjectManagement Group).

UML è un linguaggio standard per la descrizione di sistemi software (orientati agli oggetti). Si compone di nove formalismi di base (diagrammi con semantica e notazione data) e di un insieme di estensioni.

UML è un linguaggio di descrizione, non è un metodo né definisce un processo.

FORMALISMI UML

I nove formalismi di base dello UML sono:

1. Use case diagram
evidenziano la modalità (caso d'uso) con cui gli utenti (attori) utilizzano il sistema.
Possono essere usati come supporto per la definizione dei requisiti utente.
2. Class diagram
consentono di rappresentare le classi con le relative proprietà (attributi, operazioni) e le associazioni che le legano.
3. State diagram
rappresentano il comportamento dinamico dei singoli oggetti di una classe in termini di stati possibili e transizioni di stato per effetto di eventi.
4. Activity diagram
sono particolari state diagram, in cui gli stati rappresentati rappresentano azioni in corso di esecuzione. Sono particolarmente indicati per la produzione di modelli di workflow.
5. Sequence diagram
evidenziano le interazioni (messaggi) che oggetti di classi diverse si scambiano nell'ambito di un determinato caso d'uso, ordinate in sequenza temporale. A differenza dei diagrammi di collaborazione, non evidenziano le relazioni tra oggetti.
6. Collaboration diagram
descrivono le interazioni (messaggi) tra oggetti diversi, evidenziando le relazioni esistenti tra le singole istanze.
7. Object diagram
permettono di rappresentare gli oggetti e le relazioni tra essi nell'ambito di un determinato caso d'uso.
8. Component diagram
evidenziano la strutturazione e le dipendenze esistenti tra componenti software.
9. Deployment diagram
evidenziano le configurazioni dei nodi elaborativi di un sistema real-time ed i componenti, processi ed oggetti assegnati a tali nodi.

MODELLO DEI DATI

Rappresenta da un punto di vista statico e strutturale l'organizzazione logica dei dati da elaborare.

Le strutture dati sono definite mediante lo stato degli oggetti, che viene determinato dal valore assegnato ad attributi e associazioni.

Il modello dei dati viene specificato mediante il formalismo dei class diagram che permette di definire:

- classi
- attributi di ciascuna classe
- operazioni di ciascuna classe
- associazioni tra classi

Il modello dei dati è di fondamentale importanza, visto che, secondo l'approccio ad oggetti, un sistema software è costituito da un insieme di oggetti (classificati) che collaborano.

Il modello dei dati viene costruito in modo iterativo ed incrementale.

Si tratta di un processo creativo, in cui giocano un ruolo importante sia l'esperienza dell'analista che la comprensione del dominio applicativo.

Durante la fase iniziale di costruzione del modello dei dati occorre concentrarsi sulle cosiddette entity classes, ovvero quelle classi che definiscono il dominio applicativo e che sono rilevanti per il sistema.

Le control classes (che gestiscono la "logica" del sistema) e boundary classes (che rappresentano l'interfaccia utente) vengono introdotte successivamente, usando le informazioni del modello comportamentale.

Le operazioni di ciascuna classe vengono identificate a partire dal modello comportamentale, per cui vengono inizialmente trascurate.

LEZ13 – 13/11/2023

APPROCCI PER L'IDENTIFICAZIONE DELLE CLASSI

Alcuni approcci sono: Noun phrase, Common class patterns, Use case driven, CRC, Mixed.

APPROCCIO NOUN PHRASE

Una frase nominale (noun phrase) è una frase in cui il sostantivo ha una prevalenza sulla parte verbale (sono frasi di tipo assertivo).

I sostantivi delle frasi nominali usate per la stesura dei requisiti utente sono considerati candidate classes.

La lista delle candidate classes viene suddivisa in tre gruppi:

- Irrelevant (non appartengono al dominio applicativo e quindi possono essere scartate)
- Relevant (evidenziano caratteristiche di entity classes)
- Fuzzy (non si hanno sufficienti informazioni per classificarle come relevant o irrelevant, vanno analizzate successivamente)

Si assume che l'insieme dei requisiti utente sia completo e corretto.

APPROCCIO COMMON CLASS PATTERNS

Basato sulla teoria della classificazione. Le candidate classes vengono identificate a partire da gruppi (pattern) di classi predefinite, quali ad esempio Concept (es. Reservation), Events (es. Arrival), Organization (es. AirCompany), People (es. Passenger), Places (es. TravelOffice).

Non è un approccio sistematico, ma può rappresentare una utile guida.

A differenza dell'approccio noun phrase, non si concentra sul documento dei requisiti utente.

Può causare problemi di interpretazione dei nomi delle classi.

APPROCCIO USE CASE DRIVEN

Si assume che:

- Siano già stati sviluppati gli use case diagram (e possibilmente anche i sequence diagram più significativi)
- Per ogni use case sia fornita una descrizione testuale dello scenario di funzionamento

Simile all'approccio noun phrase (si considera l'insieme degli use case come insieme dei requisiti utente).

Si assume che l'insieme degli use case sia completo e corretto.

Approccio function-driven (o problem-driven secondo la terminologia object oriented)

APPROCCIO CRC

L'approccio CRC (Class – Responsibility - Collaborators) è basato su riunioni in cui si fa uso di apposite card.

Ciascuna card rappresenta una classe, e contiene tre compartimenti, che identificano:

- Il nome della classe
- Le responsabilità assegnate alla classe
- Il nome di altre classi che collaborano con la classe

Le classi vengono identificate analizzando come gli oggetti collaborano per svolgere le funzioni di sistema.

Approccio utile per verifica di classi identificate con altri metodi e per identificazione di attributi e operazioni di ciascuna classe.

CLASS
Elevator Controller
RESPONSIBILITY
1. Turn on elevator button 2. Turn off elevator button 3. Turn on fl oor button 4. Turn off fl oor button 5. Open elevator doors 6. Close elevator doors 7. Move elevator one fl oor up 8. Move elevator one fl oor down
COLLABORATION
1. Class Elevator Button 2. Class Floor Button 3. Class Elevator

APPROCCIO MIXED

Basato su elementi presenti in ciascuno degli approcci precedenti.

Un possibile scenario potrebbe essere il seguente:

1. L'insieme iniziale delle classi viene identificato in base all'esperienza dell'analista, facendosi eventualmente guidare dall'approccio common class patterns
2. Altre classi possono essere aggiunte usando sia l'approccio noun phrase che l'approccio use case driven (se gli use case diagram sono disponibili)
3. Infine l'approccio CRC può essere usato per verificare l'insieme delle classi identificate

LINEE GUIDA PER L'IDENTIFICAZIONE DELLE ENTITY CLASSES

- 1) Ogni classe deve avere un ben preciso statement of purpose.
- 2) Ogni classe deve prevedere un insieme di istanze (oggetti)
 - le cosiddette singleton classes (per la quali si prevede una singola istanza) non sono di norma classificabili come entity classes.
- 3) Ogni classe deve prevedere un insieme di attributi (non un singolo attributo).
- 4) Distinguere tra elementi che possono essere modellati come classi o come attributi.
- 5) Ogni classe deve prevedere un insieme di operazioni (anche se inizialmente le operazioni vengono trascurate, i servizi che la classe mette a disposizione sono implicitamente derivabili dallo statement of purpose)

CASI DI STUDIO:

A. University Enrolment

L'università offre:

- Diplomi di laurea e post-laurea
- Agli studenti a tempo pieno e part-time

La struttura universitaria:

- Facoltà contenente dei reparti
- Ogni grado è amministrato da un'unica divisione
- La laurea può includere corsi di altre divisioni

Sistema di immatricolazione all'università:

- Programmi di studio personalizzati
- Corsi propedeutici
- Corsi obbligatori
 - o Restrizioni (Scontri di orario, Numero massimo di studenti per classe, ecc.)

→ Esempio A.1 (utilizzo approccio Noun Phrase)

Prendere in considerazione i seguenti requisiti per il sistema di immatricolazione all'Università e identificare le classi candidate:

- Ogni laurea universitaria ha un certo numero di corsi obbligatori e una serie di corsi a scelta.

Classi rilevanti = laurea(Degree) , corso(Course)

Classi irrilevanti = corso obbligatorio(CompulsoryCourse) , corso opzionale(ElectiveCourse)

Altri requisiti:

- Ogni corso è di un determinato livello e ha dei crediti associati
- Un corso può far parte di un numero qualsiasi di lauree
- Ogni laurea specifica il valore minimo dei crediti totali necessario per il completamento della laurea
- Gli studenti possono combinare le offerte di corsi in programmi di studio adatti alle loro esigenze individuali e al corso di laurea a cui si è iscritti

Classi rilevanti = corso(Course), laurea(Degree) , studente(Student) , corso offerto(CourseOffering)

Classi irrilevanti = corso obbligatorio(CompulsoryCourse) , corso opzionale(ElectiveCourse) , programma di studio(StudyProgram)

B. Video Store

Nel video store:

- Noleggio di videocassette e dischi ai clienti
- Tutte le videocassette e i dischi con codice a barre
- Anche l'iscrizione dei clienti deve essere accompagnata da un codice a barre.

I clienti esistenti possono effettuare prenotazioni su video da raccogliere in una data specifica. Rispondere alle richieste dei clienti, comprese le richieste di informazioni sui film che la videoteca non ha in magazzino (ma può ordinare su richiesta).

→ Esempio B.1 (prima iterazione sul caso di studio B)

Prendere in considerazione i seguenti requisiti per il video Store e identificare le classi candidate:

- Il negozio tiene in magazzino una vasta libreria di titoli di film attuali e popolari. Un particolare film può essere contenuto su videocassette o dischi.

Classi rilevanti = titolo del film(MovieTitle), videocassetta(VideoTape),
disco(VideoDisk)

Classi irrilevanti = video store(videoStore), magazzino(Stock) , libreria(Library)

Altri requisiti:

- Le videocassette sono in formato "Beta" o "VHS"
- I dischi video sono in formato DVD
- Ogni film ha un periodo di noleggio particolare (espresso in giorni), con un canone di locazione per quel periodo
- La videoteca deve essere in grado di rispondere immediatamente a eventuali richieste di informazioni sulla disponibilità di magazzino di un film e su come alcuni nastri e/o dischi sono disponibili per il noleggio
- La condizione corrente di ogni nastro e disco deve essere conosciuta e registrata

Classi rilevanti = MovieTitle, VideoMedium, VideoTape, VideoDisk, BetaTape, VHSTape

Classi irrilevanti = RentalConditions(condizioni di noleggio)

C. Contact Management

La società di ricerche di mercato con una base di clienti consolidata di organizzazioni che acquistano report di analisi di mercato. L'azienda è costantemente alla ricerca di nuovi clienti.

Sistema di gestione dei contatti:

- Potenziali clienti
- Clienti effettivi
- Clienti passati

Il nuovo sistema di gestione dei contatti da sviluppare internamente deve essere a disposizione di tutti i dipendenti dell'azienda, ma con livelli diversi di accesso (i dipendenti del Servizio Clienti assumeranno la proprietà di sistema).

Il sistema consente una programmazione e una riprogrammazione flessibili delle attività correlate in modo che i dipendenti possano collaborare con successo per acquisire nuovi clienti e promuovi le relazioni esistenti

→ Esempio C.1

Prendere in considerazione i seguenti requisiti per la gestione dei contatti e individuare le classi candidate:

- Per "tenersi in contatto" con la base di clienti attuali e potenziali
- Memorizzare i nomi, i numeri di telefono, il codice postale e gli indirizzi correnti, ecc... delle organizzazioni e i contatti delle persone in queste organizzazioni
- Pianificare le attività e gli eventi per i dipendenti per quanto riguarda contatti di persone rilevanti
- I dipendenti possono pianificare attività ed eventi per altri dipendenti o per se stessi
- Un'attività è un insieme di eventi che si verificano per raggiungere un risultato (ad es. per risolvere il problema del cliente)
- Tipi tipici di eventi sono: telefonata, visita, invio di fax, organizzare la formazione, ecc..

Classi rilevanti = Organization, Contact, Employee, Task, Event

Classi irrilevanti = CurrentOrg, ProspectiveOrg, PostalAddress, CourierAddress

D. Telemarketing

La società di beneficenza vende biglietti della lotteria per raccogliere fondi per:

- Campagne a sostegno di enti di beneficenza
- Contributori passati (sostenitori) a cui si rivolge telemarketing e/o direct mailing

Ricompense (campagne bonus speciali):

- Per l'acquisto all'ingrosso

- Per attirare nuovi contributori

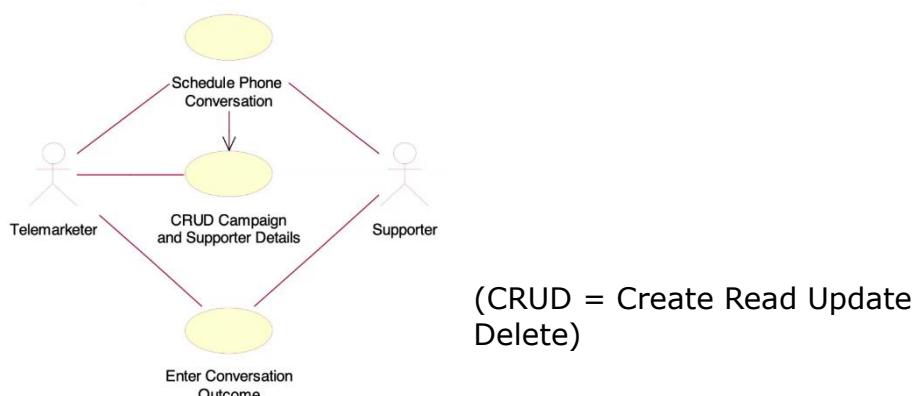
La società non prende di mira in modo casuale potenziali supporters utilizzando elenchi telefonici o mezzi analoghi.

L'applicazione telemarketing deve:

- supportare fino a cinquanta operatori di telemarketing che lavorano simultaneamente
- programmare le telefonate in base a proprietà pre-specificate e a altri vincoli noti
- comporre le chiamate telefoniche pianificate
- ripianificare le connessioni non riuscite
- organizzare altre telefonate ai sostenitori
- registrare i risultati della conversazione, incluso il ticketorder ed eventuali modifiche ai registri dei sostenitori

→ Esempio D.1

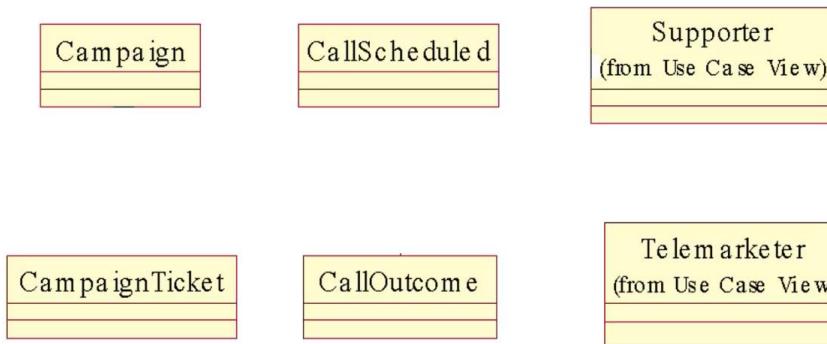
Business use case diagram:



Si consideri la seguente descrizione testuale per i casi d'uso del sistema di telemarketing e identificare le classi candidate:

- L'operatore di telemarketing richiede al sistema che la chiamata telefonica a un supporter deve essere programmata e composta
- Una volta completata la connessione, l'operatore di telemarketing offre biglietti della lotteria al supporter. Durante una conversazione, l'operatore di telemarketing potrebbe aver bisogno di accedere e modificare entrambi i dettagli della campagna e del sostenitore (CRUD, crea - leggi - aggiorna - elimina)
- Infine, l'operatore di telemarketing entra nell'esito della conversazione (risultati positivi o insoddisfacenti dell'azione di telemarketing)

Soluzione esercizio D.1:



LEZ14 – 16/11/2023

LINEE GUIDA PER LA SPECIFICA DELLE CLASSI

Nomi di classe:

- Associare ad ogni classe un nome significativo nello specifico dominio applicativo
- Adottare una convenzione standard per assegnare nomi alle classi, ad esempio: nome singolare, parole multiple devono essere congiunte, con l'iniziale di ciascuna parola in carattere maiuscolo (es. PostalAddress)
- Definire una lunghezza massima per i nomi delle classi (non più di 30 caratteri)

Attributi e operazioni:

- Considerare inizialmente solo attributi che caratterizzano possibili stati di interesse per gli oggetti
- Adottare una convenzione standard per assegnare nomi agli attributi, ad esempio: le parole devono essere scritte in carattere minuscolo, separate da un carattere di underscore (es. street_name)
- Ritardare l'aggiunta di operazioni fino al momento in cui sia disponibile il modello comportamentale, da cui vanno derivate

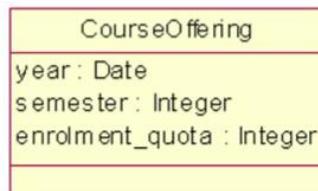
SECONDA ITERAZIONE CASI DI STUDIO PRECEDENTI

→ Esempio A.2 – University Enrolment (ricordando approccio iterativo e incrementale):

Prendere in considerazione i seguenti requisiti aggiuntivi dal Documento dei Requisiti:

La scelta dei corsi da parte di uno studente può essere limitata da scomparti di orario e dalle limitazioni al numero di studenti che possono essere iscritti al corso in corso offerta.

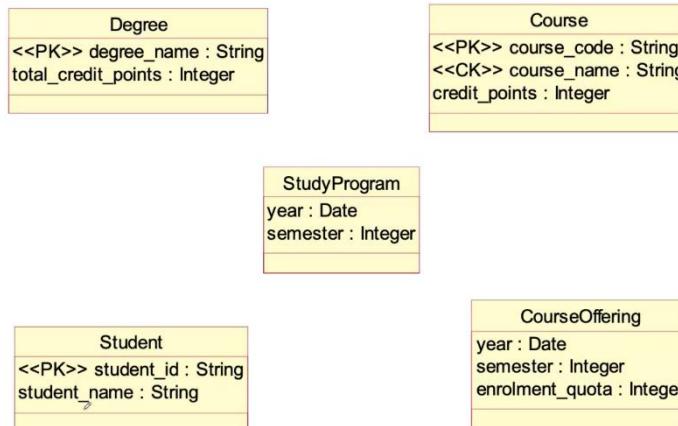
Allora:



Altri requisiti (aggiunti ai precedenti):

- Il programma di studio proposto dallo studente viene inserito nell sistema on-line di immatricolazione
- Il sistema verifica la coerenza del programma e segnala eventuali problematiche
- I problemi devono essere risolti con l'aiuto di un accademico consigliato
- Il programma di studio finale è soggetto all'approvazione accademica da parte del delegato del capo divisione ed è quindi trasmessa al segretario

Soluzione:



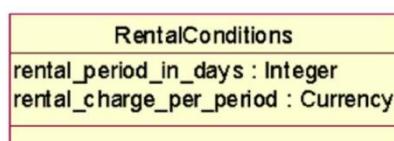
Con << >> in UML
si indicano i così
detti stereotipi

→ Esempio B.2 – Video Store

I requisiti aggiuntivi sono:

- Il costo del noleggio varia a seconda del mezzo video, nastro o disco (ma è lo stesso per il due categorie di nastri: Beta e VHS).

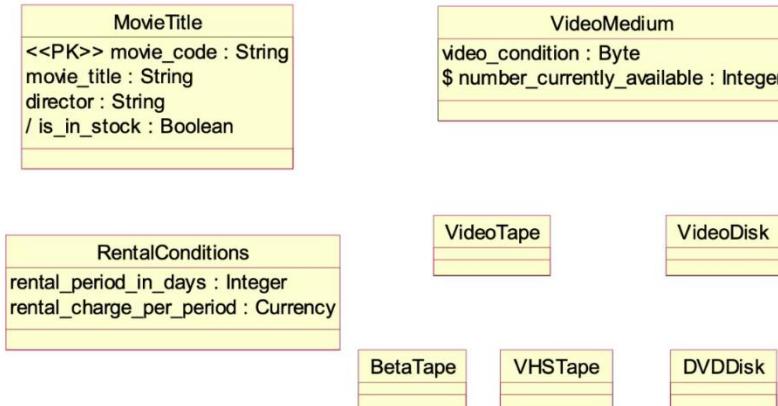
Soluzione:



Altri requisiti:

- Il sistema dovrebbe essere in grado di supportare i video futuri formati di memorizzazione oltre ai nastri VHS, Beta nastri e dischi DVD
- I dipendenti usano spesso un codice filmato, invece del titolo del film, per identificare il film
- Il titolo dello stesso film può averne più di una pubblicazione da parte di diversi registi

Soluzione:



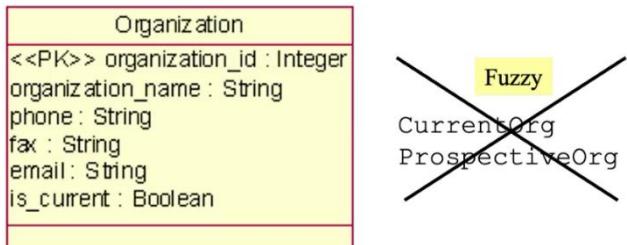
Attributo con / davanti al nome dell'attributo in UML ci dice che è un attributo derivato
Attributo con \$ davanti al nome in UML sta ad indicare un attributo di tipo statico (singolo valore condiviso da tutti gli oggetti)

→ Esempio C.2 –Contact Management

Requisiti aggiuntivi rispetto alla prima iterazione:

Un cliente è considerato corrente se esiste un contratto con tale cliente per la consegna dei nostri prodotti o servizi. La gestione dei contratti è, tuttavia, al di fuori dell'ambito del nostro sistema.

Soluzione:

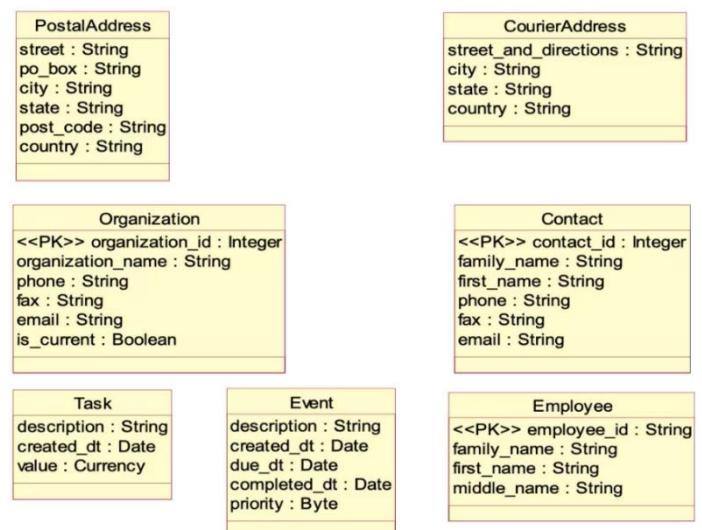


Per fuzzy si intendono classi non rilevanti

Altri requisiti:

- Reports sui contatti in base all'indirizzo postale e del corriere (ad es. clienti per codice postale)
- Vengono registrate la data e l'ora di creazione dell'attività
- Il "valore monetario" di un'attività può essere memorizzato
- Gli eventi per il dipendente vengono visualizzati sullo schermo del dipendente nella finestra di dialogo in pagine simili a calendari (un giorno per pagina).
 - o La priorità di ogni evento (bassa, media oppure alta) è distinta visivamente sullo schermo
- Non tutti gli eventi hanno un "tempo dovuto", alcuni sono "non programmati"
- L'ora di creazione dell'evento non può essere modificata, ma l'ora di scadenza sì.
- Vengono registrate la data e l'ora di completamento dell'evento
- Il sistema memorizza le identificazioni dei dipendenti che hanno creato attività e eventi, che sono programmati per fare l'evento ("dipendente dovuto") e che hanno completato l'evento

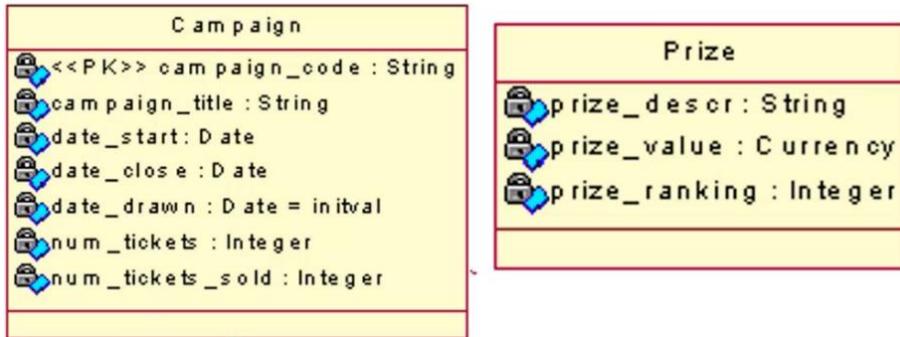
Soluzione:



Altri requisiti:

- Ogni campagna di solidarietà
 - o Ha un titolo che viene generalmente utilizzato per riferirsi ad esso
 - o Ha anche un codice univoco per il riferimento interno
 - o Funziona per un periodo di tempo fisso
- Subito dopo la chiusura della campagna, vengono estratti i premi e si avvisano i possessori di biglietti vincenti

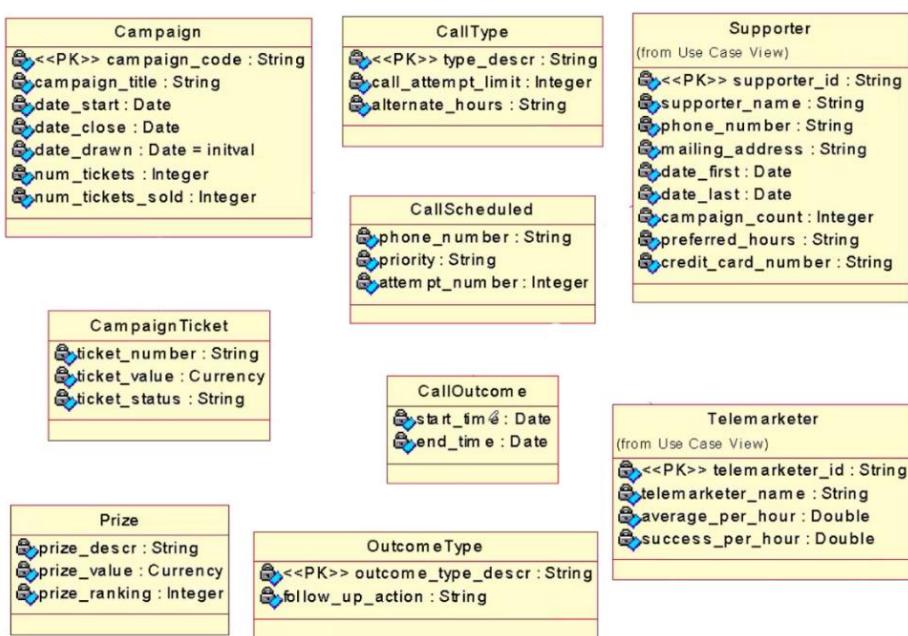
Soluzione:



Altri requisiti:

- I biglietti sono numerati in modo univoco all'interno di ogni campagna
- Il numero totale di biglietti in una campagna, il numero di biglietti venduti finora e lo stato attuale di ogni biglietto sono noti (ad es. disponibile, ordinato, pagato, vincitore del premio)
- Per determinare le prestazioni degli operatori di telemarketing della società, la durata delle chiamate e gli esiti delle chiamate (ossia con conseguenti biglietti ordinati) sono registrati
- Vengono mantenute ampie informazioni sui sostenitori
 - o Dati di contatto (indirizzo, numero di telefono, ecc.)
 - o Dettagli storici, come la prima e la data più recente in cui un sostenitore aveva partecipato a una campagna
 - o Le preferenze e i vincoli noti del sostenitore (ad esempio, gli orari di non chiamata, solito numero di carta di credito)
- Le chiamate di telemarketing vengono effettuate in base alle loro priorità
- Chiamate senza risposta o in cui è stata utilizzata una segreteria telefonica, vengono riprogrammate
 - o I tempi di ripetizione delle chiamate sono alternati
 - o Il numero di chiamate ripetute è limitato (i limiti possono essere diversi per i diversi tipi di chiamata (ad esempio una normale "sollecitazione" può avere un limite diverso da quello di una chiamata per ricordare a un sostenitore un pagamento in sospeso))
- Gli esiti delle chiamate sono classificati in successo (ad es. ticket ordinati), insuccesso, richiamare più tardi, nessuna risposta, occupato, segreteria telefonica, Fax, numero errato, disconnesso.

Soluzione:



IDENTIFICAZIONE DELLE ASSOCIAZIONI

Alcuni attributi identificati con le classi rappresentano associazioni (ogni attributo di tipo non primitivo dovrebbe essere modellato come un'associazione alla classe che rappresenta quel tipo di dato).

Ogni associazione ternaria dovrebbe essere rimpiazzata con un ciclo di associazioni binarie, per evitare problemi di interpretazione.

Nei cicli di associazioni almeno un'associazione potrebbe essere eliminata e gestita come associazione derivata, anche se per problemi di efficienza spesso si introducono associazioni ridondanti.

SPECIFICA DELLE ASSOCIAZIONI

Per assegnare nomi alle associazioni adottare la stessa convenzione usata per gli attributi (le parole devono essere scritte in carattere minuscolo, separate da un carattere di underscore).

Assegnare nomi di ruolo (rolename) alle estremità dell'associazione (i rolename diventano i nomi degli attributi nella classe all'estremità opposta dell'associazione).

Determinare la molteplicità delle associazioni (ad entrambe le estremità).

TERZA ITERAZIONE CASO DI STUDIO C

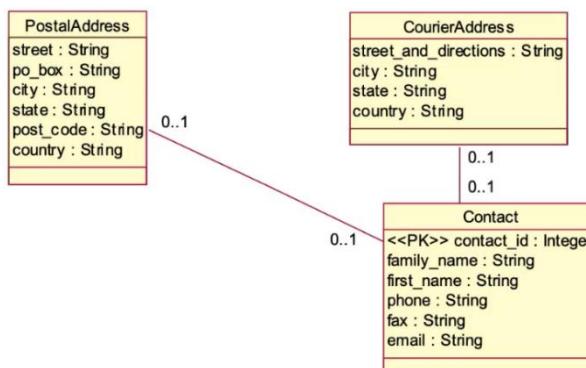
→ Esempio C.3 –Contact Management

Si fa riferimento agli esempi C.1 e C.2 - specificare le associazioni.

Si consideri il seguente ulteriore requisito:

- Il sistema permette di produrre vari report sui nostri contatti in base a indirizzi postali e di corriere

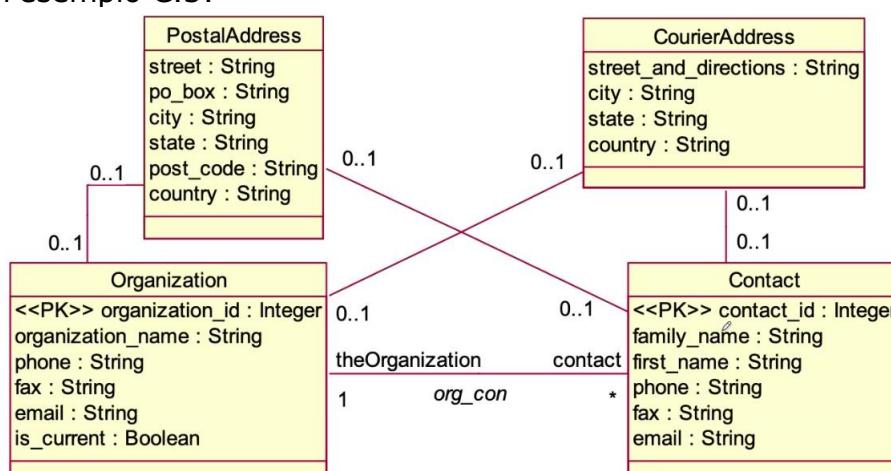
Allora:



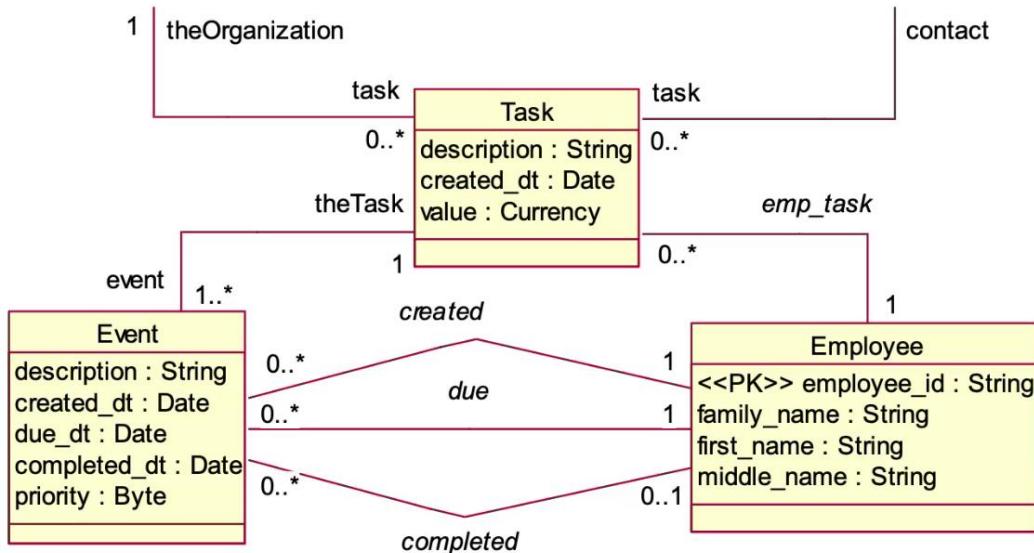
I due punti .. in UML indicano il range specificato con a sinistra il valore minimo e a destra il valore massimo

Nota: questa rappresentazione presenta dei limiti, non si può ad esempio con 0..1 esprimere il vincolo che o da entrambe le parti è 0 o da entrambe le parti è 1 ma non da una parte 0 e dall'altra 1. Interviene in questi casi il linguaggio OCL (abbreviazione che sta per Object Constraint Language) che è linguaggio usato per aggiungere vincoli ai diagrammi UML.

Soluzione1 per l'esempio C.3:



Soluzione2 per l'esempio C.3:



AGGREGAZIONE

Rappresenta una relazione di tipo "whole-part"(contenimento) tra una classe composta (superset class) e l'insieme di una o più classi componenti (subset classes).

Può assumere quattro differenti significati:

- ExclusiveOwns (esempio: Libro ha Capitolo, o Capitolo è parte di Libro)
 - o Existence-dependency
 - o Transitivity
 - o Asymmetricity
 - o Fixed property
- Owns (e.g. Car has Tire)
 - o No fixed property
- Has (esempio: Divisione ha Dipartimento)
 - o No existence dependency
 - o No fixed property
- Member (e.g. Meeting has Chairperson)
 - o Nessuna proprietà speciale tranne membership

LEZ16 – 27/11/2023

SPECIFICA DI AGGREGAZIONE IN UML

UML ha due primitive per le relazioni di contenimento, che sono:

- Aggregation
 - Semantica per riferimento
 - Si rappresenta tramite ◇
 - Corrisponde alle aggregazioni Has e Member
- Composition
 - Semantica per valore
 - Si rappresenta tramite ◆
 - Corrisponde alle aggregazioni ExclusiveOwns e Owns

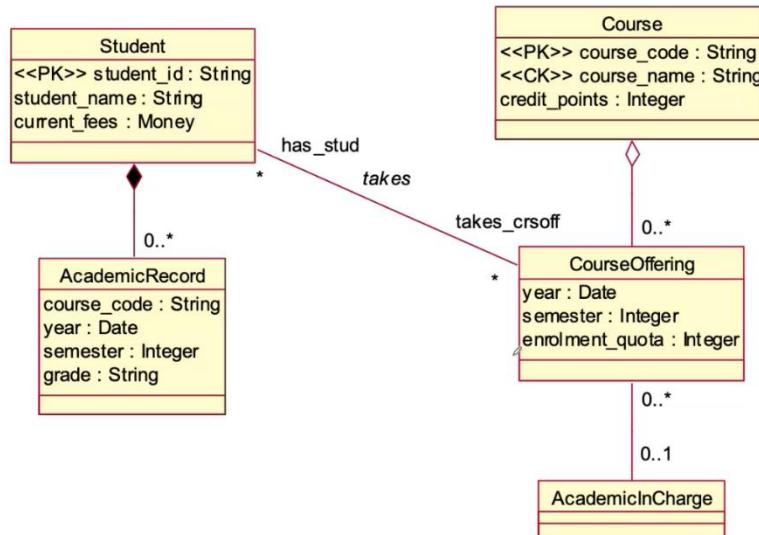
TERZA ITERAZIONE CASO DI STUDIO A

Fare riferimento agli esempi A.1 e A.2.

Prendere in considerazione i seguenti requisiti aggiuntivi:

- Il curriculum accademico dello studente deve essere disponibile su richiesta
- Il record per includere le informazioni sui voti dello studente in ogni corso a cui lo studente si è iscritto (e non si è ritirato senza penalità)
- Ogni corso ha un docente(academic) responsabile del corso, ma più docenti possono insegnarlo
 - o Ci può essere un docente diverso responsabile di un corso ogni semestre
 - o Ci possono essere docenti diversi per ciascun scorso di ciascun semestre

Soluzione:



EREDITARIETA' (GENERALIZZAZIONE)

Usata per rappresentare la condivisione di attributi ed operazioni tra classi.

Le caratteristiche comuni sono modellate in una classe più generica (superclasse), che viene specializzata nell'insieme di sottoclassi.

Una sottoclasse eredita attributi ed operazioni della superclasse.

Caratteristiche:

- Sostituibilità: un oggetto della sottoclasse è un valore legale per una variabile avente come tipo la superclasse (es. una variabile di tipo Frutta può avere un oggetto di tipo Mela come suo valore)
- Polimorfismo: la stessa operazione può avere differenti implementazioni nelle sottoclassi

SPECIFICA DI EREDITARIETA' IN UML

Rappresenta relazioni di tipo:

- "can-be" (Es. Student can be a TeachingAssistant)
- o "is-a-kind-of" (Es. TeachingAssistant is a kind of student)

Supporto ad ereditarietà multipla, Es. TeachingAssistant is also a kind of Teacher.

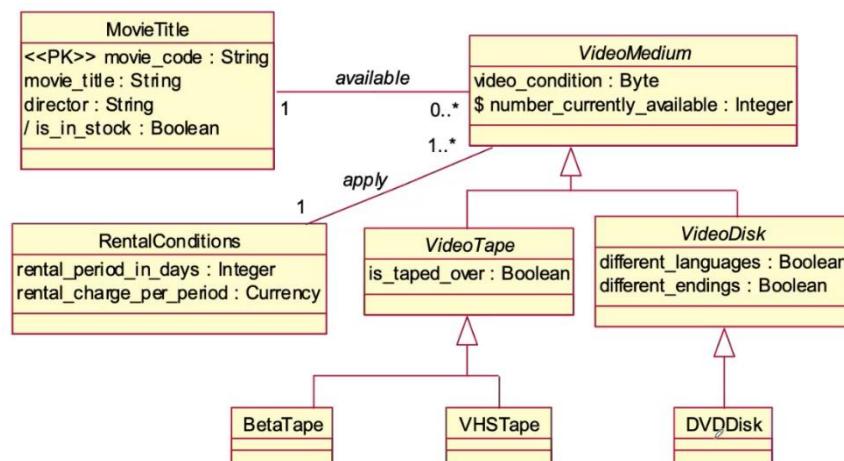
Viene rappresentata in UML con una linea, che collega la sottoclasse con la superclasse, avente una freccia diretta verso la superclasse.

TERZA ITERAZIONE CASO DI STUDIO B

Fare riferimento agli esempi B.1 e B.2

Le classi identificate nell'esempio B.2 implicano una gerarchia di generalizzazione radicata nella classe VideoMedium. Estendere il modello per includere le relazioni tra classi e specificare le relazioni di generalizzazione. Si supponga che l'archivio video debba sapere se un VideoTape è un nastro nuovo di zecca o è già stato registrato più volte (questo può essere catturato da un attributo is_taped_over). Si supponga inoltre che la capacità di archiviazione di un VideoDisk consente di contenere più versioni dello stesso film, ciascuna in una lingua diversa o con finali diversi.

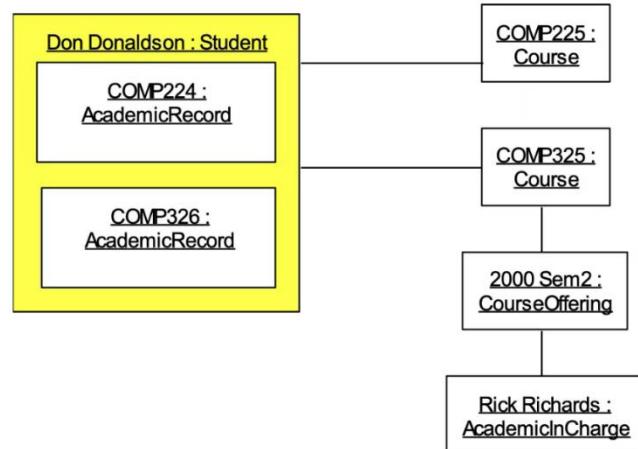
Soluzione:



OBJECT DIAGRAM

Rappresentazione grafica di istanze di classi (è la rappresentazione di oggetti). Usati per:

- modellare relazioni complesse tra classi (a scopo esemplificativo)
- illustrare le modifiche ai singoli oggetti durante l'evoluzione del sistema
- Illustrare la collaborazione tra oggetti durante l'evoluzione del sistema



Esempio object diagram con oggetti esempio

A.3:

MODELLO COMPORTAMENTALE

Rappresenta gli aspetti funzionali del sistema da un punto di vista operativo, evidenziando come gli oggetti collaborano ed interagiscono al fine di offrire i servizi che il sistema mette a disposizione.

Fa uso di vari formalismi:

- Use case diagram (per descrivere scenari di funzionamento)
- Activity diagram (per descrivere il flusso di elaborazione)
- Sequence diagram (per descrivere l'interazione tra gli oggetti)
- Collaboration diagram (per descrivere l'interazione tra gli oggetti)

Viene costruito in modo iterativo ed incrementale, usando le informazioni del modello dei dati, che a sua volta fa uso del modello comportamentale per identificare operazioni e classi aggiuntive (control classes e boundary classes).

USE CASE DIAGRAM

Può essere sviluppato a differenti livelli di astrazione (sia in fase di OOA che OOD).

Durante la fase di OOA, si concentra su COSA il sistema deve fare (scenari di funzionamento).

Un caso d'uso rappresenta:

- una funzionalità completa (flusso principale, sottoflussi e alternative)
- una funzionalità visibile dall'esterno
- un comportamento ortogonale (ogni use case viene eseguito in modo indipendente dagli altri)
- una funzionalità originata da un attore del sistema (una volta originato, il caso d'uso può interagire con altri attori)
- una funzionalità che produce un risultato significativo per un attore

IDENTIFICAZIONE DEI CASI D'USO

A partire da:

- Insieme dei requisiti utente
- Attori del sistema (insieme ai relativi obiettivi)

L'identificazione può essere facilitata facendosi guidare dalle seguenti domande:

- Quali sono i compiti principali svolti da ciascun attore?
- Un attore accede o modifica l'informazione nel sistema?
- L'attore rappresenta il tramite mediante cui il sistema viene informato di modifiche apportate in altri sistemi?
- L'attore deve essere informato di eventuali cambiamenti avvenuti nel sistema?

Durante la fase di OOA, i casi d'uso identificano le necessità degli attori del sistema.

SPECIFICA DI USE CASE DIAGRAM

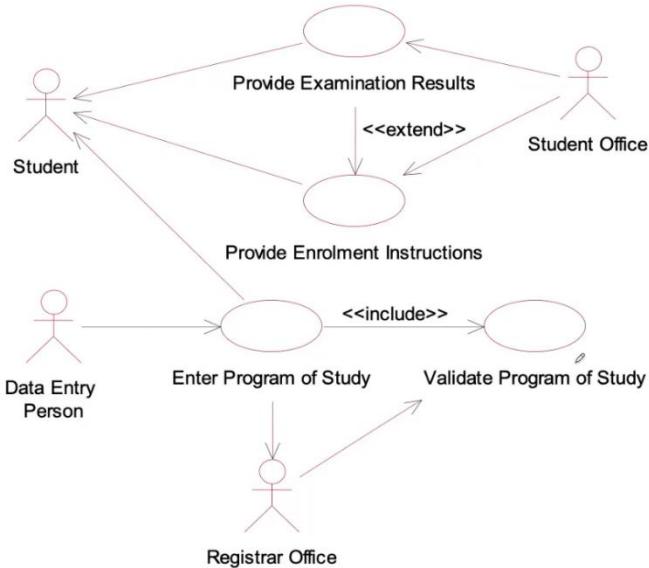
Rappresentazione grafica di attori, casi d'uso e relazioni.

Si possono rappresentare quattro tipi di relazioni:

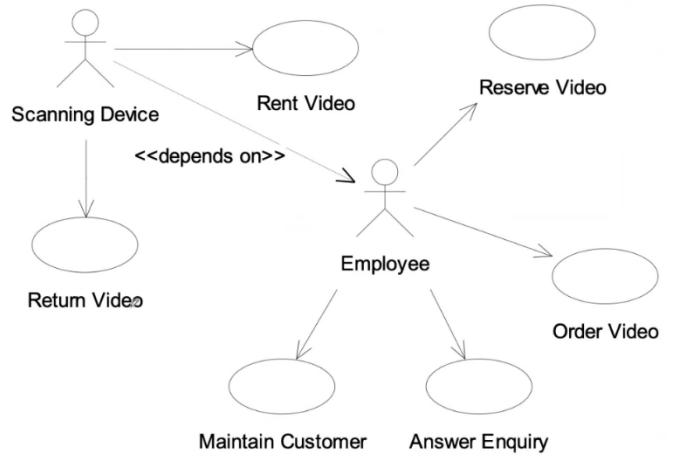
- Associazione (tra attore e caso d'uso)
- Include (identificato dallo stereotype: «include»)
 - o Un caso d'uso included è sempre necessario per completare il caso d'uso con il quale è messo in relazione

- Extend (identificato dallo stereotype : «extend»)
 - o Un caso d'uso extended può attivare il caso d'uso dal quale viene esteso (ma tale attivazione non è necessaria per completare il caso d'uso extended)
- Generalizzazione

ESEMPIO A.5 – UNIVERSITY ENROLMENT

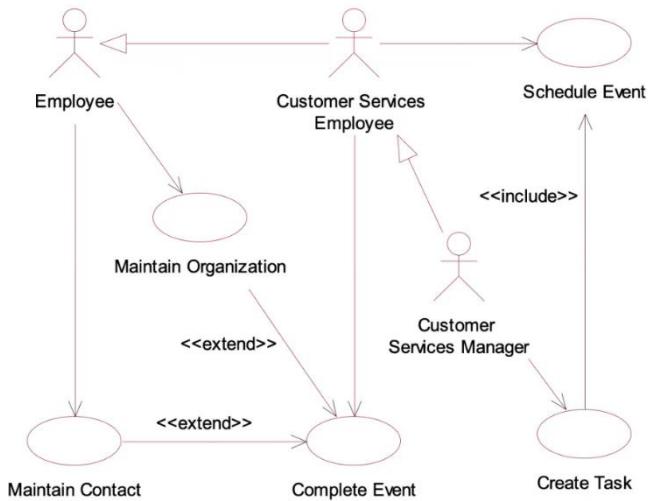


ESEMPIO B.4 – VIDEO STORE

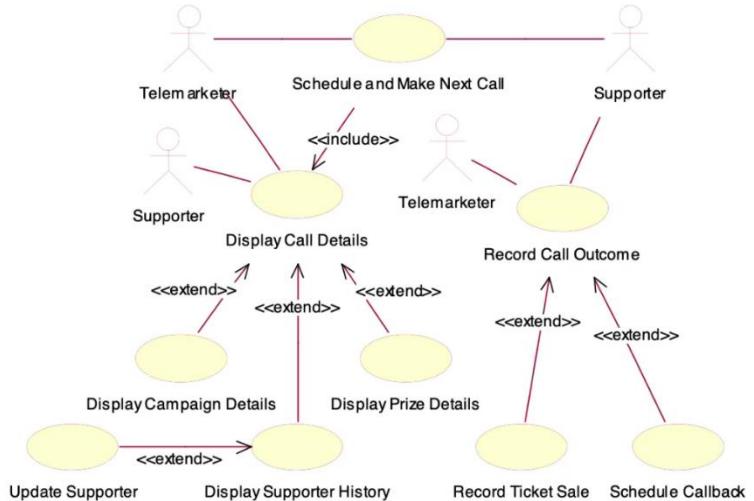


LEZ17 – 30/11/2023

ESEMPIO C.3 – CONTANT MANAGEMENT



ESEMPIO D.3 – TELEMARKETING



ACTIVITY DIAGRAM

Rappresenta a vari livelli di astrazione il flusso di esecuzione, sia sequenziale che concorrente, in una applicazione object-oriented.

È una variante degli state diagram, in cui gli stati rappresentano l'esecuzione di azioni e le transizioni sono attivate dal completamento di tali azioni.

Usato principalmente in fase di OOD per rappresentare il flusso di esecuzione delle operazioni definite nel class diagram.

In fase di OOA, viene usato per rappresentare il flusso delle attività nella esecuzione di un caso d'uso (un caso d'uso può essere associato ad uno o più activity diagram).

Poiché non vengono mostrati gli oggetti che eseguono le attività, può essere costruito anche in assenza del class diagram.

In presenza del class diagram, ogni attività può essere associata ad una o più operazioni appartenenti ad una o più classi.

SPECIFICA DI ACTIVITY DIAGRAM

Un evento (esterno) che origina un caso d'uso viene modellato come un evento che causa l'esecuzione di un activity diagram.

Gli action state vengono identificati a partire dalla descrizione testuale dello scenario di funzionamento di un caso d'uso.

Gli action state vengono quindi associati mediante transition lines (che possono essere controllate da guard conditions).

Le transizioni in uscita da un action state vengono percorse quando l'action state viene completato (l'esecuzione procede da un action state al successivo).

Un action state viene completato quando la sua elaborazione termina.

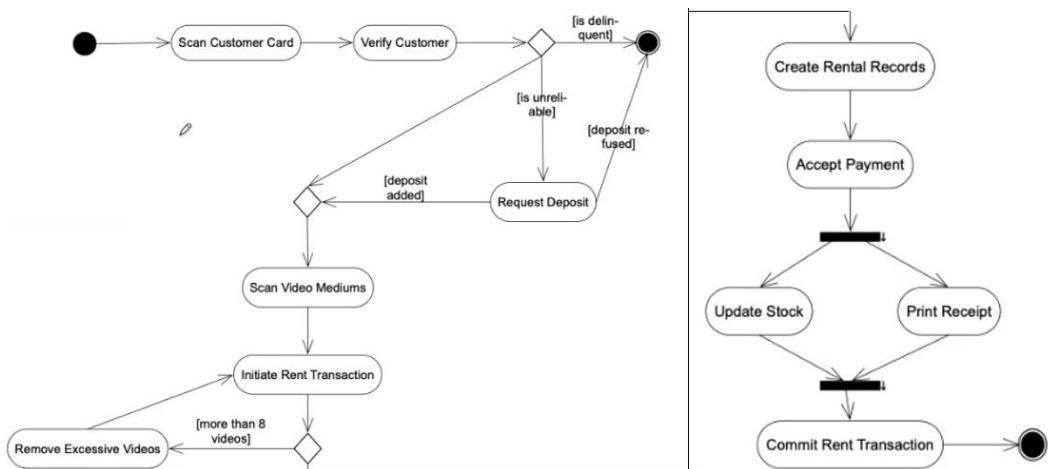
Flussi concorrenti vengono modellati con barre di sincronizzazione (barre fork-join).

Flussi alternativi vengono modellati con nodi decisionali (branch/merge diamonds).

Eventi esterni non sono generalmente modellati.

ESEMPIO B.5 (VIDEO STORE):

La specifica di activity diagram è la seguente :



DIAGRAMMI DI INTERAZIONE

Li troviamo in UML in due modi: sequenze diagram e collaboration diagram (equivalenti)

Sequence diagram:

- Describe lo scambio di messaggi tra oggetti in ordine temporale
- Usato principalmente in fase di OOA

Collaboration diagram

- Describe lo scambio di messaggi tra oggetti mediante relazioni tra gli oggetti stessi
- Usato principalmente in fase di OOD

Sequence diagram e collaboration diagram permettono di identificare le operazioni delle classi nel class diagram.

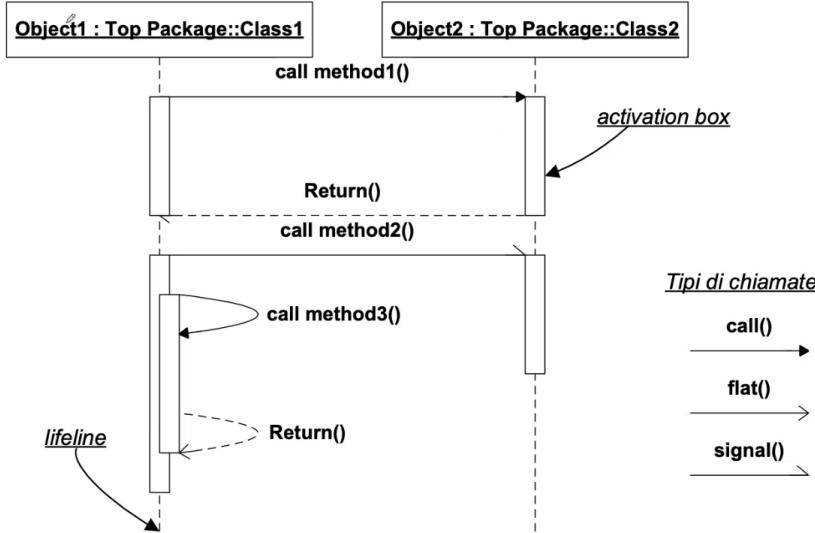
Sequence diagram e collaboration diagram sono rappresentazioni equivalenti e possono essere generati in modo automatico l'uno dall'altro.

SPECIFICA DI SEQUENCE DIAGRAM

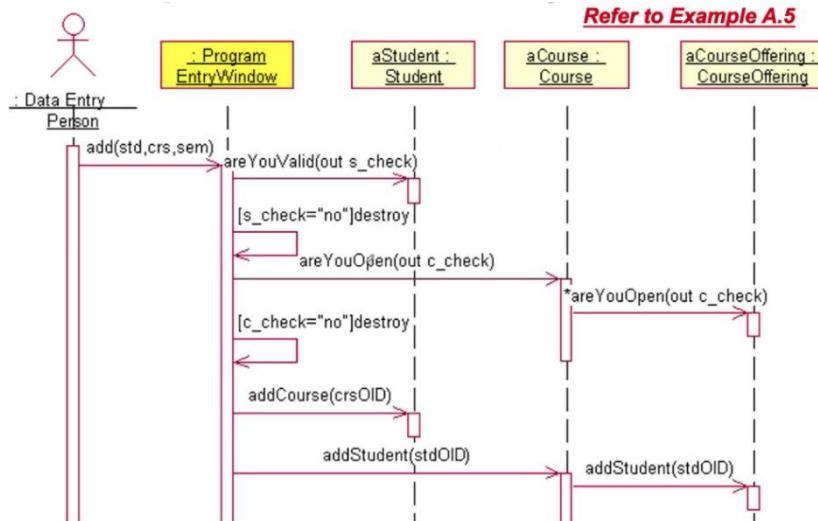
Le attività dell'activity diagram vengono mappate come messaggi (di tipo "richiesta esecuzione attività") in un sequence diagram. Un messaggio può rappresentare:

- Un signal
 - o Denota una chiamata di tipo asincrono
 - o L'oggetto mittente continua l'esecuzione dopo aver inviato il messaggio asincrono
- Una call
 - o Denota una chiamata di tipo sincrono
 - o L'oggetto mittente blocca l'esecuzione dopo aver inviato il messaggio sincrono, in attesa della risposta da parte dell'oggetto destinatario (che può o meno contenere valori di ritorno)

NOTAZIONE PER SEQUENCE DIAGRAM



ESEMPIO APPLICAZIONE SEQUENCE DIAGRAM (ESEMPIO A.6 – UNIVERSITY ENROLMENT)



LEZ18 – 04/12/2023

INTERFACCIA PUBBLICA DI CLASSE

Definisce l'insieme di operazioni che la classe mette a disposizione delle altre classi.

Durante la fase di OOA, si determina la signature dell'operazione, che consiste di: Nome dell'operazione, Lista degli argomenti formali, Tipo di ritorno.

Durante la fase di OOD, si definisce l'algoritmo che implementa l'operazione.

Una operazione può avere:

- Instance scope
- Class (static) scope
 - o rappresentata con un carattere \$ che precede il nome dell'operazione
 - o agisce su class object (classi con attributi statici)

IDENTIFICAZIONE DELLE OPERAZIONI

Dai sequence diagram ogni messaggio inviato ad un oggetto identifica un metodo della classe a cui appartiene tale oggetto.

Usando criteri aggiuntivi, come ad esempio il criterio CRUD, secondo cui ogni oggetto deve supportare le seguenti operazioni primitive (CRUD operations):

- Create (una nuova istanza)
- Read (lo stato di un oggetto)
- Update (lo stato di un oggetto)
- Delete (l'oggetto stesso)

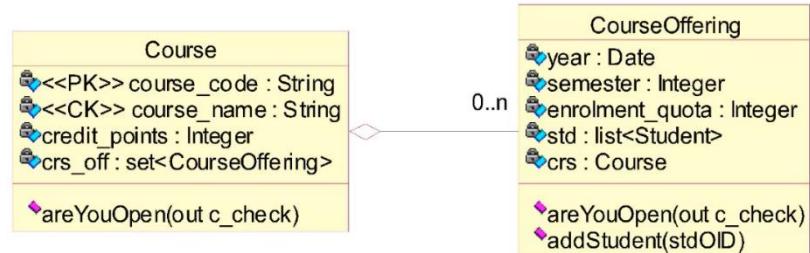
ESEMPIO A.7 – UNIVERSITY ENROLMENT

Riferendoci agli esempi A.3 e

A.6 e alle classi Course e

CourseOffering.

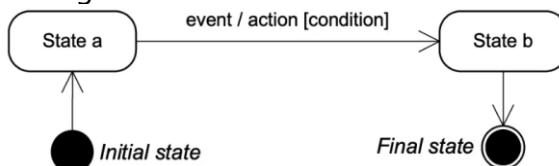
Derivare operazione dal Sequence Diagram e aggiungerle alle classi Course e CourseOffering:



MODELLO DINAMICO

Rappresenta il comportamento dinamico degli oggetti di una singola classe, in termini di stati possibili ed eventi e condizioni che originano transizioni di stato (assieme alle eventuali azioni da svolgere a seguito dell'evento verificatosi).

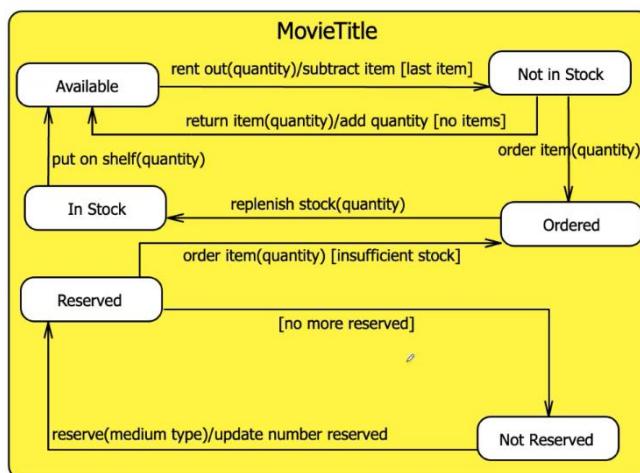
Fa uso del formalismo State Diagrams:



Viene costruito per ogni classe di controllo (per le quali è interessante descrivere il comportamento dinamico).

Usato principalmente per applicazioni scientifiche e real-time (meno frequentemente nello sviluppo di applicazioni gestionali).

ESEMPIO APPLICAZIONE DIAGRAMMA DEGLI STATI (B.5 – VIDEO STORE)



GESTIONE DELLA COMPLESSITA' NEI MODELLI OOA

Nella fase di OOA per sistemi software di grandi dimensioni occorre gestire in modo opportuno l'intrinseca complessità dei modelli.

Le associazioni tra classi nel modello dei dati formano complesse reti di interconnessione, in cui i cammini di comunicazione crescono in modo esponenziale con l'aggiunta di nuove classi. L'introduzione di gerarchie di classi permette di ridurre tale complessità da esponenziale a polinomiale, grazie all'introduzione di opportuni strati di classi che vincolano la comunicazione tra classi appartenenti allo stesso strato o a strati adiacenti.

UML PACKAGE

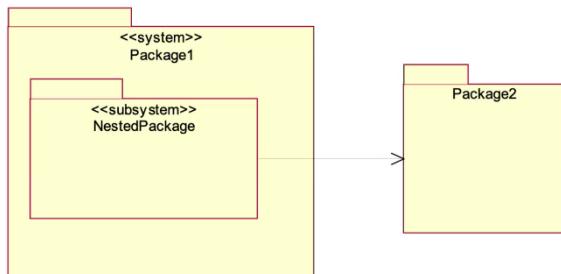
L'UML fornisce la nozione di package per rappresentare un gruppo di classi o di altri elementi (ad esempio casi d'uso).

I package possono essere annidati (il package esterno ha accesso ad ogni classe contenuta all'interno dei package in esso annidati).

Una classe può appartenere ad un solo package, ma può comunicare con classi appartenenti ad altri package.

La comunicazione tra classi appartenenti a package differenti viene controllata mediante dichiarazione di visibilità (private, protected, o public) delle classi all'interno dei package.

DIPENDENZA TRA PACKAGE



la relazione di dipendenza non è specificata, ma sta ad indicare che eventuali modifiche a Package2 potrebbero richiedere modifiche di NestedPackage.

PACKAGE DIAGRAM

In UML non esiste il concetto di package diagram.

I package possono essere creati all'interno o di class diagram o di use case diagram.

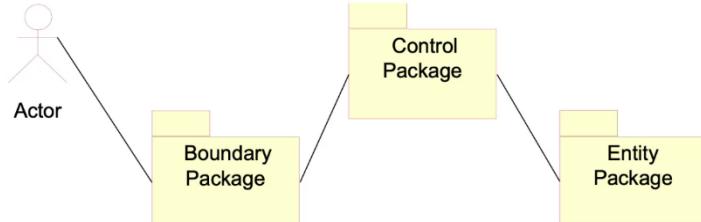
Si possono specificare due tipi di relazioni tra package:

- Generalization: implica anche dependency
- Dependency: usage dependency, access dependency, visibility dependency

APPROCCIO BCE

- Boundary package (BcE)
 - Descrive classi i cui oggetti gestiscono l'interfaccia tra un attore ed il sistema
 - Le classi catturano una porzione dello stato del sistema e la presentano all'utente in forma visuale
- Control package (BCE)
 - Descrive classi i cui oggetti intercettano l'input dell'utente e controllano l'esecuzione di uno scenario di funzionamento del sistema
 - Le classi rappresentano azioni ed attività di uno use case
- Entity package (BCE)
 - Descrive classi i cui oggetti gestiscono l'accesso alle entità fondamentali del sistema
 - Le classi corrispondono alle strutture dati gestite dal sistema

GERARCHIA DI PACKAGE BCE



OOA ESERCIZIO (da svolgere) – SISTEMA SW PER SHOPPING ONLINE

Requisiti Utente:

- Il sistema software deve supportare l'azienda X che vende computer online
- I clienti che accedono al sistema possono scegliere di acquistare un computer in configurazione standard o costruire una specifica configurazione selezionando i singoli elementi (ad es .: processore, disco, RAM, etc.)
- Per effettuare l'ordine, il cliente deve fornire le informazioni necessarie per la spedizione e per il pagamento
- Il cliente può usare il sistema per verificare online lo stato dell'ordine
- Il computer nella configurazione scelta viene inviato al cliente assieme alla relativa fattura (se richiesta)

Esercizio = sviluppare la specifica secondo OOA: soluzione in prossime pagine.

GESTIONE DI PROGETTI SOFTWARE (SOFTWARE PROJECT MANAGEMENT)

Lo sviluppo di un prodotto software è una operazione complessa che richiede una specifica attività di gestione.

La gestione di un progetto software implica la pianificazione, il monitoraggio ed il controllo di persone, processi ed eventi durante lo sviluppo del prodotto.

Il Software Project Management Plan (SPMP) è il documento che guida la gestione di un progetto software.

LE QUATTRO "P"

La gestione efficace di un progetto software si fonda sulle "quattro P":

- Persone, che rappresentano l'elemento più importante di un progetto software di successo (il SEI ha elaborato il People Management - Capability Maturity Model)
- Prodotto, che identifica le caratteristiche del software che deve essere sviluppato (obiettivi, dati, funzioni, comportamenti principali, alternative, vincoli)
- Processo, che definisce il quadro di riferimento entro cui si stabilisce il piano complessivo di sviluppo del prodotto software
- Progetto, che definisce l'insieme delle attività da svolgere, identificando persone, compiti, tempi e costi

ORGANIZZAZIONE DEI TEAM

→ Problema: sviluppare un prodotto software in 3 mesi con un impegno pianificato di 1 anno/uomo.

Soluzione immediata: 4 sviluppatori si suddividono il lavoro.

Realtà: i 4 sviluppatori potrebbero impiegare un anno ottenendo un prodotto di qualità inferiore a quello risultante dal lavoro di un singolo sviluppatore.

Motivi:

- alcuni compiti non possono essere condivisi
- necessità di frequenti interazioni

LEGGE DI BROOKS = L'aggiunta di uno sviluppatore potrebbe ritardare ulteriormente il progetto, a causa del periodo di formazione e dell'aumento delle interazioni (legge di Brooks, 1975)

APPROCCIO DEMOCRATICO

Introdotto da Weinberg (1971) e basato sul concetto di "egoless programming", secondo cui ogni sviluppatore valuta la scoperta di fault nel proprio codice come un attacco alla propria persona e non come un evento usuale.

L'approccio democratico punta ad organizzare team che lavorino con un obiettivo comune, senza un singolo leader.

Il codice appartiene al team come entità e non al singolo sviluppatore.

Vantaggi approccio democratico:

- atteggiamento positivo verso la ricerca di fault
- molto produttivo in caso di problemi difficili da risolvere (es. ambienti di ricerca)

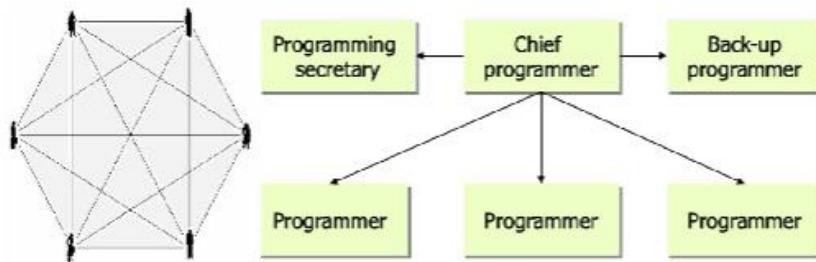
Svantaggi:

- l'approccio non può essere imposto dall'esterno ma deve nascere spontaneamente
- gli sviluppatori più anziani non desiderano essere valutati dai più giovani

APPROCCIO CON CAPO-PROGRAMMATORE (APPROCCIO GERARCHICO)

Basato su:

- specializzazione: ogni partecipante svolge i compiti per i quali è stato formato
- gerarchia: ogni sviluppatore comunica esclusivamente con il capo-programmatore, che dirige le attività ed è responsabile dei risultati



Vantaggi approccio gerarchico: diminuisce il numero di canali di programmazione.

Lo svantaggio principale è che richiede personale molto esperto per ricoprire gli incarichi di:

- capo-programmatore (è sia un manager che un tecnico, sviluppa il progetto architettonicale e le parti di codice critiche)
- programmatore di back-up (sostituisce il capo-programmatore ed è responsabile delle attività di testing)
- segretario di programmazione (responsabile della documentazione e dell'archivio di produzione)

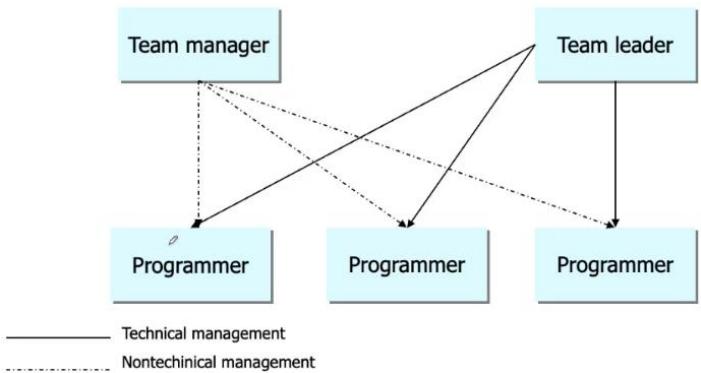
EVOLUZIONE DEGLI APPROCCI

Il capo-programmatore, essendo sia manager che tecnico, risulta essere il valutatore di sé stesso.

Va dunque sostituito con due individui:

- Team Leader (responsabile aspetti tecnici)
- Team Manager (responsabile aspetti gestionali)

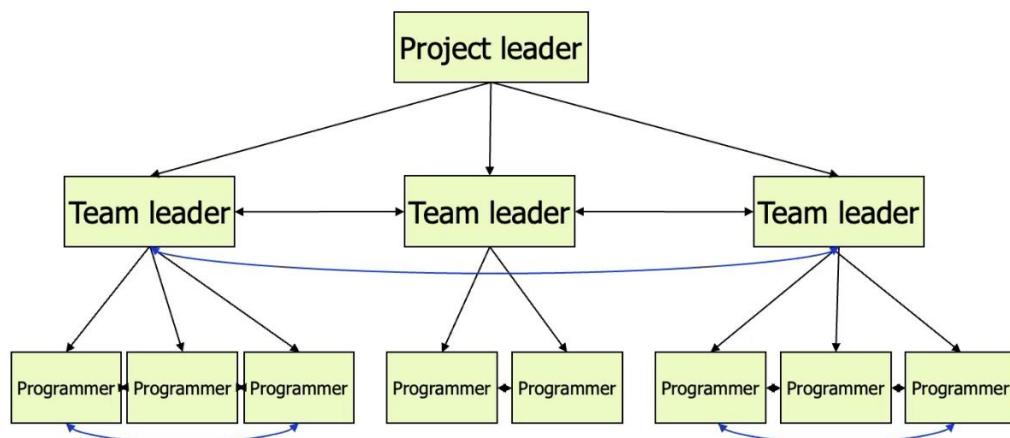
Problema: il team manager concede le ferie al programmatore senza contattare il team leader che invece è di parere contrario.



Soluzioni:

- identificare aree di responsabilità condivise
- introdurre un ulteriore livello di gestione, con un project leader che è responsabile dell'intero progetto e che comunica con i team leader

Organizzazione (scalabile) dei team per progetti SW di dimensione medio-grande, con decision-making decentralizzato



PIANIFICAZIONE DI PROGETTI SOFTWARE

Obiettivo: definire un quadro di riferimento per controllare, determinare l'avanzamento ed osservare lo sviluppo di un progetto software

Motivazione: essere in grado di sviluppare prodotti software nei tempi e costi stabiliti, con le desiderate caratteristiche di qualità

Componenti fondamentali:

- Scoping(raggio d'azione): comprendere il problema ed il lavoro che deve essere svolto
- Stime: prevedere tempi, costi e effort
- Rischi: definire le modalità per l'analisi e la gestione dei rischi
- Schedule: allocare le risorse disponibili e stabilire i punti di controllo nell'arco temporale del progetto
- Strategia di controllo: stabilire un quadro di riferimento per il controllo di qualità e per il controllo dei cambiamenti

STIME NEI PROGETTI SOFTWARE

Le attività di stima di tempi, costi ed effort nei progetti software sono effettuate con gli obiettivi di:

- ridurre al minimo il grado di incertezza
- limitare i rischi comportati da una stima

Risulta quindi necessario usare tecniche per incrementare l'affidabilità e l'accuratezza di una stima.

Le tecniche di stima possono basarsi su:

1. stime su progetti simili già completati (expert judgement by analogy)
2. "tecniche di scomposizione" (approccio bottom-up)
3. modelli algoritmici empirici

Le tecniche di scomposizione utilizzano una strategia "divide et impera" e sono basate su:

- stime dimensionali, ad es. LOC (Lines Of Code) o FP (Function Point)
- suddivisione dei task e/o delle funzioni con relativa stima di allocazione dell'effort

I modelli algoritmici empirici si basano su dati storici e su relazioni del tipo $d = f(v_i)$, dove d è il valore da stimare (es. effort, costo, durata) e v_i sono le variabili indipendenti (es. LOC o FP stimati).

LEZ20 – 18/12/2023

FUNCTION POINT -FP (UNITA' DI MISURA)

Function point (FP) è una misura ponderata di funzionalità software proposte da Albrecht (1979~1983). I function point misurano la quantità di funzionalità in un sistema basato sulla specifica di sistema (stima prima dell'implementazione).

FP viene calcolato in due passaggi:

1. Calcolo di un conteggio dei function point non regolato(unadjusted) (UFC).
2. Moltiplicare l'UFC per un fattore di complessità tecnica TCF (TCF).

Il function point finale (adjusted) è: $FP = UFC \times TCF$

Conteggi per le categorie di dati:

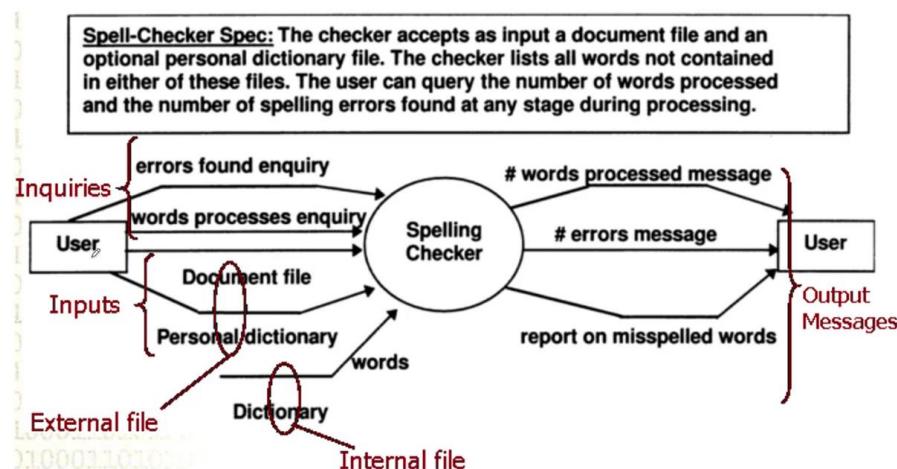
- Numero di file logici interni (ILF, Internal Logical File)
Un gruppo di dati o informazioni di controllo che viene generato, utilizzato o mantenuto dal sistema software (esempio memorizzazione dati sul db)
- Numero di file di interfaccia esterna (EIF, External Interface Files)
Un gruppo di dati o informazioni di controllo passate o condivise tra le applicazioni, ovvero le interfacce leggibili dalla macchina ad altri sistemi e/o all'utente.

Il termine "file" non significa file nel senso tradizionale dell'elaborazione dei dati. Si riferisce a un gruppo di dati correlato e non l'implementazione fisica dei gruppi di dati.

Conteggi per le categorie di transazioni:

- Numero di ingressi esterni (EI, External Inquiries)
Quegli elementi forniti dall'utente che descrivono dati orientati all'applicazione, informazioni di controllo (ad esempio nomi di file e selezioni di menu) o output di altri sistemi che entrano in un'applicazione e modificano lo stato dei file logici interni.
- Numero di uscite esterne (EO, External Outputs)
Tutti i dati univoci o le informazioni di controllo prodotti dai sistemi software, ad esempio report e messaggi.
- Numero di richieste esterne (EQ, External inQuires)
Tutte le combinazioni di ingresso/uscita univoche, in cui un ingresso provoca e genera un output immediato senza modifica dello stato dei file logici interni.

Esempio Spell Checker (correttore ortografico):



Componenti FP:

EI=2 (ingressi esterni): nome del file del documento, nome-dizionario-personale

EO=3 (uscite esterne): report di parole errate, numero di parole elaborate, numero messaggio di errori-finora

EQ=2 (richieste esterne): parole elaborate, Errori finora

EIF=2 (file di interfaccia esterna): file di documento, Dizionario personale

ILF=1 (file logici interni): dizionario

Assumere complessità media dei pesi per ogni elemento.

UFC=55

Function Type	Functional Complexity	Complexity Totals	Function Type Totals												
ILFs	<table> <tr><td>Low</td><td>X 7 =</td><td></td></tr> <tr><td><u>1</u></td><td>X 10 =</td><td><u>10</u></td></tr> <tr><td>Average</td><td>X 15 =</td><td></td></tr> <tr><td>High</td><td></td><td></td></tr> </table>	Low	X 7 =		<u>1</u>	X 10 =	<u>10</u>	Average	X 15 =		High				<u>10</u>
Low	X 7 =														
<u>1</u>	X 10 =	<u>10</u>													
Average	X 15 =														
High															
EIFs	<table> <tr><td>Low</td><td>X 5 =</td><td></td></tr> <tr><td><u>2</u></td><td>X 7 =</td><td><u>14</u></td></tr> <tr><td>Average</td><td>X 10 =</td><td></td></tr> <tr><td>High</td><td></td><td></td></tr> </table>	Low	X 5 =		<u>2</u>	X 7 =	<u>14</u>	Average	X 10 =		High				<u>14</u>
Low	X 5 =														
<u>2</u>	X 7 =	<u>14</u>													
Average	X 10 =														
High															
EIs	<table> <tr><td>Low</td><td>X 3 =</td><td></td></tr> <tr><td><u>2</u></td><td>X 4 =</td><td><u>8</u></td></tr> <tr><td>Average</td><td>X 6 =</td><td></td></tr> <tr><td>High</td><td></td><td></td></tr> </table>	Low	X 3 =		<u>2</u>	X 4 =	<u>8</u>	Average	X 6 =		High				<u>8</u>
Low	X 3 =														
<u>2</u>	X 4 =	<u>8</u>													
Average	X 6 =														
High															
EOs	<table> <tr><td>Low</td><td>X 4 =</td><td></td></tr> <tr><td><u>3</u></td><td>X 5 =</td><td><u>15</u></td></tr> <tr><td>Average</td><td>X 7 =</td><td></td></tr> <tr><td>High</td><td></td><td></td></tr> </table>	Low	X 4 =		<u>3</u>	X 5 =	<u>15</u>	Average	X 7 =		High				<u>15</u>
Low	X 4 =														
<u>3</u>	X 5 =	<u>15</u>													
Average	X 7 =														
High															
EOs	<table> <tr><td>Low</td><td>X 3 =</td><td></td></tr> <tr><td><u>2</u></td><td>X 4 =</td><td><u>8</u></td></tr> <tr><td>Average</td><td>X 6 =</td><td></td></tr> <tr><td>High</td><td></td><td></td></tr> </table>	Low	X 3 =		<u>2</u>	X 4 =	<u>8</u>	Average	X 6 =		High				<u>8</u>
Low	X 3 =														
<u>2</u>	X 4 =	<u>8</u>													
Average	X 6 =														
High															
Total Unadjusted Function Point Count			<u>55</u>												

FATTORI DI DEGREE OF INFLUENCE

Come si calcola la complessità tecnica al di là di quella funzionale? Si calcola a partire da 14 fattori chiamati fattori di grado di influenza.

Questi 14 fattori sono:

1. Reliable back-up and recovery (Backup e ripristino affidabili)
2. Data communications (Comunicazioni di dati)
3. Distributed data processing (Elaborazione distribuita dei dati)
4. Performance (Prestazione)
5. Heavily used configuration (Configurazione molto utilizzata)
6. Online data entry (Inserimento dati online)
7. Operational ease (Facilità d'uso)
8. Online update (Aggiornamento online)
9. Complex interface (Interfaccia complessa)
10. Complex processing (Lavorazioni complesse)
11. Reusability (Riusabilità)
12. Installation ease (Facilità di installazione)
13. Multiple sites (Siti multipli)
14. Facilitate change (Facilità di cambiamento)

Ad ognuno di questi fattori viene associato un valore intero compreso tra 0 e 5, dove 0 si utilizza se il fattore ha influenza irrilevante mentre 5 si utilizza quando il fattore è di influenza essenziale.

CALCOLO TFC

Ogni componente è valutato da 0 a 5, dove:

0 Non presente, o nessuna influenza

1 Influenza incidentale

2 Influenza moderata

3 Influenza media

4 Influenza significativa

5 Forte influenza dappertutto.

Il TCF varia da 0,65 (se tutti i F_i sono impostati a 0) a 1,35 (se tutti i F_j sono impostati a 5). Allora +/- 35% di regolazione.

Il TCF può essere calcolato come:

$$TCF = 0.65 + 0.01 \sum_{j=1}^{14} F_j$$

Suppose that:

$F_1 = 3$ Reliable back-up and recovery	$F_7 = 3$ Operational ease
$F_2 = 3$ Data communications	$F_8 = 3$ Online update
$F_3 = 0$ Distributed data processing	$F_9 = 0$ Complex interface
$F_4 = 5$ Performance	$F_{10} = 5$ Complex processing
$F_5 = 0$ Heavily used configuration	$F_{11} = 0$ Reusability
$F_6 = 3$ Online data entry	$F_{12} = 0$ Installation ease
	$F_{13} = 0$ Multiple sites
	$F_{14} = 3$ Facilitate change

then

$$TCF = 0.65 + 0.01(18+10) = 0.93$$

and

$$FP = 55 \times 0.93 \approx 51$$

FP vs LOC

Numerosi studi hanno tentato di mettere in relazione le metriche LOC e FP.

Il numero medio di istruzioni del codice sorgente per function point è stato derivato da casi di studio su numerosi linguaggi di programmazione.

I linguaggi sono stati classificati in diversi livelli in base al rapporto tra LOC e FP. Il nominal level aumenta con l'aumentare delle generazioni (esempio: C++ ha nomination level pari a 6.00, mentre linguaggio Basic Assembly ha nominal level pari a 1.00).

Language	Nominal level	Source statements per function point		
		Low	Mean	High
First generation	1.00	220	320	500
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
Basic (interpreted)	2.50	70	128	165
Second generation	3.00	55	107	165
Fortran	3.00	75	107	160
Algol	3.00	68	107	165
Cobol	3.00	65	107	170
CMS2	3.00	70	107	135
Jovial	3.00	70	107	165
Pascal	3.50	50	91	125

COCOMO (ESEMPIO DI MODELLO ALGORITMICO)

COCOMO (COnstructive COst Model) è il modello introdotto da Boehm (1981) per determinare il valore dell'effort.

Il valore ottenuto per l'effort viene successivamente utilizzato per determinare durata e costi di sviluppo.

COCOMO comprende 3 modelli:

- Basic (per stime iniziali)
- Intermediate (usato dopo aver suddiviso il sistema in sottosistemi)
- Advanced (usato dopo aver suddiviso in moduli ciascun sottosistema)

La stima dell'effort viene effettuata a partire da:

- stima delle dimensioni del progetto in KLOC
- stima del modo di sviluppo del prodotto, che misura il livello intrinseco di difficoltà nello sviluppo, tra:
 - o organic (per prodotti di piccole dimensioni)
 - o semidetached (per prodotti di dimensioni intermedie)
 - o embedded (per prodotti complessi)

Nel 1995 è stato introdotto COCOMO II, più flessibile e sofisticato rispetto alla versione precedente.

MODELLO INTERMEDIATE, MODO ORGANIC→ **Passo 1**

Determinare l'effort nominale usando la formula:

$$\text{effort nominale} = 3.2 \times (\text{KLOC})^{1.05} \text{ MM}$$

Esempio: $3.2 \times (33)^{1.05} = 126 \text{ MM}$

→ **Passo 2**

Ottenerne la stima dell'effort applicando un fattore moltiplicativo C basato su 15 cost drivers:

$$\text{effort} = \text{effort nominale} \times C$$

Esempio: $126 \times 1.15 = 145 \text{ MM}$

→ **C (cost driver multiplier)** si ottiene come produttoria dei cost driver ci.

Ogni ci; determina la complessità del fattore i che influenza il progetto e può assumere uno tra più valori assegnati con variazioni intorno al valore unitario (valore nominale)

TABELLA DI COST DRIVER (INTERMEDIATE COCOMO)

Cost Drivers	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*	0.87	1.00	1.15	1.30		
Computer turnaround time	0.87	1.00	1.07	1.15		
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

COCOMO TIME SCHEDULE

Stima del tempo T alla consegna (product delivery):

– Modo *organic* $T = 2.5 E^{0.38}$ (months M)

– Modo *semi-detached* $T = 2.5 E^{0.35}$

– Modo *embedded* $T = 2.5 E^{0.32}$

STIMA COSTI DI SVILUPPO (DEVELOPMENT COSTS ESTIMATION)

I costi di sviluppo (C) sono stimati dall'allocazione dello sforzo di sviluppo (E) su fasi e attività del personale, ad esempio:

- 16% progettazione preliminare
 - o 50% responsabile di progetto
 - o 50% analista
- 62% di progettazione, codifica e test dettagliati
 - o 75% programmatore/analista
 - o 25% programmatore
- 22% Integrazione
 - o 30% analista
 - o 70% programmatore/analista

Il costo per persona/mese di ciascuna categoria di personale (ad esempio, project manager, analista, programmatore, ecc.) viene poi utilizzato per ottenere i costi di sviluppo.

PIANIFICAZIONE TEMPORALE

Dopo aver scelto il modello di processo, identificato i task da eseguire e stimato durata, costi ed effort, è necessario effettuare la pianificazione temporale ed il controllo dei progetti.

La pianificazione temporale consiste nel definire una "rete di task" in base ai seguenti principi fondamentali:

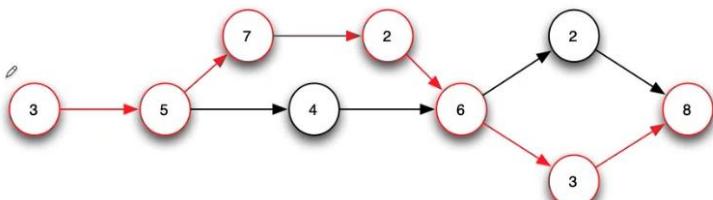
- ripartizione: scomposizione di processo e prodotto in parti (task e funzioni) di dimensioni ragionevoli
- interdipendenza: identificazione delle dipendenze reciproche tra i task individuati
- allocazione di risorse: determinazione di numero di persone, effort e date di inizio/fine da assegnare ad ogni task
- responsabilità definite: individuazione delle responsabilità assegnate a ciascun task
- risultati previsti: definizione dei risultati prodotti al termine di ogni task
- punti di controllo (milestone): identificazione dei punti di controllo della qualità da associare al singolo task o a gruppi di task

STRUMENTI DI PIANIFICAZIONE

2 strumenti:

→ 1) Diagramma PERT (Program Evaluation and Review Technique)

- grafo in cui ogni nodo rappresenta un task ed ogni arco un legame di precedenza
- consente di determinare:
 - o il cammino critico (sequenza di task che determina la durata minima di un progetto)
 - o la stima del tempo di completamento di ciascun task, mediante applicazione di modelli statistici
 - o i limiti temporali di inizio e termine di ciascun task

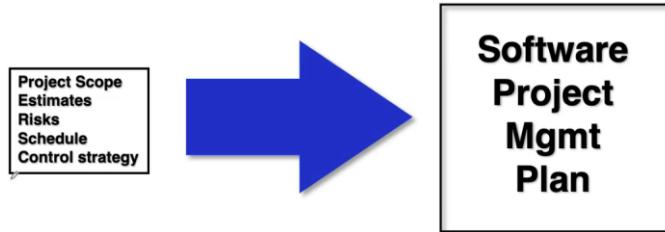


→ 2) Carta di Gantt

- diagramma a barre che consente di visualizzare l'allocazione temporale dei task
- non appare nessuna indicazione dei legami di precedenza, quindi viene integrata con un diagramma PERT



SPMP (Software Project Management Plan)



Esempio template per contenuti del SPMP:

Title page	5. Managerial process plans	6. Technical process plans
Signature page	5.1 Start-up plan	6.1 Process model
Change history	5.1.1 Estimation plan	6.2 Methods, tools and techniques
Preface	5.1.2 Staffing plan	6.3 Infrastructure plan
Table of contents	5.1.3 Resource acquisition plan	6.4 Product acceptance plan
List of figures	5.1.4 Project staff training plan	7. Supporting process plans
List of tables	5.2 Work plan	7.1 Configuration management plan
1. Overview	5.2.1 Work activities	7.2 Verification and validation plan
1.1 Project summary	5.2.2 Schedule allocation	7.3 Documentation plan
1.1.1 Purpose, scope and objective	5.2.3 Resource allocation	7.4 Quality assurance plan
1.1.2 Assumptions and constraints	5.2.4 Budget allocations	7.5 Reviews and audits
1.1.3 Project deliverables	5.3 Control plan	7.6 Problem-resolution plan
1.1.4 Schedule and budget summary	5.3.1 Requirements control plan	7.7 Subcontractor management plan
1.2 Evolution of the plan	5.3.2 Schedule control plan	7.8 Process improvement plan
2. References	5.3.3 Budget control plan	8. Additional plans
3. Definitions	5.3.4 Quality control plan	Annexes
4. Project organization	5.3.5 Reporting plan	Index
4.1 External interfaces	5.3.6 Metrics collection plan	
4.2 Internal structure	5.4 Risk management plan	
4.3 Roles and responsibilities	5.5 Closeout plan	

LEZ22 – 08/01/2024 (Soluzione esercizio assegnato in lez17)

OOA ESERCIZIO (da svolgere) – SISTEMA SW PER SHOPPING ONLINE

Requisiti Utente:

- Il sistema software deve supportare l'azienda X che vende computer online
- I clienti che accedono al sistema possono scegliere di acquistare un computer in configurazione standard o costruire una specifica configurazione selezionando i singoli elementi (ad es.: processore, disco, RAM, etc.)
- Per effettuare l'ordine, il cliente deve fornire le informazioni necessarie per la spedizione e per il pagamento
- Il cliente può usare il sistema per verificare online lo stato dell'ordine
- Il computer nella configurazione scelta viene inviato al cliente assieme alla relativa fattura (se richiesta)

Esercizio = sviluppare la specifica secondo OOA:

1. Si produca inizialmente il modello dei dati costruendo un class diagram in cui le operazioni di ciascuna classe possono essere omesse. Per ciascuna associazione devono invece essere specificate le molteplicità, mentre i nomi di ruolo possono essere omessi.
2. Successivamente, si produca una porzione di modello comportamentale identificando attori e casi d'uso e specificando la descrizione di un caso d'uso a scelta, sia in forma testuale che usando un activity diagram.
3. A partire dal caso d'uso identificato, si produca un sequence diagram che mostri una possibile interazione tra gli oggetti del sistema.
4. Infine, a partire dal sequence diagram, si produca un raffinamento del class diagram iniziale, identificando le operazioni ed eventuali classi, associazioni o attributi aggiuntivi.

SVOLGIMENTO:

Class Diagram

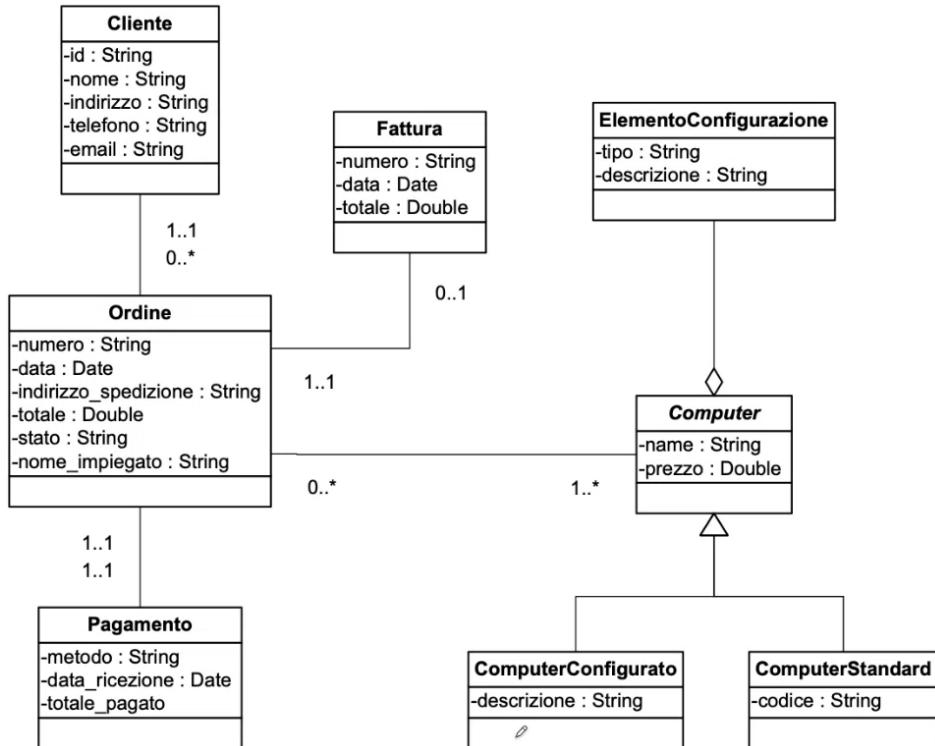
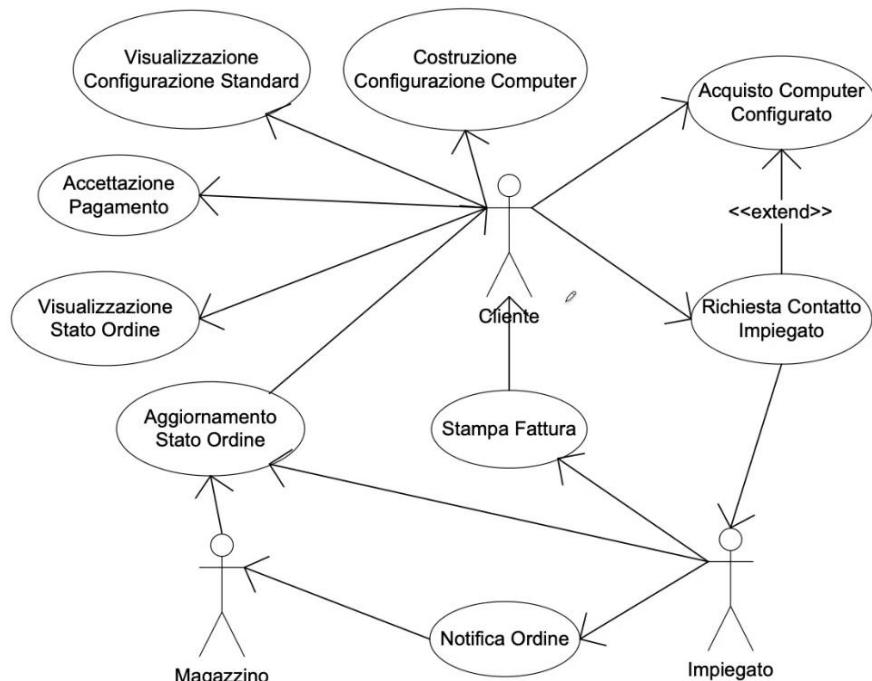


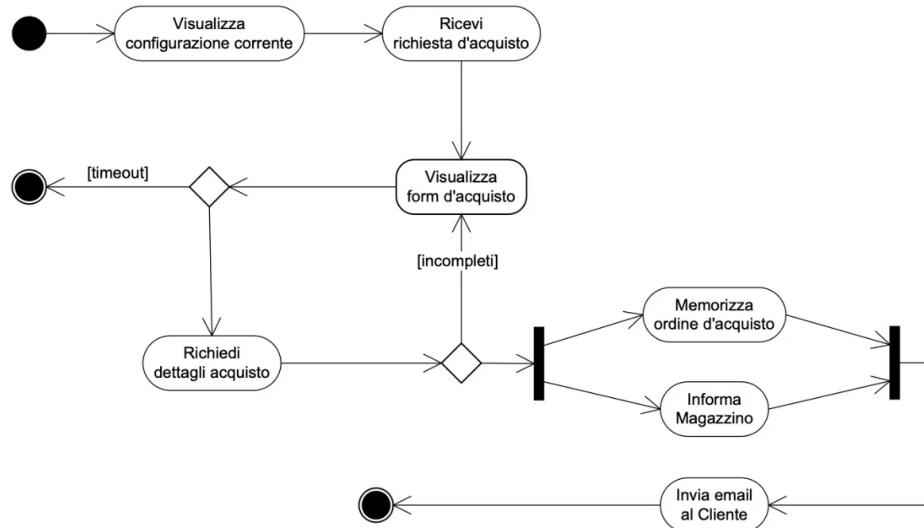
Diagramma casi d'uso (Use Case Diagram)



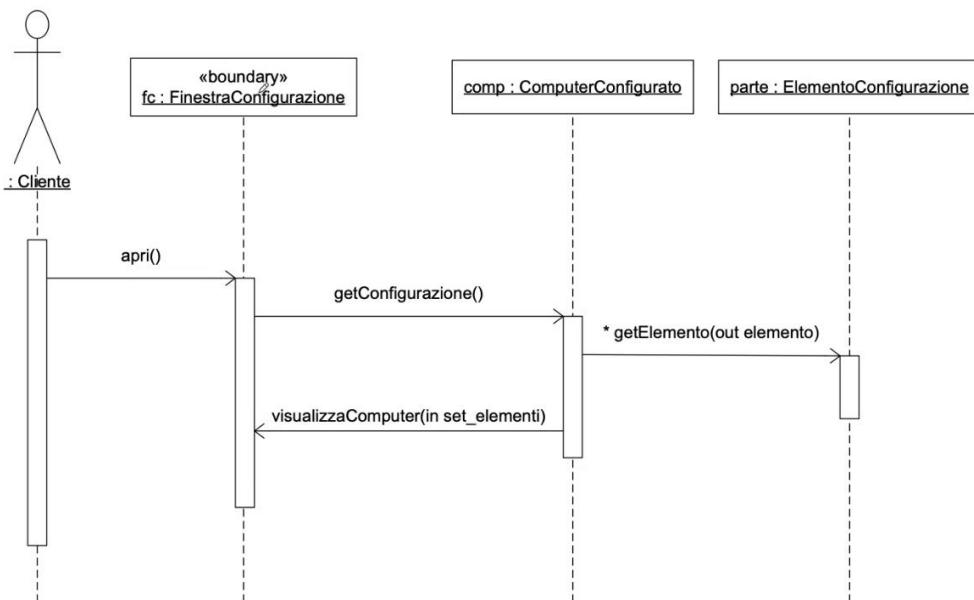
Use case acquisto computer configurato (descrizione testuale)

Brief Description	Il caso d'uso permette al Cliente di inserire un ordine di acquisto....
Actors	Cliente
Preconditions	L'interfaccia utente mostra a video i dettagli del computer configurato, insieme al prezzo relativo...
Main Flow	Il sistema assegna un numero unico e l'identificatore del cliente all'ordine di acquisto e ne memorizza le informazioni nel database...
Alternative Flows	Il Cliente attiva la funzionalità di acquisto prima di aver fornito tutte le informazioni necessarie...
Postconditions	L'ordine di acquisto viene registrato nel database di sistema.

Use case acquisto computer configurato (descrizione flusso con activity diagram)

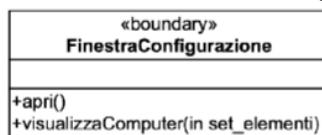


Sequence Diagram (visualizza configurazione corrente)



Nota: manca classe di controllo in questo sequence diagram! (BE e non BCE)

Raffinamento Class Diagram



ALTRI ESERCIZI

Ripetere l'esercizio sviluppando i requisiti utente di altri sistemi software (si possono "inventare" ipotizzando di analizzare lo specifico dominio applicativo ...). Ad esempio:

- Gestione filiali di una banca
- Gestione contratti di una compagnia di assicurazione
- Gestione scommesse sportive
- Gestione corsi universitari
- Gestione posta elettronica
- Etc.

LEZ23 – 04/03/2024 (INIZIO SECONDA PARTE DEL CORSO)

LA FASE DI PROGETTO

Fase in cui si decidono le modalità di passaggio da "che cosa" deve essere realizzato nel sistema software a "come" la realizzazione deve aver luogo.

La fase di progetto prende in input il documento di specifica (analisi dei requisiti) e produce un documento di progetto che guida la successiva fase di codifica.

La fase di progetto può essere suddivisa in due sotto-fasi:

- progetto architetturale (o preliminare), in cui il sistema software complessivo viene suddiviso in più sottosistemi (decomposizione modulare)
- progetto dettagliato, in cui ogni sottosistema identificato viene progettato in dettaglio, scegliendo algoritmi e strutture dati specifiche

PRINCIPI DI PROGETTAZIONE

- a) stepwise refinement (usato anche in fase di analisi dei requisiti)
- b) astrazione (usato anche in fase di analisi dei requisiti)
- c) decomposizione modulare
- d) modularità
- e) information hiding
- f) riusabilità

STEPWISE REFINEMENT

Il procedere per raffinamenti successivi (stepwise refinement) è una strategia di progettazione top-down proposta da Wirth (1971) nell'ambito della programmazione strutturata.

Il raffinamento è un processo di elaborazione che parte dalla specifica di una funzione (o di dati) in cui non si descrive il funzionamento interno della funzione o la struttura interna dei dati. Il raffinamento elabora la specifica aggiungendo ad ogni passo un livello di dettaglio maggiore. L'importanza del raffinamento deriva dalla legge di Miller (1956): "at any one time a human being can concentrate on at most 7 + 2 chunks (quanta of information)". Il raffinamento è un concetto complementare al concetto di astrazione.

ASTRAZIONE

L'astrazione consiste nel concentrarsi sugli aspetti essenziali di una entità e nell'ignorare i dettagli secondari.

Il concetto di livello di astrazione è stato introdotto da Djikstra (1968) nell'ambito dei sistemi operativi, al fine di descriverne l'architettura a strati.

Nell'ambito del processo software, ogni passo rappresenta un raffinamento del livello di astrazione della soluzione. Ciò significa concentrarsi su cosa è e cosa fa una entità del sistema software prima di decidere come debba essere realizzata.

I principali tipi di astrazione sono:

- astrazione procedurale (es. funzioni C)
- astrazione dei dati (es. data encapsulation)

Con il termine data encapsulation ci si riferisce ad una struttura dati insieme alle azioni eseguite su di essa (es. stack, struttura dati con le azioni che realizzano il meccanismo LIFO).

L'uso di data encapsulation in fase di progetto permette di ottenere vantaggi sia in fase di codifica che in fase di manutenzione.

TIPO DI DATO STRATTO

Un tipo di dato astratto (abstract data type) identifica un tipo di dato insieme alle azioni eseguite sulle istanze del tipo di dato.

Un tipo di dato astratto combina astrazione procedurale e astrazione dei dati.

L'uso dei tipi di dati astratti permette di migliorare la qualità del software in relazione agli attributi di riusabilità e manutenibilità.

Una classe C++ è un esempio di tipo di dato astratto che inoltre supporta il meccanismo di Ereditarietà.

MODULARITA'

Prodotti software fatti di un unico, monolitico, blocco di codice sono difficili a:

- manutenere
- correggere
- capire
- riusare

La soluzione consiste nel suddividere il prodotto software in segmenti più piccoli detti moduli.

Definizione di modularità secondo lo standard IEEE Std 610.12:

la misura in cui il software è composto da componenti discreti in modo tale che una modifica a un componente abbia un impatto minimo sugli altri componenti.

DECOMPOSIZIONE MODULARE

- Un modulo è un elemento software che:
- contiene istruzioni, logica di elaborazione e strutture dati
- può essere compilato separatamente e memorizzato all'interno di una libreria software
- può essere incluso in un programma
- può essere usato invocando segmenti di modulo identificati da un nome e da una lista di parametri
- può usare altri moduli

La suddivisione in moduli di un sistema software (decomposizione modulare) produce come risultato l'identificazione di una architettura dei moduli (structure chart).

L'architettura dei moduli di un sistema software descrive la struttura dei moduli, il modo in cui tali moduli interagiscono e la struttura dei dati manipolati.

La decomposizione modulare si basa sul principio del "divide et impera".

Detti p1 e p2 due problemi, C la complessità ed E l'effort si ha che:

$$\begin{aligned}
 C(p_1) &> C(p_2) \Rightarrow E(p_1) > E(p_2) \\
 C(p_1+p_2) &> C(p_1) + C(p_2) \\
 &\Downarrow \\
 E(p_1+p_2) &> E(p_1) + E(p_2)
 \end{aligned}$$

Una buona divisione di un prodotto software in moduli è quella che permette di ottenere:

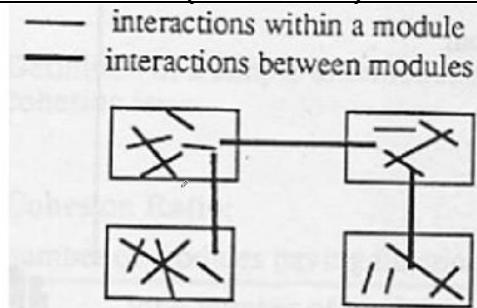
- Massima coesione (cohesion) interna ai moduli
- Minimo grado di accoppiamento (coupling) tra i moduli

Infatti, massima coesione e minimo coupling permettono di incrementare:

- Comprensibilità
- Manutenibilità
- Estensibilità
- Riusabilità

del prodotto software.

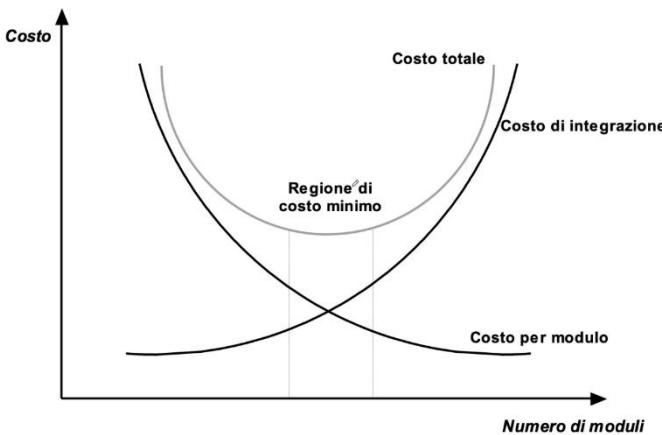
COESIONE(COHESION)/ACCOPIAMENTO(COUPLING) RISPETTO A MODULARITA'



Coesione linee all'interno dei singoli moduli (confinate all'interno del modulo e non interagiscono con altri)(si cerca di massimizzare la coesione).

Accoppiamento rappresentato con linee più spesse che rappresentano interazioni tra moduli.

MODULARITA' E COSTO DEL SOFTWARE



COESIONE

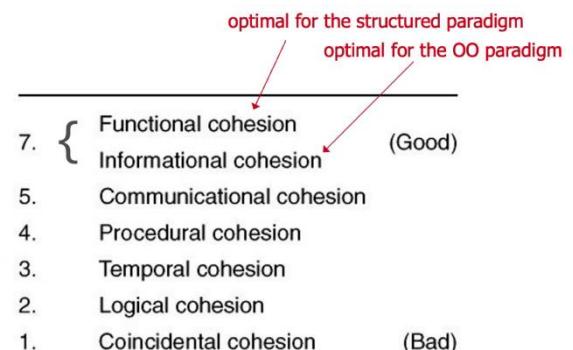
Per eseguire una funzione sono necessarie varie azioni.

Le azioni possono essere concentrate in un singolo modulo oppure sparse tra tanti.

Coesione di un modulo = Misura in cui il modulo espletà internamente tutte le azioni necessarie a espletare una data funzione (cioè senza interagire con le azioni interne ad altri moduli). Coesione misura dunque il grado di interazione interna al modulo tra le azioni di una funzione.

Livelli di Coesione (1 è il peggiore, 7 il migliore):

1. Coincidental (nessuna relazione tra gli elementi del modulo).
2. Logical (elementi correlati, di cui uno viene selezionato dal modulo chiamante)
3. Temporal (relazione di ordine temporale tra gli elementi).
4. Procedural (elementi correlati in base ad una sequenza predefinita di passi da eseguire).
5. Communicational (elementi correlati in base ad una sequenza predefinita di passi che vengono eseguiti sulla stessa struttura dati).
6. Informational (ogni elemento ha una porzione di codice indipendente e un proprio punto di ingresso ed uscita; tutti gli elementi agiscono sulla stessa struttura dati).
7. Functional (tutti gli elementi sono correlati dal fatto di svolgere una singola funzione)



ESEMPIO COINCIDENTAL COHESION (COESIONE COINCIDENTALE)

Module functions:

- print next line
- invert characters of the second string parameter
- add 7 to the fifth parameter
- perform int-double conversion to the fourth parameter

ESEMPIO LOGICAL COHESION (COESIONE LOGICA)

Module performing all input and output

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
- ...
37. Code for keyboard input

ESEMPIO TEMPORAL COHESION (COESIONE TEMPORALE)

Module functions:

- Open old_master_file
- Open new_master_file
- Open transaction_file
- Open print_file
- Initialize sales_region_table
- Read first transaction_file records
- Read first old_master_file record

ESEMPIO PROCEDURAL COHESION (COESIONE PROCEDURALE)

Module functions:

- Read part_number from database
- Use part_number to update repair_record on maintenance_file

ESEMPIO COMMUNICATIONAL COHESION (COESIONE COMUNICATIVA)

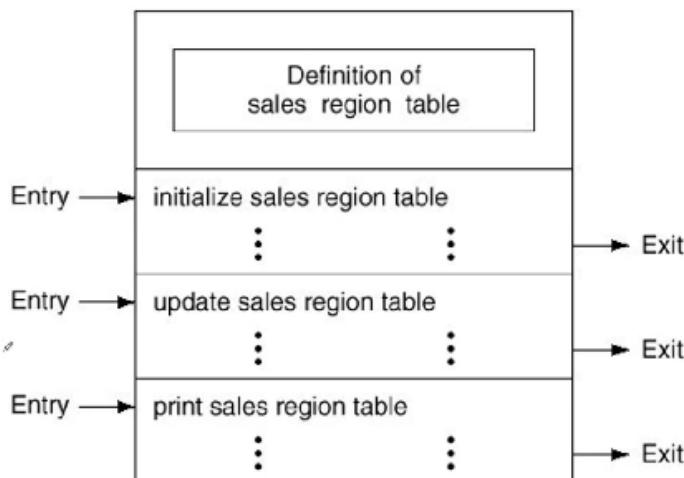
Ex. 1: module functions

- Update record_a in database
- Write record_a to the trajectory_file

Ex. 2: module functions

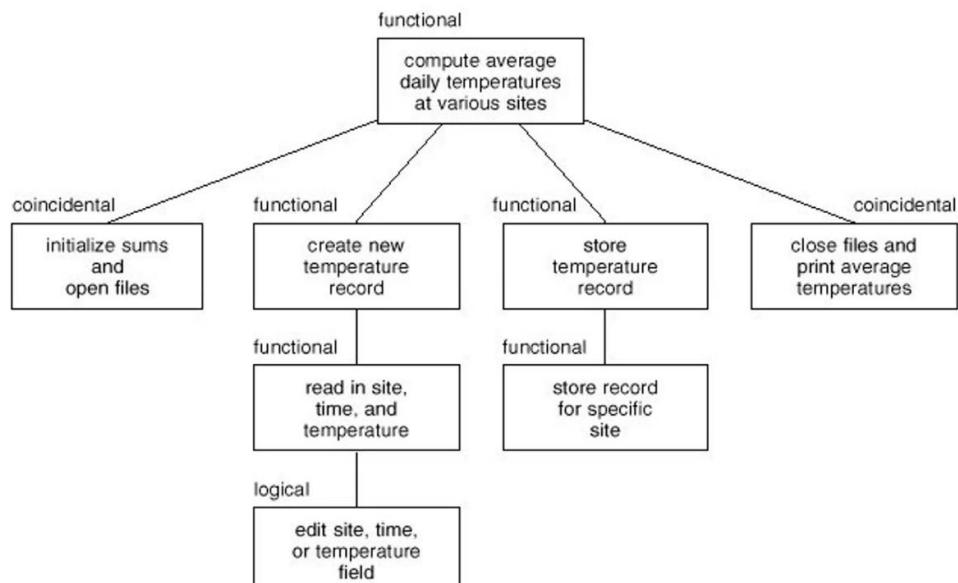
- Calculate new_trajectory
- Send new_trajectory to the printer

ESEMPIO INFORMATIONAL COHESION



LEZ24 - 06/03/2024

ESEMPIO ARCHITETTURA COMPOSTA DA MODULI CON COESIONE



COUPLING(ACCOPPIAMENTO)

Misura il grado di accoppiamento tra moduli.

Livelli di coupling (1 è il peggiore, 5 il migliore):

1. Content (un modulo fa diretto riferimento al contenuto di un altro modulo).
2. Common (due moduli che accedono alla stessa struttura dati)
3. Control (un modulo controlla esplicitamente l'esecuzione di un altro modulo).
4. Stamp (due moduli che si passano come argomento una struttura dati, della quale si usano solo alcuni elementi).
5. Data (due moduli che si passano argomenti omogenei, ovvero argomenti semplici o strutture dati delle quali si usano tutti gli elementi).

FATTORI CHE INFLUENZANO IL COUPLING

La forza dell'accoppiamento dipende da:

- il numero di riferimenti di un modulo con un altro
- la quantità di dati trasmessi/condivisi tra moduli
- la complessità dell'interfaccia tra moduli
- l'entità del controllo esercitato da un modulo su un altro

ESEMPIO CONTENT COUPLING

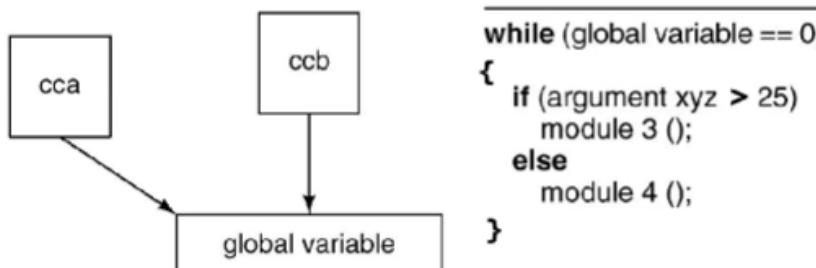
p--->q

Es. 1: il modulo p modifica un'istruzione del modulo Q (go to)

Es. 2: p si riferisce ai dati locali del modulo q in termini di alcuni numeri spostati all'interno di q

Es. 3: p si ramifica in un'etichetta locale di q

ESEMPIO COMMON COUPLING

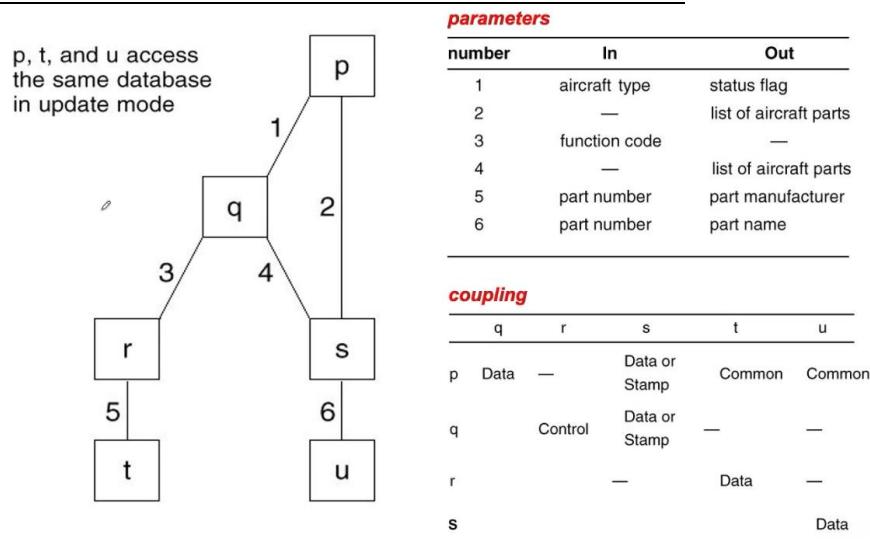


ESEMPIO CONTROL COUPLING

Il modulo p chiama il modulo q

- e chiede a q di eseguire un'azione,
- q restituisce un flag (ad esempio "task non completato")
- e chiede anche a p di eseguire un'azione (ad es. "stampa un messaggio di errore")

ESEMPIO STRUTTURA COMPOSTA DA MODULI CON COUPLING



INFORMATION HIDING

I concetti di astrazione procedurale e astrazione dei dati sono derivati da un concetto più generale detto information hiding, introdotto da Parnas(1971).

La tecnica di information hiding consiste nel definire e progettare i moduli in modo che i dettagli implementativi (procedura e dati) non siano accessibili ad altri moduli che non abbiano necessità di conoscere tali dettagli.

I vantaggi della tecnica di information hiding si riscontrano quando è necessario apportare modifiche (fasi di testing e manutenzione).

RIUSABILITÀ

La riusabilità fa riferimento all'utilizzo di componenti sviluppati per un prodotto all'interno di un prodotto differente.

Per componente riusabile si intende non solo un modulo o un frammento di codice, ma anche progetti, parti di documenti, insiemi di test data o stime di costi e durata.

Vantaggi:

- netta diminuzione di costi e tempi di produzione del software
- incremento dell'affidabilità dovuto all'uso di componenti già convalidati

La riusabilità nella fase di progetto si applica a moduli software, application framework(che incorpora la logica di controllo di un progetto), design pattern(che identifica una soluzione di progetto ricorrente in applicazioni dello stesso tipo), architetture software comprendenti i tre precedenti.

OBJECT ORIENTED DESIGN – OOD

La fase OOD è composta dalle seguenti due sottofasi:

- OOD preliminare (o architettonico, o di sistema): definisce la strategia globale per costruire una soluzione che risolva il problema specificato al momento dell'OOA. Vengono prese decisioni che riguardano l'organizzazione generale del software (architettura di sistema)
- OOD dettagliato (o oggetti): fornisce la definizione completa delle classi e le associazioni da implementare, nonché le strutture dati e l'algoritmo dei metodi che implementano le operazioni delle classi.

Secondo uno sviluppo iterativo e incrementale il modello OOA viene "trasformato" nel modello OOD, che aggiunge i dettagli tecnici della soluzione hardware/software che definisce il modo in cui il software deve essere attuato.

ARCHITETTURA DI SISTEMA (SYSTEM ARCHITECTURE)

Un'architettura di sistema definisce la struttura dei componenti software del sistema insieme alle relazioni tra tali componenti e i principi che guidano la progettazione e l'evoluzione del sistema.

Evoluzione delle architetture di sistema:

1. Architetture basate su mainframe (nota:mainframe è monoprocessore ma multi-utente)
2. Architetture di condivisione di file
3. Architetture client/server (C/S):
 - 3.1 A due livelli (thin client, fat client)
 - 3.2 Tre livelli (strato superiore, strato intermedio, strato inferiore)
4. Architetture ad oggetti distribuiti
5. Architetture component-based (basate su componenti)
6. Architetture service-oriented (orientati ai servizi)

Le architetture da 3 a 6 sono indicate come architetture distribuite, o architetture di sistemi software distribuiti.

DISTRIBUTED SOFTWARE SYSTEMS (SISTEMI SW DISTRIBUITI)

L'elaborazione di un sistema software distribuito è distribuita su un insieme di host di esecuzione indipendenti, collegati da un'infrastruttura di rete.

Il set di host di esecuzione indipendenti viene visualizzato dagli utenti come singolo host di esecuzione.

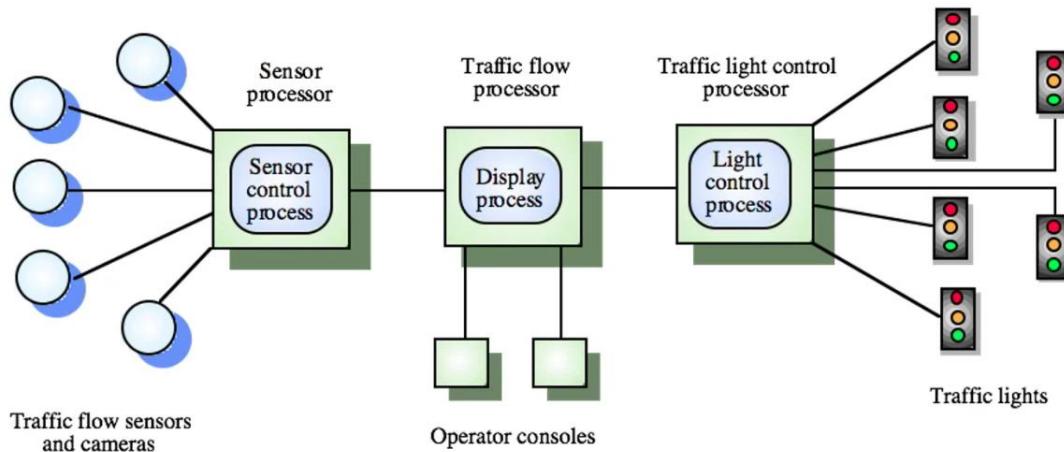
La tecnologia middleware ha svolto un ruolo essenziale nella transizione da architetture centralizzate a architetture distribuite.

Il middleware si riferisce al livello software che fornisce connettività.

Tale livello si trova tra il livello applicazione e il livello operazione di sistema e fornisce una serie di servizi per stabilire l'interazione richiesta tra i vari processi eseguiti da host di rete.

ESEMPIO DI SISTEMA SOFTWARE DISTRIBUITO

Sistema di controllo del traffico: composto da più processi che vengono eseguiti su processori diversi (potrebbero essere eseguiti anche su un singolo processore: è l'insieme di processi che rendono distribuito un sistema software)



CARATTERISTICHE PRINCIPALI DEI SISTEMI DISTRIBUITI:

Vantaggi:

- Condivisione di dati e risorse
- Apertura, capacità di gestire risorse eterogenee
- Concorrenza
- Scalabilità
- Bilanciamento del carico
- Tolleranza ai guasti
- Trasparenza (esempio trasparenza di locazione dei dati)
- Adattabilità agli scenari di elaborazione aziendale

Fattori critici:

- qualità del servizio (prestazioni, affidabilità, ecc.)
- interoperabilità
- sicurezza

ARCHITETTURE CLIENT/SERVER (C/S ARCHITECTURES)

Ogni processo svolge il ruolo di client o server.

Il processo client interagisce con l'utente nel modo seguente:

- fornisce l'interfaccia utente per raccogliere le richieste degli utenti
- inoltra le richieste ai server, utilizzando la tecnologia middleware
- visualizza le risposte del server all'utente tramite l'interfaccia utente

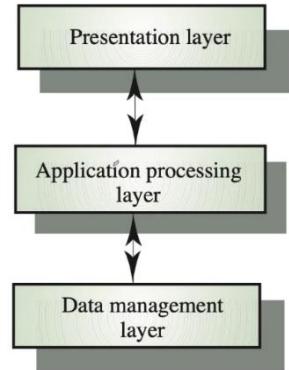
Il processo server (o l'insieme di processi eseguiti da un dato host) fornisce servizi ai clienti, come segue:

- risponde alle richieste dei client (non è il server che avvia la conversazione con il client)
- nasconde all'utente la complessità dell'intero sistema C/S (un dato server può a sua volta fungere da client che inoltra la richiesta iniziale a un server secondario, senza rendere il client e l'utente consapevoli della catena di inoltro).

Un'architettura C/S partiziona le applicazioni software in termini di una serie di processi, ciascuno dei quali funge da client, server o ambedue.

APPLICATION LAYERS

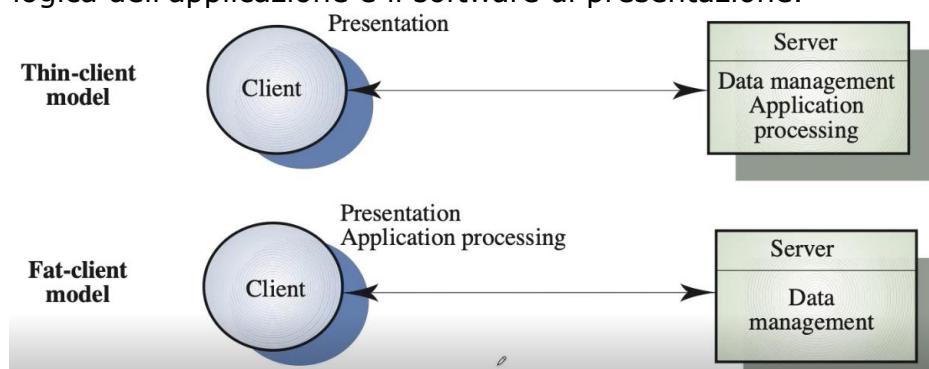
- Presentation layer, si occupa della raccolta dell'input dell'utente e della presentazione dei risultati di una computazione agli utenti di sistema
- Application processing layer, si occupa di fornire funzionalità specifiche dell'applicazione, ad esempio in un sistema bancario le funzioni bancarie come l'apertura dell'account, la chiusura ecc...
- Data management layer, si occupa di gestire l'accesso ai dati dell'applicazione



LEZ25 – 11/03/2024

TWO-TIER ARCHITETTURE CLIENT SERVER

- Modello thin-client (modello a client leggero): il client è semplicemente responsabile dell'esecuzione del software di presentazione, tutta la gestione dei dati e tutta l'elaborazione dell'applicazione sono effettuate sul server.
- Modello fat-client: il server è responsabile solo della gestione dei dati; il software sul client implementa la logica dell'applicazione e il software di presentazione.

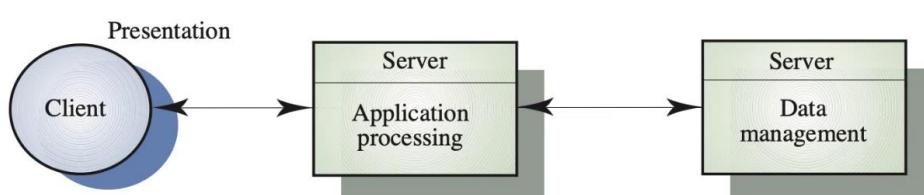


THREE-TIER ARCHITETTURE CLIENT SERVER

Ognuno dei livelli dell'architettura dell'applicazione viene eseguito su un processore separato.

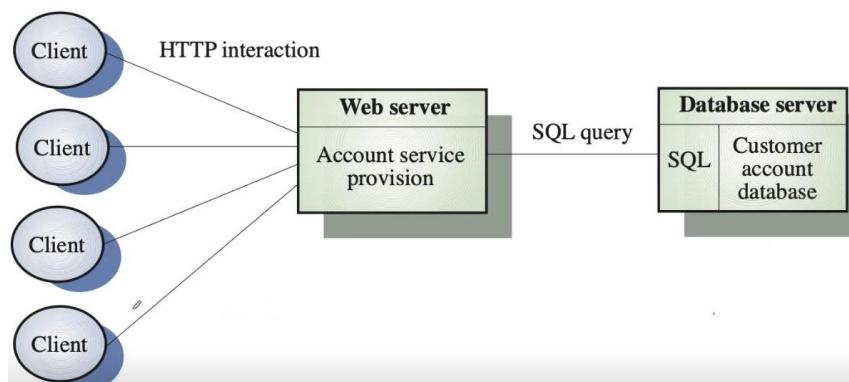
Consente prestazioni migliori rispetto a un approccio al client di tipo two-tier ed è più semplice da gestire rispetto a un approccio fat-client a due livelli.

Un'architettura più scalabile: con l'aumento delle richieste, è possibile aggiungere altri server.



Esempio C/S three-tier architecture:

Internet Banking System



ARCHITETTURE A OGGETTI DISTRIBUITE

- Nessuna distinzione tra client e server!

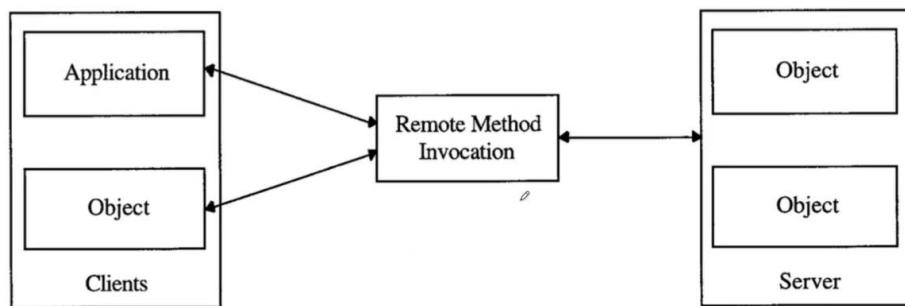
Ogni oggetto distribuito funge sia da client (invia messaggi di richiesta, cioè invocando metodi) e come server (fornendo messaggi di risposta, ad es. esecuzione del metodo richiamato).

La comunicazione remota tra gli oggetti viene effettuata in modo trasparente grazie all'uso di middleware basati su software con concetto di bus (denominato object request broker, ORB):

- bus astratto: specifica dell'interfaccia che fornisce servizi di comunicazione e di scambio di dati (control transfer model e type model per i valori scambiati)
- implementazione del bus: implementazione del bus astratto per una data piattaforma HW/SW (=> netta separazione tra interfaccia e implementazione).

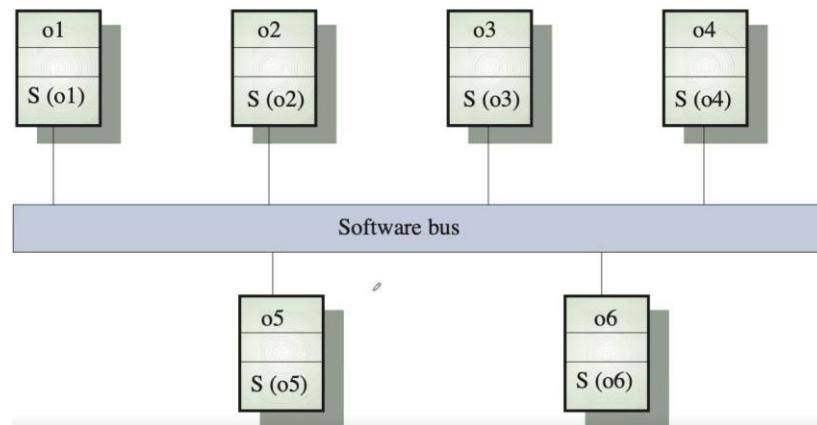
Le applicazioni basate su architetture ad oggetti distribuiti sono costituite da un insieme di oggetti che vengono eseguiti su piattaforme distribuite ed eterogenee e che comunicano tramite chiamate al metodo remoto.

L'esempio più famoso di abstract bus (bus astratto) è CORBA (che sta per Common Object Request Broker Architecture), che è uno standard (una specifica) pubblicato da OMG.



(Remote method invocation è un'interfaccia fornita dall'ORB (object request broker))

Esempio di distributed object architecture:



ARCHITETTURE COMPONENT-BASED

Idea: architettura basata su componenti nasce con l'idea di prendere componenti preconfezionate che forniscono determinate funzionalità e invece di codificarle mi limito a assemblare questi componenti tra di loro.

Le architetture basate su componenti(component-based) definiscono i prodotti software assemblati da un insieme di componenti software, che sono progettati per lavorare insieme come parte di un component framework.

Un component framework fa uso di architetture di software generico per formalizzare determinate classi di applicazioni.

I sistemi software component-based supportano l'efficienza nello sviluppo di sistemi software i cui requisiti presentano livelli significativi di variabilità.

È quindi necessario identificare e implementare il software in astrazioni che racchiudono soluzioni efficienti e affidabili per problemi standard di coordinazione e sintesi.

Queste astrazioni, o "componenti", vengono utilizzate per costruire nascondendo i dettagli di attuazione dei sistemi di struttura più piccola.

Un componente può essere utilizzato in molte applicazioni diverse, e può essere riconfigurato quando l'applicazione lo richiede.

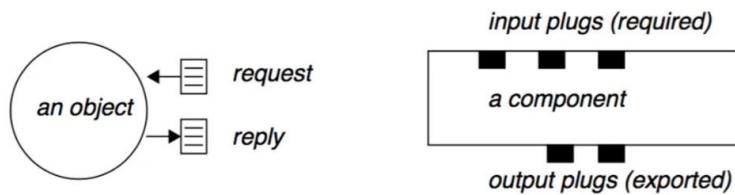
- ➔ Black-box-reus: un elemento essenziale per la creazione di sistemi software basati su componenti è il riutilizzo chiamato "a scatola nera" del software.
Mettere insieme i componenti è semplice, poiché ogni componente ha un set limitato di "plugs" con regole fisse che specificano come possono essere collegate con altri componenti.
Invece di dover adattare la struttura di un software a modificarne la funzionalità, un utente inserisce il comportamento desiderato nei parametri del componente.

Aspetti importanti dei componenti sono:

- encapsulamento di strutture software come componenti astratte (variability)
- composizione dei componenti associando i loro parametri a valori specifici, oppure altri componenti (adattabilità)
-

Oggetti vs Componenti:

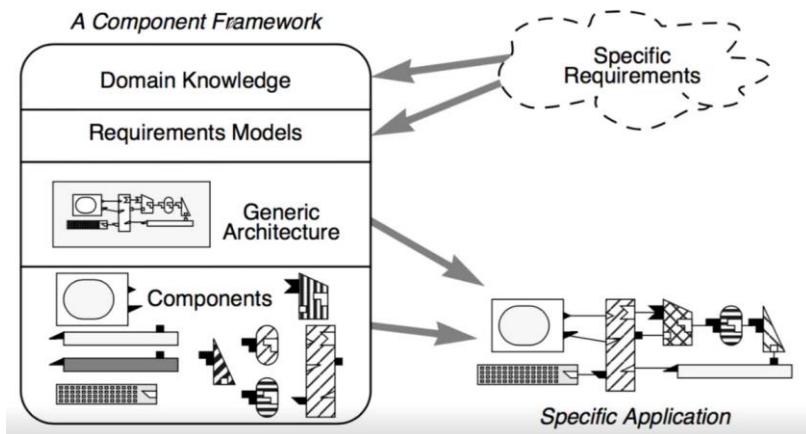
- Gli oggetti incapsulano i servizi, mentre i componenti sono astrazioni (che possono essere usate per costruire sistemi orientati agli oggetti)
- Gli oggetti hanno identità, stato e comportamento e sono sempre entità in fase di esecuzione; i componenti d'altra parte sono generalmente entità statiche necessarie in fase di compilazione del sistema (e non esistono necessariamente in fase di esecuzione).
- I componenti possono essere di granularità più fine o più grossolana rispetto agli oggetti: ad esempio, classi, modelli, mix-in, moduli; i componenti dovrebbero avere un'interfaccia di composizione esplicita, che è controllabile per tipo.



COMPONENT FRAMEWORK

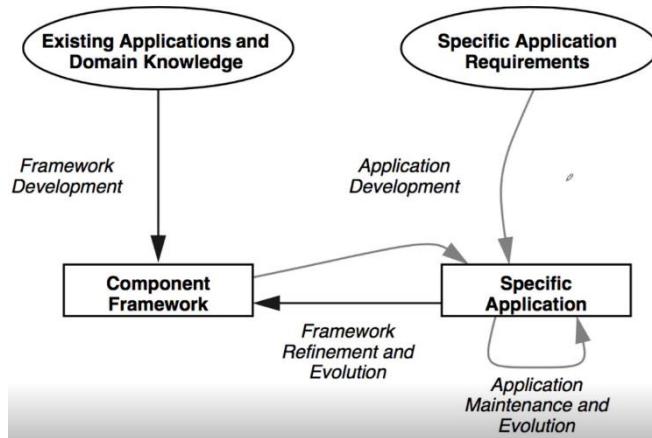
Il component framework non mette semplicemente a disposizione una libreria di componenti, ma fornisce molto di più in ottica sviluppo sw come architetture software generiche che danno il modo di assemblare componenti per fornire funzionalità di alto livello, tali funzionalità soddisfano un certo insieme di requisiti.

I component framework sono specifici per ogni dominio.



Quindi quello che fa il component framework è catturare un insieme di requisiti generici per un particolare dominio, dopodiché si occupa anche di fornire la relativa architettura software che in base all'insieme di componenti che vengono messe a disposizione permette di realizzare applicazioni che soddisfano modelli di requisiti.

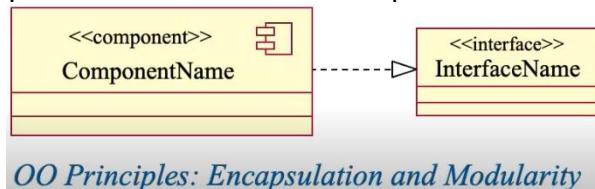
FRAMEWORK DEVELOPER vs APPLICATION DEVELOPER



UML COMPONENTS

Una parte modulare di un sistema che nasconde il suo contenuto e il cui aspetto è sostituibile all'interno del suo ambiente.

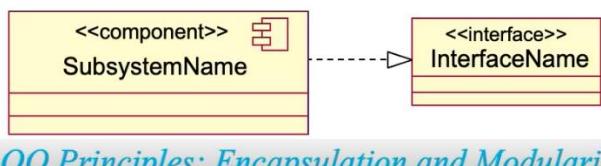
- Definisce il suo comportamento in termini di interfacce fornite e richieste che possono essere cablate tra loro
- Può essere sostituito in fase di progettazione o di esecuzione da un componente che offre funzionalità equivalenti in base alla compatibilità delle sue interfacce.



OO Principles: Encapsulation and Modularity

COS'E' UN SOTTOSISTEMA?

La parte di un sistema che incapsula comportamento, espone un insieme di interfacce, e contiene altri elementi model.
(modellato come componente)



OO Principles: Encapsulation and Modularity

SERVICE ORIENTED ARCHITECTURE (SOA)

Una SOA è un'architettura software distribuita che è costituita da più servizi autonomi. I servizi sono distribuiti in modo tale da poter eseguire su nodi diversi con diversi servizi provider.

Con una SOA, l'obiettivo è quello di sviluppare applicazioni software composte da servizi distribuiti, in modo tale che i singoli servizi possano essere eseguiti su piattaforme diverse ed essere implementate in diverse lingue.

PROTOCOLLI SOA

I protocolli standard basati su internet (internet-based) sono forniti per consentire servizi di comunicazione e per lo scambio di informazioni.

Ogni servizio ha una descrizione del servizio, che consente alle applicazioni di individuare e comunicare con il servizio.

La descrizione del servizio definisce il nome del servizio, l'ubicazione del servizio e i suoi requisiti per lo scambio di dati.

SERVICE PROVIDERS AND CONSUMERS (FORNITORI E CONSUMATORI DI SERVIZI)

Un fornitore di servizi supporta i servizi utilizzati da più client.

A differenza delle architetture client/server, le SOA si basano sul concetto di servizi ad accoppiamento debole(loosely coupled services) che possono essere scoperti e collegati dai client (anche denominati consumatori di servizi) con l'assistenza di service brokers.

CONCETTI DI PROGETTAZIONE SOA

1. Un obiettivo importante della SOA è quello di progettare i servizi come componenti autonome riutilizzabili.
2. I servizi sono destinati ad essere autonomi e debolmente accoppiati, il che significa che le dipendenze tra i servizi sono ridotte al minimo.
3. Invece di un servizio che dipende da un altro, servizi coordinati sono forniti in situazioni in cui è necessario accedere a più servizi e l'accesso ad essi deve essere in sequenza.
4. Vengono descritti diversi modelli di architettura software per applicazioni orientate ai servizi:
 - Modelli di broker, tra cui la registrazione del servizio, il servizio d'intermediazione (brokering), e individuazione dei servizi
 - Modelli di transazione, incluso il two-phase commit, modelli di transazione composti e di lunga durata
 - Modelli di negoziazione

SERVICES DESIGN PRINCIPLES

I principi di progettazione dei servizi sono:

- Loose coupling (servizi indipendenti il più possibile)
- Service contract (contratto tra consumatore e provider, il provider garantisce di essere in grado di soddisfare quanto promette e il consumatore si impegna a usare il servizio come si deve)
- Autonomia (ogni servizio il più possibile indipendente)
- Astrazione (dettagli implementativi dei servizi vengono nascosti)
- Riusabilità
- Componibilità
- Stateless
- Discoverability (i provider vogliono che i consumatori siano informati del servizio)

LEZ26 – 13/03/2024

SOFTWARE ARCHITECTURAL BROKER PATTERNS (MODELLI DI BROKER)

DELL'ARCHITETTURA SOFTWARE

In una SOA, gli oggetti broker fungono da intermediari tra client e servizi.

Nel modello Broker (noto anche come modello Object Broker o Object Request Broker pattern), il broker funge da intermediario tra il Client e i servizi.

Registrazione dei servizi presso il broker.

I clienti individuano i servizi tramite il broker.

Dopo che il broker ha negoziato la connessione tra il cliente e il servizio, la comunicazione tra il cliente e il servizio può essere diretta o tramite il broker.

TRASPARENZA

Il broker fornisce 2 tipi di trasparenza: la platform transparency e la location transparency.

- Local transparency (trasparenza di locazione) significa che se il servizio viene spostato in un'altra posizione, i clienti non vengono informati di questo spostamento e solo il broker ha bisogno di esserne a conoscenza.
- Platform transparency (trasparenza della piattaforma) significa che ogni servizio può essere eseguito su una piattaforma hardware/software diversa e non ha bisogno di mantenere informazioni sulle piattaforme su cui si eseguono altri servizi.

BROKERED COMMUNICATION Con la brokered communication (comunicazione negoziata), il client non ha bisogno di conoscere l'ubicazione di un determinato servizio, il client interroga il broker per i servizi forniti.

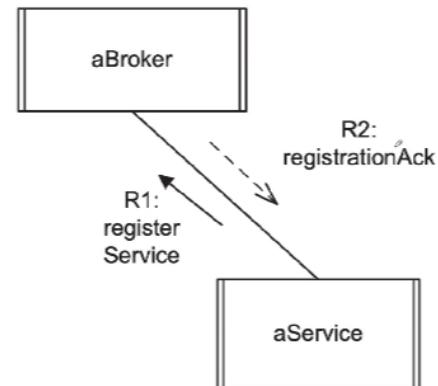
Come prima cosa è necessario che il servizio si registri presso un broker come descritto dal Service Registration pattern.

SERVICE REGISTRATION PATTERN

Il servizio deve registrare le informazioni di servizio con l'intermediario, compreso il nome del servizio, una sua descrizione e l'ubicazione in cui viene fornito.

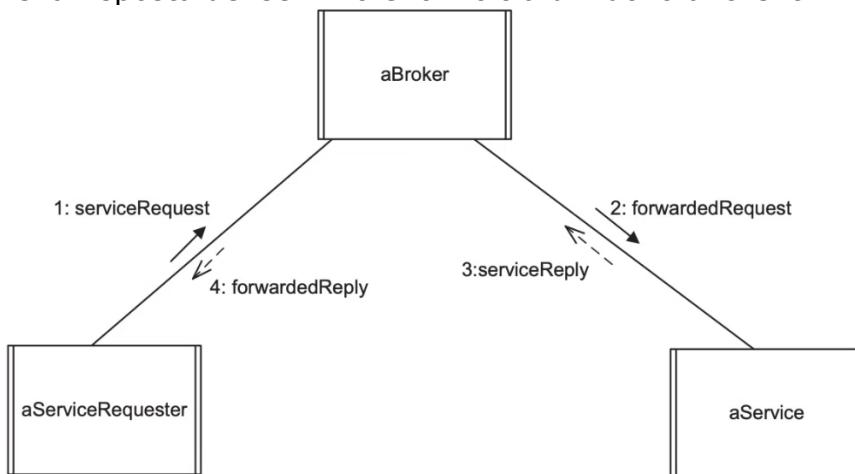
Sequenza dei messaggi:

1. Il servizio invia una register service request (richiesta di registro del servizio) all'intermediario
2. Il broker registra il servizio nel registro dei servizi e invia un avviso di conferma di registrazione al servizio.



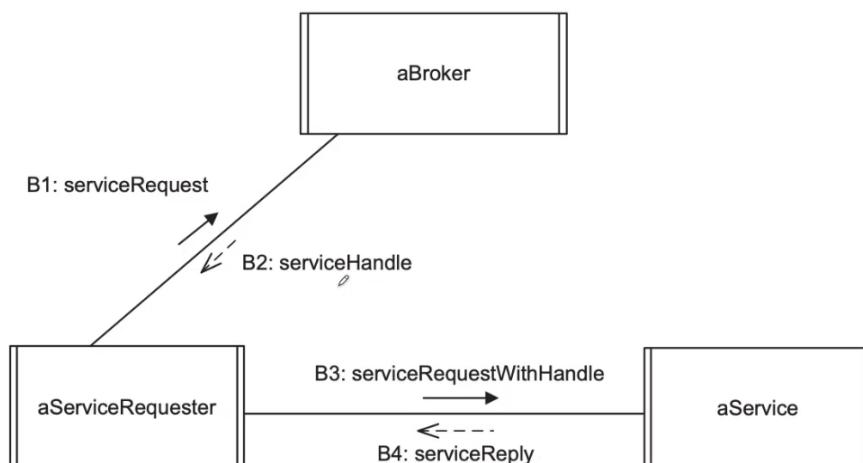
BROKER FORWARDING PATTERN (WHITE PAGES)

1. Un client invia un messaggio che identifica il servizio - ad esempio per prelevare contanti da una banca
2. Il broker riceve la richiesta del cliente, determina l'ubicazione del servizio (l'ID del nodo in cui risiede il servizio) e inoltra il messaggio al servizio all'ubicazione specificata
3. Il messaggio arriva al servizio e il servizio e la richiesta del servizio è invocata
4. Il broker riceve la risposta del servizio e lo inoltra di nuovo al client



BROKER HANDLE PATTERN (WHITE PAGES)

Il modello Broker Handle mantiene il vantaggio di trasparenza della posizione(location transparency) con l'aggiunta del vantaggio di riduzione del traffico di messaggi. Questo perché invece di inoltrare ogni messaggio client al servizio, il broker restituisce un handle di servizio al client, che viene poi utilizzato per la comunicazione tra cliente e servizio (che possono quindi comunicare tra loro senza più il tramite del broker). Questo modello è particolarmente utile quando il client e il servizio hanno probabilmente un dialogo e devono scambiare diversi messaggi tra di loro.

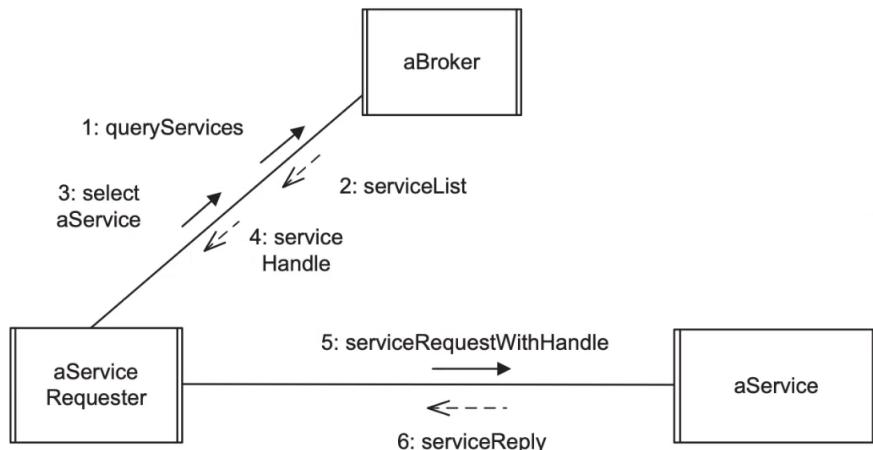


SERVICE DISCOVERY PATTERN (YELLOW PAGES)

Nelle white pages brokering il cliente conosce il servizio richiesto, ma non la posizione. Un modello di intermediazione diverso sono le yellow pages brokering, analogo alle pagine gialle dell'elenco telefonico, il cliente conosce il tipo del servizio richiesto, ma non il servizio specifico.

Questo modello è noto anche come modello di individuazione dei servizi perché permette al cliente di scoprire nuovi servizi:

1. Il cliente invia una richiesta di query al broker, richiedendo tutti i servizi di un determinato tipo
2. Il broker risponde con un elenco di tutti i servizi che corrispondono alla richiesta del cliente
3. Il cliente, eventualmente dopo aver consultato l'utente, seleziona un servizio specifico
4. Il broker restituisce l'handle del servizio, che il client utilizza per comunicare direttamente con il servizio



TECHNOLOGY SUPPORT FOR SOA

Sebbene le SOA siano concettualmente platform-indipendenti, sono attualmente forniti con molto successo su piattaforme con tecnologia Web Services.

Un servizio Web è un servizio a cui si accede utilizzando protocolli standard basati su Internet e XML.

WEB SERVICES PROTOCOLS

I client e i servizi dell'applicazione devono avere un protocollo di comunicazione per la comunicazione tra componenti.

XML (Extensible Markup Language) è una tecnologia che consente l'interoperabilità di sistemi diversi attraverso lo scambio di dati e testo.

Il protocollo SOAP (Simple Object Access Protocol), che è un protocollo leggero sviluppato dal World Wide Web Consortium (W3C), si basa su XML e HTTP per consentire scambio di informazioni in un ambiente distribuito.

SOAP definisce un approccio unificato per l'invio di file con codifica XML e si compone di tre parti:

1. una busta che definisce un framework per descrivere ciò che si trova in un messaggio e come elaborarlo
2. un insieme di regole di codifica per esprimere le istanze di tipi di dati definiti, e
3. una convenzione per la rappresentazione delle chiamate e delle risposte a procedura remota

WEB SERVICES (SERVIZI WEB)

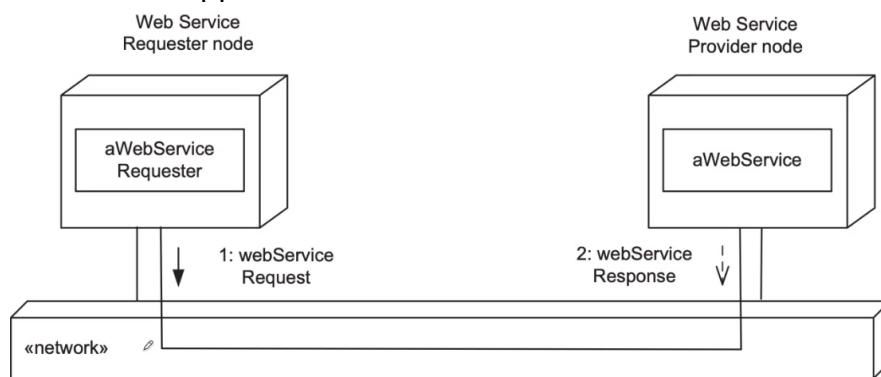
Le applicazioni forniscono servizi per i clienti.

Un esempio di servizi applicativi sono i Web services che utilizzano il World Wide Web per la comunicazione application-to-application.

Da un punto di vista software, i servizi Web sono le interfacce per programmi applicativi (API) che forniscono un mezzo standard di comunicazione tra diverse applicazioni software in tutto il mondo.

Dal punto di vista di un'applicazione aziendale, un Web service è una funzionalità aziendale fornita da una società sotto forma di un servizio esplicito su Internet per l'uso da parte di altre aziende o programmi.

Un servizio Web è fornito da un provider di servizi e può essere composto da altri servizi per formare nuovi servizi e applicazioni.



REGISTRATION SERVICES

È previsto un servizio di registrazione per i servizi al fine di mettere i servizi a disposizione dei clienti.

I servizi registrano i propri servizi con un servizio di registrazione, tramite un processo denominato pubblicazione o registrazione del servizio.

La maggior parte dei broker, come CORBA e i broker di servizi Web, forniscono un servizio di registrazione.

Per i servizi Web, viene fornito un service registry (registro dei servizi) per consentire ai servizi di essere pubblicati e localizzati tramite il World Wide Web.

I fornitori di servizi registrano i loro servizi insieme alle descrizioni associate in un service registry.

I clienti che cercano un servizio possono cercarlo sul service registry per trovare un servizio adatto. Il linguaggio WSDL (Web Services Description Language) è un linguaggio basato su XML utilizzato per descrivere cosa un servizio fa, dove risiede e come invocarlo.

BROKERING AND DISCOVERY SERVICES

In un ambiente distribuito, un broker di oggetti è un intermediario nelle interazioni tra i clienti e i servizi.

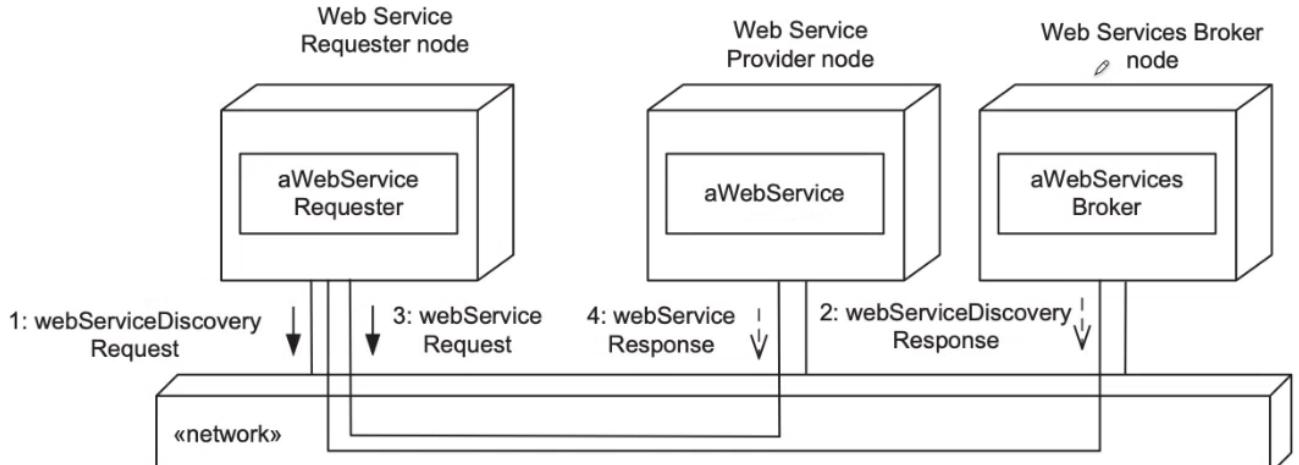
Un esempio di tecnologia di intermediazione è un Web services broker.

Le informazioni su un servizio Web possono essere definite dal framework Universal Description, Discovery, and Integration (UDDI) per l'integrazione di servizi Web.

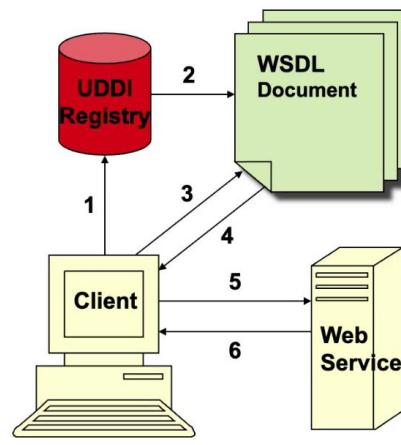
Una specifica UDDI è costituita da diversi documenti e uno schema XML che definisce un Protocollo basato su SOAP per la registrazione e l'individuazione di servizi Web.

Un broker di servizi Web può utilizzare il framework UDDI per fornire ai clienti un meccanismo per trovare dinamicamente servizi sul Web.

Esempio di Web Service Broker:

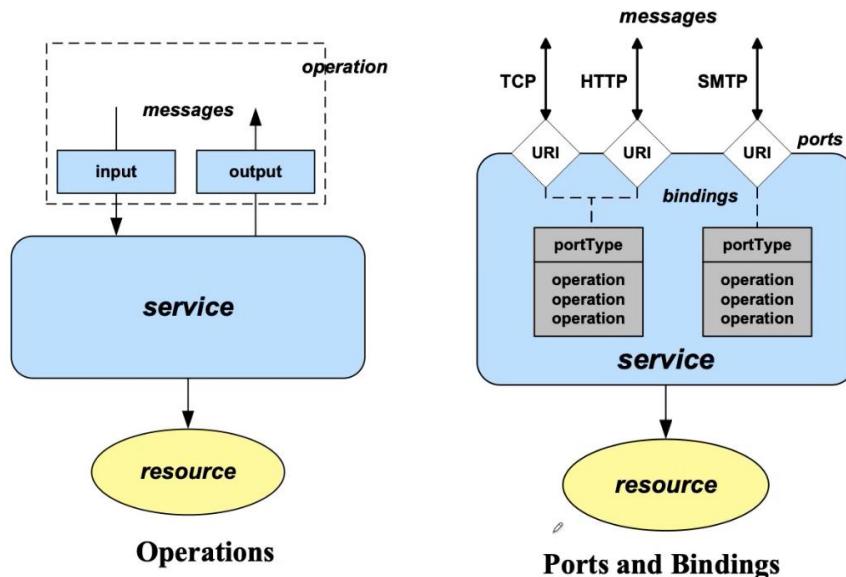


WEB SERVICE PROTOCOLS AND STANDARDS



1. Client queries UDDI registry to locate service
2. Registry refers client to WSDL document
3. Client accesses WSDL document
4. WSDL provides data to interact with web service
5. Client sends SOAP-message request
6. Web service returns SOAP-message response

WSLD



REST

REST è l'acronimo di Representational State Transfer, coniato per descrivere uno stile architettonico dei sistemi in rete.

RESTful API: un'API basata sulle risorse che utilizza il protocollo http

REST-BASED NETWORK CHARACTERISTICS

- Client-Server. Uno stile di interazione basato su pull
- Stateless: la comunicazione client-server è vincolata senza che il contesto client venga archiviato sul server
- Cache: i client e gli intermediari possono memorizzare nella cache le risposte
- Interfaccia uniforme: tutte le risorse sono accessibili con un'interfaccia generica (ad esempio, HTTP GET, POST, PUT, DELETE), semplificando e disaccoppiando così l'architettura
- Risorse denominate: il sistema è composto da risorse che sono denominate utilizzando un URL (o URI)
- Rappresentazioni di risorse interconnesse: le rappresentazioni delle risorse sono interconnesse utilizzando URL, consentendo in tal modo a un client di passare da uno stato all'altro.

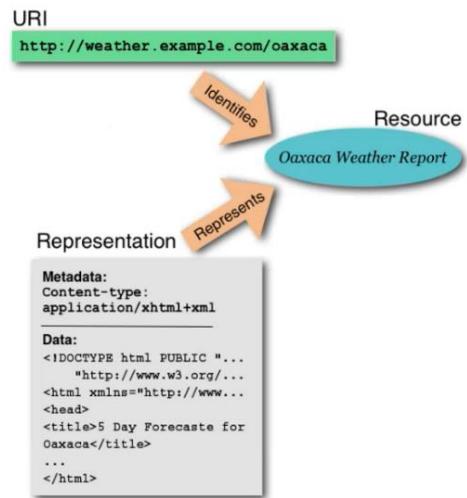
RESOURCES

Risorse:

- ogni entità distinguibile è una risorsa.
- una risorsa può essere un Web site, una pagina HTML, un documento XML, un servizio Web, un dispositivo fisico, ecc...

Gli URL identificano le risorse:

- Le risorse sono univoche e identificate da un URL (Assioma 0 di Tim Berners-Lee Web Design)



RESTful API

L'API RESTful utilizza i verbi HTTP disponibili per

eseguire operazioni CRUD in base al

"contesto":

- Collezione: un insieme di elementi (ad es. .. : /users)
- Elemento: un elemento specifico in una raccolta (ad es. .. : /users/{id})

VERB	Collection	Item
POST	Create a new item.	Not used
GET	Get list of elements.	Get the selected item.
PUT	Not used	Update the selected item.
DELETE	Not used	Delete the selected item.

CONVENTIONAL vs REST-BASED DESIGN

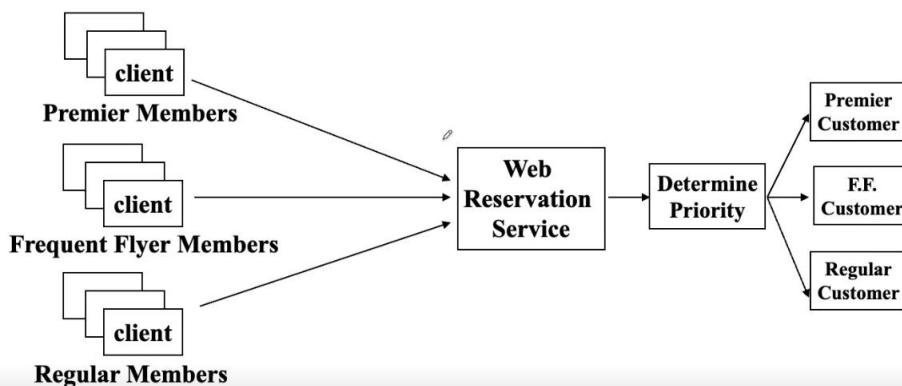
Scenario di esempio:

- Una compagnia aerea vuole fornire un servizio di prenotazione via Web per far effettuare prenotazioni di voli ai clienti tramite il web.
- La compagnia aerea vuole assicurarsi che i suoi membri premier ottengano servizio immediato, i suoi membri frequent flyer ottengono servizio accelerato, tutti gli altri ottengono un servizio regolare.

Due approcci principali alla progettazione e all'implementazione del servizio di prenotazione web sono:

- 1) Approccio a URL singolo: basato sul service design web convenzionale
- 2) Approccio Multiple URLs: sfrutta il design basato su REST

→ Nell'approccio single URL (approccio 1) il servizio Web è responsabile dell'esaminazione delle richieste in arrivo dei clienti per determinarne la loro priorità e li elaborano di conseguenza.

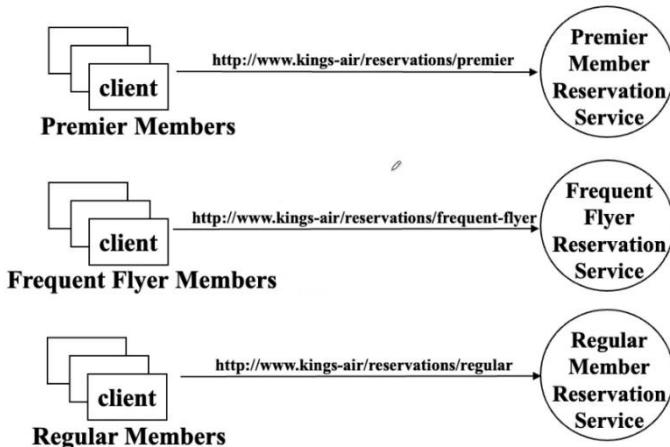


Gli svantaggi dell'approccio single URL sono i seguenti:

- i clienti devono apprendere la regola per l'espressione delle priorità e l'applicazione del servizio Web deve essere scritto per capire la regola.
- basato sull'errata presupposizione che un URL sia "costoso" e che il loro utilizzo debba essere razionato.
- il servizio Web è un punto centrale di errore e un collo di bottiglia (Il bilanciamento del carico è una sfida)
- viola l'Assioma 0 del Web Design di Tim Berners-Lee

→ Nell'approccio multiple URLs (approccio 2)

Un URL per i membri premier, un URL diverso per i frequent flyer e un altro ancora per i clienti abituali.



Vantaggi dell'approccio multiple URLs sono:

- È facile capire cosa fa ogni servizio semplicemente esaminando l'URL
- Non c'è bisogno di introdurre regole (le priorità vengono elevate al livello di un URL. "Quello che vedi è quello che ottieni")
- È facile implementare la priorità alta (è sufficiente assegnare una macchina veloce all'URL del membro premier)
- Non c'è alcun collo di bottiglia e nessun punto centrale di guasto
- coerente con l'assioma 0

LEZ27 – 18/03/2024

SOFTWARE ARCHITECTURAL TRANSACTION PATTERNS (MODELLI DI TRANSAZIONE DELL'ARCHITETTURA SOFTWARE)

Un servizio spesso incapsula dati o fornisce l'accesso ai dati che devono essere letti o aggiornati dai clienti.

Molti servizi devono fornire operazioni di aggiornamento.

DEF: Una transazione è una richiesta da parte di un client ad un servizio che consiste in due o più operazioni che svolgono un'unica funzione logica e che deve essere compilato nella sua interezza o non deve essere completato affatto.

PROPRIETA' DELLE TRANSAZIONI

Le transazioni hanno le seguenti proprietà, a volte indicate come proprietà ACID:

- Atomicità (A). Un'operazione è un'unità indivisibile di lavoro. È interamente completato (committed) o Interrotto (rolled back)
- Consistenza (C). Dopo l'esecuzione della transazione, il sistema deve essere in uno stato consistente
- Isolamento (I). Il comportamento di una transazione non deve essere influenzato da altre operazioni
- Durabilità (D). Le modifiche sono permanenti dopo che la transazione viene completata. Questi cambiamenti devono sopravvivere ai guasti del sistema. Questa operazione viene anche definita persistenza.

Esempio: transazione bancaria

TWO-PHASE COMMIT PROTOCOL

Protocollo di commit a due fasi.

Il modello Two-Phase Commit Protocol risolve il problema della gestione delle transazioni atomiche in sistemi distribuiti sincronizzando gli aggiornamenti sui diversi nodi.

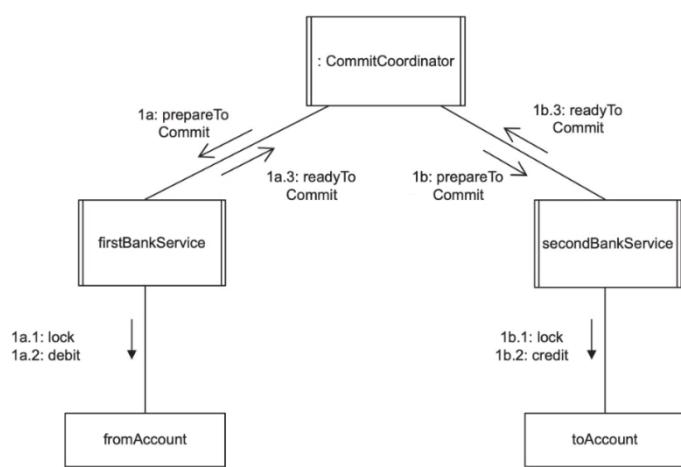
Il coordinamento dell'operazione è assicurato dal CommitCoordinator.

Per ogni nodo è disponibile un servizio per i partecipanti.

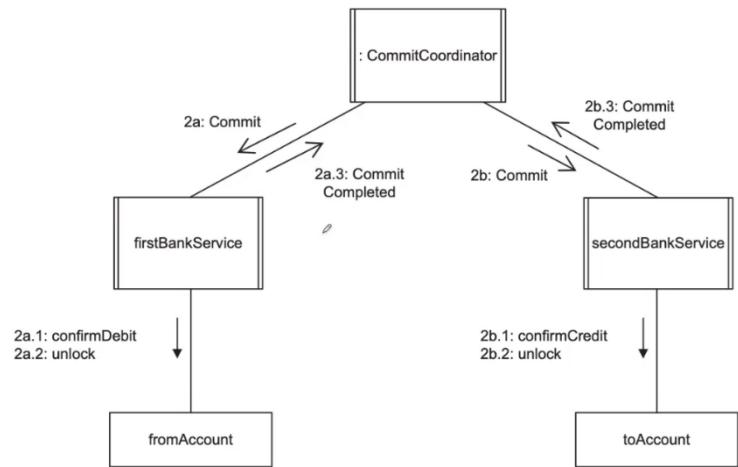
Ci sono due partecipanti all'operazione di bonifico bancario:

- firstBankService, che mantiene il conto da cui il denaro viene trasferito (from)
- secondBankService, che mantiene il conto su cui viene depositato il denaro in fase di trasferimento (to)

Fase1 protocollo(communication diagram):



Fase2 protocollo:



COMPOUND TRANSACTION PATTERN

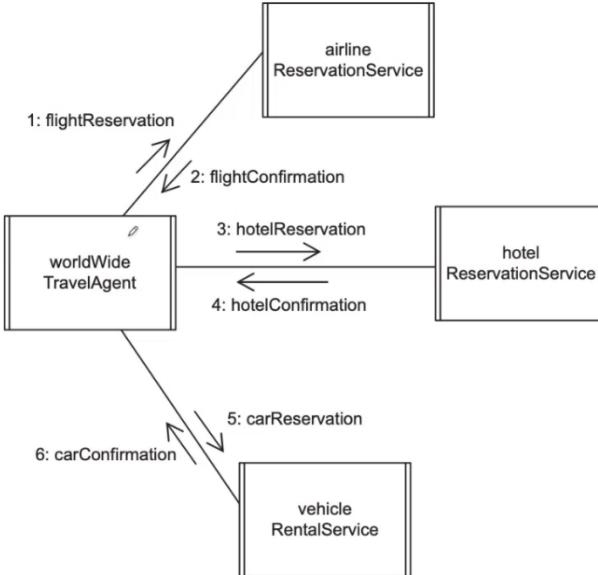
Modello di transazione composta.

La precedente operazione di bonifico bancario è un esempio di transazione flat, che ha una caratteristica "tutto o niente".

Una transazione composta, al contrario, potrebbe richiedere solo un Rollback parziale. Il modello di transazione composta può essere utilizzato quando il requisito di transazione del cliente può essere suddiviso in transazioni atomiche flat più piccole, in cui ogni transazione può essere eseguita separatamente e ripristinata separatamente.

Ad esempio, se un'agenzia di viaggi fa una prenotazione d'aereo, seguita da una prenotazione alberghiera e da una prenotazione di auto, è più flessibile trattare questa prenotazione come costituita da tre operazioni flat. Trattare un'operazione composta consente di modificare o cancellare una parte della prenotazione senza che le altre subiscano effetti.

Esempio:



LONG-LIVING TRANSACTION PATTERN

Modello di transazione di lunga durata.

Un'operazione di lunga durata è un'operazione che ha un essere umano nel ciclo e questo potrebbe richiedere molto tempo e possibilmente tempo indefinito per l'esecuzione, perché il comportamento umano individuale è imprevedibile.

Il modello di transazione di lunga durata divide una transazione di lunga durata in due o più transazioni separate (di solito due) in modo che il processo decisionale umano avvenga tra le coppie successive (come la prima e la seconda) di transazioni.

Esempio di Long-Living Transaction:

Prendi in considerazione una prenotazione aerea con un coinvolgimento umano nell'operazione (l'acquirente umano).

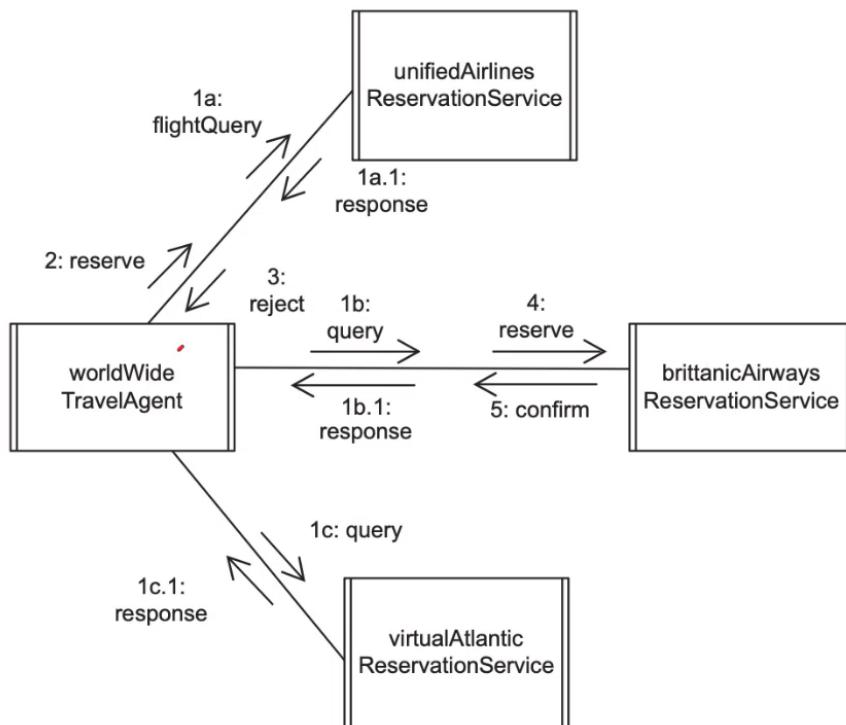
Innanzitutto, una query transaction visualizza i posti ancora disponibili.

La query transaction è seguita da una reserve transaction.

Con questo approccio, è necessario ricontrolare la disponibilità del posto prima della prenotazione.

Una postazione disponibile al momento della query potrebbe non essere più disponibile al momento della prenotazione perché diversi agenti potrebbe interrogare lo stesso volo contemporaneamente.

Se è disponibile un solo posto, il primo agente otterrà il sedile ma non gli altri.



NEGOTIATION PATTERN

Modello di negoziazione.

In alcune SOA, il coordinamento tra i servizi comporta negoziazioni tra agenti software in modo che possano prendere decisioni in modo cooperativo.

Nel modello di negoziazione (noto anche come Multi-Agent Negotiation pattern), un agente client agisce per conto dell'utente e fa una proposta a un agente di servizio.

L'agente di servizio tenta di soddisfare la proposta del cliente, che potrebbe comportare la comunicazione con altri servizi.

Dopo aver determinato le opzioni disponibili, l'agente del servizio offre all'agente client una o più opzioni che si avvicinano di più alla proposta originale dell'agente client.

L'agente client può quindi richiedere una delle opzioni, proporre ulteriori opzioni, oppure rifiutare l'offerta. Se l'agente di servizio è in grado di soddisfare la richiesta dell'agente client, accetta la richiesta; in caso contrario, rifiuta la richiesta.

NEGOTIATION SERVICES (Servizi di negoziazione)

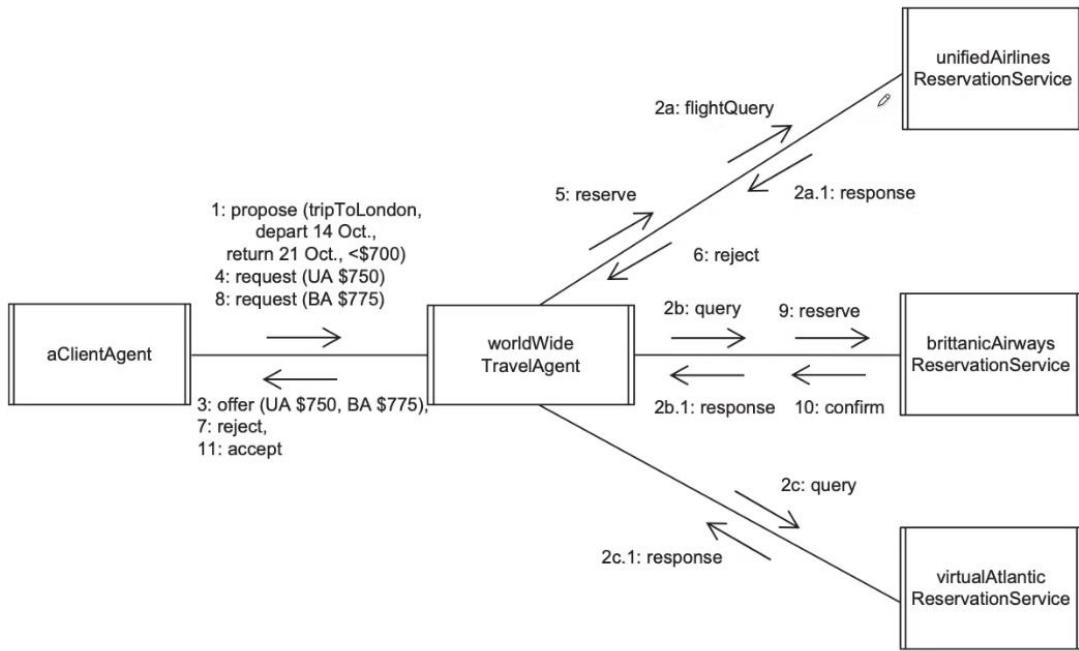
L'agente client, che agisce per conto del cliente, può eseguire una delle cose seguenti:

- Proporre un servizio. L'agente client propone un servizio all'agente di servizio. Questo servizio proposto è negoziabile, il che significa che l'agente client è disposto a prendere in considerazione le controfferte.
- Richiedere un servizio. L'agente client richiede un servizio all'agente del servizio. Questo servizio richiesto non è negoziabile, il che significa che l'agente client non è disposto a prendere in considerazione le controfferte.
- Rifiutare un'offerta di servizio. L'agente client rifiuta un'offerta fatta dall'agente di servizio

L'agente di servizio, che agisce per conto del servizio, può eseguire una delle seguenti operazioni:

- Offrire un servizio. In risposta a una proposta del cliente, un agente di servizio propone una controproposta
- Rifiutare una richiesta/proposta del cliente. L'agente di servizio rifiuta il servizio proposto o richiesto dall'agente client
- Accettare una richiesta/proposta del cliente. L'agente di servizio accetta il servizio proposto o richiesto dall'agente client

ESEMPIO NEGOTIATION PATTERN



SERVICE INTERFACE DESIGN IN SOA

Progettazione dell'interfaccia di servizio in SOA.

I nuovi servizi vengono inizialmente progettati utilizzando i criteri di strutturazione delle classi.

Durante la modellazione dinamica dell'interazione, viene determinata l'interazione tra gli oggetti client e gli oggetti servizio.

L'approccio adottato per la progettazione delle operazioni di servizio è simile a quello utilizzato nella progettazione dell'interfaccia di classe.

I messaggi che arrivano a un servizio costituiscono la base per la progettazione delle operazioni del servizio. I messaggi vengono analizzati per determinare il nome dell'operazione, nonché per determinare i parametri di input e output.

SERVICE COORDINATION IN SOA

Coordinamento dei servizi in SOA

Nelle applicazioni SOA che coinvolgono più servizi, in genere è richiesto il coordinamento di questi servizi.

Per garantire un accoppiamento libero tra i servizi, spesso è meglio separare i dettagli del coordinamento dalla funzionalità del singolo servizio.

Nella SOA sono previsti diversi tipi di coordinamento, tra cui l'orchestrazione(Orchestration) e la coreografia(Choreography)

ORCHESTRATION & CHOREOGRAPHY

L'orchestrazione è costituita da una logica di coordinamento del flusso di lavoro controllata centralmente per il coordinamento di più servizi partecipanti

- Ciò consente il riutilizzo di servizi esistenti incorporandoli in nuove applicazioni di servizio

La coreografia fornisce un coordinamento distribuito tra i servizi e può essere utilizzata quando è necessario il coordinamento tra diverse organizzazioni aziendali

- Pertanto, la coreografia può essere utilizzata per la collaborazione tra servizi di diversi fornitori di servizi forniti da diverse organizzazioni aziendali
- Mentre l'orchestrazione è controllata centralmente, la coreografia implica un controllo distribuito

COORDINATION (COORDINAMENTO)

Poiché i termini orchestrazione e coreografia sono spesso usati in modo intercambiabile, il termine più generale coordinamento (coordination) viene utilizzato per descrivere il controllo e la sequenza di servizi diversi in base alle esigenze di un'applicazione SOA, indipendentemente dal fatto che siano controllati centralmente o che coinvolgano il controllo distribuito.

I modelli di transazione possono essere utilizzati anche per il coordinamento dei servizi.

DETAILED OOD (OOD dettagliato)

L'OOD preliminare fornisce la piattaforma di esecuzione HW/SW a cui deve conformarsi la sottofase di progettazione dettagliata.

La parte principale della sottofase OOD dettagliata è dedicata a perfezionare ciò che è stato prodotto nella fase OOA.

L'obiettivo è quello di trasformare i modelli OOA, che sono stati definiti nel dominio del problema, in modelli definiti nel dominio della soluzione, che a loro volta verranno utilizzati nella fase di codifica.

La sottofase OOD dettagliata fornisce la progettazione dettagliata delle unità architettoniche identificate nella sottofase OOD preliminare, attraverso l'aggiunta di dettagli tecnici (o producendo modelli aggiuntivi a un livello di astrazione ridotto).

La sottofase OOD dettagliata definisce la collaborazione tra gli oggetti, che è alla base di ogni programma orientato agli oggetti.

Tale collaborazione si definisce attraverso la realizzazione di casi d'uso e operazioni.

Le collaborazioni sono per OOD ciò che i casi d'uso sono per OOA: se i casi d'uso guidano l'OOA, le collaborazioni guidano l'OOD.

REALIZATION OF USE CASES

Realizzazione di casi d'uso.

I casi d'uso introdotti al momento dell'OOA sono realizzati attraverso collaborazioni al momento dell'OOD.

Un singolo caso d'uso è tipicamente realizzato da un insieme di collaborazioni, a causa del diverso livello di astrazione.

Una collaborazione ha:

- una parte comportamentale
 - o rappresenta la dinamica che mostra come gli elementi statici collaborano
 - o definito mediante l'uso di diagrammi di comunicazione
- una parte strutturale
 - o rappresenta aspetti statici della collaborazione
 - o definito elaborando il diagramma di classe OOA con i dettagli implementativi, che portano a diagrammi di struttura compositi

COLLABORATION – CONTROL MANAGEMENT

Sistema di immatricolazione all'Università.

Esempio: aggiungere uno studente (Student) a un'offerta di corsi (CourseOffering)

Azioni da eseguire:

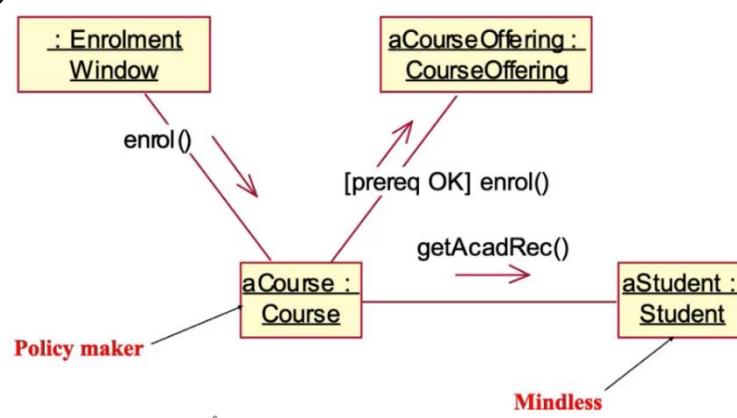
1. identificare gli insegnamenti propedeutici all'offerta formativa
2. verificare se lo studente soddisfa i prerequisiti

Si consideri che:

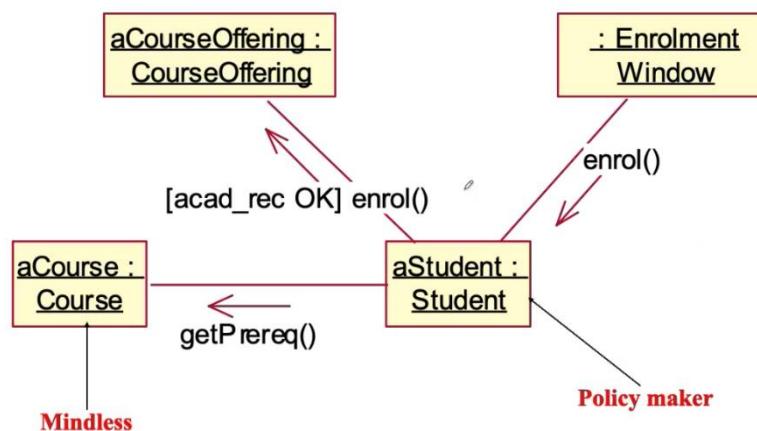
- il messaggio enrol () viene inviato dall'oggetto boundary :EnrolmentWindow
- sono coinvolte tre classi di entità: CourseOffering, Course e Student

Ci sono almeno quattro soluzioni possibili (con caratteristiche di accoppiamento di classe diverse), e sono le seguenti:

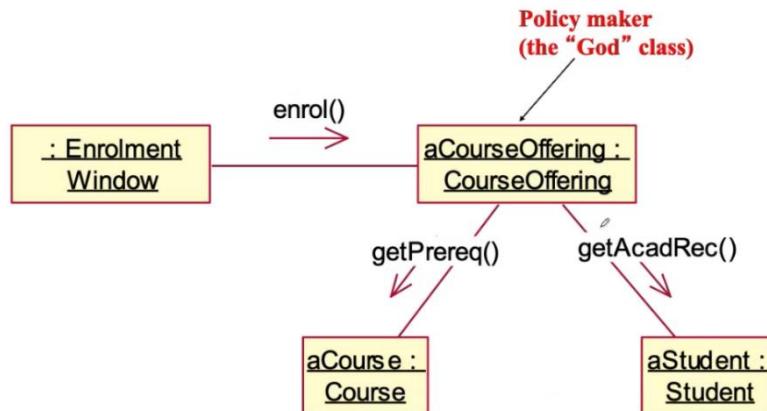
1)



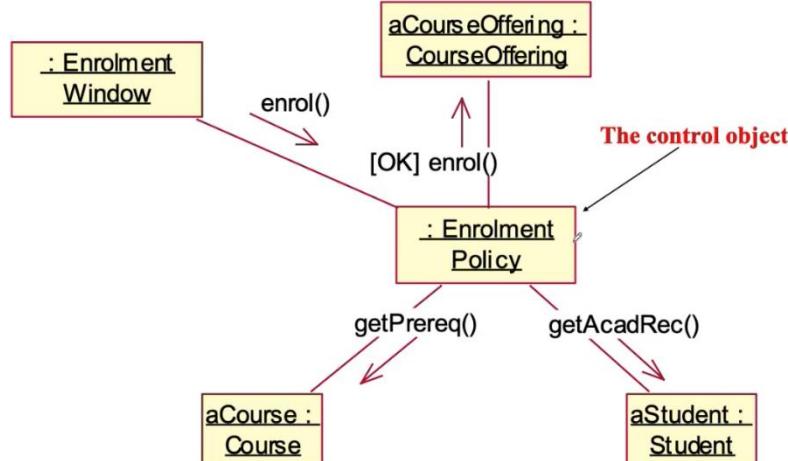
2)



3)



4)



Utilizzo Boundary-Control-Entity (BCE) Approach

COUPLING ISSUES (PROBLEMI DI ACCOPPIAMENTO)

L'approccio BCE specifica tre livelli di classi.

Gli oggetti comunicano all'interno di un livello e tra i livelli adiacenti.

Accoppiamento intrastrato (intra-layer coupling)

- Desiderabile
- Localizza la manutenzione e l'evoluzione del software in singoli strati

Accoppiamento interstrato (inter-layer coupling)

- Da ridurre al minimo
- Interfacce di comunicazione da definire con cura

La legge di Demeter può essere utilizzata per ridurre l'inter-layer class coupling.

LEZ28-25/03/2024

LEGGE DI DEMETER

Il destinatario di un messaggio (di una invocazione di metodo) può essere solo uno dei seguenti oggetti:

1. L'oggetto stesso del metodo (cioè this in Java)
2. Un oggetto che è un argomento nella firma del metodo
3. Un oggetto a cui fa riferimento l'attributo dell'oggetto (legge forte --> attributi ereditati)
4. non possono essere utilizzati per identificare l'oggetto di destinazione)
5. Un oggetto creato dal metodo
6. Un oggetto a cui fa riferimento una variabile globale

Questa legge è conosciuta anche come il "non parlare allo straniero", e ha lo scopo di evitare che il codice diventi illegibile.

UML STRUCTURED CLASS

Una classe strutturata contiene roles(ruoli) o parts(parti) che ne formano la struttura e ne realizzano il comportamento.

- Descrive la struttura di implementazione interna

Le parti stesse possono anche essere classi strutturate

- Consente la struttura gerarchica per consentire una chiara espressione di modelli multilivello

Un connettore viene utilizzato per rappresentare un'associazione in un contesto particolare

- Rappresenta i percorsi di comunicazione tra le parti

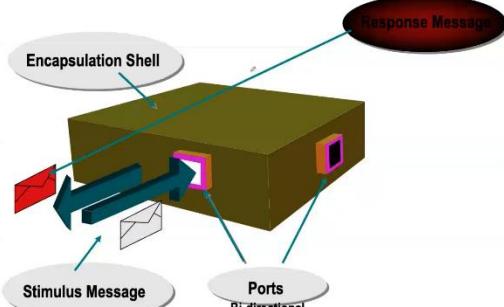
Differenza tra parti e ruoli: i ruoli seguono tipicamente una semantica per riferimento, mentre le parti seguono una semantica per valore. Quindi i ruoli rappresentano una sorta di attributi rappresentati per riferimento, mentre le parti sono contenute all'interno della classe strutturata.

STRUCTURED CLASS USAGE (utilizzo delle classi strutturate)

Le classi strutturate possono essere utilizzate come blocchi predefiniti primari di un'applicazione

- Forniscono una rappresentazione grafica degli elementi di design
- Si nascondono i dettagli dell'implementazione
- o Potente strumento di astrazione: lo stesso costrutto si applica a più livelli semantici
- o Comunicazione chiara e comprensione dell'architettura del sistema
- Incapsulamento rigoroso del comportamento
 - o Interazioni limitate alle comunicazioni basate su messaggi passate attraverso interfacce esterne (porte)

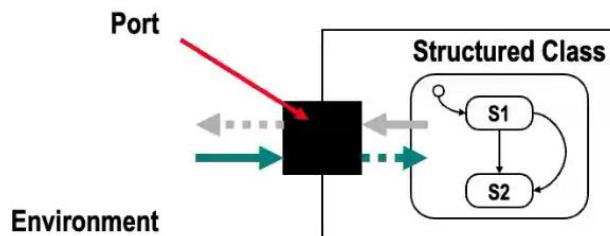
VISIONE CONCETTUALE DELLE STRUCTURED CLASS



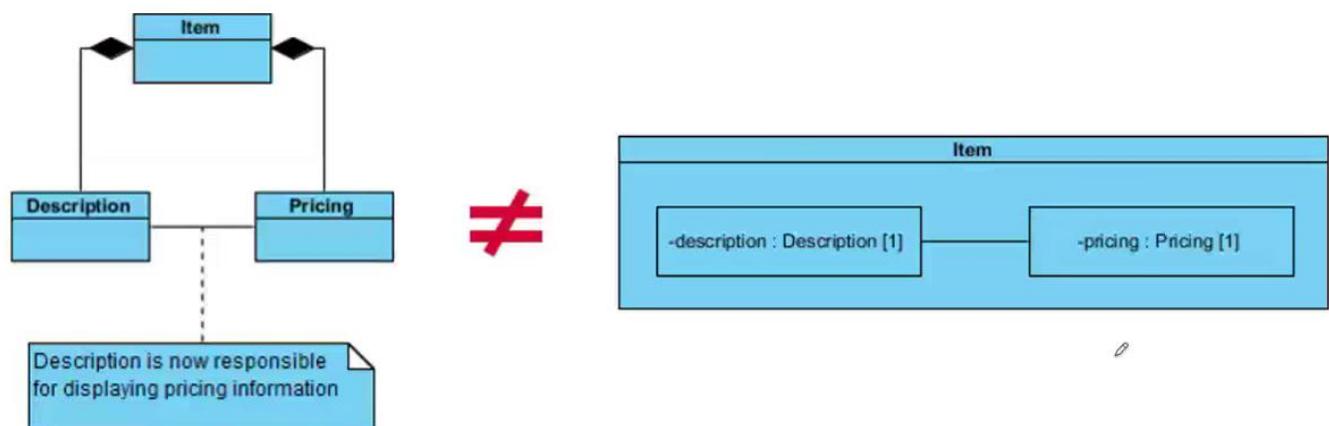
STRUCTURED CLASS: UNITA' AUTONOMA

L'incapsulamento rigoroso garantisce che l'implementazione sia indipendente dall'ambiente

- Le porte possono svolgere un ruolo di mediazione bidirezionale
 - o L'ambiente vede solo la porta della classe strutturata
 - o Il comportamento interno è costruito secondo la "specifica" fornita dalla sua interfaccia
- Le classi strutturate possono essere progettate in modo indipendente e testate



CLASS DIAGRAM vs COMPOSITE STRUCTURE DIAGRAM



La classe strutturata permette di avere informazione più precisa

PLATFORM CONFIGURATION

Una configurazione della piattaforma descrive la soluzione hardware/software che definisce il modo in cui le funzionalità del sistema possono essere distribuite tra i nodi fisici

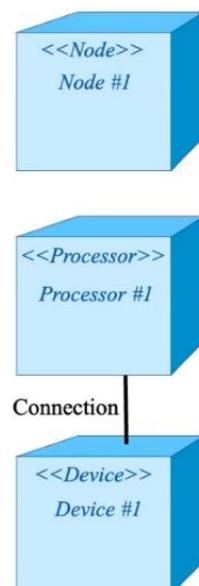
- Spiegare la relazione tra gli elementi del modello e la loro implementazione, nonché la loro distribuzione

Si ottiene tramite:

- Definizione della configurazione della piattaforma mediante l'utilizzo di un diagramma di distribuzione (deployment diagram)
- Assegnazione di elementi di sistema (artefatti) ai nodi dei diagrammi di distribuzione

DEPLOYMENT MODEL MODELING ELEMENTS

- Nodo, può rappresentare i seguenti tipi di elementi:
 - Risorsa computazionale fisica in fase di esecuzione
 - Nodo processore - Esegue il software di sistema
 - Nodo dispositivo
 - o Dispositivo di supporto
 - o In genere controllato da un processore
- Connessione
 - Meccanismo di comunicazione
 - Supporto fisico
 - Protocollo software

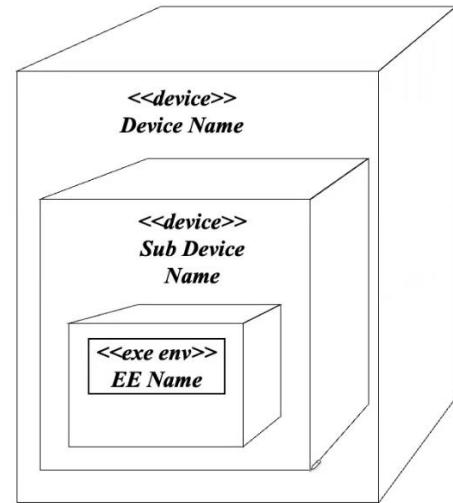


COS'E' UN NODO?

Rappresenta una risorsa di calcolo in fase di esecuzione e in genere dispone almeno di memoria e spesso capacità di elaborazione.

Tipi:

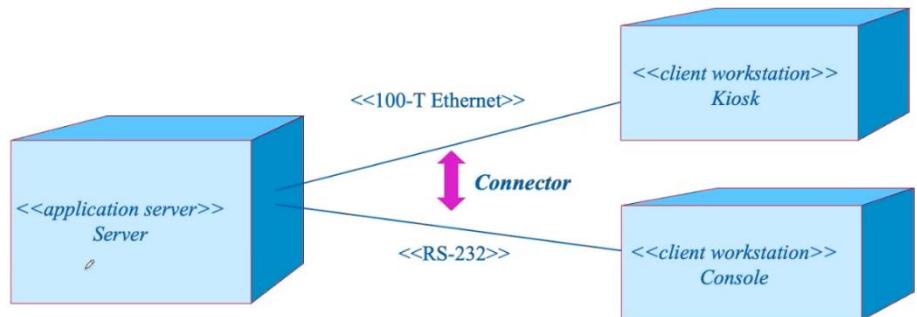
- Dispositivo - Risorsa computazionale fisica con capacità di elaborazione. I dispositivi possono essere nidificati
- Ambiente di esecuzione - Rappresenta particolari piattaforme di esecuzione



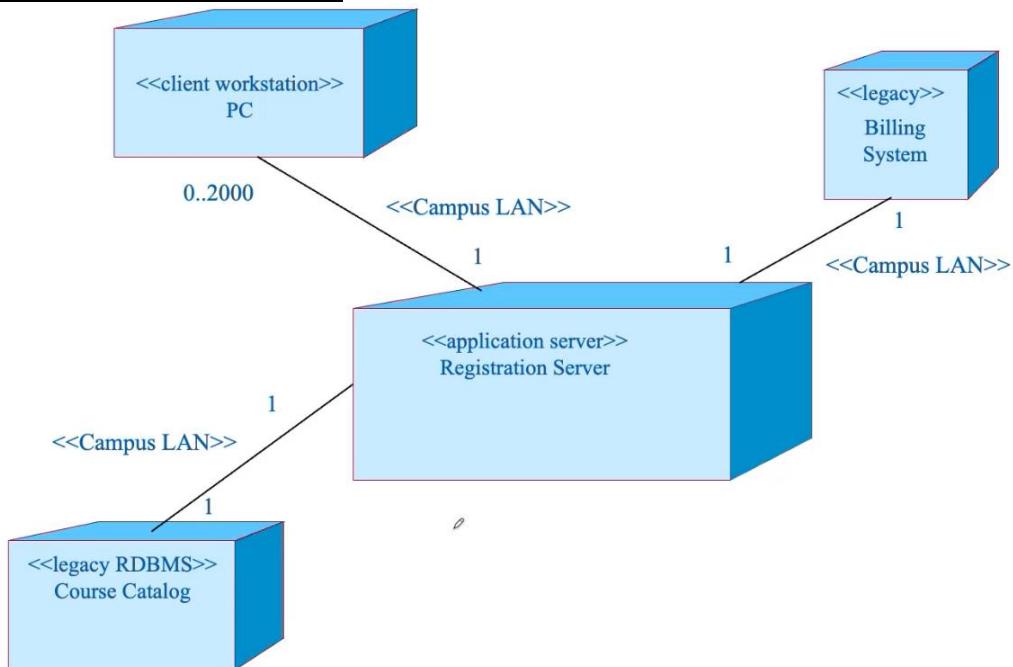
COS'E' UN CONNETTORE?

Un connettore rappresenta un meccanismo di comunicazione descritto da:

- Physical medium (supporto fisico)
- Software protocol (protocollo software)



ESEMPIO DEPLOYMENT DIAGRAM:



PROCESS-TO-NODE ALLOCATION CONSIDERATIONS

(considerazioni sull'allocazione da processo a nodo)

Modelli di distribuzione

Tempo di risposta e throughput del sistema

Riduzione al minimo del traffico tra reti

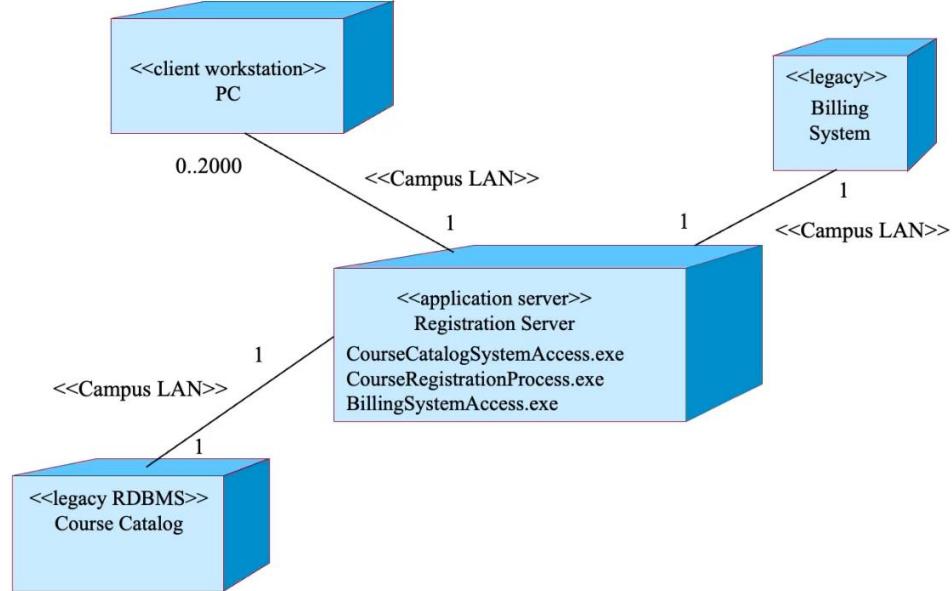
Capacità del nodo

Larghezza di banda media di comunicazione

Disponibilità di hardware e collegamenti di comunicazione

Requisiti di reinstradamento

ESEMPIO DEPLOYMENT DIAGRAM CON PROCESSI



COS'E' DEPLOYMENT

Deployment(distribuzione) è l'assegnazione, o mappatura, di artefatti software ai nodi fisici durante l'esecuzione.

- Gli artefatti sono le entità distribuite nei nodi fisici
 - o I processi vengono assegnati ai computer

Gli artefatti modellano le entità fisiche

- File, eseguibili, tabelle di database, pagine Web e così via.

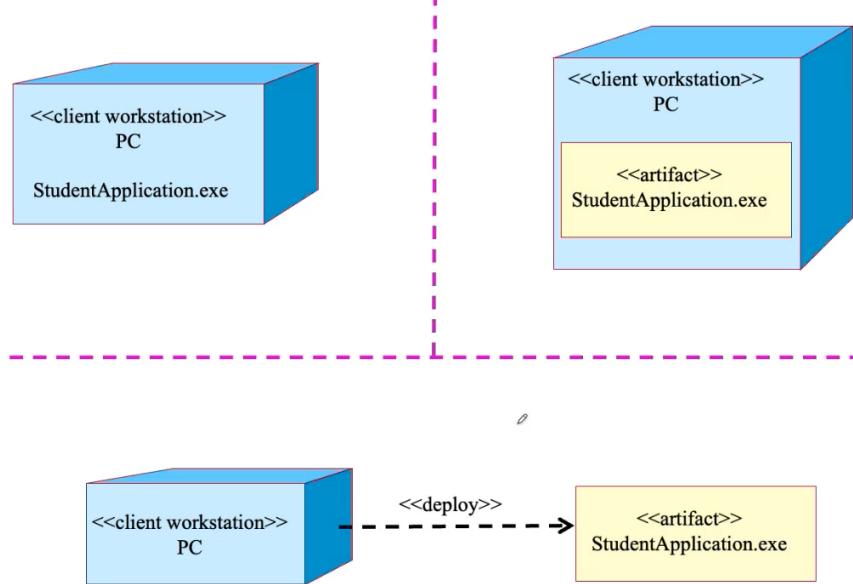
I nodi modellano le risorse computazionali

- Computer, unità di archiviazione

NB: il componente UML1 viene rimpiazzato in UML2 dal costrutto artifact

ESEMPIO DEPLOYING ARTIFACTS TO NODES

3 modi per rappresentare il deployment degli artefatti sui nodi

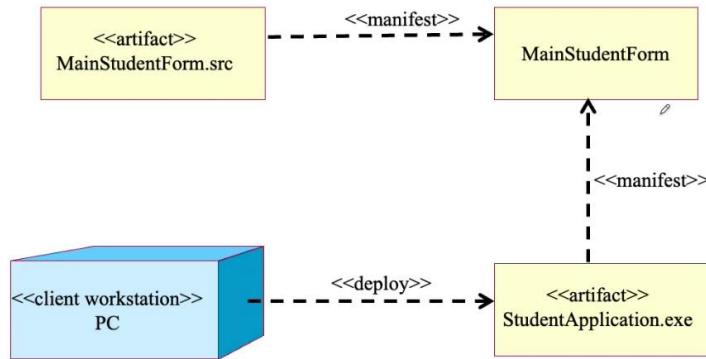


COS'E' MANIFESTATION?

Per manifestation(manifestazione) in UML2 si intende l'implementazione fisica di un elemento del modello come un artefatto. Quindi la manifestation è una relazione tra l'elemento un elemento di un modello e il relativo artefatto(ovvero l'entità fisica) che implementa quell'elemento.

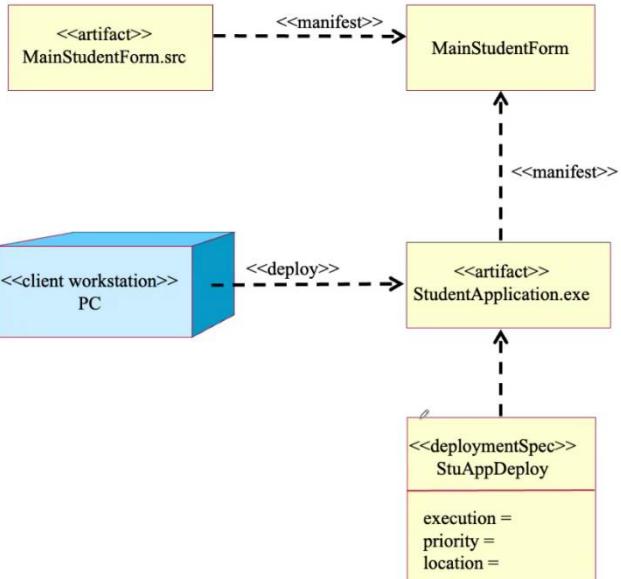
Gli elementi del modello vengono in genere implementati come set di artefatti. Esempi di elementi del modello sono i file di origine, i file eseguibili, il file di documentazione.

Esempio di manifestation:



COS'E' UNA DEPLOYMENT SPECIFICATION?

Una specifica di distribuzione è una specifica dettagliata dei parametri della distribuzione di un artefatto in un nodo. Essa può definire valori che parametrizzano l'esecuzione.



LEZ29 – 27/03/2024

DESIGNING SOA (caso di studio)

Sistema di shopping online basato sul Web.

Nel sistema di acquisto online basato sul Web, i clienti possono richiedere l'acquisto di uno o più articoli dal fornitore.

Il cliente fornisce dati personali, come l'indirizzo e i dati della carta di credito.

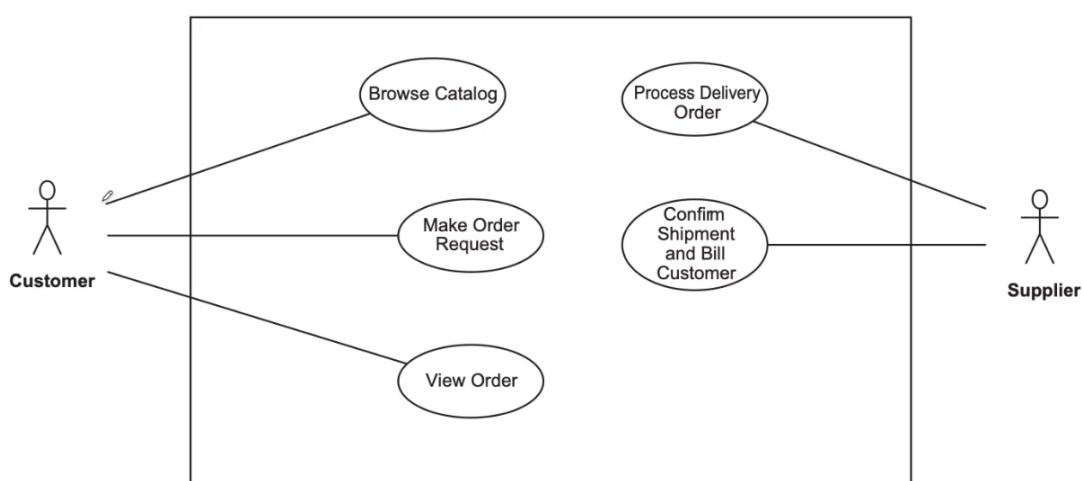
Queste informazioni vengono memorizzate in un account cliente. Se la carta di credito è valida, viene creato un ordine di consegna che viene inviato al fornitore.

Il fornitore controlla l'inventario disponibile, conferma l'ordine e inserisce una data di spedizione pianificata.

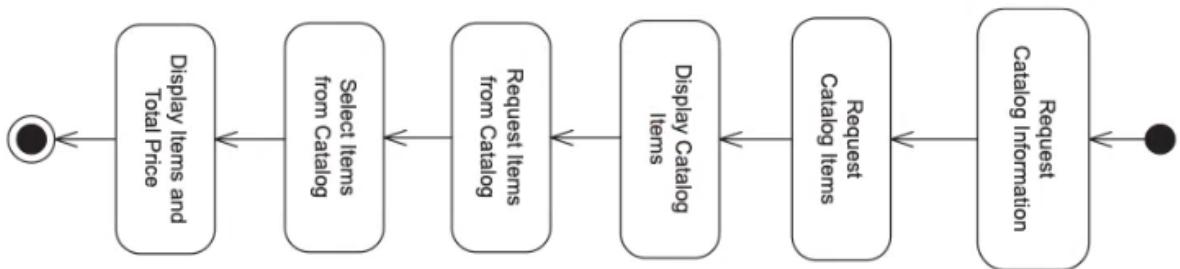
Quando l'ordine viene spedito, il cliente viene avvisato e viene addebitato sul conto della carta di credito del cliente.

USE CASE MODELING (MODELLAZIONE CASI D'USO)

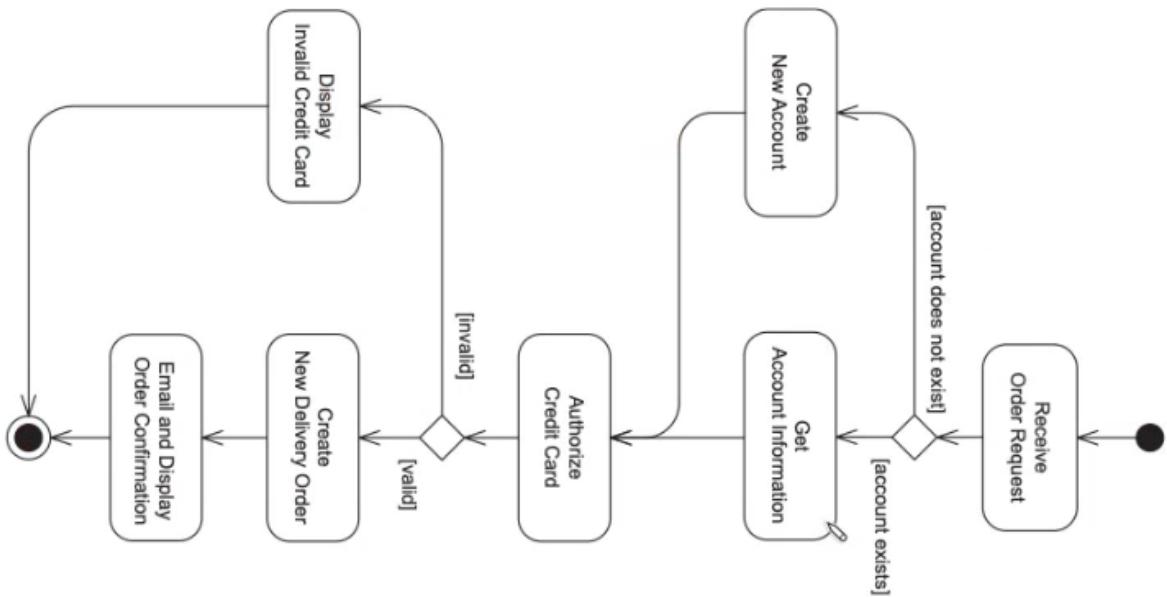
Use case
diagram:



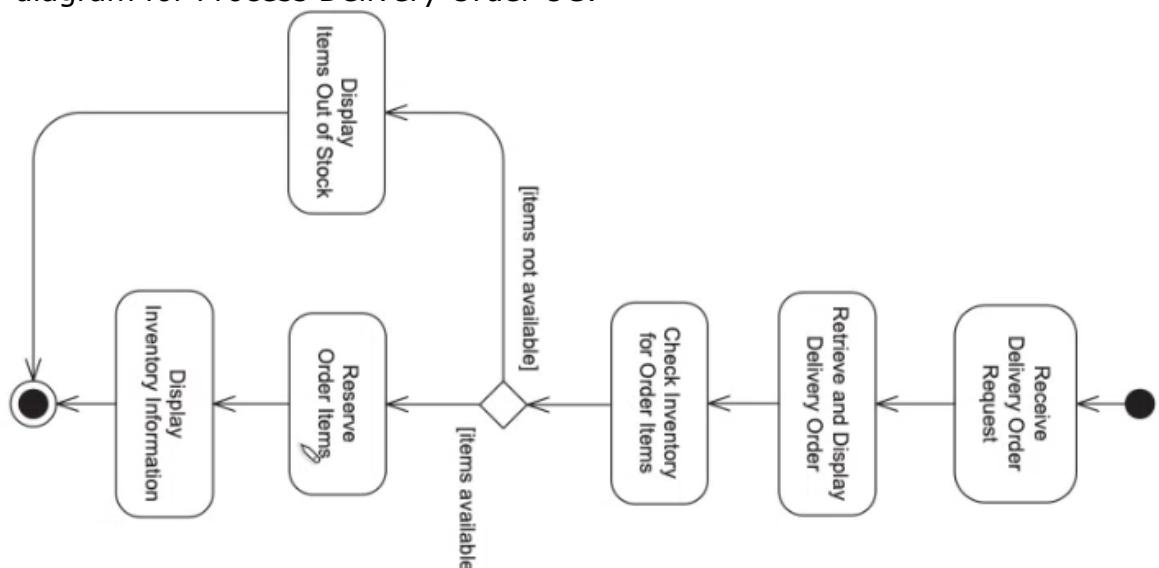
Activity diagram for Browse Catalog UC:



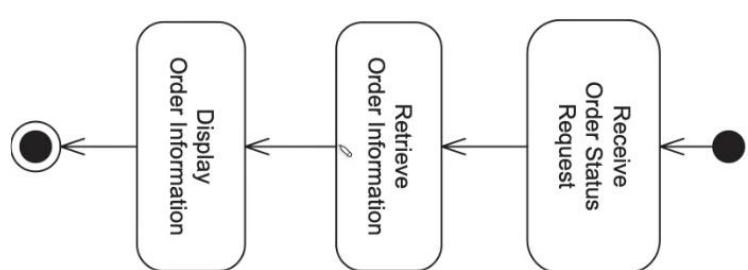
Activity diagram for Make Order Request UC:



Activity diagram for Process Delivery Order UC:



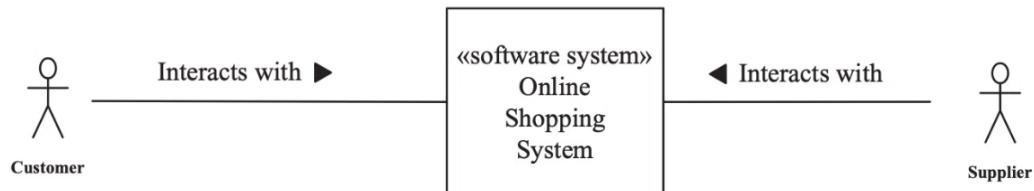
Activity diagram for View Order UC:



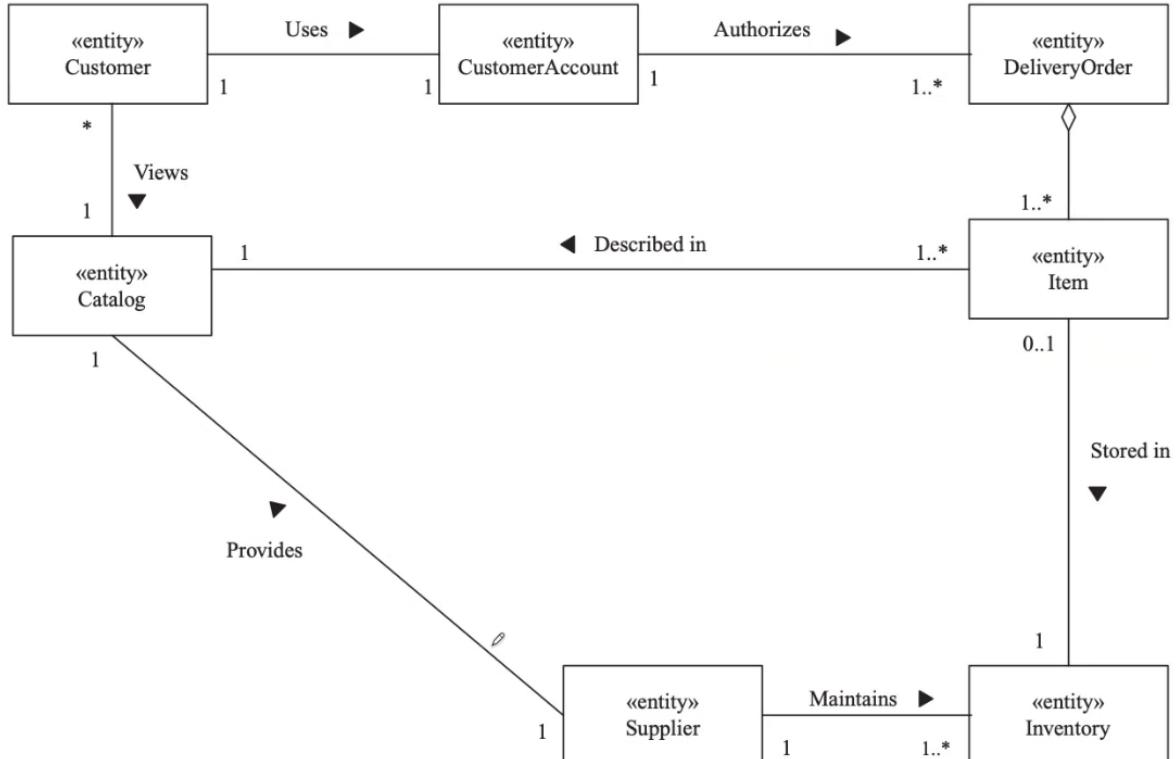
STATIC MODELING (MODELLAZIONE STATICA)

[Iniziamo a far uso di UML2]

Software System Context Class Diagram (diagramma delle classi di contesto):



Entity Class Diagram:



Entity Classes:

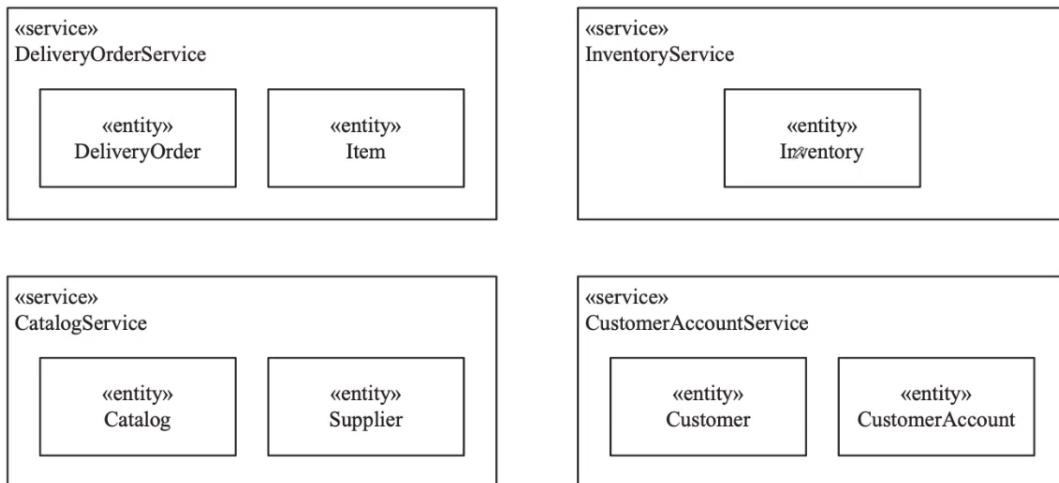
<table border="1"> <tr><th>«entity» DeliveryOrder</th></tr> <tr><td>orderId : Integer orderStatus : OrderstatusType accountId : Integer amountDue: Real authorizationId: Integer supplierId : Integer creationDate : Date plannedShipDate : Date actualShipDate : Date paymentDate: Date</td></tr> </table>	«entity» DeliveryOrder	orderId : Integer orderStatus : OrderstatusType accountId : Integer amountDue: Real authorizationId: Integer supplierId : Integer creationDate : Date plannedShipDate : Date actualShipDate : Date paymentDate: Date	<table border="1"> <tr><th>«entity» Customer</th></tr> <tr><td>customerId : Integer customerName : String address : String telephoneNumber : String faxNumber : String emailId : EmailType</td></tr> </table>	«entity» Customer	customerId : Integer customerName : String address : String telephoneNumber : String faxNumber : String emailId : EmailType	<table border="1"> <tr><th>«entity» Item</th></tr> <tr><td>itemId : Integer unitCost : Real quantity : Integer</td></tr> </table>	«entity» Item	itemId : Integer unitCost : Real quantity : Integer
«entity» DeliveryOrder								
orderId : Integer orderStatus : OrderstatusType accountId : Integer amountDue: Real authorizationId: Integer supplierId : Integer creationDate : Date plannedShipDate : Date actualShipDate : Date paymentDate: Date								
«entity» Customer								
customerId : Integer customerName : String address : String telephoneNumber : String faxNumber : String emailId : EmailType								
«entity» Item								
itemId : Integer unitCost : Real quantity : Integer								
<table border="1"> <tr><th>«entity» Inventory</th></tr> <tr><td>itemID : Integer itemDescription : String quantity : Integer price : Real reorderTime : Date</td></tr> </table>	«entity» Inventory	itemID : Integer itemDescription : String quantity : Integer price : Real reorderTime : Date	<table border="1"> <tr><th>«entity» Catalog</th></tr> <tr><td>itemId : Integer itemDescription : String unitCost : Real supplierId : Integer itemDetails : linkType</td></tr> </table>	«entity» Catalog	itemId : Integer itemDescription : String unitCost : Real supplierId : Integer itemDetails : linkType	<table border="1"> <tr><th>«entity» Supplier</th></tr> <tr><td>supplierId : Integer supplierName: String address : String telephoneNumber : String faxNumber : String email : EmailType</td></tr> </table>	«entity» Supplier	supplierId : Integer supplierName: String address : String telephoneNumber : String faxNumber : String email : EmailType
«entity» Inventory								
itemID : Integer itemDescription : String quantity : Integer price : Real reorderTime : Date								
«entity» Catalog								
itemId : Integer itemDescription : String unitCost : Real supplierId : Integer itemDetails : linkType								
«entity» Supplier								
supplierId : Integer supplierName: String address : String telephoneNumber : String faxNumber : String email : EmailType								
<table border="1"> <tr><th>«entity» CustomerAccount</th></tr> <tr><td>accountId : Integer cardId : String cardType : String expirationDate: Date</td></tr> </table>			«entity» CustomerAccount	accountId : Integer cardId : String cardType : String expirationDate: Date				
«entity» CustomerAccount								
accountId : Integer cardId : String cardType : String expirationDate: Date								

CLASS STRUCTURING (STRUTTURAZIONE DELLE CLASSI)

Le classi entity determinate nella sezione precedente sono integrate in un'architettura orientata ai servizi per mezzo di service classes.

Catalog Service, Customer Account Service, Delivery Order Service, e Inventory Service sono classi di servizio che forniscono l'accesso alle classi di entità.

Service and Entity Classes:

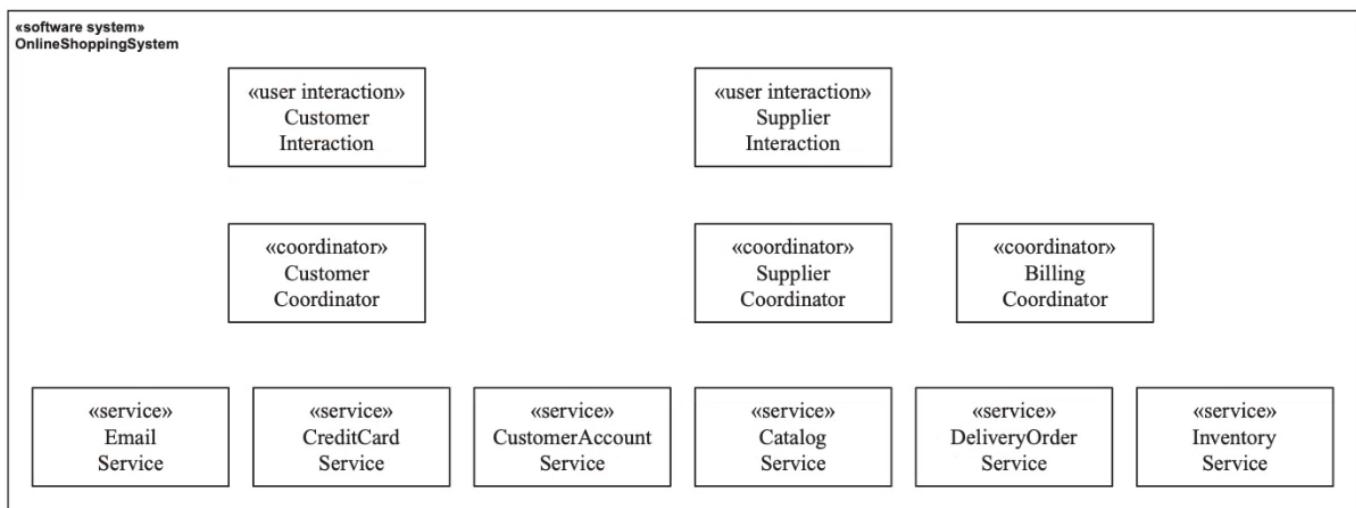


Altre classi:

C'è anche una classe di servizio, Credit Card Service, che si occupa dell'autorizzazione e dell'addebito della carta di credito. Un'altra classe di servizio è il servizio di posta elettronica, che consente al sistema di acquisti online di inviare messaggi di posta elettronica ai clienti.

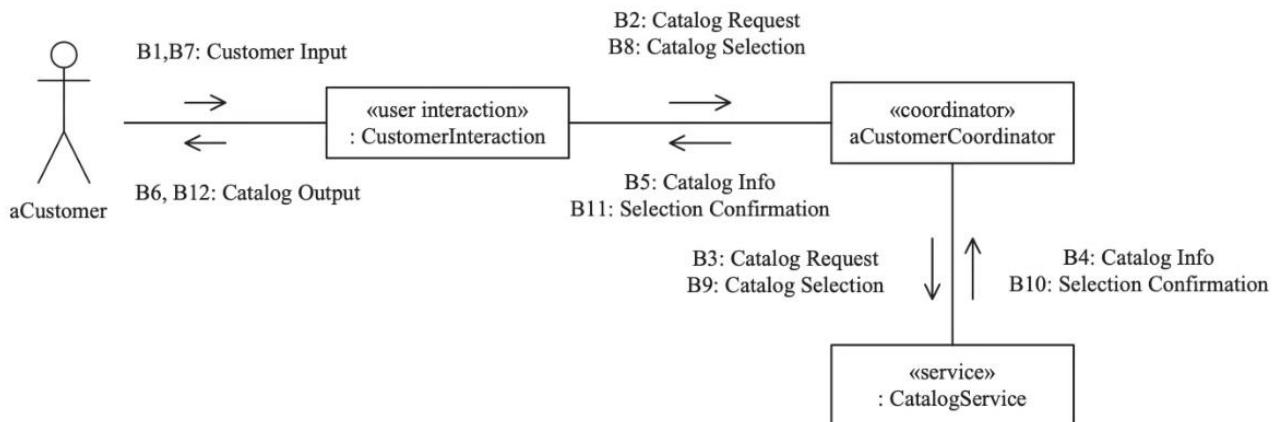
Le User interaction classes sono necessarie per interagire con gli utenti esterni, in particolare l'interazione con il cliente e l'interazione con il fornitore, che corrispondono agli attori nei casi d'uso.

Inoltre, per coordinare e sequenziare l'accesso di clienti e fornitori ai servizi di shopping online, sono previste due classi di coordinatori, Customer Coordinator e Supplier Coordinator. Un terzo coordinatore autonomo, il Billing Coordinator, è necessario per gestire la fatturazione ai clienti.

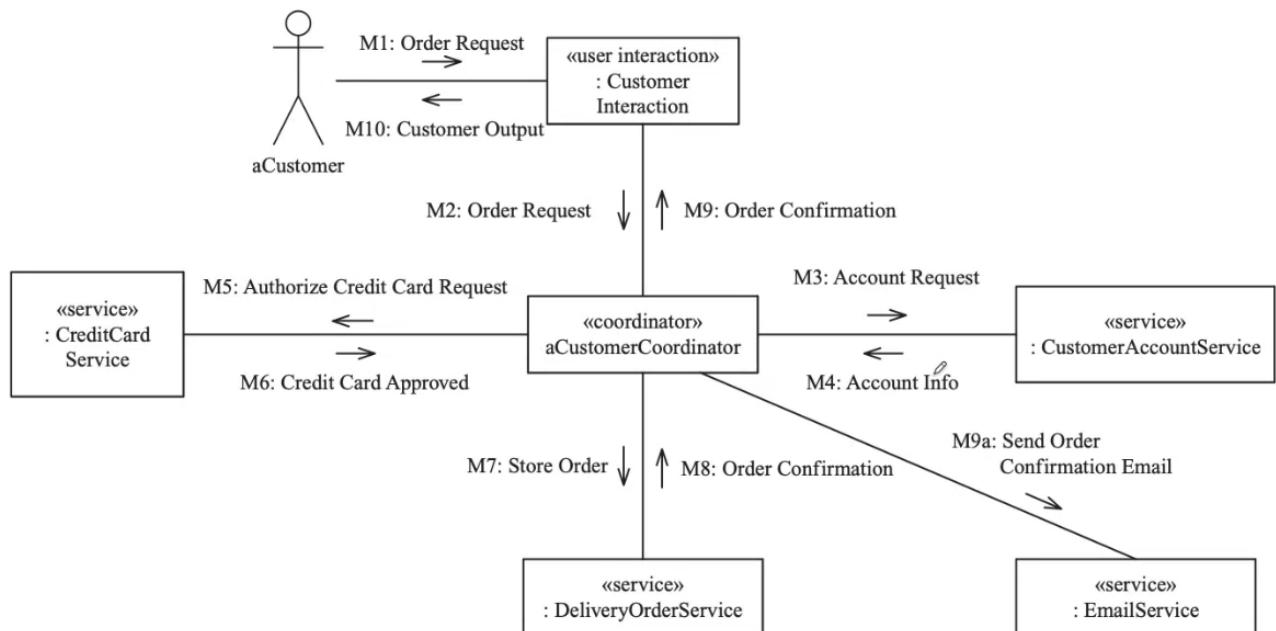


DYNAMIC MODELING (A differenza di quanto visto nella prima parte del corso ora utilizziamo in questa fase i communication diagram al posto dei sequence diagram)

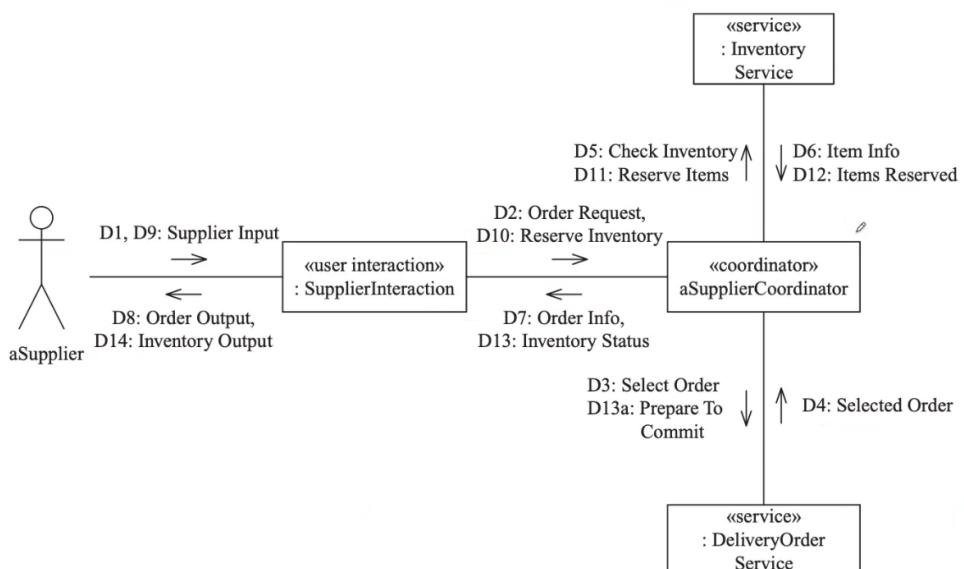
Communication diagram for the Browse Catalog UC:



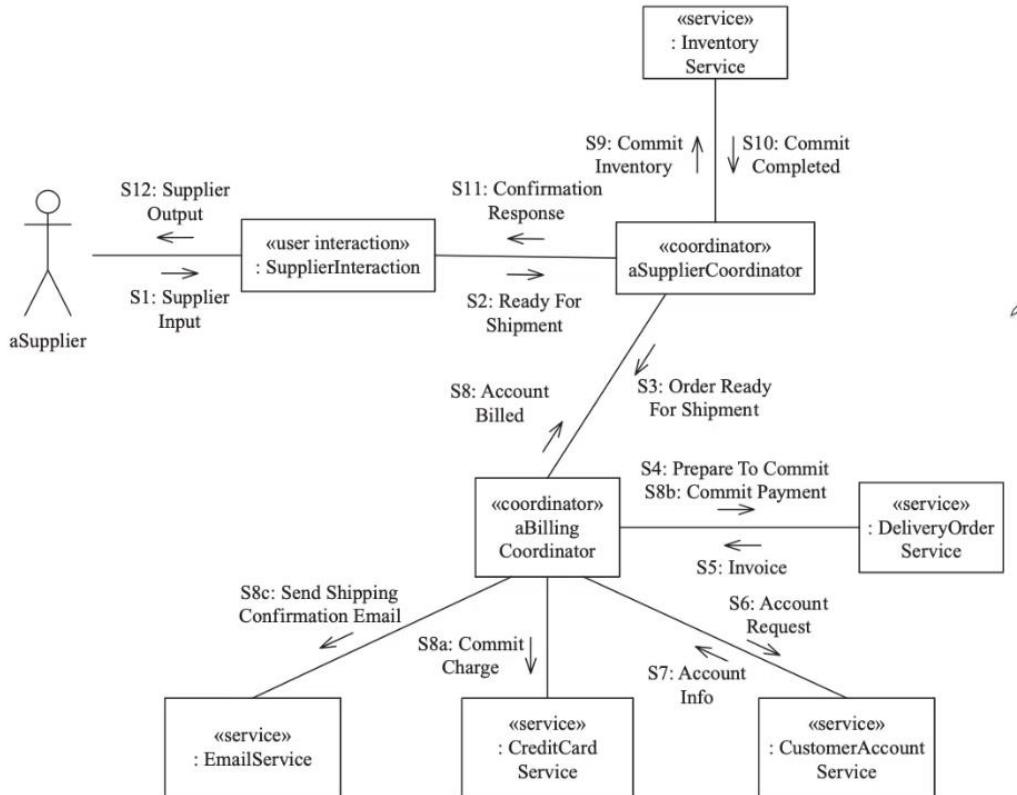
Communication diagram for the Make Order Request UC:



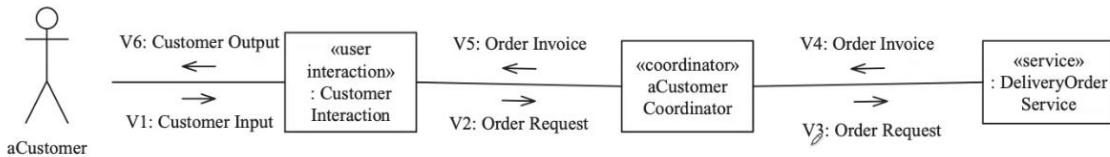
Communication diagram for the Process Delivery Order UC:



Communication diagram for the Confirm Shipment and Bill Customer UC:



Communication diagram for the View Order UC:



DESIGN MODELING (PARTE DI PROGETTAZIONE)

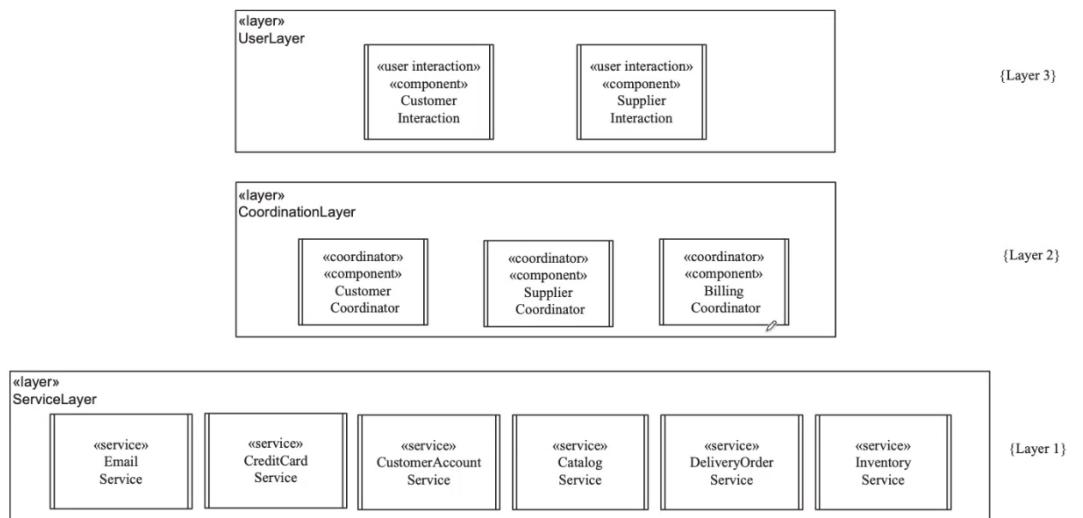
Il sistema di acquisto online è progettato come un'architettura a più livelli basata sul modello di architettura Layers of Abstraction.

L'architettura software è costituita da tre livelli: un livello di servizio (dati), un livello coordinatore e un livello di interazione con l'utente. Inoltre, poiché questo sistema deve essere altamente flessibile e distribuito, si decide di progettare un'architettura orientata ai servizi, in cui i componenti distribuiti possano scoprire i servizi e comunicare con essi.

Ogni componente è raffigurato con lo stereotipo del componente (di che tipo di componente si tratta, come specificato dai criteri di strutturazione del componente).

La progettazione delle interfacce dei componenti e dei servizi è determinata dall'analisi dei diagrammi di comunicazione per ciascun caso d'uso.

Architettura stratificata(BCE):



Modelli di comunicazione architetturale:

- Comunicazione sincrona dei messaggi con Reply
- Broker Handle
- Service discovery
- Comunicazione bidirezionale asincrona dei messaggi
- Two-Phase commit Commit in due fasi

Concurrent software design(progettazione di software simultanei):

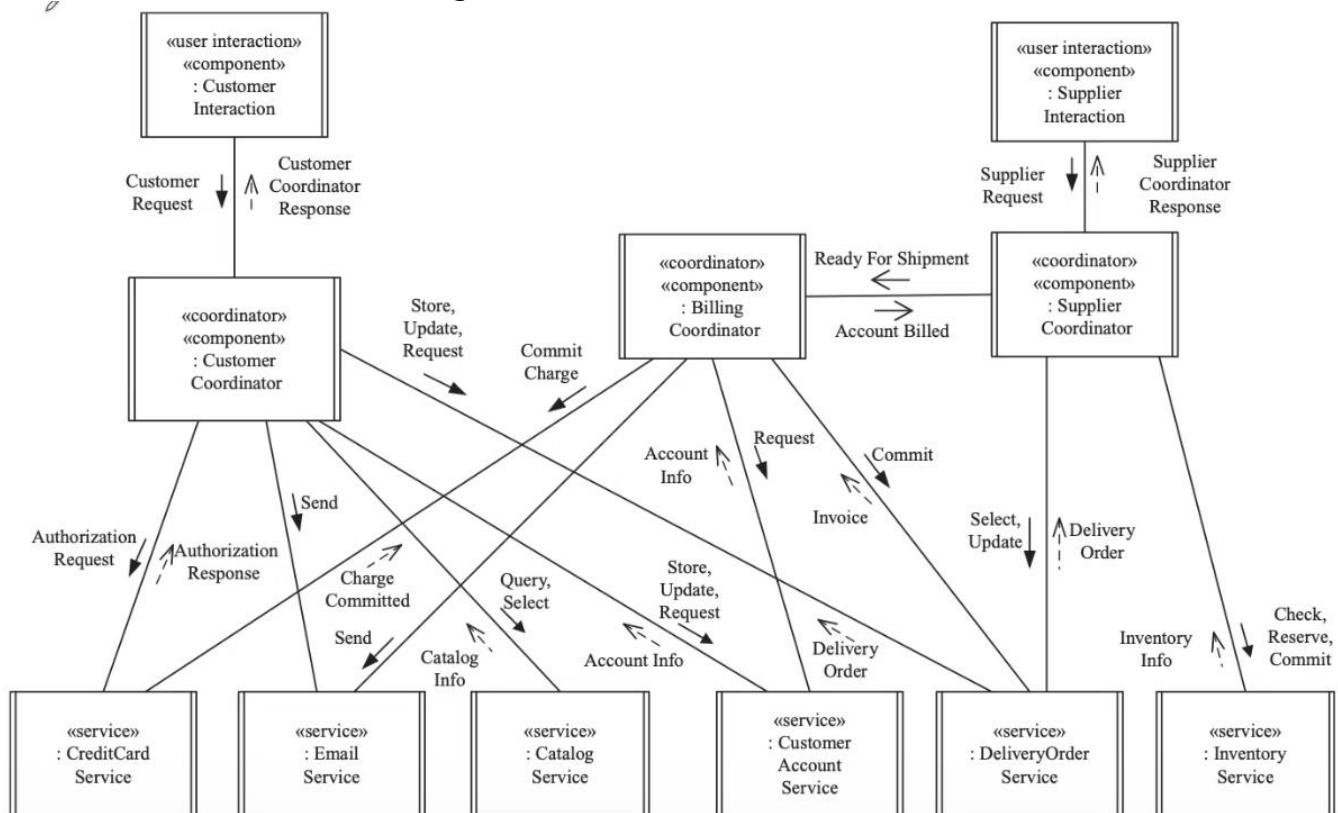
Per semplificare la progettazione, in questo caso di studio è stato ampiamente utilizzato il modello Synchronous Message Communication with Reply.

Questo approccio ha lo svantaggio di sospendere il client mentre attende una risposta dal servizio.

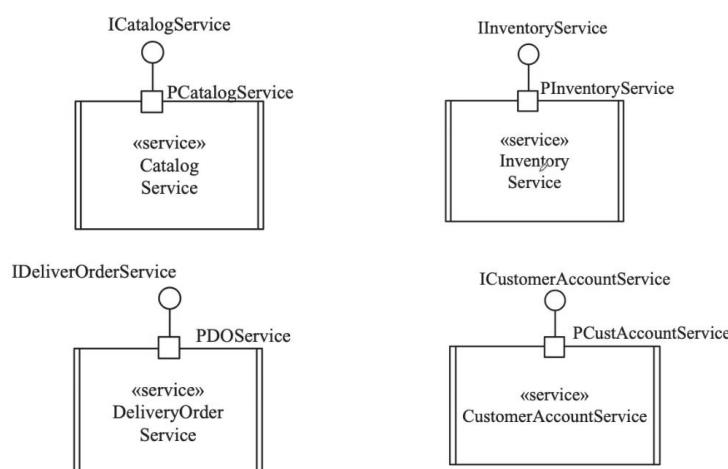
Una progettazione alternativa per evitare di sospendere il client consiste nell'utilizzare il modello Asynchronous Message Communication with Callback.

Il modello di comunicazione asincrona bidirezionale viene utilizzato per consentire al coordinatore del fornitore e al coordinatore della fatturazione di comunicare tra loro in entrambe le direzioni.

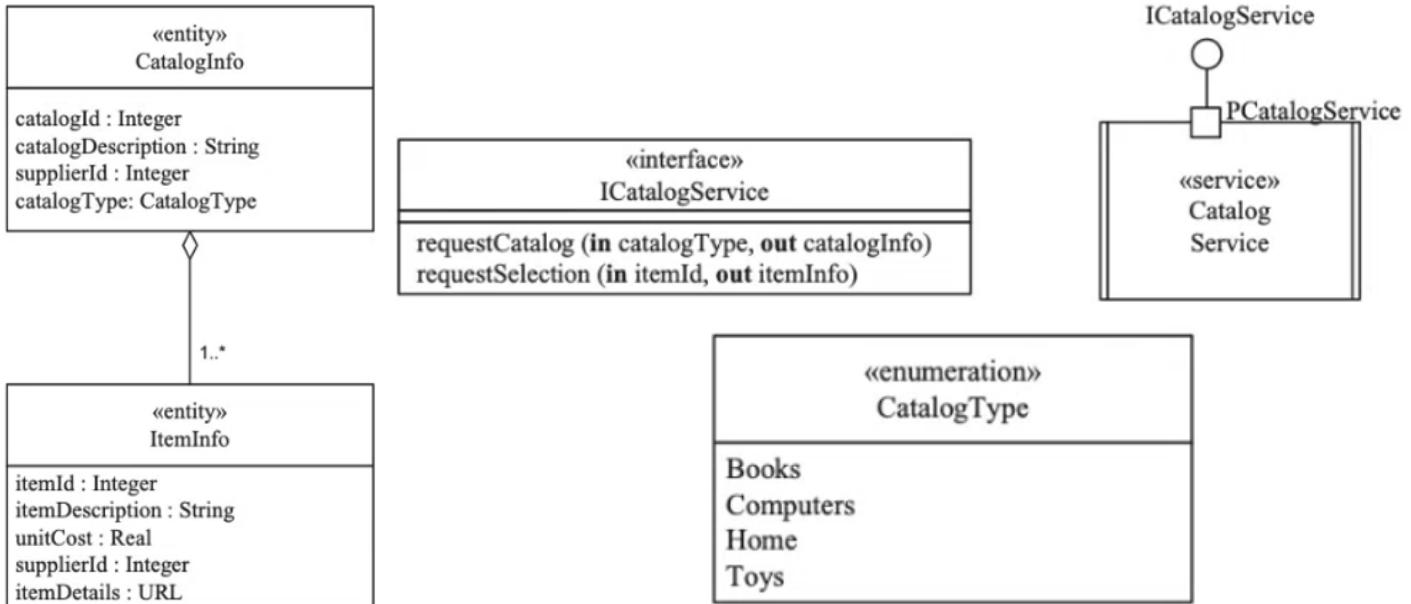
Concurrent Communication Diagram:



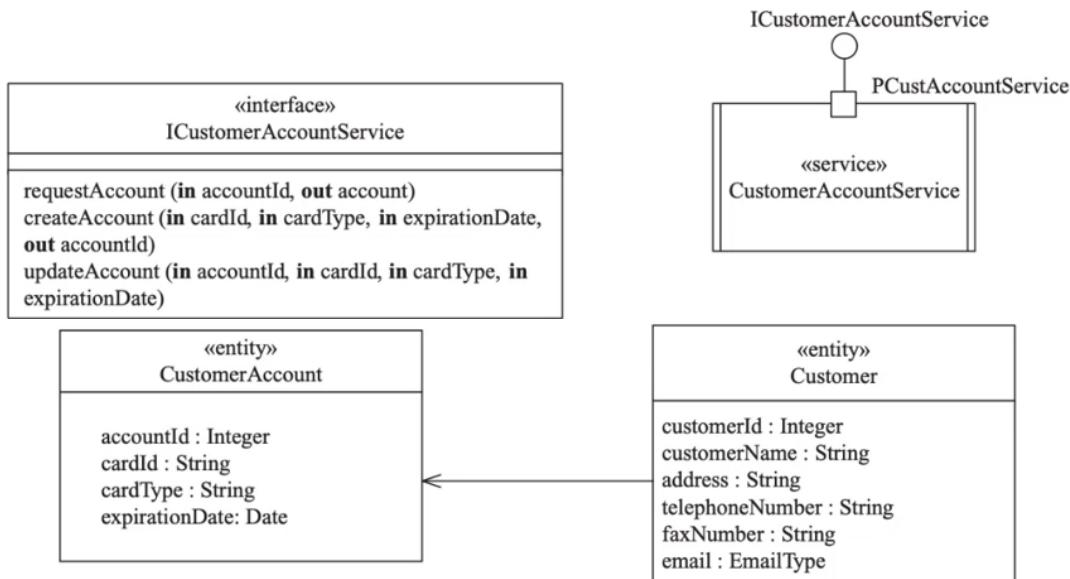
Component Ports e Interfaces for Services:



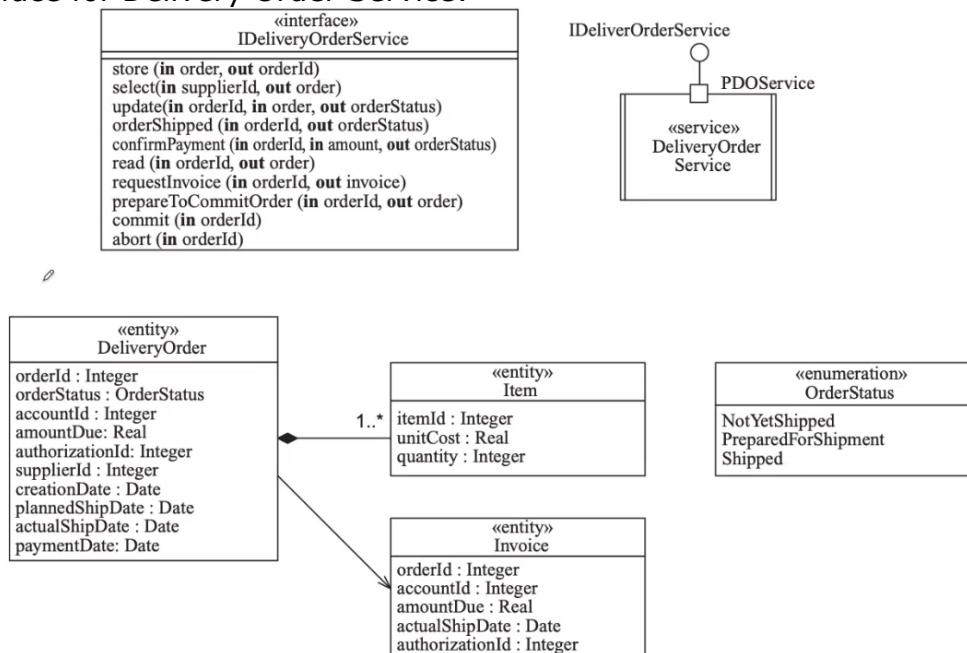
Service Interface for Catalog Service:



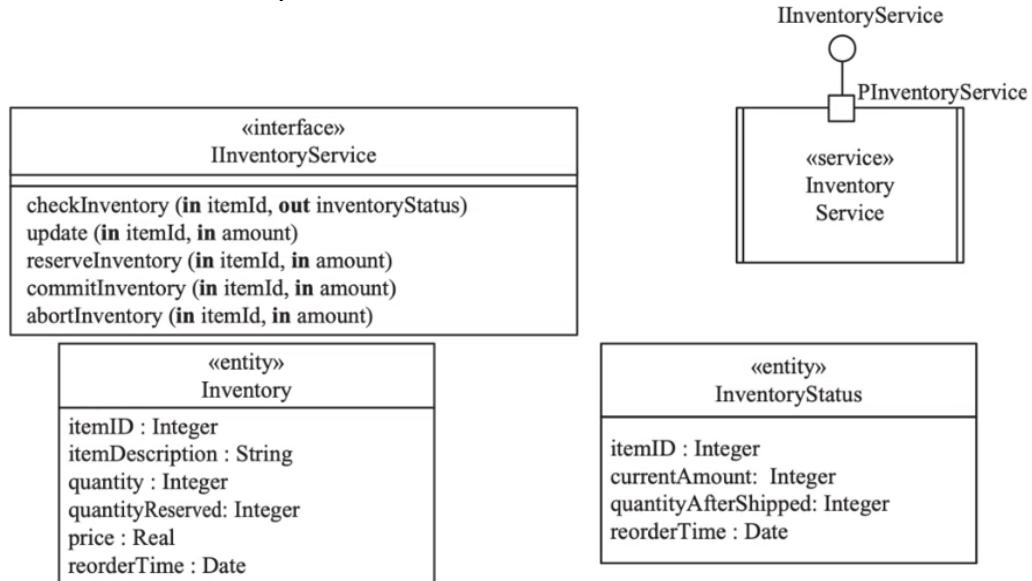
Service Interface for Customer Account Service:



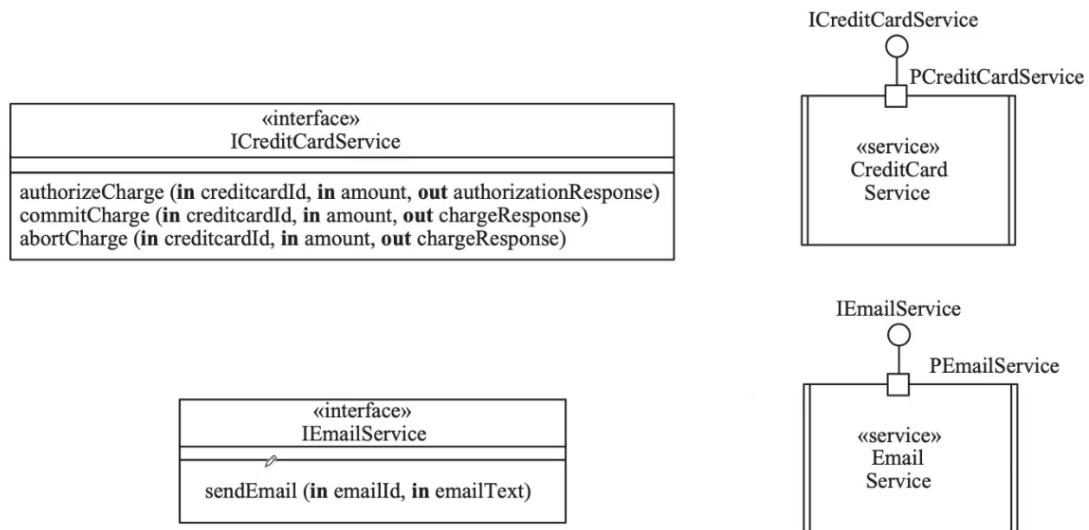
Service Interface for Delivery Order Service:



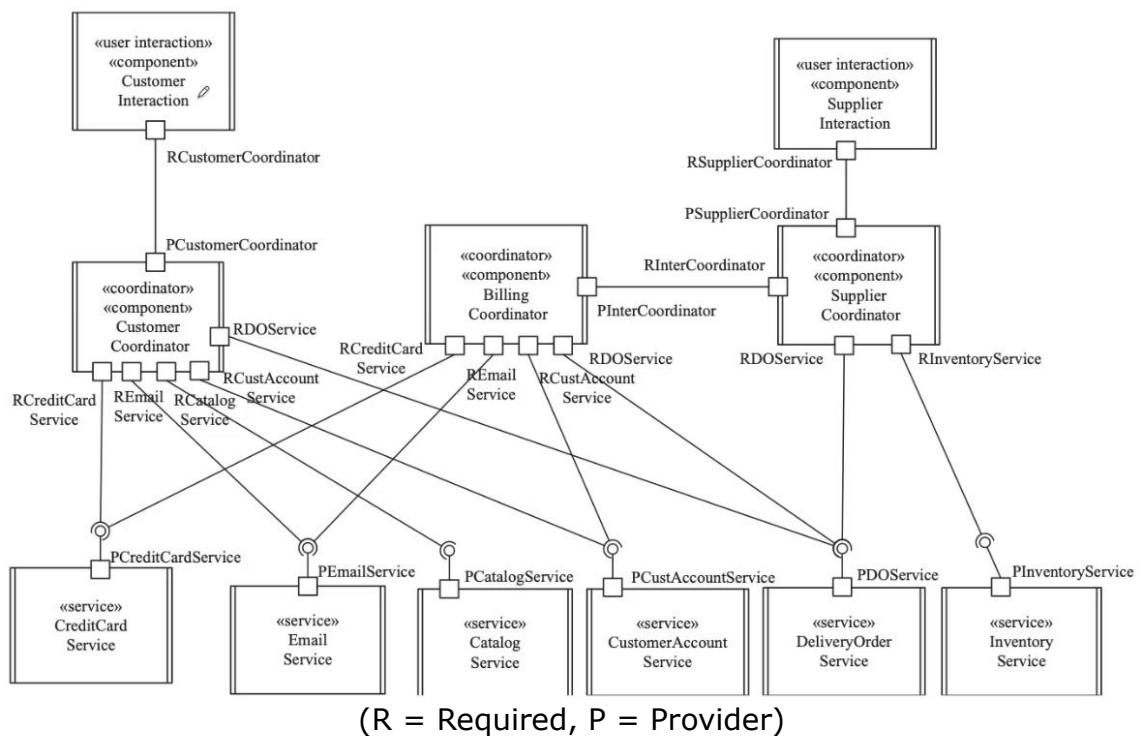
Service Interface for Inventory Service:



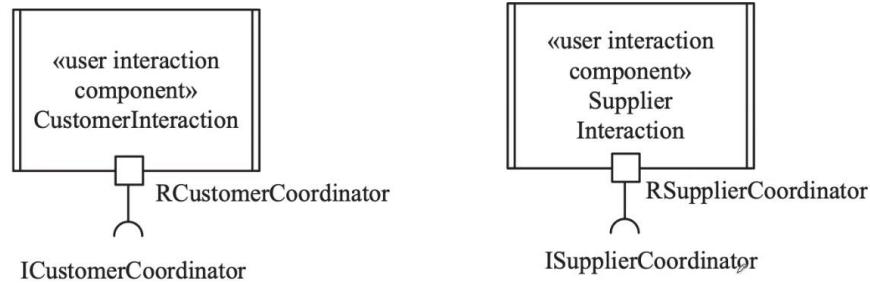
Service Interface for Credit Card e Email services:



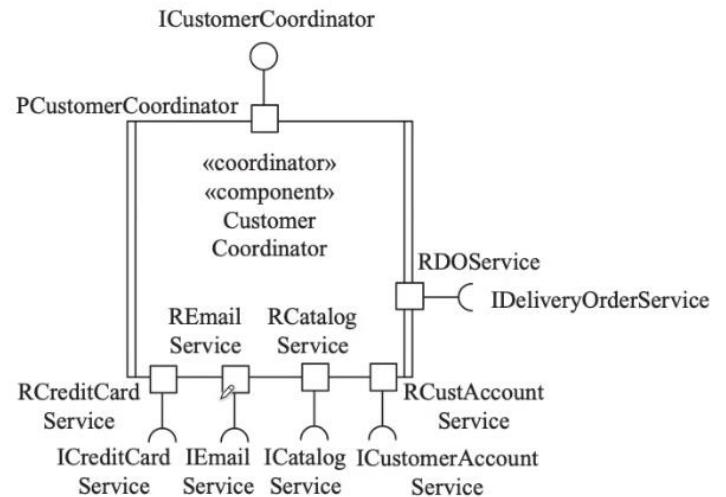
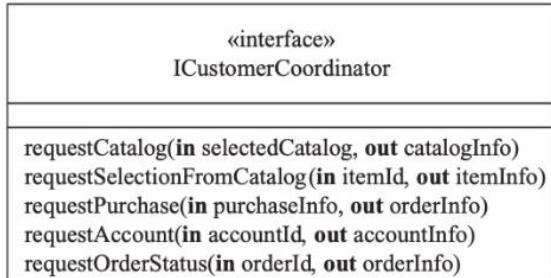
Service-Oriented Software Architecture:



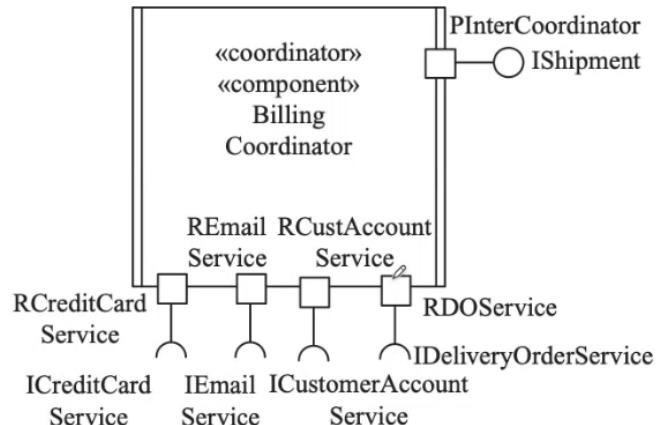
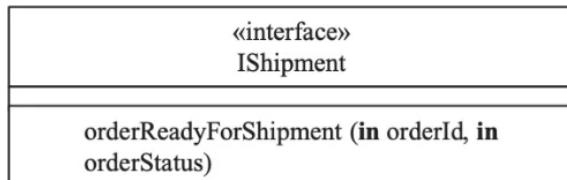
Component ports and interfaces for Customer Interaction e Supplier Interaction:



Component ports and interfaces for Customer Coordinator:



Component ports and interfaces for Billing Coordinator:



SERVICE REUSE (riusabilità dell'applicazione)

Con il paradigma SOA, una volta che i servizi sono stati progettati e le relative interfacce specificate, le informazioni sull'interfaccia del servizio possono essere registrate con un broker di servizi.

I servizi possono essere utilizzati e composti in nuove applicazioni.

Questo caso di studio ha descritto un sistema di shopping online.

Tuttavia, potrebbero essere progettati altri sistemi di commercio elettronico che vogliono utilizzare i servizi forniti dal sistema di acquisto online, come il servizio di catalogo, il servizio di ordine di consegna e il servizio inventario.

DESIGN PATTERNS

Introduzione: La progettazione di software orientato ad oggetti è una attività complessa. Una delle maggiori difficoltà che il progettista deve affrontare è l'individuazione di un insieme di oggetti:

- correttamente individuato
- il più possibile riusabile
- definendo al meglio le relazioni tra classi e le gerarchie di ereditarietà

"Ogni pattern descrive un problema che si ripete più e più volte nel nostro ambiente, descrive poi il nucleo della soluzione del problema, in modo tale che si possa riusare la soluzione un milione di volte, senza mai applicarla alla stessa maniera."

CARATTERISTICHE dei Design Pattern:

- Rappresentano soluzioni a problematiche ricorrenti che si incontrano durante le varie fasi di sviluppo del software.
- Organizzano l'esperienza di OOD favorendo il riuso.
- Evitano al progettista di 'reinventare la ruota' ogni volta
- Permettono di imparare dal lavoro degli altri (evitando errori comuni ...)
- Permettono di definire un linguaggio comune che semplifica la comunicazione tra gli addetti ai lavori.
- Indirizzano verso la scrittura di codice che usa strutture valide.
- Portano di norma ad una buona progettazione: semplificano la manutenzione (adattiva, perfettiva, preventiva e correttiva).
- Non risolvono tutti i problemi!

CLASSIFICAZIONE:

Un primo criterio riguarda lo scopo (purpose). Introduciamo le 3 classi seguenti

- Pattern Creazionali: i pattern di questo tipo sono relativi alle operazioni di creazione di oggetti.
- Pattern Strutturali: sono utilizzati per definire la struttura del sistema in termini della composizione di classi ed oggetti. Si basano sui concetti OO di ereditarietà e polimorfismo.
- Pattern Comportamentali: permettono di modellare il comportamento del sistema definendo le responsabilità delle sue componenti e definendo le modalità di interazione.

Un secondo criterio riguarda il raggio di azione (scope):

- Classi: pattern che definiscono le relazioni fra classi e sottoclassi. Le relazioni sono basate prevalentemente sui concetti di ereditarietà e sono quindi statiche (definite a tempo di compilazione).
- Oggetti: pattern che definiscono relazioni tra oggetti, che possono cambiare durante l'esecuzione e sono quindi più dinamiche.

Classificazione completa:

		Scopo		
		Creazionale	Strutturale	Comportamentale
Raggio d'azione	Classi	Factory Method	Adapter (class)	Interpreter Template Method
		Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of responsibility Iterator Mediator Memento Observer State Strategy Visitor

DESCRIZIONE dei Pattern:

- Nome e Classificazione: il nome illustra l'essenza di un pattern definendo un vocabolario condiviso tra i progettisti; la classificazione lo identifica in termini di scopo e raggio di azione.
 - Motivazione: scenario che descrive in modo astratto il problema al quale applicare il pattern; può includere la lista eventuali pre-condizioni necessarie a garantirne l'applicabilità.
 - Applicabilità: Descrive le situazioni in cui il pattern può essere applicato.
 - Struttura: descrive graficamente la configurazione di elementi che risolvono il problema (relazioni, responsabilità, collaborazioni).
- ➔ È uno schema di soluzione, non una soluzione per un progetto specifico
- Partecipanti: classi ed oggetti che fanno parte del pattern con le relative responsabilità.
 - Conseguenze: risultati che si ottengono applicando il pattern.
 - Implementazioni: tecniche e suggerimenti utili all'implementazione del pattern.
 - Codice di esempio: frammenti di codice che illustrano come implementare in un certo linguaggio di programmazione (es. java o c++) il pattern.
 - Usi conosciuti: esempi di applicazione in sistemi reali
 - Pattern correlati: altri pattern correlati

UN CONCETTO UTILE: I FRAMEWORK

Un Framework non è una semplice libreria.

Un Framework rappresenta il design riusabile di un sistema (o di una sua parte), definito da un insieme di classi astratte.

Un framework è lo scheletro di un'applicazione che viene personalizzato (customized) da uno sviluppatore.

Il programmatore di applicazioni implementa le interfacce e classi astratte ed ottiene automaticamente la gestione delle funzionalità richieste.

I Framework permettono quindi di definire lo scopo e la struttura statica di un sistema.

Sono un buon esempio di progettazione orientata agli oggetti.

Permettono di raggiungere due obiettivi: il riuso del design e il riuso del codice.

Classe Astratta: una classe astratta è una classe che possiede almeno un metodo non implementato, definito "astratto".

Una classe astratta è quindi un template per le sottoclassi da cui deriveranno le specifiche applicazioni.

Un framework è rappresentato da un'insieme di classi astratte e dalle loro interrelazioni.

I Pattern sono spesso mattoni per la costruzione di framework.

ABSTRACT FACTORY (IL PATTERN ABSTRACT FACTORY)

Scopo: fornire una interfaccia per la creazione di famiglie di oggetti tra loro correlati.

Motivazione: realizzazione di uno strumento per lo sviluppo di user interface (UI) in grado di supportare diversi tipi di look & feel. Per garantire la portabilità di una applicazione tra look & feel diversi, gli oggetti non devono essere cablati nel codice.

Classificazione: creazionale basato su oggetti.

ESEMPIO ABSTRACT FACTORY:

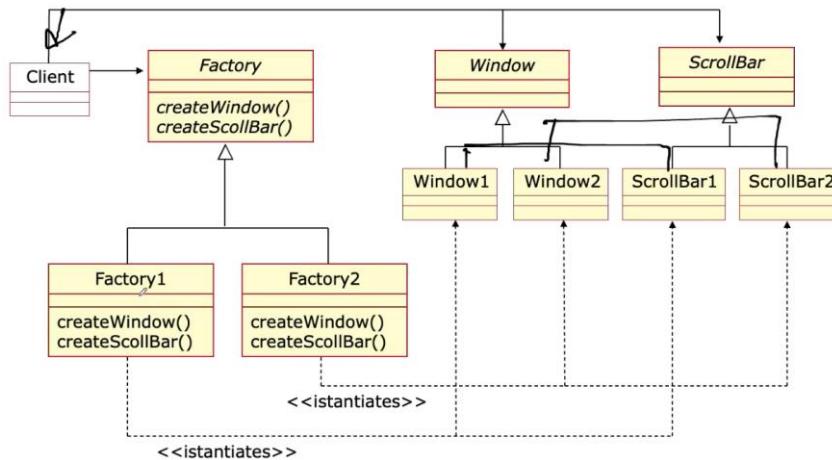
Consideriamo uno strumento per realizzare UI con due soli elementi, in grado di supportare due look & feel diversi

- 1) Look&Feel1 -> Window1 e ScrollBar1
- 2) Look&Feel2 -> Window2 e ScrollBar2

L'applicazione client deve realizzare una UI che rispetti le relazioni tra gli oggetti e che sia portabile da un L&F ad un altro:

- Il client non dovrebbe istanziare direttamente gli oggetti
- Bisogna evitare che il client accoppi (sbagliando) Window1 e ScrollBar2

Usiamo una Abstract Factory:



Senza utilizzare l'Abstract Factory l'applicazione client deve esplicitamente istanziare gli oggetti. Il rispetto delle relazioni è cablato nel codice e deve essere noto al client.

```

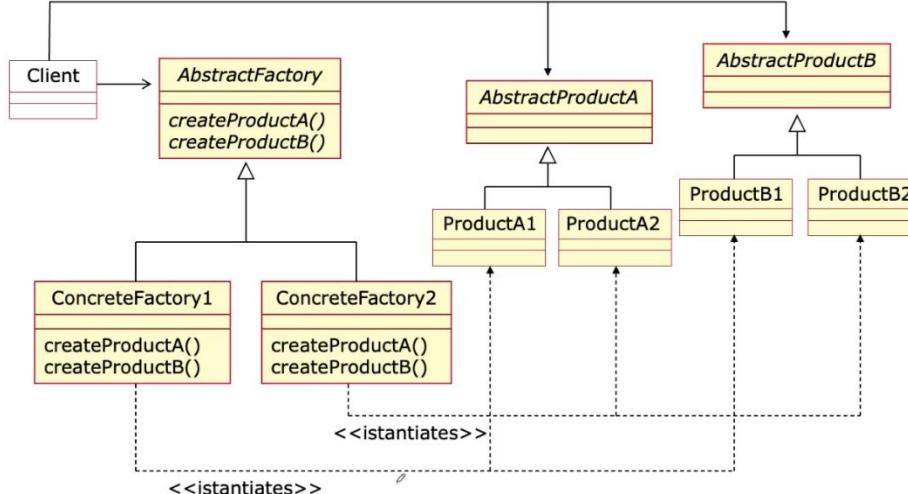
Window w = new Window1();
...
ScrollBar s = new ScrollBar1();
  
```

Con l'Abstract Factory la responsabilità è demandata alla Factory:

```

Factory f = new Factory1();
Window w = f.createWindow();
...
ScrollBar s = f.createScrollBar();
  
```

Quindi:



ALTRÉ CARATTERISTICHE:

Altre caratteristiche sono applicabilità e il campo partecipanti.

Applicabilità:

- A sistema che deve essere indipendente dalle modalità di creazione dei prodotti con cui opera
- A sistema che deve poter essere configurato per usare famiglie di prodotti diverse
- Il client non deve essere legato ad una specifica famiglia

Partecipanti

- AbstractFactory e ConcreteFactory
- AbstractProduct e ConcreteProduct
- Applicazione Client

Conseguenze

- Le classi concrete sono isolate e sotto controllo
- La famiglia di prodotti può essere cambiata rapidamente perché la factory completa compare in un unico punto del codice
- Aggiungere nuove famiglie di prodotti richiede ricompilazione perché l'insieme di prodotti gestiti è legato all'interfaccia della factory

FACTORY METHOD (IL PATTERN FACTORY METHOD)

Scopo: Definire una interfaccia per la creazione di un oggetto, che consenta di decidere a tempo di esecuzione quale specifico oggetto istanziare.

Motivazione: E' un pattern ampiamente usato nei framework, dove le classi astratte definiscono le relazioni tra gli elementi del dominio, e sono responsabili per la creazione degli oggetti concreti.

Classificazione: creazionale basato su classi.

ESEMPIO DI FACTORY METHOD

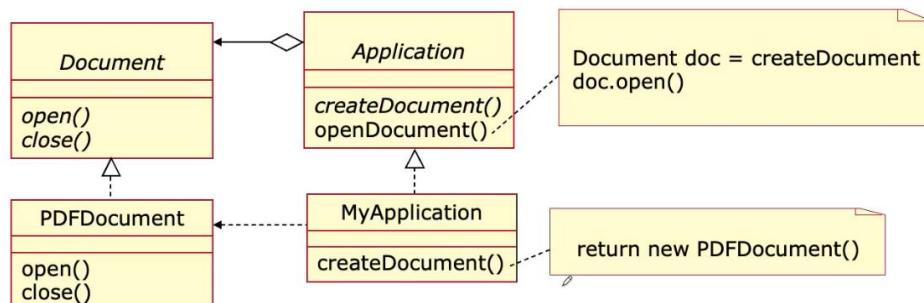
Consideriamo un framework di gestione di documenti di tipo diverso.

Le due astrazioni chiave sono Application e Document.

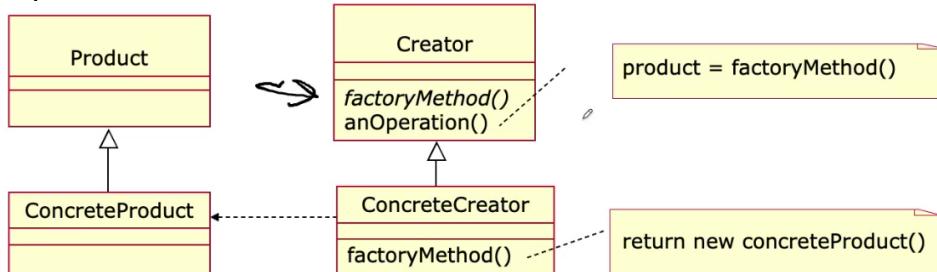
Gli utilizzatori devono definire delle sotto classi per ottenere delle implementazioni adatte all'applicazione specifica.

Application contiene la logica per sapere quando un nuovo documento sarà creato, ma non per sapere quale tipo di documento creare.

Il pattern Factory encapsula la conoscenza della specifica classe da creare al di fuori del framework:



Struttura Factory Method:



ALTRE CARATTERISTICHE DEL PATTERN FACTORY METHOD

Applicabilità

- Una classe non è in grado di sapere in anticipo le classi di oggetti che deve creare.
- Una classe vuole che le sue sottoclassi scelgano gli oggetti da creare.
- Le classi delegano la responsabilità di creazione.

Partecipanti

- Product e ConcreteProduct
- Creator e ConcreteCreator

Conseguenze: elimina la necessità di riferirsi a classi dipendenti dall'applicazione all'interno del codice.

ADAPTER (IL PATTERN ADAPTER)

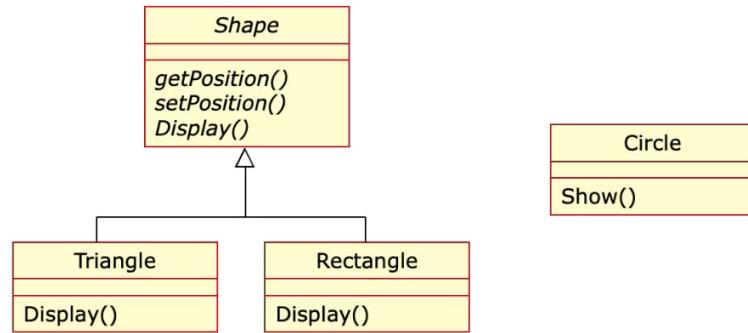
Scopo: Convertire l'interfaccia di una classe esistente incompatibile con un client, in una compatibile.

Motivazione: Consideriamo un editor che consente di disegnare e comporre elementi grafici. L'astrazione chiave è un singolo oggetto grafico. Supponiamo di voler integrare un nuovo componente, ma che questo non abbia una interfaccia compatibile con l'editor.

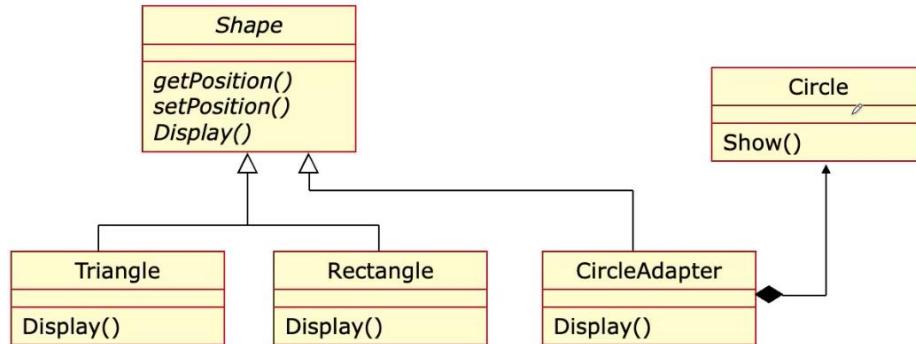
Classificazione: strutturale basato su classi/oggetti.

ESEMPIO DI ADAPTER

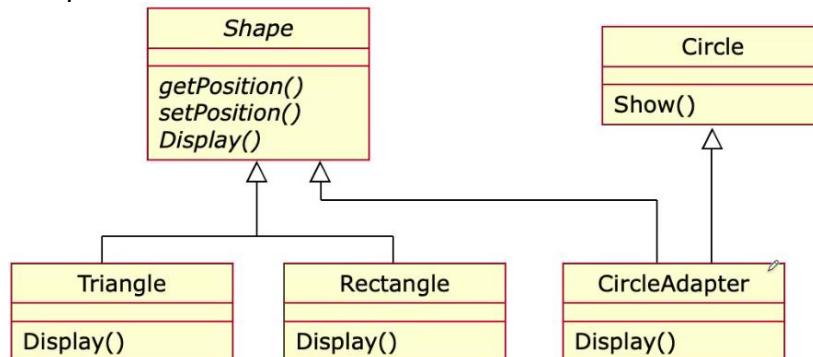
Supponiamo di voler integrare il componente Circle nell'editor che già supporta le forme Triangle e Rectangle.



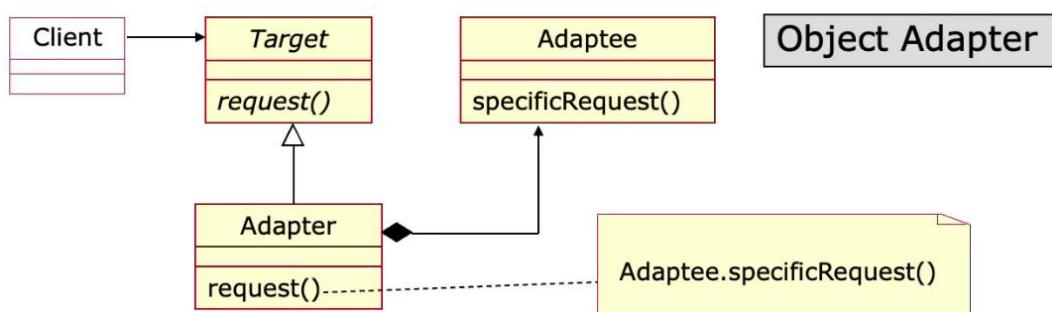
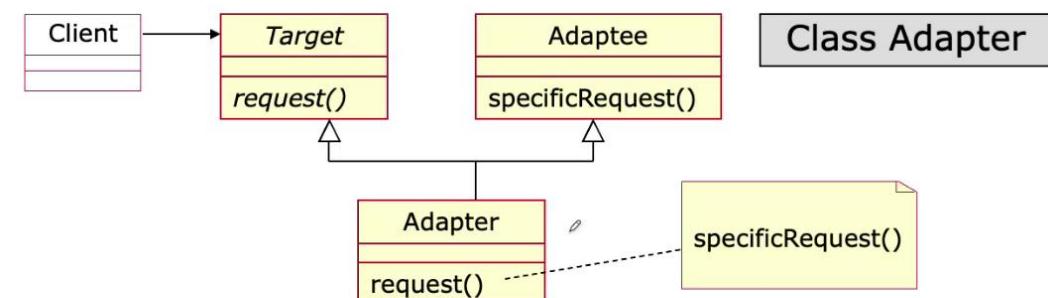
Soluzione1: Object Adapter



Soluzione2: Class Adapter



Struttura:



ALTRÉ CARATTERISTICHE DEL PATTERN ADAPTER:

Applicabilità: si usa quando si vuole riusare una classe esistente, ma con interfaccia incompatibile con quella desiderata.

Partecipanti: Client, Target, Adapter ed Adaptee.

Conseguenze: E' necessario prendere in considerazione l'effort necessario all'adattamento.

COMPOSITE (IL PATTERN COMPOSITE)

Scopo: Comporre oggetti in strutture che consentano di trattare i singoli elementi e la composizione in modo uniforme.

Motivazione: Le applicazioni grafiche consentono di trattare in modo uniforme sia le forme geometriche di base (linee, cerchi, ...) sia gli oggetti complessi che si creano a partire da questi elementi semplici.

Molti editor grafici ad esempio hanno la funzione raggruppa.

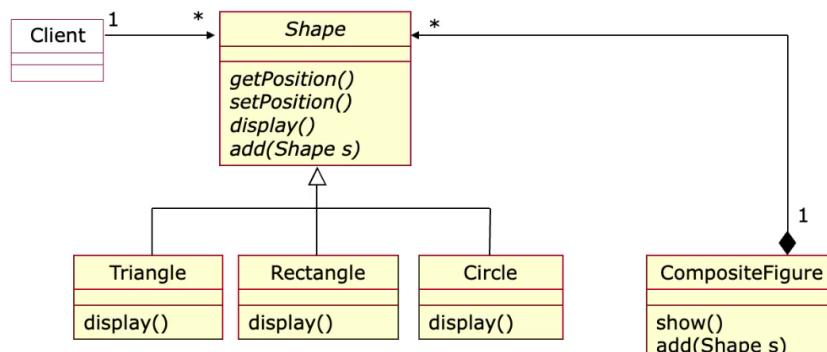
Classificazione: strutturale basato su oggetti.

ESEMPIO DI COMPOSITE

Consideriamo un'applicazione grafica in grado di gestire gli oggetti elementari Triangle, Rectangle, e Circle.

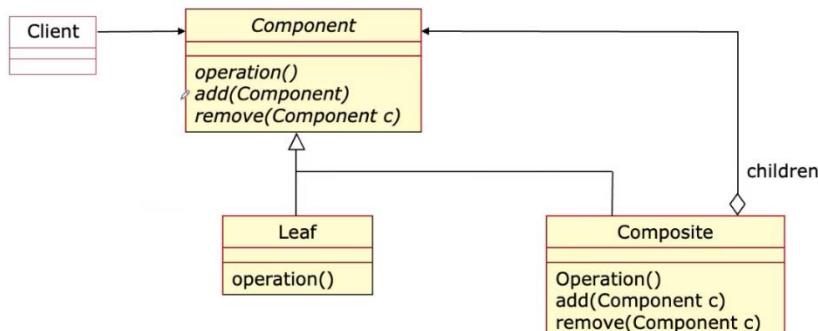
Un ulteriore requisito è che l'applicazione deve poter permettere il raggruppamento dinamico di oggetti elementari in oggetti composti.

Gli oggetti elementari e quelli composti devono essere trattati in modo uniforme.



La sola relazione di composizione non soddisfa tutti i requisiti: CompositeFigure è una collezione di oggetti elementari ma non è essa stessa una figura geometrica (Shape).
→ La relazione di ereditarietà permette di considerare CompositeFigure una figura geometrica.

Struttura:



ALTRÉ CARATTERISTICHE DEL PATTERN COMPOSITE

Applicabilità: si usa quando si vogliono rappresentare gerarchie di oggetti in modo che oggetti semplici e oggetti composti siano trattati in modo uniforme.

Partecipanti: Component e Composite, Leaf, Client.

Conseguenze:

- I client sono semplificati perché gli oggetti semplici e quelli composti sono trattati allo stesso modo.
- L'aggiunta di nuovi oggetti Leaf o Composite è semplice, e questi potranno sfruttare il codice dell'applicazione Client già esistente.
- Può rendere il sistema troppo generico. Non è possibile fare in modo che un oggetto composito contenga solo un certo tipo di oggetti.

DECORATOR (IL PATTERN DECORATOR)

Scopo: Aggiungere dinamicamente funzionalità (responsabilità) ad un oggetto.

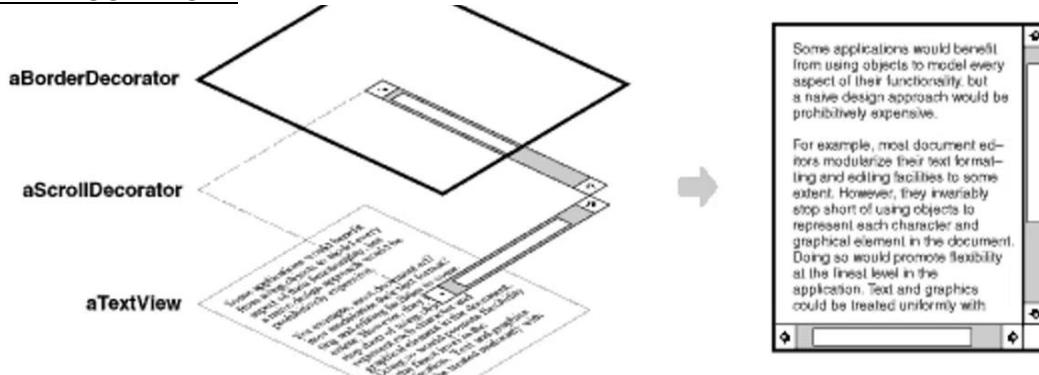
→ Il subclassing è una alternativa statica e il cui scope è a livello di classe e non di singolo oggetto.

Motivazione: Uno scenario classico di applicabilità per questo pattern è la realizzazione di interfacce utente.

Responsabilità quali il testo scorrevole o un particolare bordo devono essere aggiunti a livello di singolo oggetto.

Classificazione: strutturale basato su oggetti.

ESEMPIO DI DECORATOR:



Come combinare i tre oggetti **aBorderDecorator**, **aScrollDecorator** e **aTextView** per raggiungere l'obiettivo della dinamicità?

Il primo approccio che scartiamo è quello del subclassing.

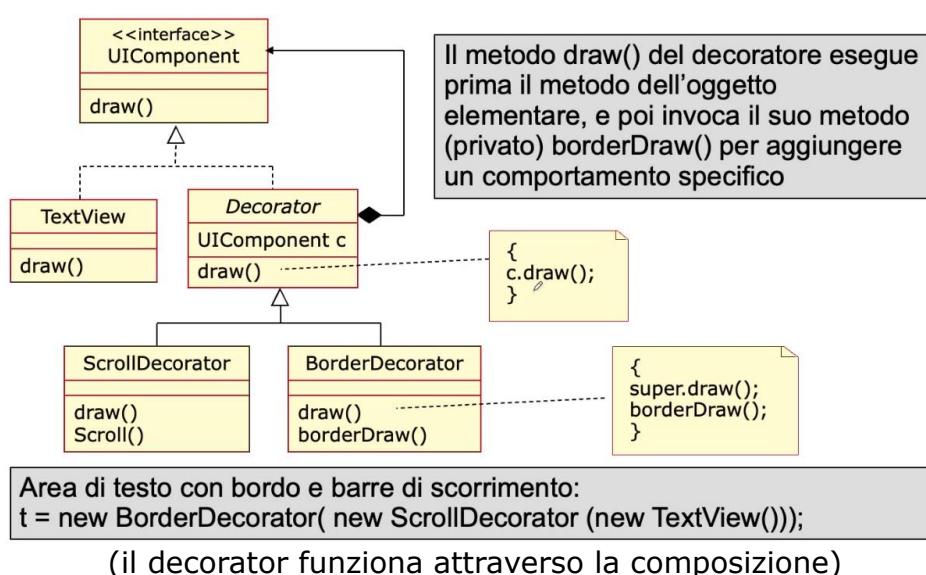
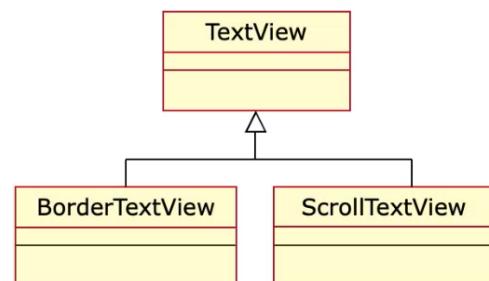
- E' un approccio statico. Una volta creato **BorderTextView** non posso più cambiarlo.
- Devo creare una sottoclasse per ogni esigenza
- E per creare una finestra che abbia sia il bordo, sia il testo scorrevole?
- **BorderAndScrollTextView?**
- **ScrollAndBorderTextView?**

Un approccio più flessibile è quello di racchiudere un oggetto elementare in un altro, che aggiunge una responsabilità particolare.

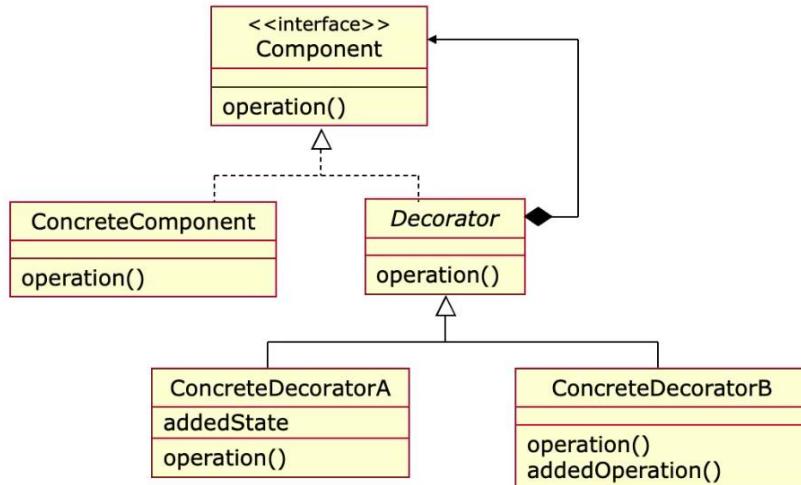
L'oggetto contenitore è chiamato Decorator.

Il Decorator ha una interfaccia conforme all'oggetto da decorare.

Il Decorator trasferisce le richieste all'oggetto decorato ma può svolgere funzioni aggiuntive (ad esempio aggiungere un bordo) prima o dopo il trasferimento della richiesta.



Struttura Decorator:



ALTRÉ CARATTERISTICHE DEL PATTERN DECORATOR

Applicabilità

- Si applica quando è necessario aggiungere responsabilità agli oggetti in modo trasparente e dinamico.
- Si applica quando il subclassing non è adatto.

Partecipanti

- Component e ConcreteComponent
- Decorator e ConcreteDecorator(s)

Conseguenze:

- Maggiore flessibilità rispetto all'approccio statico
- Evita di definire strutture gerarchiche complesse

NOTE AGGIUNTIVE SUL PATTERN DECORATOR

In Java il Decorator è particolarmente usato nella definizione degli Stream di I/O:

BufferedInputStream bin = new BufferedInputStream(new FileInputStream("test.dat"));
E' simile al pattern Composite, ma la finalità è diversa. Decorator serve ad aggiungere responsabilità in modo dinamico.

E' simile al pattern Adapter, ma quest'ultimo si limita ad un adattamento (limitato) di una interfaccia.

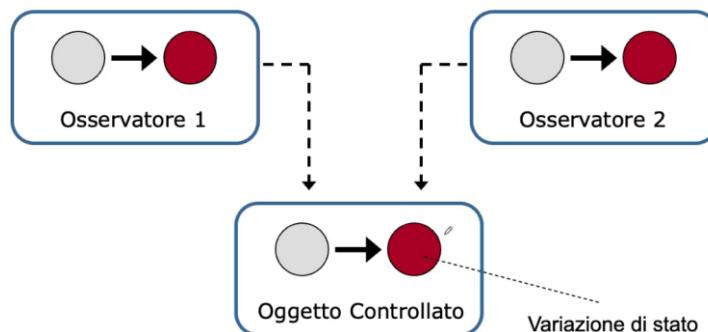
OBSERVER (IL PATTERN OBSERVER)

Scopo: Definire una dipendenza uno a molti tra oggetti, mantenendo basso il grado di coupling. In altre parole la variazione dello stato di un oggetto deve essere osservata da altri oggetti, in modo che possano aggiornarsi automaticamente.

Motivazione: Lo scenario classico è quello di applicazioni con GUI, realizzate secondo il paradigma Model-View-Control. Quando il Model cambia, gli oggetti che implementano la View devono aggiornarsi

Classificazione: comportamentale basato su oggetti

Idea di fondo:



OBSERVER: UNA POSSIBILE SOLUZIONE (NON BUONA)

Una prima soluzione potrebbe essere quella di utilizzare, nell'oggetto osservato, attributi pubblici oppure metodi pubblici che leggono il valore di un attributo protetto.

Non è una buona soluzione:

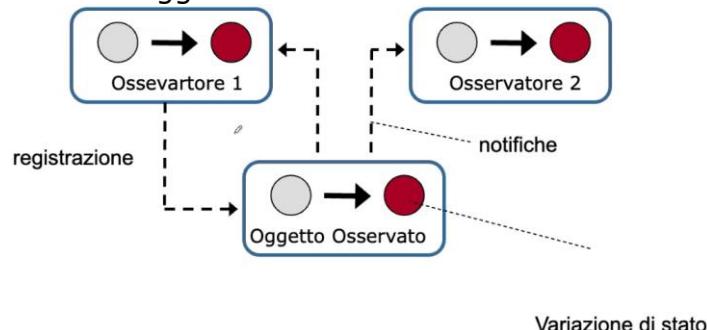
- non è scalabile, se aumentano troppo gli osservatori l'oggetto osservato è sovraccaricato dalle richieste
- gli osservatori dovrebbero continuamente interrogare l'oggetto osservato
- variazioni rapide potrebbero comunque non essere rilevate da qualche osservatore

LEZ32 – 10/04/2024

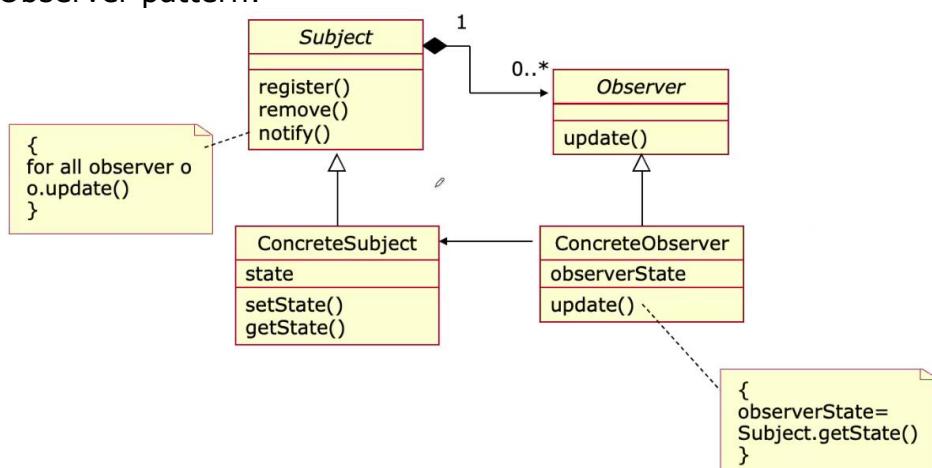
OBSERVER APPROCCIO CORRETTO

Il pattern Observer prevede che gli osservatori si registrino presso l'oggetto osservato. In questo modo è l'oggetto osservato che notifica ogni cambiamento di stato agli osservatori. Quando l'osservatore rileva la notifica può interrogare l'oggetto osservato, oppure può svolgere altre operazioni indipendenti dal valore specifico dello stato.

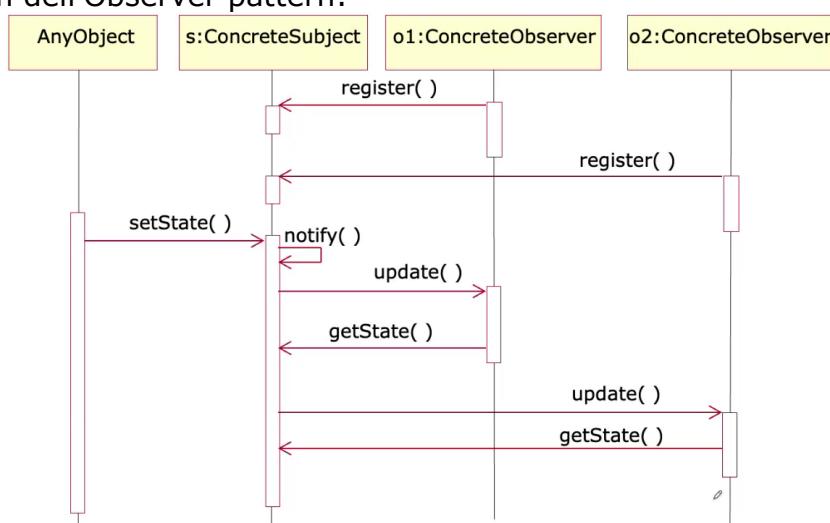
Gli osservatori possono essere aggiunti a runtime:



Struttura dell'Observer pattern:



Sequence diagram dell'Observer pattern:



ALTRÉ CARATTERISTICHE DEL PATTERN OBSERVER

Applicabilità:

- Si applica quando una azione può essere scomposta in due ambiti, ciascuno dei quali encapsulato in oggetti separati per mantenere basso il livello di coupling.
 - Gestire le modifiche di oggetti conseguenti alla variazione dello stato di un oggetto.

Partecipanti:

- Subject e ConcreteSubject
 - Observer e ConcreteObserver

Consequenze:

- l'accoppiamento tra Subject ed Observer è astratto
 - il Subject conosce solo la lista degli osservatori
 - la notifica è una comunicazione di tipo broadcast
 - il Subject non si occupa di quanti sono gli Observer registrati
 - attenzione perché una modifica al Subject scatena una serie di modifiche su tutti gli osservatori e su tutti gli oggetti da questi dipendenti

TEMPLATE METHOD (IL PATTERN COMPORTAMENTALE TEMPLATE METHOD)

Scopo: Definire la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi alle sottoclassi.

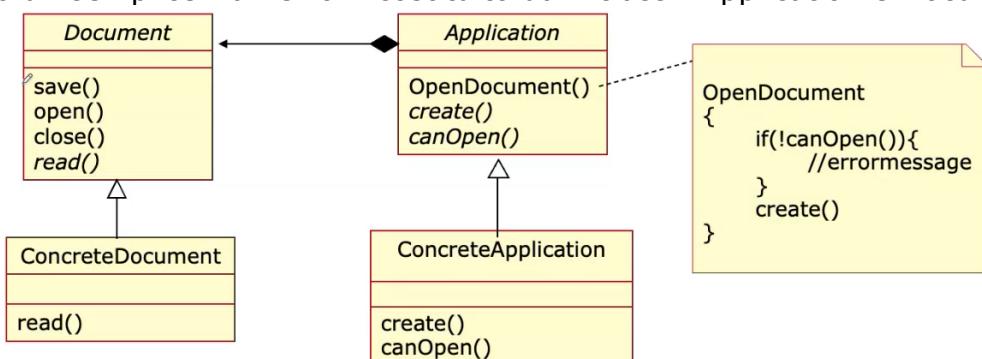
Le sottoclassi ridefiniscono solo alcuni passi dell'algoritmo ma non la sua struttura.

Motivazione: consideriamo un framework per costruire applicazioni in grado di gestire documenti diversi. Il Template Method definisce un algoritmo in base ad operazioni astratte che saranno definite nelle sottoclassi specifiche.

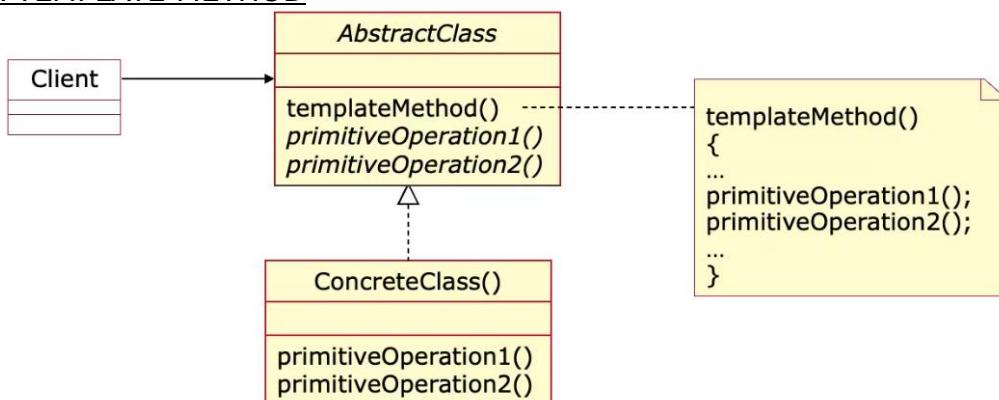
Classificazione: comportamentale basato su classi.

ESEMPIO DI TEMPI ATE METHOD

Consideriamo un semplice framework costituito da 2 classi: Application e Document.



STRUTTURA TEMPLATE METHOD



ALTRÉ CARATTERISTICHE DEL PATTERN TEMPLATE METHOD

Applicabilità:

- E' utilizzato per implementare la parte invariante di un algoritmo, lasciando alle sottoclassi la definizione degli step variabili
 - E' utile quando ci sono comportamenti comuni che possono essere inseriti nel template

Partecipanti:

- AbstractClass e ConcreteClass
- Client

Conseguenze:

- I metodi template permettono il riuso del codice
- Creano una struttura di controllo invertito dove è la classe padre che chiama le operazioni ridefinite nei figli e non viceversa
- Per controllare l'estendibilità delle sottoclassi, i metodi richiamati dal template sono chiamati metodi gancio (hook)
- I metodi hook possono essere implementati, offrendo un comportamento standard, che la sottoclasse può volendo ridefinire

NOTE AGGIUNTIVE SUL PATTERN TEMPLATE METHOD

Il Template Method è simile al Factory Method

- Invocazione di metodi astratti tramite interfaccia
- Implementazione dei metodi rimandata a classi concrete non note

Indirizzano però problemi diversi

- Il Template Method è il metodo che invoca i metodi astratti, al fine di generalizzare un algoritmo
- Il Factory Method è un metodo astratto che deve creare e restituire l'istanza di classe concreta, al fine di sganciare il cliente dalla scelta del tipo specifico

STRATEGY (IL PATTERN COMPORTAMENTO BASATO SU OGGETTI CHIAMATO STRATEGY)

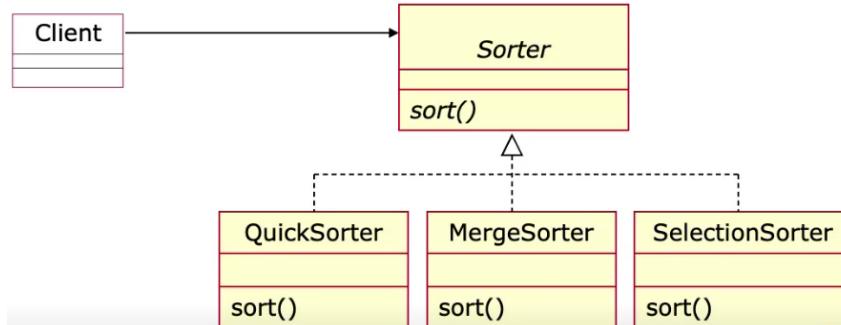
Scopo: Definire ed encapsulare una famiglia di algoritmi in modo da renderli intercambiabili indipendentemente dal client che li usa.

Motivazione: Consideriamo la famiglia degli algoritmi di ordinamento. Ne esistono diversi (QuickSort, BubbleSort, MergeSort, etc). Costruiamo una applicazione che li supporti tutti, che possa essere facilmente estendibile, e che permetta una scelta rapida del tipo di algoritmo.

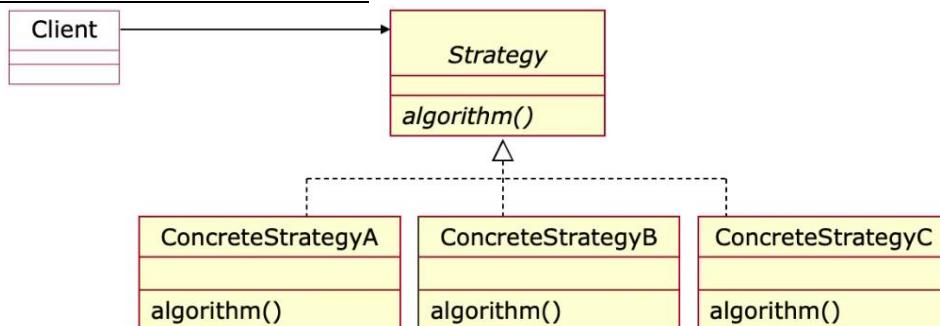
Classificazione: comportamentale basato su oggetti

ESEMPIO DI TEMPLATE STRATEGY

Gli algoritmi di ordinamento devono essere indipendenti dal vettore di dati su cui operano, e dal resto dell'implementazione dell'applicazione.



STRUTTURA DEL TEMPLATE STRATEGY



ALTRÉ CARATTERISTICHE DEL PATTERN STRATEGY

Applicabilità:

- Molte classi correlate differiscono solo per il comportamento (il pattern fornisce un modo per avere una interfaccia comune)
- Sono necessarie più varianti di uno stesso algoritmo, a seconda dei tipi di dato in ingresso o delle condizioni operative.

Partecipanti:

- Strategy e ConcreteStrategy
- Client

Conseguenze:

- Il pattern separa l'implementazione degli algoritmi dal contesto dell'applicazione (usare il subclassing della classe Client per aggiungere un algoritmo non sarebbe stata una buona scelta)
- Le diverse strategie eliminano i blocchi condizionali che sarebbero necessari inserendo tutti i diversi comportamenti in una unica classe
- Lo svantaggio principale è che i client devono conoscere le diverse strategie

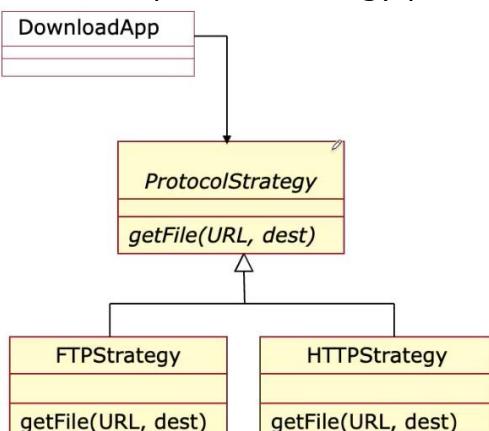
ESERCIZI SU DESIGN PATTERN

➤ ESERCIZIO1:

- Realizzare una applicazione per gestire il download da siti http e ftp. La selezione avviene in base all'inizio dell'url (ftp o http).
- Suggerimento: usare una factory e uno strategy pattern

Soluzione:

- L'applicazione deve gestire il download da siti supportando solo due protocolli.
- E' ragionevole scrivere una applicazione che sia estendibile senza problemi (manutenzione evolutiva).
- Usiamo un pattern Strategy per implementare il download secondo i due protocolli

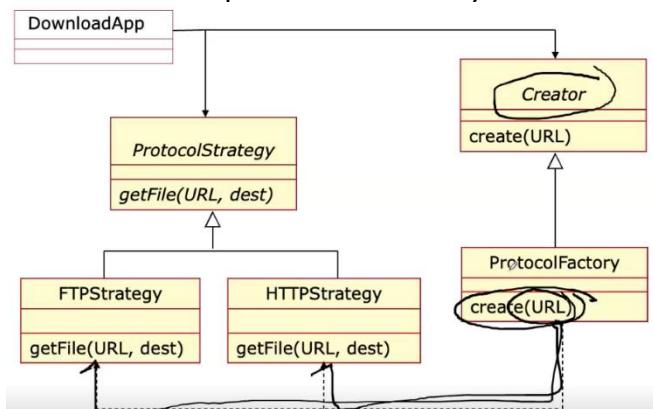


```

download( String url, String dest ) {
    ProtocolStrategy ps=null;
    url = url.toLowerCase();
    if( url.startsWith( "ftp" ) ) {
        ps = new FtpStrategy();
    }
    else if( url.startsWith( "http" ) ) {
        ps = new HttpStrategy();
    }
    else {
        //No operation available
    }
    ps.getFile( url, dest );
}
  
```

In questo modo ogni volta che vogliamo aggiungere un nuovo protocollo dobbiamo modificare la classe DownloadApp. DownloadApp in questo scenario deve conoscere alcuni dettagli implementativi: ad esempio deve sapere esattamente quali protocolli sono supportati e come si chiamano le relative classi di gestione.

Introduciamo quindi una Factory:



```

abstract class Creator {
    public abstract ProtocolStrategy create( String url );
}

class ProtocolFactory extends Creator {
    public ProtocolStrategy create( String ind ){
        ind = ind.toLowerCase();
        if( ind.startsWith( "ftp" ) ) {
            return new FtpStrategy();
        }
        else if( ind.startsWith( "http" ) ) {
            return new HttpStrategy();
        }
        else {
            //throw an exception...
        }
    }
}
  
```

La Factory Contiene la logica per creare la corretta sottoclasse di ProtocolStrategy

```
abstract class ProtocolStrategy {
    abstract void getFile(String url, String dest);
}
```

```
class FtpStrategy extends ProtocolStrategy {
    public void getFile(String url, String dest) {
        //...method implementation
    }
}
```

```
class Download{
```

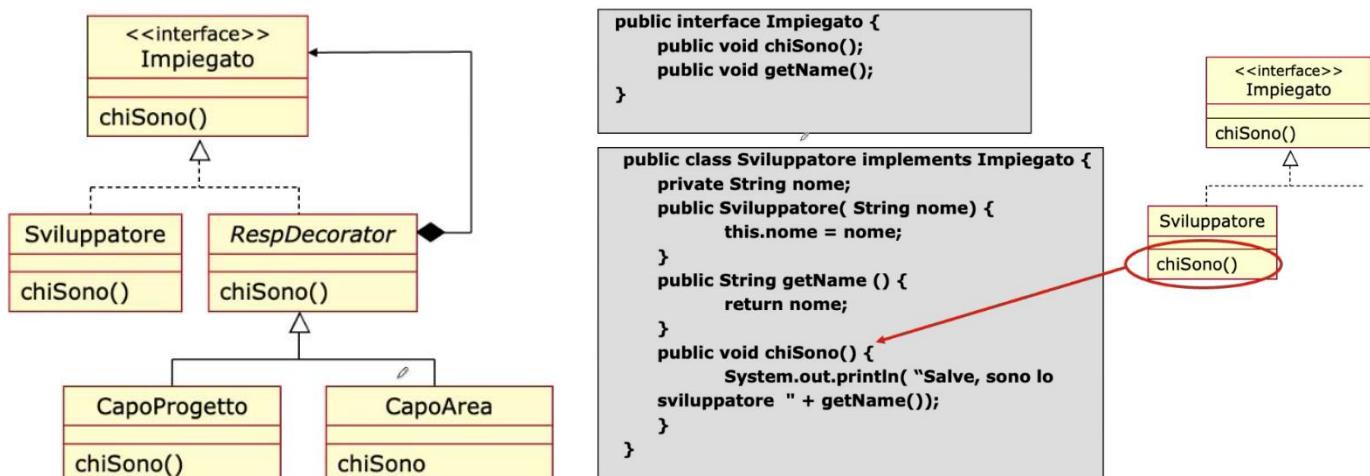
```
...
    download( String url, String dest ) {
        ProtocolFactory p = new ProtocolFactory();
        ProtocolStrategy ps = p.create(url);
        ps.getFile(url,dest);
    }
}
```

➤ ESERCIZIO2:

- Descrivere un modello a oggetti che rappresenti gli Impiegati di una azienda.
- Gli impiegati hanno tutti il metodo chiSono() che visualizza il nome e la particolare responsabilità.
- Gli impiegati possono avere delle responsabilità aggiuntive: possono essere CapoArea o CapoProgetto, in modo non esclusivo.
- Una particolare categoria di impiegati che ci interessa sono gli Sviluppatori
- Suggerimento: usare il Decorator Pattern per le responsabilità, e l'ereditarietà per contraddistinguere gli Sviluppatori dagli Impiegati

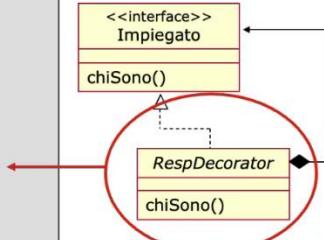
Soluzione: Dall'esame dei requisiti si può provare a disegnare una struttura con queste caratteristiche:

- Deve esistere una classe Impiegato che definisce una interfaccia comune (metodo chiSono())
- Deve esistere una classe Sviluppatore che eredita da Impiegato
- Definiamo con il pattern Decorator due classi CapoArea e CapoProgetto



```
abstract class RespDecorator implements Impiegato {
    protected Impiegato responsabile;
    public RespDecorator(Impiegato imp) {
        responsabile = imp;
    }
    public String getName() {
        return responsabile.getName();
    }
    public void chiSono() {
        responsabile.chiSono();
    }
}
```

```
public class CapoProgetto extends RespDecorator {
    public CapoProgetto ( Impiegato imp ) {
        super( imp );
    }
    public void chiSono() {
        super.chiSono();
        System.out.println("e sono anche un CapoProgetto!");
    }
}
```



```
Impiegato pippo = new Sviluppatore("Pippo");
pippo.chiSono();
```

Salve, sono lo sviluppatore Pippo

```
ippo = new CapoProgetto(ippo));
ippo.chiSono();
```

Salve, sono lo sviluppatore Pippo e sono anche un CapoProgetto!

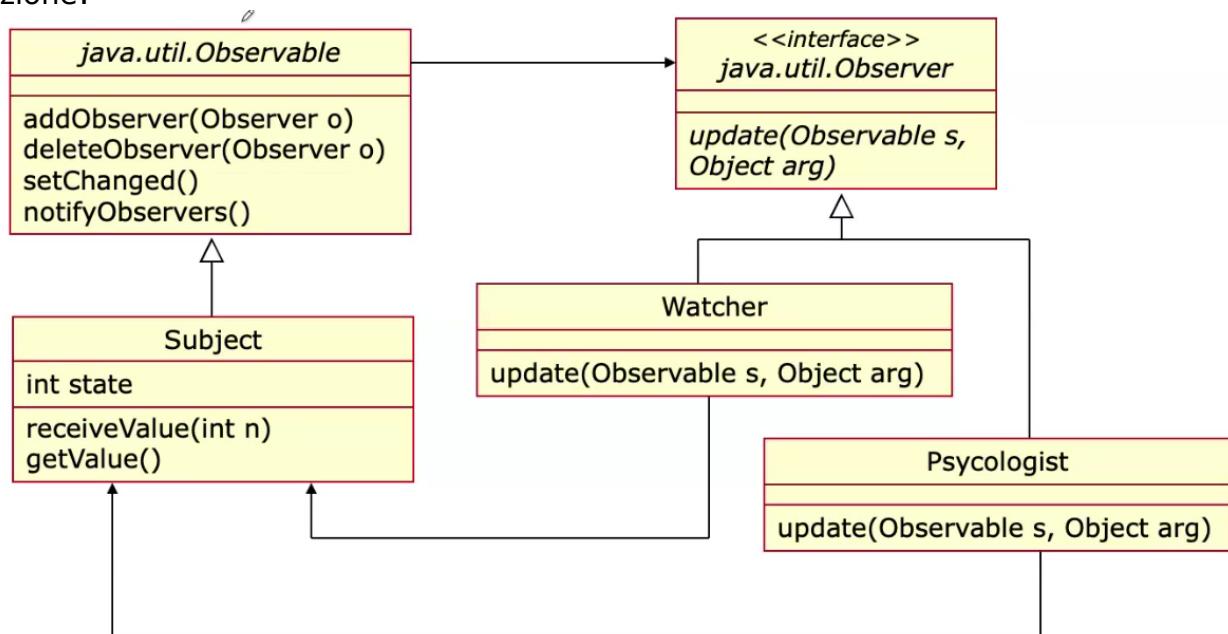
```
Impiegato pluto = new CapoProgetto(new Sviluppatore("Pluto"));
pluto.chiSono();
```

Salve, sono lo sviluppatore Pluto e sono anche un CapoProgetto!

➤ **ESERCIZIO3:**

- Consideriamo un oggetto Subject che riceve valori dall'esterno e che in modo casuale cambia il suo stato interno accettando o meno il valore ricevuto.
- Due oggetti Watcher e Psicologist devono monitorare il cambiamento di valore di Subject.
- Utilizzare il pattern Observer sfruttando le capability di Java.
- Suggerimento: usare le interfacce Observer ed Observable

Soluzione:



```

import java.util.Observer;
import java.util.Observable;

public class Subject extends Observable {
    private int value = 0;
    public void receiveValue( int newNumber ) {
        if (Math.random() < .5) {
            System.out.println( "Subject : Ho cambiato il mio stato" );
            value = newNumber;
            this.setChanged();
        } else
            System.out.println( "Subject : Stato interno invariato" );
        this.notifyObservers();
    }

    public int returnValue()
    {
        return value;
    }
}
  
```

```

import java.util.Observer;
import java.util.Observable;

public class Watcher implements Observer {
    private int changes = 0;

    public void update(Observable obs, Object arg) {
        System.out.println( "Watcher: Lo stato del subject è: "
        + ((ObservedSubject) obs).returnValue() + ".");
        changes++;
    }

    public int observedChanges() {
        return changes;
    }
}
  
```

```

public class Esempio {
    public static void main (String[] args) {
        ObservedSubject s = new ObservedSubject();
        Watcher o = new Watcher();
        Psychologist p = new Psychologist();
        s.addObserver( o );
        s.addObserver( p );

        for(int i=1;i<=10;i++){
            System.out.println( "Main : Invio il numero " + i );
            s.receiveValue( i );
        }
        System.out.println( "Il subject ha cambiato stato " +
o.observedChanges() + " volte." );
        ...
    }
}

```

Main : Invio il numero 1

Subject : Stato interno invariato

Main : Invio il numero 2

Subject : Ho cambiato il mio stato

Watcher: Lo stato del subject è: 2.

...

Il subject ha cambiato stato 4 volte.

LEZ33 – 15/04/2024

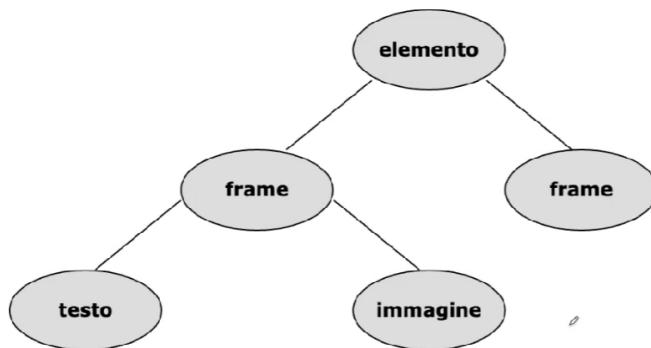
➤ ESERCIZIO4:

- Realizzare un semplice Editor in grado di gestire elementi grafici costituiti dai seguenti elementi elementari: frame, testi, immagini
- L'editor deve supportare due diversi algoritmi di formattazione
- L'editor deve supportare elementi grafici complessi come le barre di scorrimento o dei bordi grafici
- Suggerimento: usare i pattern Composite, Strategy, Decorator.

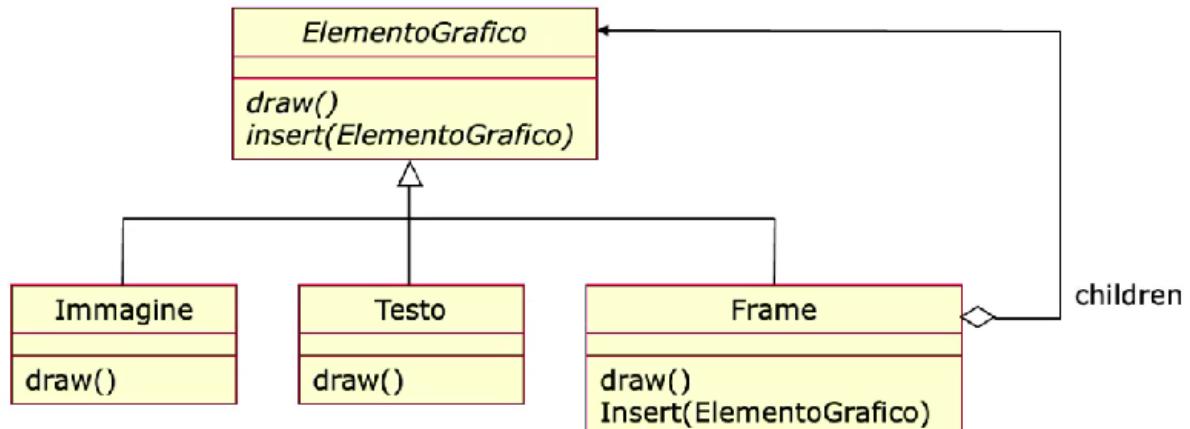
Soluzione:

- La struttura degli elementi tipografici può essere resa con un Composite Pattern
- La gestione degli algoritmi di formattazione può essere disaccoppiata ed estendibile utilizzando il pattern Strategy
- Il controllo delle funzionalità grafiche può essere ottenuto utilizzando il Decorator Pattern per gli elementi grafici

Per semplicità consideriamo la rappresentazione di elementi semplici (frame, testo, immagine)

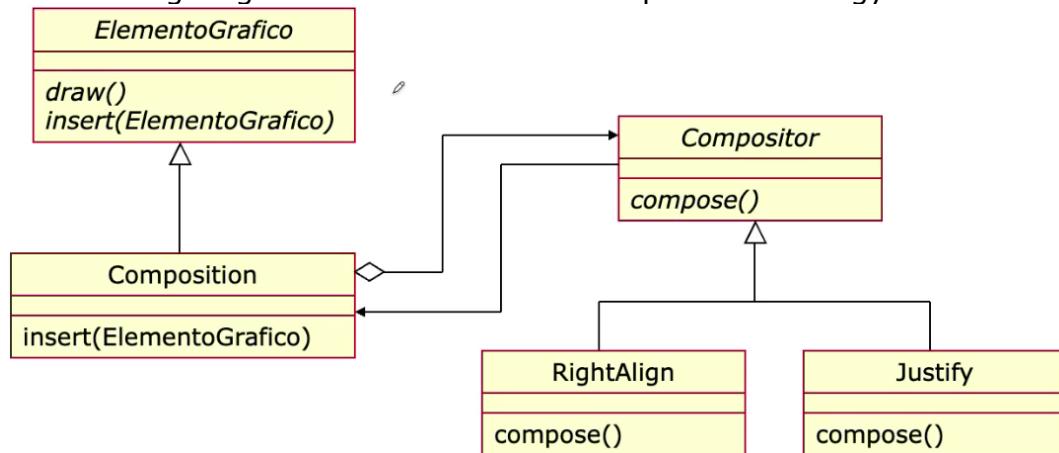


Rappresentazione degli elementi grafici: il pattern Composite

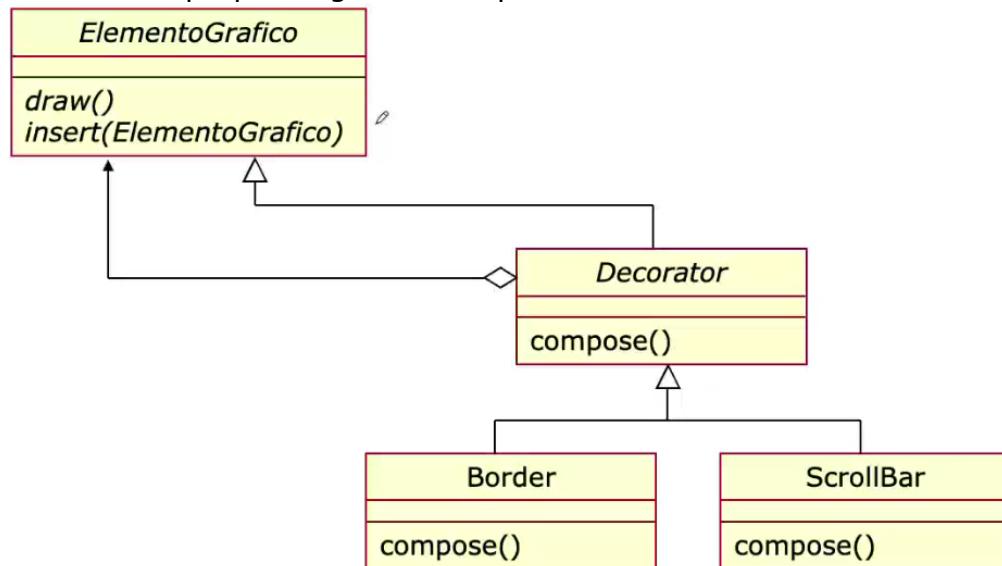


Nota: In Java le classi foglia (Immagine e Testo) devono necessariamente implementare il metodo insert() previsto dalla classe astratta. In questo caso una possibile soluzione è che queste generino una eccezione apposita.

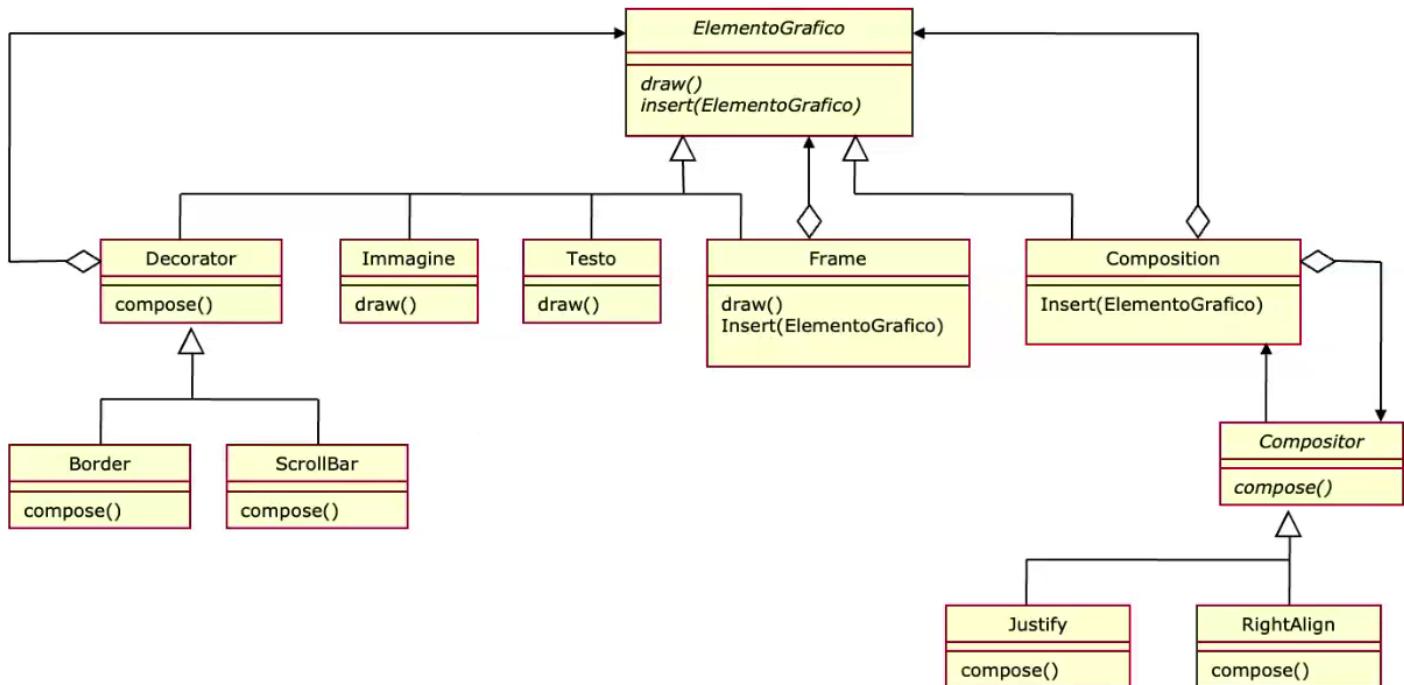
Rappresentazione degli algoritmi di formattazione: il pattern Strategy



Rappresentazione delle proprietà grafiche: il pattern Decorator



Mettendo tutto insieme:



SINGOLI ESEMPI DI APPLICAZIONE DEI DESIGN PATTERNS

1. ADAPTER - ESEMPIO

Scenario: consideriamo un'applicazione per lavorare con oggetti geometrici.

Questi oggetti saranno gestiti dall'applicazione tramite un'interfaccia particolare (Polygon), che offre un insieme di metodi che gli oggetti grafici devono implementare.

Si ha a disposizione una classe (Rectangle) che si vorrebbe riutilizzare, ma tale classe ha un'interfaccia diversa che non si vuole modificare.

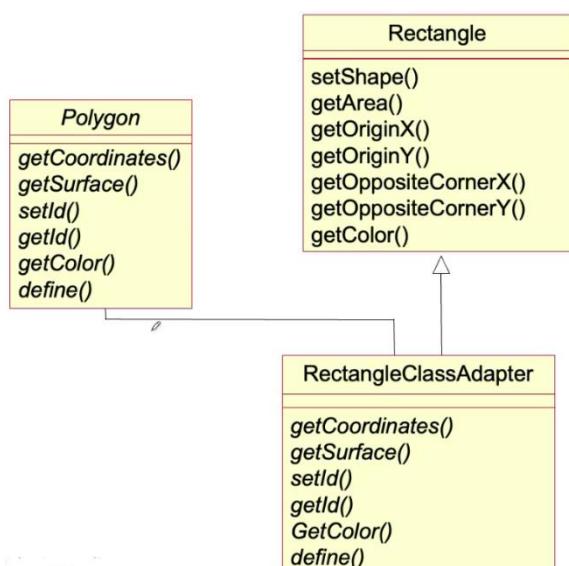
Polygon	Rectangle
<code>getCoordinates() getSurface() setId() getId() getColor() define()</code>	<code>setShape() getArea() getOriginX() getOriginY() getOppositeCornerX() getOppositeCornerY() getColor()</code>

```
public class Rectangle {  
    private float x0, y0;  
    private float height, width;  
    private String color;  
  
    public void setShape(float x, float y, float a, float l, String c) {  
        x0 = x;  
        y0 = y;  
        height = a;  
        width = l;  
        color = c;  
    }  
  
    public float getArea() {  
        return x0 * y0;  
    }  
  
    public float getOriginX() {  
        return x0;  
    }  
  
    public float getOriginY() {  
        return y0;  
    }  
  
    public float getOppositeCornerX() {  
        return x0 + height;  
    }  
  
    public float getOppositeCornerY() {  
        return y0 + width;  
    }  
  
    public String getColor() {  
        return color;  
    }  
}
```

```
public interface Polygon {  
    public void define(float x0, float y0, float x1, float y1, String color);  
  
    public float[] getCoordinates();  
  
    public float getSurface();  
  
    public void setId(String id);  
  
    public String getId();  
  
    public String getColor();  
}
```

Caso1-class adapter:

La costruzione del Class Adapter per il Rectangle è basato sulla sua estensione. Per questo obiettivo viene creata la classe RectangleClassAdapter che estende Rectangle e implementa l'interfaccia Polygon.



```

public class RectangleClassAdapter extends Rectangle implements Polygon {
    private String name = "NO NAME";

    public void define(float x0, float y0, float x1, float y1, String color) {
        float a = x1 - x0;
        float l = y1 - y0;
        setShape(x0, y0, a, l, color);
    }

    public float getSurface() {
        return getArea();
    }

    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = getOriginX();
        aux[1] = getOriginY();
        aux[2] = getOppositeCornerX();
        aux[3] = getOppositeCornerY();
        return aux;
    }

    public void setId(String id) {
        name = id;
    }

    public String getId() {
        return name;
    }
}
  
```

Ereditato da Rectangle

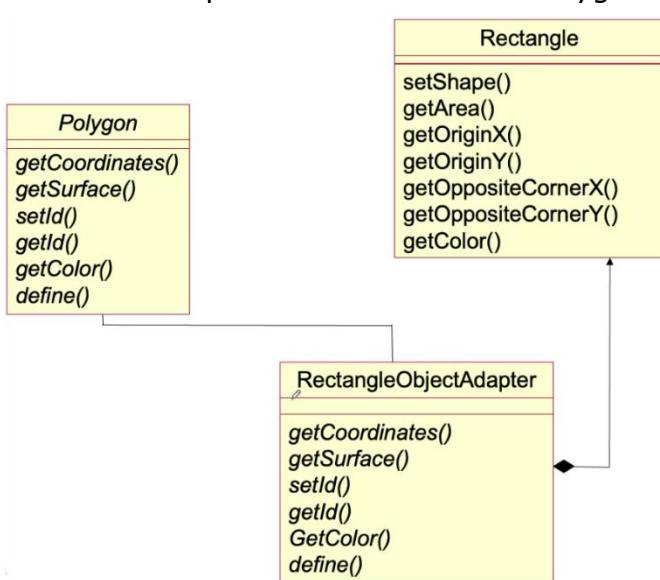
Un possibile client:

```

public class ClassAdapterExample {
    public static void main(String[] arg) {
        Polygon block = new RectangleClassAdapter();
        block.setId("Demo");
        block.define(3, 4, 10, 20, "RED");
        System.out.println("The area of " + block.getId() + " is "
            + block.getSurface() + ", and it's " + block.getColor());
    }
}
  
```

Caso2-Object Adapter:

La costruzione dell'Object Adapter per il Rectangle, si basa nella creazione di una nuova classe (RectangleObjectAdapter) che avrà al suo interno un'oggetto della classe Rectangle, e che implementa l'interfaccia Polygon.



```

public class RectangleObjectAdapter implements Polygon {
    Rectangle adaptee;
    private String name = "NO NAME";

    public RectangleObjectAdapter() {
        adaptee = new Rectangle();
    }

    public void define(float x0, float y0, float x1, float y1, String col) {
        float a = x1 - x0;
        float l = y1 - y0;
        adaptee.setShape(x0, y0, a, l, col);
    }

    public float getSurface() {
        return adaptee.getArea();
    }

    public float[] getCoordinates() {
        float aux[] = new float[4];
        aux[0] = adaptee.getOriginX();
        aux[1] = adaptee.getOriginY();
        aux[2] = adaptee.getOppositeCornerX();
        aux[3] = adaptee.getOppositeCornerY();
        return aux;
    }

    public void setId(String id) {
        name = id;
    }

    public String getId() {
        return name;
    }

    public String getColor() {
        return adaptee.getColor();
    }
}
  
```

Istanzia un oggetto Rectangle
"adaptee" e invoca i suoi metodi:

un possibile client:

```
public class ObjectAdapterExample {  
    public static void main(String[] args) {  
        Polygon block = new RectangleObjectAdapter();  
        block.setId("Demo");  
        block.define(3, 4, 10, 20, "RED");  
        System.out.println("The area of " + block.getId() + " is "  
                           + block.getSurface() + ", and it's " + block.getColor());  
    }  
}
```

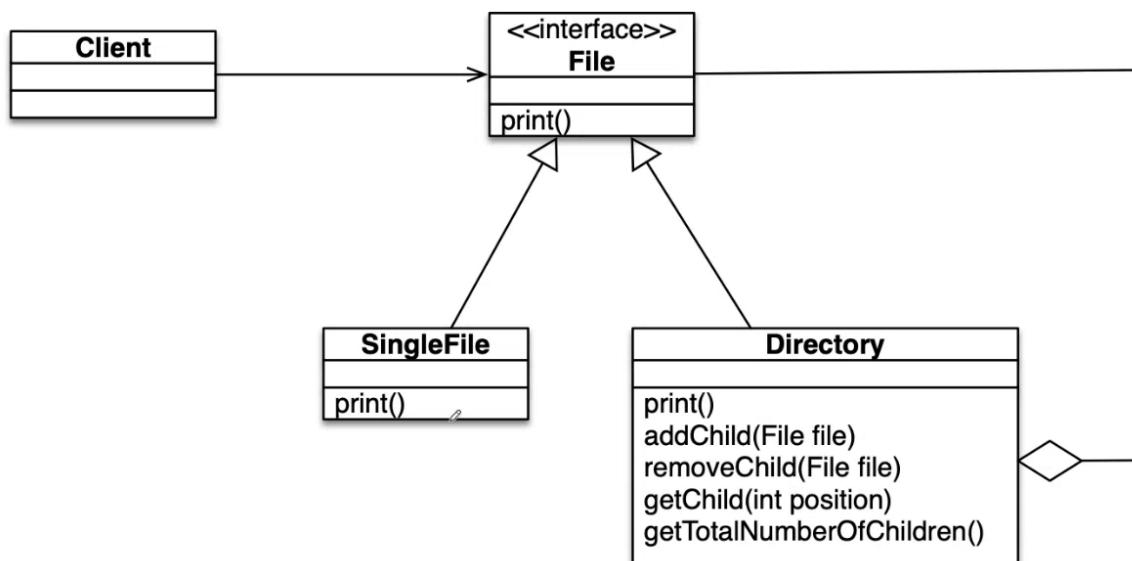
Quindi per quanto riguarda questo esempio i partecipanti sono:

- l'interfaccia Polygon (interfaccia che il client utilizza)
- la classe Rectangle da adattare (implementa una interfaccia che deve essere adattata)
- l'Adapter ossia le classi RectangleClassAdapter e la RectangleObjectAdapter (adatta l'interfaccia dell'Adaptee verso la Target Interface)

2.COMPOSITE - ESEMPIO

Scenario: pensiamo al FileSystem che presenta una struttura ad albero e che può essere composto da elementi semplici (files) e da contenitori (cartelle). L'obiettivo è quello di permettere al Client di accedere e navigare il File System senza conoscere la natura degli elementi che lo compongono in modo da trattare tutti gli elementi nello stesso modo.

Per fare questo il Client userà la stessa interfaccia per l'accesso mentre l'implementazione nasconderà la gestione degli oggetti a seconda della loro reale natura.



Creiamo l'interfaccia di interrogazione per l'accesso a file e directory

```
public interface File {  
    void print();  
}
```

Implementiamo la classe che gestisce i File:

```
public class SingleFile implements File {  
    private final String fileName;  
  
    public SingleFile(String fileName) {  
        this.fileName = fileName;  
    }  
  
    @Override  
    public void print() {  
        System.out.println(fileName);  
    }  
}
```

```

public class Directory implements File {
    private final String directoryName;
    private final List<File> children;

    public Directory(String directoryName, List<File> children) {
        this.directoryName = directoryName;
        this.children = new ArrayList<>(children);
    }

    public void addChild(File file) {
        this.children.add(file);
    }

    public void removeChild(File file) {
        this.children.remove(file);
    }

    public File getChild(int position) {
        if (position < 0 || position >= children.size()) {
            throw new RuntimeException("Invalid position " + position);
        }
        return children.get(position);
    }

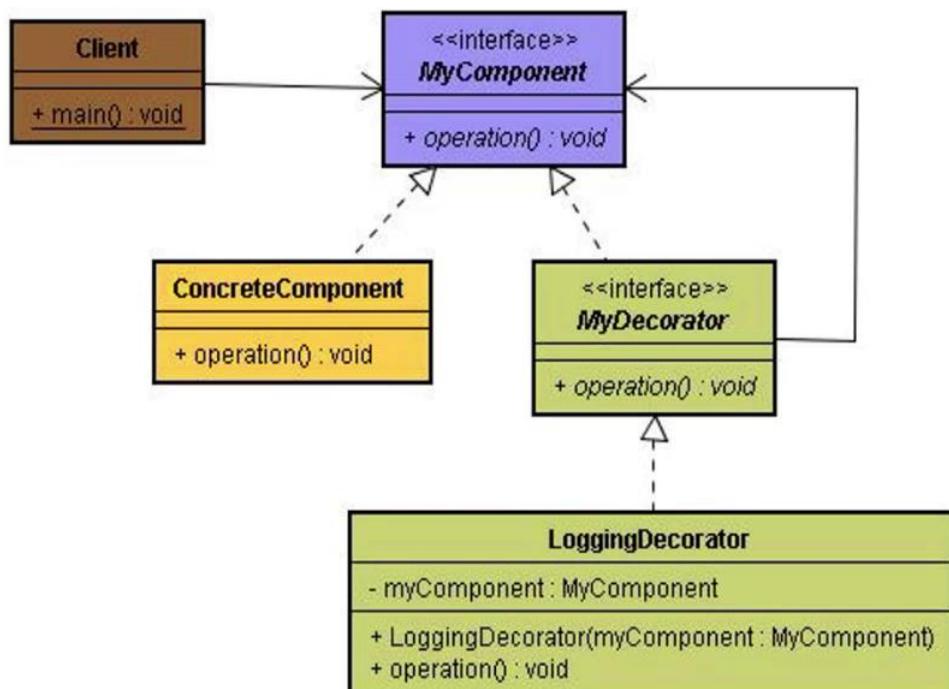
    private int getTotalNumberOfChildren() {
        return children.size();
    }

    @Override
    public void print() {
        System.out.println("Printing the contents of the directory - " +
directoryName);
        children.forEach(File::print);
        System.out.println("Done printing the contents of the directory - " +
directoryName);
    }
}

```

3.DECORATOR - ESEMPIO

Scenario: abbiamo l'esigenza di monitorare l'invocazione di un metodo ma non abbiamo la possibilità di modificare il codice. Utilizziamo il pattern Decorator per esigenze di debug pertanto "wrappiamo" un metodo con delle semplici istruzioni print-screen.



Dichiariamo l'interfaccia Component che dichiara il metodo interessato

```
package patterns.decorator;
public interface MyComponent {
    public void operation();
}
```

Implementiamo il metodo dichiarato nell'interfaccia MyComponent creando la classe ConcreteComponent

```
package patterns.decorator;
public class ConcreteComponent implements MyComponent {
    public void operation(){
        System.out.println("Hello World");
    }
}
```

Definiamo l'interfaccia MyDecorator che si occupa di ereditare il metodo interessato da MyComponent e di interporsi con le classi di decorazione concrete

```
package patterns.decorator;
interface MyDecorator extends MyComponent {
}
```

Creiamo la classe LoggingDecorator che implementa l'interfaccia MyDecorator ed aggiunge le informazioni di debug prima e dopo l'esecuzione del metodo interessato

```
public class LoggingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public LoggingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }

    public void operation() {
        System.out.println("First Logging");
        myComponent.operation();
        System.out.println("Last Logging");
    }
}
```

La classe Client invoca la classe concrera LogginDecorator passando al costruttore il componente concreto, successivamente invoca il metodo operation()

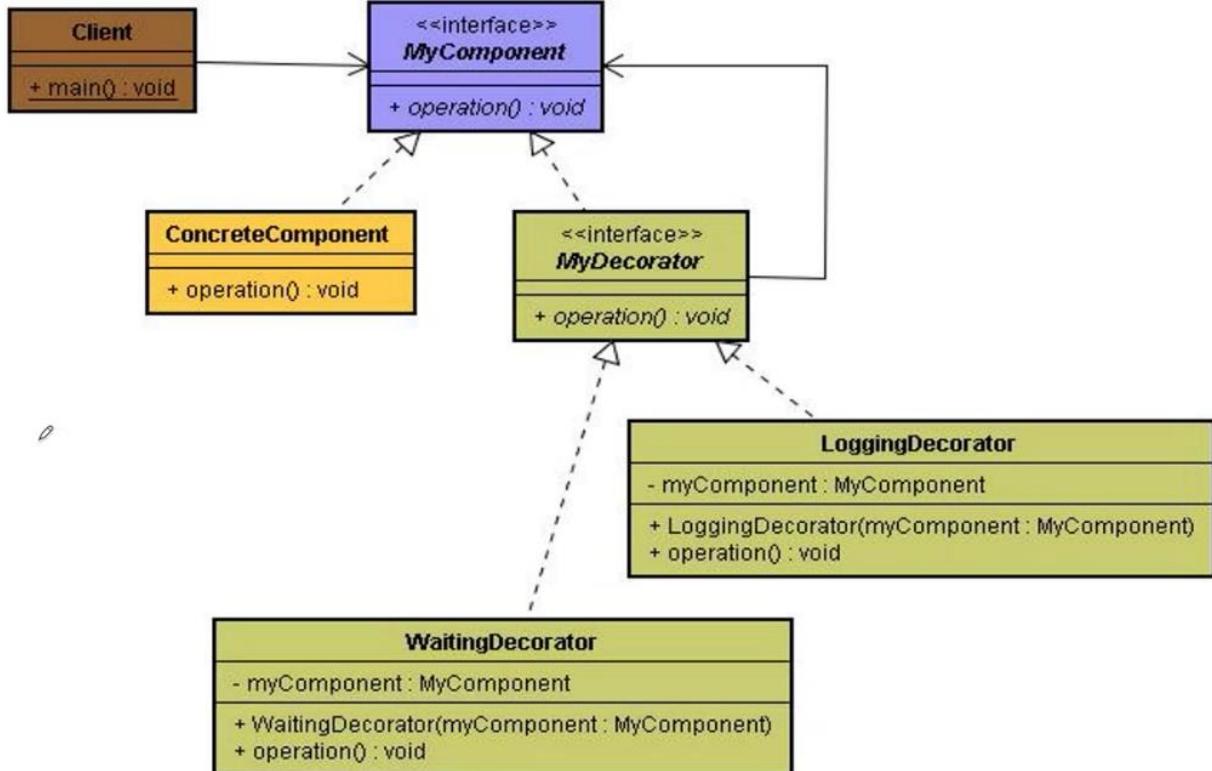
```
class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new ConcreteComponent());
        myComponent.operation();
    }
}
```

Quale sarà l'output? Sarà:

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Hello World
Last Logging
```

Partendo dall'esempio decorator, possiamo creare una moltitudine di classi concrete Decorator che aggiungono nuove funzionalità.
 Per esempio possiamo creare una classe WaitingDecorator che preveda una pausa durante l'esecuzione.

Vediamo come diventa il Class Diagram a seguito dell'inserimento di questa nuova classe.



```

public class WaitingDecorator implements MyDecorator {

    MyComponent myComponent = null;

    public WaitingDecorator(MyComponent myComponent){
        this.myComponent = myComponent;
    }

    public void operation() {
        try {
            System.out.println("Waiting...");
            Thread.sleep( 1000 );
        }
        catch (Exception e) {}

        myComponent.operation();
    }
}
  
```

Il client invoca in modo annidato i Decorator tramite il loro costruttore come segue:

```

class Client {
    public static void main(String[] args) {
        MyComponent myComponent = new LoggingDecorator(new WaitingDecorator(new
ConcreteComponent()));
        myComponent.operation();
    }
}
  
```

Output:

```
$JAVA_HOME/bin/java patterns.decorator.Cliente
First Logging
Waiting...
Hello World
Last Logging
```

4.FACTORY METHOD - ESEMPIO

Scenario: supponiamo di avere una applicazione che legge dei dati da un file di testo contenente le informazioni relative a delle rilevazioni di letture di contatori per acqua e gas. Nel nostro codice abbiamo una classe dedicata a questo scopo, che legge i vari formati dei file, la classe AcquisizioneLettura.

```
public class AcquisizioneLettura {
    public AcquisizioneLettura() {
    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia

        if (dataType.equals("gas")) {
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")) {
            fileLetturaReader = new H2OLetturaReader(fileName);
        }

        while (fileLetturaReader.hasNextLettura()) {
            lettura = fileLetturaReader.getNextLettura();

            if (lettura.verifica()) {
                lettura.calcolaconsumo();
                lettura.registra();
            } else {
                lettura.scarta();
            }
        }
    }
}

public class AcquisizioneLettura {
    public AcquisizioneLettura() {
    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia

        if (dataType.equals('gas')) {
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (dataType.equals('H2O')) {
            fileLetturaReader = new H2OLetturaReader(fileName);
        }

        if(dataType.equals('EE')){
            fileLetturaReader = new EELetturaReader(fileName);
        }

        while (fileLetturaReader.hasNextLettura()) {
            lettura = fileLetturaReader.getNextLettura();

            if (lettura.verifica()) {
                lettura.calcolaconsumo();
                lettura.registra();
            } else {
                lettura.scarta();
            }
        }
    }
}
```

Gas letturaReader e H2OLetturaReader sono classi specializzate nella lettura di file in formato testo che implementano l'interfaccia FileLetturaReader. Questa implementazione è corretta ma poco flessibile.

Succede però che il nostro cliente inizia a vendere anche energia elettrica e deve di conseguenza acquisire anche i file con le relative letture. E' quindi necessario gestire un tipo aggiuntivo di file.

È stato necessario aprire la classe e introdurre la modifica.

Operazione non pratica, soprattutto se prevediamo di ripeterla in futuro.
 Dovremmo separare il codice soggetto a modifiche da quello sempre uguale, come fare?
 Ad esempio incapsulando la creazione di FileLetturaReader all'interno di una nuova classe FileReaderFactory:

```

public class ReaderFactory {
    private FileLetturaReader fileLetturaReader;
    public ReaderFactory() {
    }

    public FileLetturaReader getFileLetturaReader(String fileName, String dataType) {
        if (dataType.equals("gas")) {
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")) {
            fileLetturaReader = new H2OLetturaReader(fileName);
        }
        if (dataType.equals("EE")) {
            fileLetturaReader = new EELetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

public class AcquisizioneLetture {
    private ReaderFactory factory;
    public AcquisizioneLetture(ReaderFactory factory) {
        this.factory = factory;
    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia
        // meglio, no?
        fileLetturaReader = factory.getFileLetturaReader(fileName,dataType);
        //
        while (fileLetturaReader.hasNextLettura()) {
            lettura = fileLetturaReader.getNextLettura();

            if (lettura.verifica()) {
                lettura.calcolaconsumo();
                lettura.registra();
            } else {
                lettura.scarta();
            }
        }
    }
}

```

Abbiamo così ottenuto la chiusura di AcquisizioneLetture alle modifiche e la possibilità di riutilizzare ReaderFactory anche altrove, isolando in essa le modifiche.



Poi accade che acquisiamo due importanti clienti, che vendono acqua e gas ed utilizzano un proprio formato XML di interscambio dati.
 Adeguiamo il nostro codice alle nuove esigenze, scrivendo due nuove factory class ad hoc per loro, Cliente1ReaderFactory e Cliente2ReaderFactory che derivano dalla nostra ReaderFactory.

```

public class Cliente1ReaderFactory extends ReaderFactory{
    public Cliente1ReaderFactory() {
    }

    public FileLetturaReader getFileLetturaReader(String fileName, String dataType) {
        if (dataType.equals("gas")) {
            fileLetturaReader = new Cliente1GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")) {
            fileLetturaReader = new Cliente1H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

public class Cliente2ReaderFactory extends ReaderFactory {
    public Cliente2ReaderFactory() {
    }

    public FileLetturaReader getFileLetturaReader(String fileName,
                                                String dataType) {
        if (dataType.equals("gas")) {
            fileLetturaReader = new Cliente2GasLetturaReader(fileName);
        }
        if (dataType.equals("H2O")) {
            fileLetturaReader = new Cliente2H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

```

Corretto, ma notiamo due cose:

- AcquisizioneLetture ha sempre bisogno che gli venga passato un factory per funzionare, tanto che viene passato nel costruttore.
- Le operazioni fatte sulla lettura, come abbiamo visto all'inizio, sono sempre quelle. Sarebbe utile quindi portare il factory direttamente dentro AcquisizioneLetture per rendere la classe autosufficiente, ma senza perdere la flessibilità ottenuta fino ad ora.

Torniamo alla prima versione di AcquisizioneLetture, ma questa volta incapsuliamo la creazione delle classi FileLetturaReader all'interno di un metodo astratto e rendiamo quindi astratta tutta la classe.

Da questa deriviamo le varie versioni per i vari clienti, implementando in ognuna di esse il metodo che incapsula la creazione delle classi FileLetturaReader.

Ora abbiamo una AcquisizioneLetture per ogni cliente, ognuna contenente la propria logica di lettura dei file.

```

public abstract class AcquisizioneLetture {
    public AcquisizioneLetture() {

    }

    public void parseFile(String fileName, String dataType) {
        ArrayList letturaArray = new ArrayList();
        FileLetturaReader fileLetturaReader; //questa è una interfaccia
        Lettura lettura; //questa è una interfaccia

        fileLetturaReader = getFileLetturaReader(fileName, dataType);
        //
        while (fileLetturaReader.hasNextLettura()) {
            lettura = fileLetturaReader.getNextLettura();

            if (lettura.verifica()) {
                lettura.calcolaconsumo();
                lettura.registra();
            } else {
                lettura.scarta();
            }
        }
    }

    protected abstract FileLetturaReader getFileLetturaReader(String fileName, String fileType);
}

```

La classe base non conosce il FileLetturaReader su cui itera e da cui ricava le letture, perché questo dipende dalle classi derivate. Abbiamo conservato disaccoppiamento e flessibilità.

```

public class AcquisizioneLettureConcreta extends AcquisizioneLetture {
    public AcquisizioneLettureConcreta() {
    }

    protected FileLetturaReader getFileLetturaReader(String fileName,
                                                    String fileType) {
        FileLetturaReader fileLetturaReader;
        if (fileType.equals("gas")) {
            fileLetturaReader = new GasLetturaReader(fileName);
        }
        if (fileType.equals("H2O")) {
            fileLetturaReader = new H2OLetturaReader(fileName);
        }
        if (fileType.equals("EE")) {
            fileLetturaReader = new EELetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

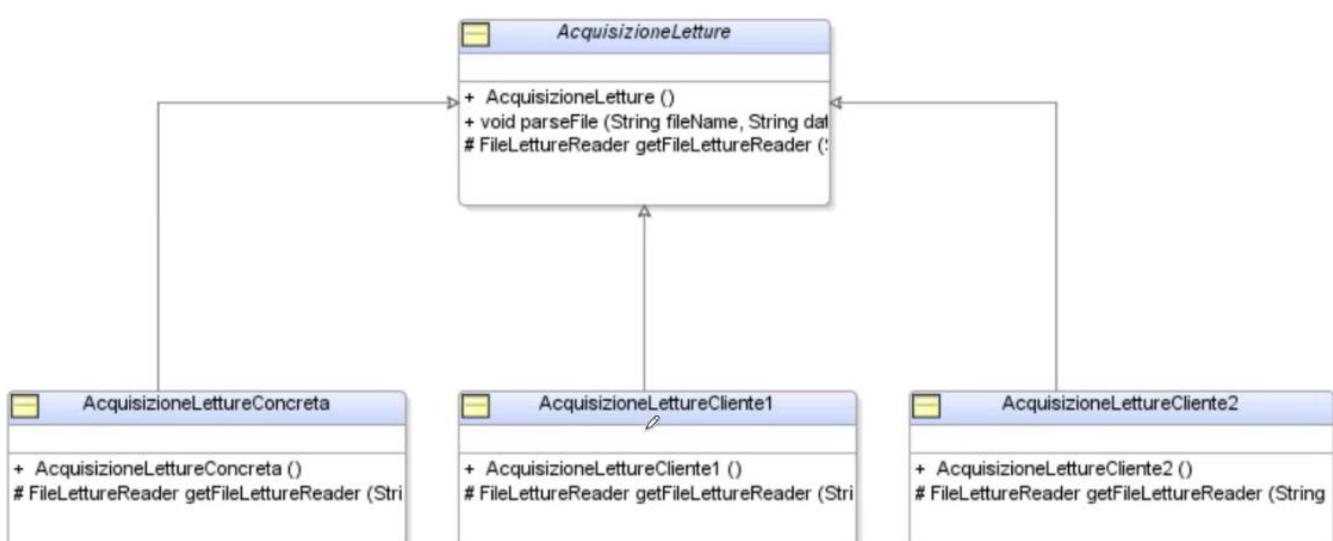
public class AcquisizioneLettureCliente1 extends AcquisizioneLetture {
    public AcquisizioneLettureCliente1() {
    }

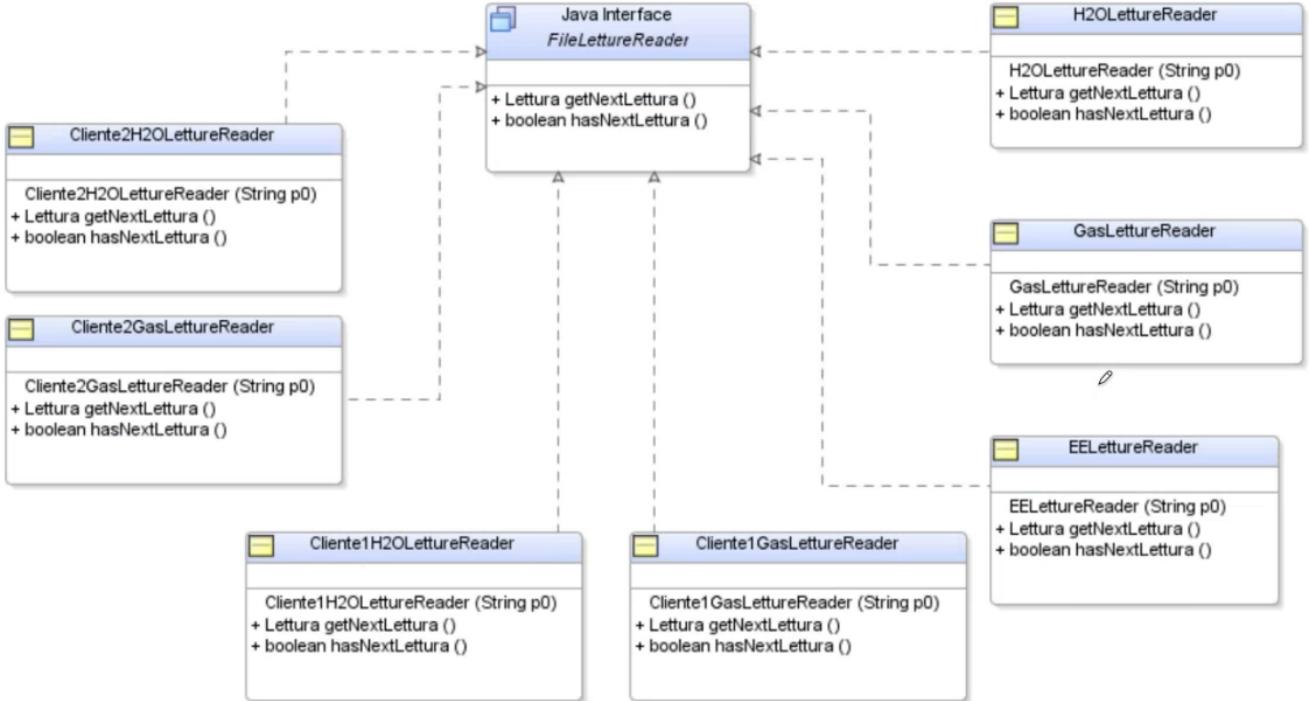
    protected FileLetturaReader getFileLetturaReader(String fileName,
                                                    String fileType) {
        FileLetturaReader fileLetturaReader;
        if (fileType.equals("gas")) {
            fileLetturaReader = new Cliente1GasLetturaReader(fileName);
        }
        if (fileType.equals("H2O")) {
            fileLetturaReader = new Cliente1H2OLetturaReader(fileName);
        }
        return fileLetturaReader;
    }
}

```

Ricordiamo il diagramma UML del Factory Method, abbiamo costruito:

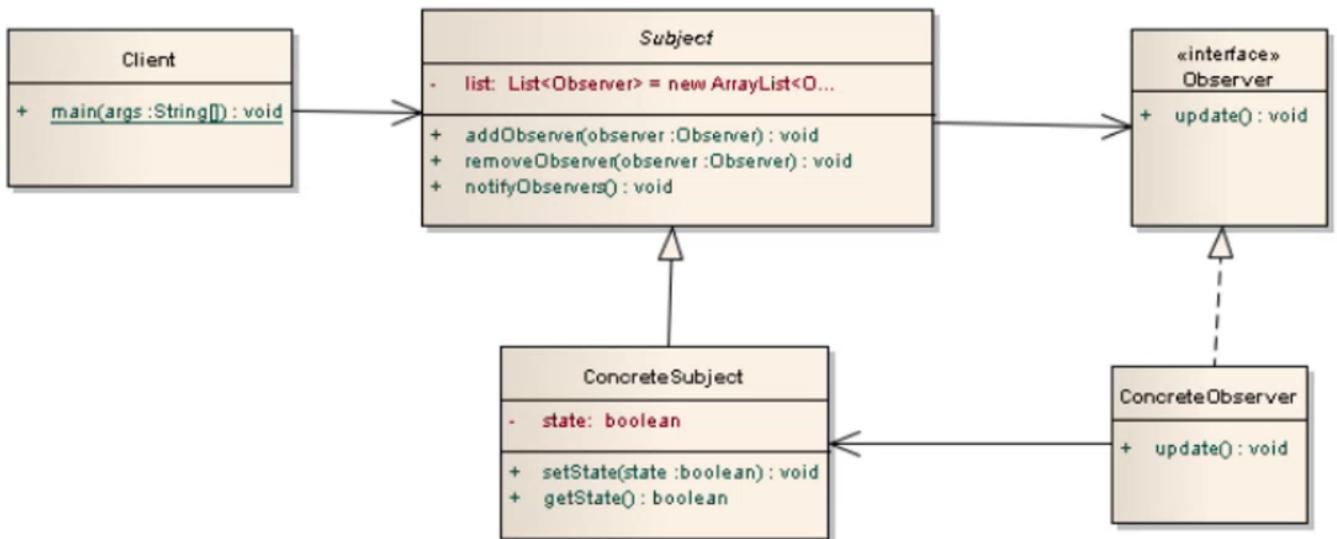
Creator	AcquisizioneLetture	ConcreteProduct	GasLetturaReader
Product	FileLetturaReader		H2OLetturaReader
ConcreteCreator	AcquisizioneLettureConcreta AcquisizioneLettureCliente1 AcquisizioneLettureCliente2		EELetturaReader Cliente1GasLetturaReader Cliente1H2OLetturaReader Cliente2GasLetturaReader Cliente2H2OLetturaReader





5.OBSERVER - ESEMPIO

Scenario: due osservatori sono interessati al cambio di stato di un soggetto osservato. Successivamente uno di essi cancella la sottoscrizione e non riceve più notifiche. In questo caso dobbiamo creare il Subject e l'Observer e le classi concrete ConcreteSubject e ConcreteObserver.



```

public interface Observer {
    public void update();
}

```

L'interfaccia dell'Observer definisce il metodo che dovrà essere implementato dagli osservatori. Pertanto quando interverranno delle modifiche al soggetto osservato, verrà invocato il metodo update() di tutti gli osservatori.

Il Subject include una lista degli osservatori che si registrano presso il soggetto osservato tramite i metodi addObserver() e si cancellano tramite il metodo removeObserver(). Mentre invece il metodo notifyObservers() viene invocato dalla classe concreta ConcreteSubject quando interviene un cambio di stato.

```

public abstract class Subject {

    private List<Observer> list = new ArrayList<Observer>();

    public void addObserver(Observer observer) {
        list.add( observer );
    }

    public void removeObserver(Observer observer) {
        list.remove( observer );
    }

    public void notifyObservers() {
        for(Observer observer: list) {
            observer.update();
        }
    }
}

public class ConcreteObserver implements Observer {

    @Override
    public void update() {
        System.out.println("Sono " + this + ": il Subject e' stato modificato!");
    }
}

```

Il ConcreteObserver implementa il metodo update() per definire l'azione da intraprendere quando interviene un cambio di stato del Subject.

LEZ34 - 17/04/2024

PRELIMINARY DESIGN MEASURES (MISURE PRELIMINARI DI PROGETTAZIONE)

Inter-modular measures (misure intermodulari), che tengono conto delle dipendenze tra i moduli, in base all'architettura di sistema sviluppata in fase di progettazione (design phase)

- Le misure degli attributi dei singoli moduli sono chiamate intra-modular measures (utilizzate durante la progettazione dettagliata e l'implementazione)

Un modulo è una sequenza contigua di istruzioni delimitata da alcuni elementi, che ha un certo identificatore (pensiamo al modulo come una parte di software compilata separatamente che può essere invocata e far parte di una libreria, ad esempio concetto di funzione o concetto di classe).

Durante la fase di progettazione preliminare, le decisioni relative all'architettura di sistema hanno un impatto importante e significativo sulla qualità del software risultante. Alcuni attributi di qualità indirizzati da quali scelte vengono fatte a livello di progettazione preliminare sono facilità di implementazione, affidabilità e manutenibilità.

Le misure preliminari di progettazione hanno il potenziale di fornire il feedback necessario sulle caratteristiche dell'architettura software in fase di sviluppo.

PRE-DESIGN E RELAZIONE CON IL CODICE

La relazione tra la progettazione preliminare e il codice (implementato dalla progettazione) include relazioni uno-a-uno tra:

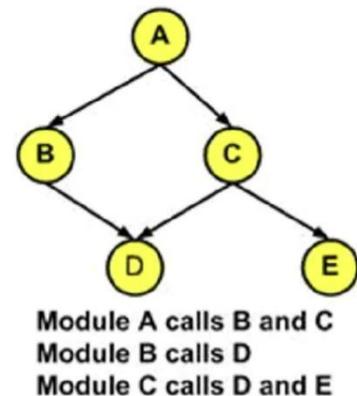
- moduli indicati nel design e moduli nel codice
- connessioni intermodulari indicate nel progetto e riferimenti intermodulari nel codice
- interfacce dati intermodulari indicate nel progetto e dati condivisi intermodulari nel codice

ARCHITETTURA DEI MODULI (SCHEMA STRUTTURALE)

L'architettura dei moduli di un sistema software può essere rappresentata da un grafo, $S=\{N, R\}$

Ogni nodo n nell'insieme dei nodi (N) corrisponde a un modulo.

Ogni spigolo r nell'insieme delle relazioni (R) indica una relazione (ad esempio, chiamata di procedura, flusso di dati, ecc.) tra due sottosistemi.



MODULARITY (MODULARITÀ)

Definizione: la misura in cui il software è composto da componenti discrete in modo tale che una modifica a un componente abbia un impatto minimo sugli altri componenti.

Un'elevata modularità è auspicabile perché si ritiene che i programmi con bassa modularità siano più soggetti a errori, meno manutenibili, meno riutilizzabili, ecc.

La modularità è un attributo qualitativo che può essere misurato in termini dei seguenti attributi secondari: cohesion(coesione), coupling(accoppiamento), morphology(morfologia), information flow(flusso di informazioni).

Nel dettaglio degli attributi secondari utili a misurare la modularità:

- Coesione: il grado con cui ogni modulo individualmente realizza una funzione
- Accoppiamento: grado di interdipendenza tra moduli
- Morfologia: l'aspetto morfologico della architettura dei moduli, si occupa di andare a misurare la forma della architettura dei moduli cercando di capire quale è la migliore per quanto riguarda la modularità
- Flusso di informazioni: le interconnessioni tra moduli non solo dal punto di vista di scambio dati ma anche dal punto di vista del flusso di controllo, quindi quando parliamo di information flow facciamo riferimento sia al flusso di controllo(quindi la dipendenza tra moduli) che al flusso dei dati(quindi ai dati che si scambiano i moduli). Si misura tramite tecniche che permettono di identificare concetti quali fan-in e fan-out che equivalgono a quanto un modulo riceve in ingresso e quanto un modulo restituisce in uscita.

MORFOLOGIA

La morfologia si riferisce alla forma complessiva dell'architettura del sistema software.

Si caratterizza per:

- Size (Dimensione): numero di nodi e spigoli
- Depth (Profondità): percorso più lungo dalla radice a un nodo foglia
- Width (Larghezza): numero massimo di nodi a qualsiasi livello
- Edge-to-node ratio (Rapporto edge-to-node): misura della densità di connettività

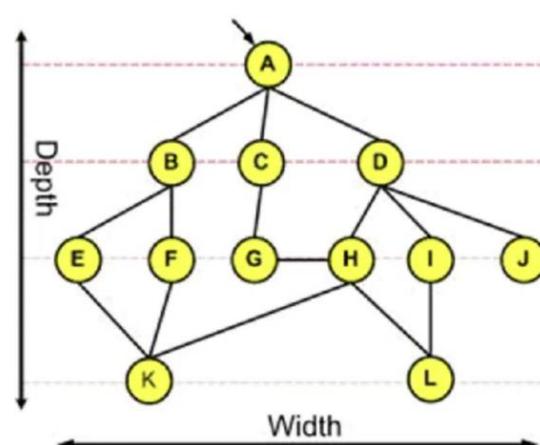
Esempio:

Size = 12 nodi, 15 archi

Depth = 4

Width = 6

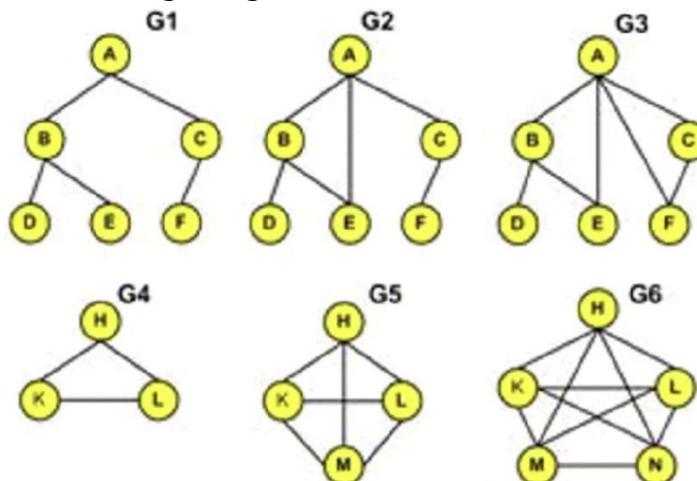
e/n = 1.25



TREE IMPURITY (IMPURITA' DELL'ALBERO)

L'impurità dell'albero $m(G)$ misura quanto il grafo G è diverso da un albero (ipotesi: un buon design dovrebbe essere il più possibile "ad albero").

Il più piccolo $m(G)$ denota il design migliore.



La misura $m(G)$ di impurità dell'albero viene definita come:

$$m(G) = \frac{\text{number of edges more than spanning tree}}{\text{maximum number of edges more than spanning tree}}$$

$$m(G) = \frac{2(e - n + 1)}{(n - 1)(n - 2)}$$

Esempio:

$$\begin{aligned} m(G_1) &= 0 & m(G_2) &= 0.1 & m(G_3) &= 0.2 \\ m(G_4) &= 1 & m(G_5) &= 1 & m(G_6) &= 1 \end{aligned}$$

INTERNAL REUSE (Y IN AND WINCHESTER MEASURE)

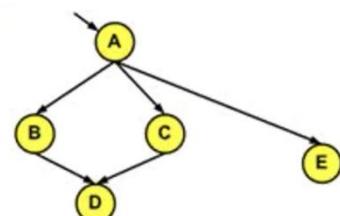
Elemento che va a misurare il concetto di morfologia.

Il riutilizzo interno è una misura che indica l'entità del riutilizzo dei moduli all'interno dello stesso prodotto (diverso dal riutilizzo esterno).

$$r(G) = e - n + 1$$

Il minor $r(G)$ significa meno riutilizzo.

Criticità: non è in grado di tenere conto delle chiamate ripetute e non può tenere conto delle dimensioni del modulo riutilizzato.



Module D is reused by
modules B and C

Example:

$$\begin{aligned} r(G1) &= 0 & r(G2) &= 1 \\ r(G3) &= 2 & r(G4) &= 1 \\ r(G5) &= 3 & r(G6) &= 6 \end{aligned}$$

INFORMATION FLOW

Le misure di information flow presuppongono che la complessità di un modulo dipenda da 2 fattori:

- la complessità del codice del modulo

- la complessità delle interfacce del modulo, cioè le sue connessioni al suo ambiente

Il livello totale del flusso di informazioni attraverso un sistema, in cui i moduli sono visti come componenti atomici, è un attributo inter-modulare.

Il livello totale del flusso di informazioni tra un singolo modulo e il resto del sistema è un attributo intra-modulare.

Le misure del flusso di informazioni sono conteggi basati sulle interconnessioni che un modulo ha con altri moduli in un sistema, ovvero il fan-in e il fan-out di un modulo.

Le misure del flusso informativo si basano su:

- Flusso locale di informazioni:
 - o diretto: un modulo chiama un altro modulo
 - o indiretto: flusso derivante dai valori restituiti
- Flusso globale di informazioni: informazioni che vengono passate tra i moduli tramite una struttura dati globale

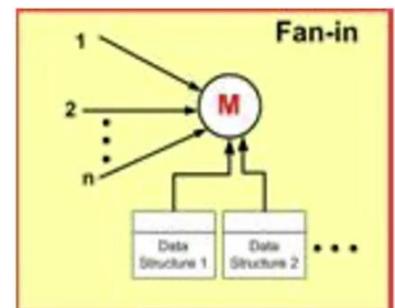
Misure del flusso di informazioni:

- può essere utilizzato per identificare le parti critiche di un sistema software
- sono pensati per identificare i punti di stress nel sistema
- fornire una panoramica dei potenziali problemi di progettazione

FAN-IN E FAN-OUT

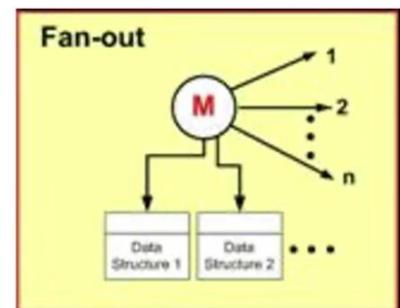
Il fan-in di un modulo M è il numero di flussi locali (diretti + indiretti) che terminano con M (quindi che terminano ad un certo modulo) più il numero di flussi globali (strutture dati) da cui il modulo M ricava dati.

Il fan-out di un modulo M è il numero di flussi locali (diretti + indiretti) a partire da M più il numero di flussi globali (strutture dati) aggiornato da M.



Un elevato fan-out di un modulo indica che influenza/controlla molti altri moduli.

Un elevato fan-in di un modulo indica che è influenzato/controllato da molti altri moduli.



Un modulo con fan-in e fan-out elevati è normalmente al centro del sistema.

Un modulo con fan-in e fan-out bassi si trova normalmente alla periferia del sistema.

Un elevato fan-in e fan-out indica un modulo spesso complesso che può essere soggetto a errori perché:

- I moduli svolgono più di una funzione. Se la struttura del progetto deve essere modificata, l'aspetto da modificare non si troverà in un unico posto ma in più punti e tutto ciò che ha a che fare con esso non è all'interno di un particolare modulo ma con più moduli interconnessi (e quindi con alti fan-in e fan-out). Pertanto, molti moduli dovranno essere modificati.
- I moduli non sono adeguatamente raffinati, cioè la struttura del sistema potrebbe mancare di un livello di astrazione (se la struttura è troppo ampia e meno profonda o il contrario, si potrebbe dire che i moduli sono inadeguatamente raffinati).

INFORMATION FLOW MEASURE (HENRY & KAFURA)

Information flow (IF) per il modulo Mi:

$$IF(M_i) = [fan-in(M_i) \times fan-out(M_i)]^2$$

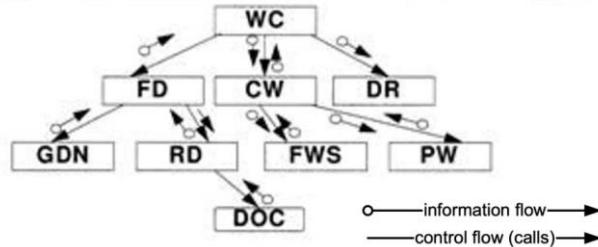
IF per un sistema con n moduli è:

$$IF = \sum_{i=1}^n IF(M_i)$$

La misura originale di Henry-Kafura include sia i flussi di controllo che i flussi di informazioni durante il conteggio fan-in e fan-out.

La variante Shepperd (più utilizzata) include solo i flussi di informazioni.

ESEMPIO DI INFORMATION FLOW MEASURE (SHEPPERD VARIANT)



Module	fan-in	fan-out	$[(\text{fan-in})(\text{fan-out})]^2$	
WC	2	2	16	
FD	2	2	16	
CW	3	3	81	
DR	1	0	0	
GDN	0	1	0	
RD	2	1	4	
FWS	1	1	1	
PW	1	1	1	
				$\text{IF} = 119$
				(Shepperd variant)

MISURE STRUTTURALI (SIA PER DETAILED DESIGN CHE PER IMPLEMENTATION)

La struttura può avere 3 componenti:

- Struttura del flusso di controllo: Sequenza di esecuzione delle istruzioni del programma
- Flusso di dati: tenere traccia dei dati man mano che vengono creati o gestiti dal programma
- Struttura dei dati: l'organizzazione dei dati stessi indipendentemente dal programma

La misurazione strutturale viene utilizzata in strumenti software per:

- deingegnerizzazione (reverse engineering)
- test (path coverage - copertura del percorso)
- ristrutturazione del codice
- analisi del flusso di dati

OBIETTIVO E DOMANDE

Come rappresentare la "struttura" di un programma?

- Control flowgraph (o semplicemente FlowGraph)

Come definire la "complessità" in termini di struttura?

- Cyclomatic complexity

BASIC CONTROL STRUCTURE (STRUTTURE DI CONTROLLO DI BASE)

Le strutture di controllo di base (BCS) sono un insieme di meccanismi essenziali del flusso di controllo utilizzati per costruire la struttura logica del programma.

Tipi di BCS:

- Sequenza: ad esempio, un elenco di istruzioni senza altri BCS coinvolti.
- Selezione: ad esempio, if ... then... else.
- Iterazione: ad esempio, do ... while; repeat... until.

Esistono altri tipi di strutture di controllo, o strutture di controllo avanzate (ACS):

- Chiamata di procedura/funzione/agente
- Ricorsione (autochiamata)
- Interrupt
- Concurrence

LEZ35 – 22/04/2024

FLOWGRAPH (DIAGRAMMA DI FLUSSO)

La struttura del flusso di controllo è solitamente modellata da un diagramma di flusso FG, cioè un grafo diretto (di-graph) $FG = \{N, E\}$.

Ogni nodo n nell'insieme dei nodi (N) corrisponde ad una istruzione del programma

- Nodi di procedura: nodi senza grado 1
- Nodi predicati: nodi con grado diverso da 1
- Nodo iniziale: nodi con grado 0
- Nodi terminali (finali): nodi senza grado 0

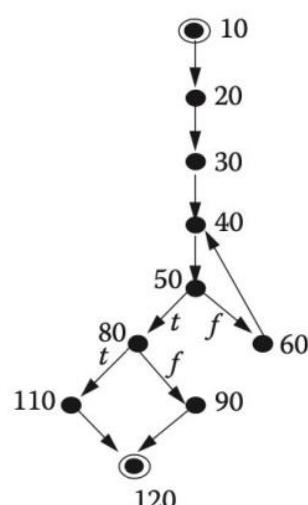
dove out-degree è il numero di archi che lasciano un nodo e in-degree è il numero di archi che entrano in un nodo.

Ogni arco diretto 'e' nell'insieme degli archi 'E' indica il flusso di controllo da una dichiarazione di programma ad un'altra dichiarazione.

Esempio:

```

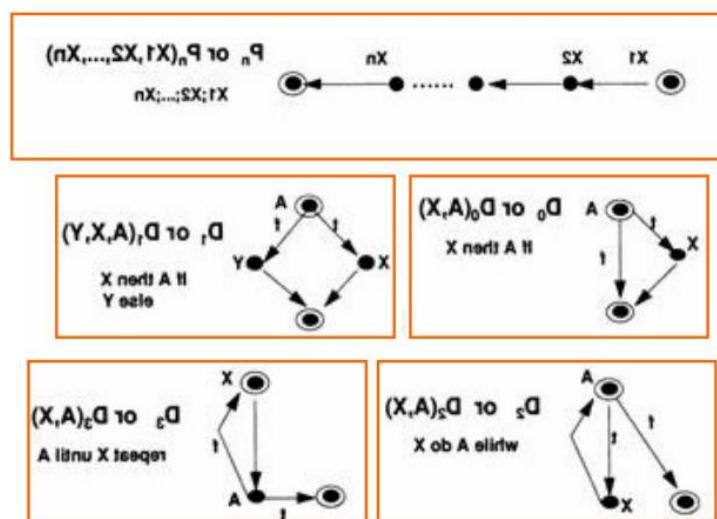
10 INPUT P
20 Div = 2
30 Lim = INT(SQR(P))
40 Flag = P/Div - INT(P/Div)
50 IF Flag = 0 OR Div = Lim THEN 80
60 Div = Div + 1
70 GO TO 40
80 IF Flag <> 0 OR P>4 THEN 110
90 PRINT Div; "Smallest factor of"; P; "."
100 GO TO 120
110 PRINT P; " is prime"
120 END
    
```



COSTRUTTI DI DIAGRAMMA DI FLUSSO

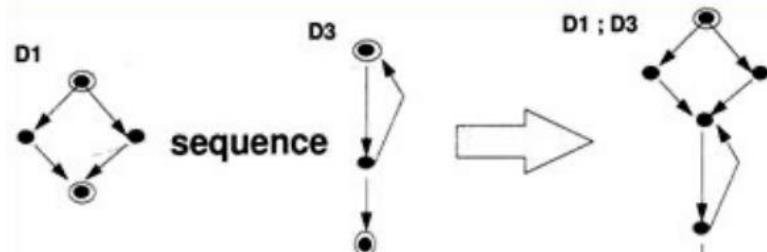
Basic CS	Sequence		
	Selection		
	Iteration		
	Procedure/ function call		
Advanced CS	Recursion		
Interrupt		Concurrence	

MODELLI COMUNI NEI FLOWGRAPGH (DIAGRAMMI DI FLUSSO)



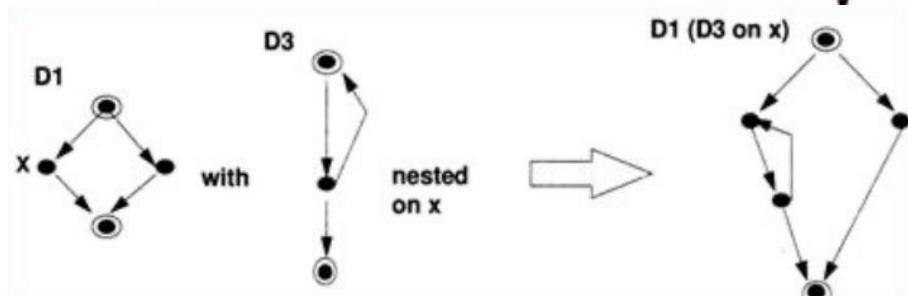
SEQUENCING (SEQUENZIAMENTO)

Siano F_1 e F_2 due flussi. Allora, la sequenza di F_1 e F_2 (mostrata da $F_1; F_2$) è un diagramma di flusso formato dalla fusione del nodo terminale di F_1 con il nodo iniziale di F_2 .



NESTING (NIDIFICAZIONE)

Siano F_1 e F_2 due flussi. Allora, il nesting di F_2 su F_1 a x , mostrato da $F_1(F_2)$, è un diagramma di flusso formato da F_1 sostituendo l'arco da x con l'intero F_2 .



PRIME FLOWGRAPHS (DIAGRAMMI DI FLUSSO PRINCIPALI)

I diagrammi di flusso 'prime' sono grafi di flusso che non possono essere decomposti non banalmente mediante sequenziamento e nesting.

Numeri primi comuni:

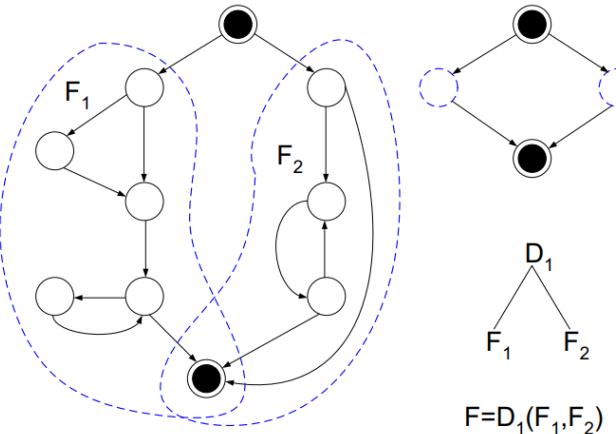
- P1
- D0 D-structures (tipico della programmazione strutturata)
- D1
- D2
- D3

PRIME DECOMPOSITION

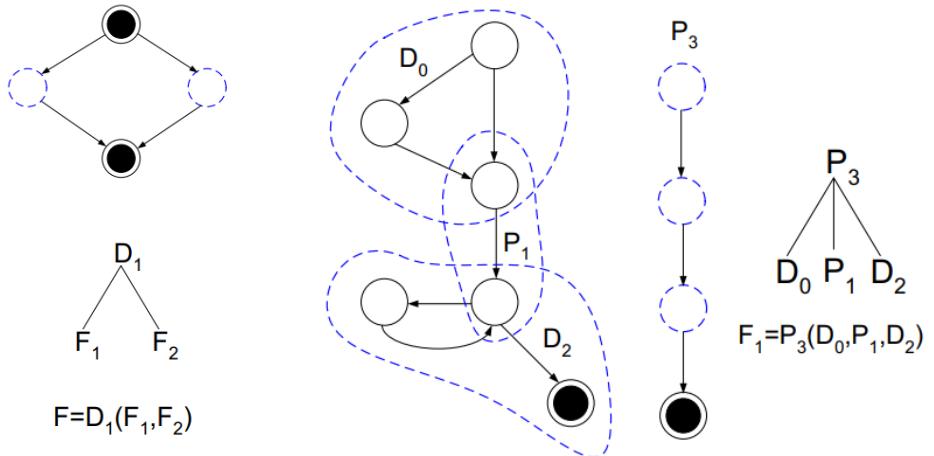
Teorema della decomposizione prima (Fenton-Whitty): ogni diagramma di flusso ha una unica decomposizione in una gerarchia di diagrammi di flusso primitivi, chiamata "albero di decomposizione".

Come effettuare decomposizione a partire da grafo di flusso e ottenere corrispondente albero di decomposizione:

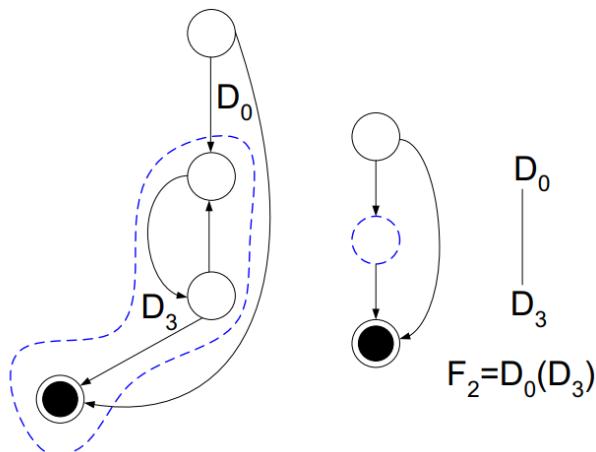
1)



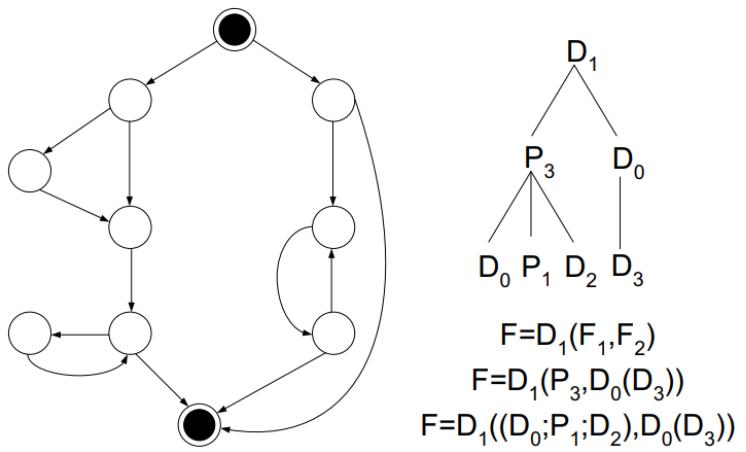
2a)



2b)



3)



FINALE)

METRICHE GERARCHICHE / MISURAZIONE GERARCHICA

La misurazione gerarchica è un modo per definire le misure del diagramma di flusso utilizzando l'albero di decomposizione.

Si basa sull'idea che possiamo misurare un attributo flowgraph con:

1. definire la misura per i prime flowgraphs;
2. descrivere come l'operazione di sequencing (sequenziamento) influisce sull'attributo;
3. descrivere come l'operazione di nesting (annidamento) influisce sull'attributo.

Esempio di misura del diagramma di flusso che può fornire informazioni sulla complessità della struttura del codice sono:

- Depth of nesting (Profondità di nidificazione)
- D-structuredness (Strutturazione di D)

DEPTH OF NESTING

La profondità di nidificazione $n(F)$ per un diagramma di flusso F può essere misurata in termini di:

- Primitive di base (primes):

$$n(P_1) = 0 ; n(P_2) = n(P_3) = \dots = n(P_k) = 1$$

$$n(D_0) = n(D_1) = n(D_2) = n(D_3) = 1$$
- Sequenziamento:

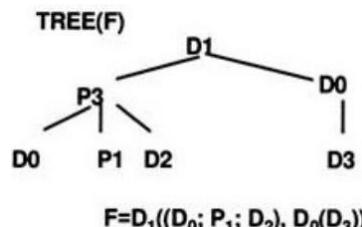
$$n(F_1; F_2; \dots; F_k) = \max\{ n(F_1), n(F_2), \dots, n(F_k) \}$$
- Annidamento:

$$n(F(F_1, F_2, \dots, F_k)) = 1 + \max\{ n(F_1), n(F_2), \dots, n(F_k) \}$$

Esempio:

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$

$$\begin{aligned} n(F) &= n(D_1((D_0; P_1; D_2), D_0(D_3))) = \\ &= 1 + \max\{ n(D_0; P_1; D_2), n(D_0(D_3)) \} = \\ &= 1 + \max\{ \max\{ n(D_0), n(P_1), n(D_2) \}, 1 + n(D_3) \} = \\ &= 1 + \max\{ \max\{ 1, 0, 1 \}, 2 \} = 1 + \max\{ 1, 2 \} = 3 \end{aligned}$$



D-STRUCTUREDNESS (D-STRUTTURA)

La maggior parte delle definizioni popolari di programmazione strutturata asseriscono che un programma è strutturato se può essere composto usando solo un piccolo numero di costrutti ammissibili (ad esempio, sequenza, selezione e iterazione).

Considerando il diagramma di flusso di un programma, possiamo decidere se è strutturato o meno. A tal fine può essere utilizzata la misura di strutturazione D .

La definizione informale di programmazione strutturata può essere formalmente espressa affermando che un programma è strutturato se e solo se è D -strutturato.

La D -structuredness $d(F)$ per un diagramma di flusso F può essere misurata in termini di:

- Primitive di base (primes):

$$d(P_1) = 1 ; d(D_0) = d(D_1) = d(D_2) = d(D_3) = 1$$
 0 altrimenti
- Sequenziamento:

$$d(F_1; F_2; \dots; F_k) = \min\{ d(F_1), d(F_2), \dots, d(F_k) \}$$
- Annidamento:

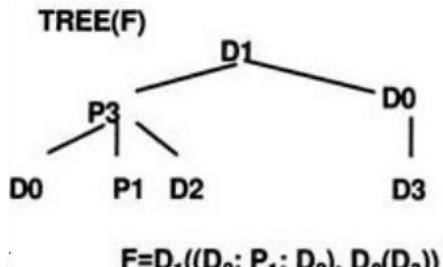
$$d(F(F_1, F_2, \dots, F_k)) = \min\{ d(F), d(F_1), d(F_2), \dots, d(F_k) \}$$

Esempio:

$$F = D_1((D_0; P_1; D_2), D_0(D_3))$$

$$\begin{aligned} d(F) &= d(D_1((D_0; P_1; D_2), D_0(D_3))) = \\ &= \min\{ d(D_1), d(D_0; P_1; D_2), d(D_0(D_3)) \} = \\ &= \min\{ d(D_1), \min\{ d(D_0), d(P_1), d(D_2) \}, \min\{ d(D_0), d(D_3) \} \} = \\ &= \min\{ 1, \min\{ 1, 1, 1 \}, \min\{ 1, 1 \} \} = \min\{ 1, 1, 1 \} = 1 \end{aligned}$$

=> f è D -strutturato (costruito soltanto utilizzando primitive di base)



CYCLOMATIC COMPLEXITY (COMPLESSITÀ CICLOMATICA)

La complessità di un programma può essere misurata dal numero ciclomatico del diagramma di flusso del programma.

Il numero ciclomatico può essere calcolato in 2 modi diversi:

1. Basato su diagrammi di flusso.

Per un programma con il diagramma di flusso F , la complessità ciclomatica $v(F)$ è misurata come segue:

$$v(F) = e - n + 2$$

dove e è il numero di archi (che rappresentano rami e cicli), e n è il numero di nodi (che rappresenta un blocco di codice sequenziale).

Il numero ciclomatico misura effettivamente il numero di percorsi linearmente indipendenti attraverso F (un insieme di percorsi è linearmente indipendente se nessun percorso nell'insieme è una combinazione lineare di qualsiasi altro percorso).

2. Basato su codici.

Basandoci sul codice la complessità ciclomatica è misurata come segue:

$$v(F) = 1 + d$$

dove d è il numero di nodi predicati (cioè, nodi con grado maggiore di 1), che rappresenta il numero di punti di decisione nel programma.

La complessità delle primitive del flowgraph dipende solo dai predicati in essi. $v(F)$ può essere definito come una misura gerarchica.

Domande sulla complessità ciclomatica:

1) Qual è la complessità delle primitive?

La complessità delle primitive è il numero di predicati più uno: $v(F) = 1 + d$.

2) Qual è la complessità del sequenziamento?

La complessità di una sequenza è uguale alla somma delle complessità dei componenti meno il numero di componenti più uno.

$$v(F_1; F_2; \dots; F_n) = \sum_{i=1}^n v(F_i) - n + 1$$

3) Qual è la complessità del nesting su un dato primo?

La complessità dei componenti di annidamento su un primo F è uguale alla complessità di F più la somma delle complessità dei componenti meno il numero di componenti.

$$v(F(F_1; F_2; \dots; F_n)) = v(F) + \sum_{i=1}^n v(F_i) - n$$

ESEMPIO FLOWGRAPH-BASED

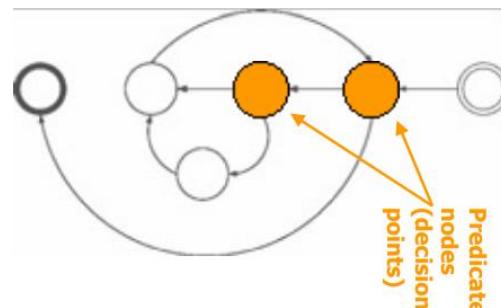
$$v(F) = e - n + 2$$

$$v(F) = 7 - 6 + 2$$

$$v(F) = 3$$

oppure

$$v(F) = 1 + d \quad v(F) = 1 + 2 = 3$$



ESEMPIO CODE-BASED

```
#include <stdio.h>
main()
{
int a ;
scanf ("%d", &a);
if ( a >=10)
    if ( a < 20 ) printf ("10 < a< 20 %d\n" , a);
    else printf ("a >= 20 %d\n" , a);
else printf ("a <= 10 %d\n" , a);
}
```

$$v(F) = 1 + d = 1 + 2 = 3$$

MISURA DI COMPLESSITÀ DI McCabe's DENOMINATA COMPLESSITÀ ESSENZIALE

La complessità essenziale di un programma con flowgraph F è data da:

$$ev(F) = v(F) - m$$

dove m è il numero di D_0, D_1, D_2 e D_3 sotto-flowgraphs di F .

Esempio:

$$v(F) = 5$$

$$ev(F) = 5 - 4 = 1$$

La complessità essenziale indica la misura in cui il diagramma di flusso può essere ridotto scomponendo tutti i sotto-flussi D_0, D_1, D_2 e D_3 ($ev(F) = 1$ per un programma strutturato con diagramma di flusso F).

CRITICITA' DELLA COMPLESSITA' CICLOMATICA

Vantaggi:

- Misurazione obiettiva della complessità.

Svantaggi:

- Può essere utilizzato solo a livello di componente.
- Due programmi aventi lo stesso numero di complessità ciclomatica possono richiedere uno sforzo di programmazione diverso.
- Richiede una progettazione completa o visibilità del codice

LEZ36 – 29/04/2024

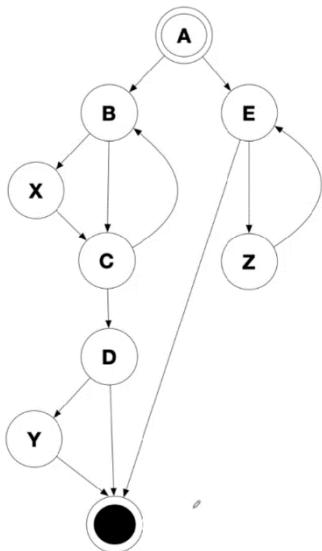
ESERCIZI SU METRICHE DI STRUTTURA

➤ Esercizio1: Dare il flow-graph del modulo descritto dalla formula

$$F = D1((D3(D0); D0), D2)$$

ed esprimerne pseudo codice, depth of nesting, D-structuredness, complessità ciclomatica

Soluzione: flow-graph



pseudo-codice

```

BEGIN
if A then
  BEGIN
    repeat
      if B then X;
      until C;
      if D then Y;
    END
  else
    while E do Z;
  END
END
  
```

depth of nesting

$$\begin{aligned}
 F &= D1(F1, F2) \text{ dove} \\
 F1 &= P2(D3(D0); D0) \\
 F2 &= D2 \\
 n(F) &= 1 + \max\{F1, F2\} \\
 &= 1 + \max\{d((D3(D0); D0), d(D2)\} \\
 &= 1 + \max\{\max\{d(D3(D0)), d(D0)\}, 1\} \\
 &= 1 + \max\{\max\{1 + \max\{d(D0), 1\}, 1\} \\
 &= 1 + \max\{\max\{1+1, 1\}, 1\} \\
 &= 1 + \max\{2, 1\} \\
 &= 1 + 2 = 3
 \end{aligned}$$

D-structuredness

$$F = D1(F1, F2) \text{ dove}$$

$$F1 = P2(D3(D0); D0)$$

$$F2 = D2$$

complessità ciclomatica

$$v(F) = e - n + 2 =$$

$$= 13 - 9 + 2 = 6$$

$$v(F) = 1 + d =$$

$$= 1 + 5 =$$

$$= 6$$

$$ev(F) = v(F) - m = 6 - 5 = 1$$

$$d(F) = \min\{d(F1), d(F2)\} =$$

$$= \min\{\min\{d(D3), d(D0)\}, 1\} =$$

$$= \min\{\min\{1, 1\}, 1\} =$$

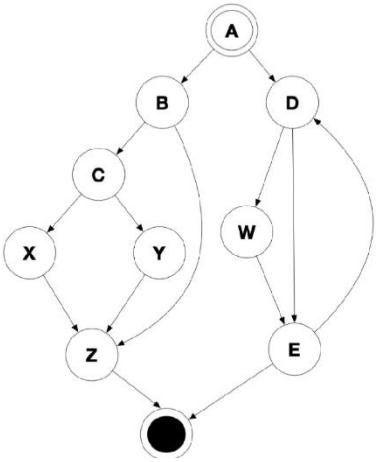
$$= \min\{1, 1\} = 1$$

➤ Esercizio2: dare il flow-graph del modulo descritto dalla formula

$$F = D1((D0(D1); P1), D3(D0))$$

ed esprimerne pseudo codice, depth of nesting, D-structuredness, complessità ciclomatica

Soluzione: flow-graph



pseudo-codice

```

BEGIN
if A then
  if B then
    Do [if C then X;
        else Y;
      Z]
  else
repeat
  if D then W;
until E;
END

```

depth of nesting

$$F = D1(F1, F2) \text{ dove}$$

$F1 = P2(D0(D1); P1)$

$F2 = D3(D0)$

$$\begin{aligned}
 n(F) &= 1 + \max\{F1, F2\} \\
 &= 1 + \max\{n(\underline{\text{D0(D1)}}); \underline{n(P1)}, n(\underline{\text{D3(D0)}})\} = \\
 &= 1 + \max\{\max\{1 + n(D1), 1\}, 1 + \max\{1\}\} = \\
 &= 1 + \max\{\max\{2, 1\}, 1 + 1\} = \\
 &= 1 + \max\{2, 2\} = \\
 &= 1 + 2 = 3
 \end{aligned}$$

D-structuredness

$$F = D1(F1, F2) \text{ dove} \\ F1 = P2(D0(D1); P1) \\ F2 = D3(D0)$$

$$\begin{aligned}
 d(F) &= \min\{d(F1), d(F2)\} = \\
 &= \min\{d(D0(D1)), d(P1), d(D3), d(D0)\} = \\
 &= \min\{d(D0), d(D1), d(P1), d(D3), d(D0)\} = \\
 &= \min\{1, 1, 1, 1, 1\} = 1
 \end{aligned}$$

complessità ciclomatica

$$v(F) = e - n + 2 =$$

$$= 14 - 10 + 2 = 6$$

$$\begin{aligned} v(F) &= 1 + d = \\ &= 1 + 5 = \\ &= 6 \end{aligned}$$

$$\text{ev}(F) = v(F) - m = 6 - 5 = 1$$

QUALITA' DEL SOFTWARE

Definizione IEEE(Standard per una metodologia di metriche di qualità del software):
La qualità del software è il grado in cui il software possiede una combinazione desiderata di attributi. Punti di vista:

- Trascendente (eccellenza innata)
 - Utente : adatto per l'uso (soddisfare le esigenze dell'utente), godere del suo utilizzo, soddisfazione dell'utente.
 - Prodotto : attributi desiderati del software (affidabilità, correttezza, ecc.) ; conformità ai requisiti.
 - Organizzazione : costi e profitti (maggiore efficienza, maggiore efficacia,. valore aggiunto all'organizzazione, prodotto commerciabile)

IL TRIANGOLO DELLA QUALITA' (QUALITY MODEL INTRODOTTO DA McCall)

Gli indici di qualità introdotti da McCall per valutare la qualità del software in questo modello sono i seguenti:

- Transition activities(perfective and adaptive maintainance), contenente gli indici
 - i1 correttezza
 - i2 affidabilità
 - i3 efficienza
 - i4 integrità
 - i5 usabilità
 - Revision activities(corrective maintenance)
 - i6 manutenibilità
 - i7 flessibilità
 - i8 testabilità
 - Operations activities
 - i9 portabilità
 - i10 riusabilità
 - i11 interoperabilità
 - i12 evolvibilità

Definizioni delle operazioni di qualità degli indici:

i1 Correttezza=fino a che punto un prodotto soddisfa le specifiche e soddisfa gli obiettivi degli utenti (fa quello che voglio?)

- i2 Affidabilità= in che misura si può prevedere che un prodotto svolga la sua funzione prevista con la necessaria precisione (lo fa sempre con precisione?)
- i3 Efficienza= quantità di risorse di calcolo e di codice richiesti da un prodotto per svolgere una funzione (unzionerà bene sul mio hardware?)
- i4 Integrità= misura in cui l'accesso al software o ai dati da parte di persone non autorizzate le persone possono essere controllate (è sicuro?)
- i5 Usabilità= sforzo necessario per apprendere, operare, preparare input e interpretare output di un prodotto (posso eseguirlo?)
- i6 Manutenibilità= sforzo necessario per individuare e correggere un errore in un programma operativo (posso correggerlo?)
- i7 Testabilità= sforzo necessario per testare un prodotto per garantire che esso svolga la funzione prevista (posso testarlo?)
- i8 Flessibilità= sforzo necessario per modificare un prodotto operativo (posso modificarlo?)
- i9 Portabilità= sforzo necessario per trasferire un prodotto da un ambiente hardware e/o software ad un altro (sarò in grado di usarlo su un'altra macchina?)
- i10 Riutilizzabilità= misura in cui un prodotto (o parti di esso) può essere riutilizzato in altre applicazioni (sarò in grado di riutilizzare parte del software?)
- i11 Interoperabilità= sforzo necessario per accoppiare un prodotto con un altro (sarò in grado di interfacciarmi con un altro sistema?)
- i12 Evolubilità= sforzo necessario per aggiornare il prodotto al fine di soddisfare nuove esigenze (è facile aggiornarlo quando le esigenze cambiano?)

Nel caso di questo modello la qualità del software è definita in termini di qualità degli indici:
 $q = (i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8, i_9, i_{10}, i_{11}, i_{12})$

Ogni indice di qualità può essere a sua volta definito in termini di attributi di qualità:

$$ij = (a_1, a_2, \dots, a_n)$$

Ogni attributo può essere a sua volta definito in termini di sottoattributi, che sono misurati per ottenere il valore dell'attributo.

I valori degli attributi contribuiscono a quantificare gli indici pertinenti.

La valutazione degli indici di qualità consente di verificare la capacità del sistema di soddisfare i livelli di qualità desiderati (per gli indici di interesse).

Gli attributi di qualità sono 10, e sono:

- a1. Complessità = livello di comprensione e verificabilità degli elementi del software e le loro interazioni
- a2. Accuratezza = precisione dei calcoli e dell'output
- a3. Completezza = piena implementazione delle funzionalità richieste
- a4. Consistenza = uso di tecniche e notazioni di progettazione e implementazione uniformi
- a5. Tolleranza di errore = continuità di funzionamento garantita in condizioni sfavorevoli
- a6. Tracciabilità = grado in cui può essere stabilita una relazione tra due o più prodotti del processo di sviluppo
- a7. Espandibilità = memorizzazione o funzioni possono essere espanso
- A8. Generalità = ampiezza delle applicazioni potenziali
- A9. Modularità = disposizioni di moduli altamente indipendenti
- A10. Auto-documentazione = documenti in linea

INDICI E ATTRIBUTI DI QUALITÀ DEL SOFTWARE (+/- PER IMPATTO POSITIVO/NEGATIVO)

<i>i₁</i> Correctness	<i>i₂</i> Reliability	<i>i₇</i> Flexibility	<i>i₁₂</i> Evolubility
<ul style="list-style-type: none"> + Completeness + Consistency + Traceability 	<ul style="list-style-type: none"> + Error Tolerance + Consistency + Accuracy - Complexity 	<ul style="list-style-type: none"> + Traceability + Consistency - Complexity + Modularity + Generality + Auto-documentation 	<ul style="list-style-type: none"> + Consistency - Complexity + Modularity + Expandability + Generality + Auto-documentation

IL METODO CHECKLIST (LISTE DI CONTROLLO)

Il livello di qualità di ciascun attributo è valutato utilizzando tecniche basate su checklist. Il risultato della lista di controllo è elaborato secondo algoritmi appropriati per ottenere un punteggio normalizzato, indicando il livello di qualità di ciascun attributo. Una volta valutato il punteggio di ciascun attributo, i valori rilevanti sono raggruppati per ogni attributo, sintetizzato e normalizzato per ottenere il livello di qualità per ogni indice. L'attributo deve essere considerato con il proprio segno (positivo o negativo).

VALUTAZIONE DELLE CHECKLIST

Una checklist è composta da diverse domande e per ogni domanda sono previste quattro risposte con un valore dato.

Il calcolo non terrà conto delle questioni individuate come non applicabili.

Se la domanda non è valida, allora il punteggio assegnato a quella domanda è il più basso tra quelli possibili.

Il team di valutazione della lista di controllo è composto da almeno quattro persone, con competenze diverse in modo da poter confrontare diversi punti di vista e la valutazione finale è la più precisa. La composizione raccomandata della squadra riguarda uno specialista della garanzia della qualità, un capo progetto, un ingegnere di sistema, un analista software.

CALCOLO DEL VALORE DEGLI ATTRIBUTI

Per ogni attributo (ad eccezione della modularità) il calcolo del punteggio viene eseguito secondo la seguente formula:

$$V_{attribute} = \frac{\sum_{i=1}^{\# questions} V_{answer_i}}{\sum_{i=1}^{\# questions} max(V_{answer_i})}$$

Dove V_{answer_i} è il valore dato dalla risposta scelta per la domanda i , e $\max(V_{answer_i})$ è il suo valore massimo.

VALUTAZIONE DELL'INDICE DI QUALITÀ'

Utilizzando i risultati del calcolo del punteggio attributi e l'impatto di ciascun attributo sull'indice pertinente consente di calcolare il livello di qualità dell'indice come segue:

$$V_{index} = \frac{\sum V_{attribute(+)} + \sum (1 - V_{attribute(-)})}{\text{number of attr. associated to index}}$$

Dove $V_{attribute(+)}$ e $V_{attribute(-)}$ sono i punteggi degli attributi che incidono positivamente e negativamente sull'indice (rispettivamente).

**SCALA DEL LIVELLO DI ACCETTAZIONE
(ACCEPTANCE LEVEL SCALE)**

L'output della formula che produce V_{index} è un valore compreso tra 0 e 1.

Il seguente schema può essere utilizzato per identificare i valori soglia per l'accettazione della qualità:

V_{index}

Quality Level

$0.66 < V \leq 1$ High

$0.33 < V \leq 0.66$ Medium

$0 < V \leq 0.33$ Low

ESEMPI CHECKLIST

Supponiamo che il livello di qualità di un'architettura di sistema sia da valutare (in fase di progettazione architettonica).

Esempi di liste di controllo per il punteggio dei seguenti attributi sono illustrati nelle diapositive successive: a1 Complessità, a8 Generalità, a9 Modularità.

Per quanto riguarda la a1.complessità (impatto negativo):

- 1) I moduli, le procedure e i nomi delle strutture sono significativi o conformi a una norma, se esiste.
 - 0 : Sì, quasi sempre
 - 1 : Molto spesso
 - 2 : Raramente
 - 3 : No
- 2) È il formalismo utilizzato per descrivere l'architettura di sistema solo uno, o più quelli sono presenti.
 - 0 : Formalismo unico, standard e scelto correttamente
 - 1 : pochi formalismi, standard e scelti correttamente
 - 2 : Pochi formalismi e qualcuno fuori dallo standard
 - 3 : Un sacco di formalismi, qualcuno fuori dallo standard
- 3) La progettazione globale del sistema è strutturata in modo adeguato e facilmente comprensibile da parte di persone prive di conoscenze specifiche sul sistema.
 - 0 : Sì
 - 1 : Quasi positivo
 - 2 : Non adeguatamente strutturato
 - 3 : Male strutturato, e difficilmente comprensibile
- 4) È possibile dedurre la classe di informazioni di ogni singolo dato presente nel modello logico. Tutti gli utilizzi di questi dati sono dichiarati nella documentazione e realizzati con lo scopo di leggere o scrivere quella classe di informazioni.
 - 0 : no
 - 1 : raramente
 - 2 : Molto spesso
 - 3 : approssimativamente sempre.

Applicazioni metriche:

- 5) Se è possibile effettuare misurazioni sulla complessità del programma, in base al grafico di chiamata del programma, utilizzare le metriche sotto definite.
 - 5.1) Complessità gerarchica: è il numero medio di moduli per livello di albero chiamante, cioè il numero totale di moduli divisi per il numero di livelli di albero
 - 0 : da 0 a 4
 - 1 : da 4 a 8
 - 2 : da 8 a 12
 - 3 : inferiore a 2 o superiore a 12
 - 5.2) Complessità strutturale: è il numero medio di chiamate per ogni modulo, cioè il numero di chiamate inter-modulo divise per i moduli numerici.
 - 0 : meno di 2
 - 1 : 2 fino a 4
 - 2 : 4 fino a 8
 - 3 : maggiore di 8

Per quanto riguarda l'attributo a8.generalità (stavolta impatto positivo):

- 1) Tutte le funzioni offerte dai moduli (e dalle loro interfacce) sono opportunamente pianificate in modo da poterle riutilizzare in alcune implementazioni non pianificate all'inizio del progetto. (cioè: una percentuale specifica [18%], o qualsiasi percentuale di qualsiasi valore)
 - 0 : no
 - 1 : raramente
 - 2 : Molto spesso
 - 3 : sempre
- 2) Per quanto riguarda i moduli che non hanno una generalità sufficiente, è possibile renderli più generici con uno sforzo minimo (circa il 10% dello sforzo globale per ottenerli ciascuno).
 - 0 : no
 - 1 : raramente
 - 2 : Molto spesso
 - 3 : sempre

3) Sono tutte le manipolazioni della struttura dati combinate in moduli specificamente dedicati a questo.

- 0 : no
- 1 : raramente
- 2 : Molto spesso
- 3 : sempre

Per quanto riguarda l'attributo a9.modularità, valutazione di coesione(1) e coupling(2):
Coesione

1) Che è la distribuzione percentuale dei moduli di sistema secondo i seguenti quattro tipi:
1A moduli con coesione coincidente
1B moduli con coesione logica o temporale
1C moduli con coesione procedurale o comunicativa
1D moduli con coesione informativa o funzionale

Accoppiamento

2) Che è la distribuzione percentuale delle modalità di comunicazione (accoppiamento) tra moduli secondo i seguenti quattro tipi:
2A coppie di moduli con accoppiamento contenuto
2B coppie di moduli con accoppiamento comune
2C coppie di moduli con accoppiamento di controllo
2D coppie di moduli con timbro o accoppiamento dati

In generale

3) La scomposizione del sistema è organizzata in modo gerarchico

- 0 : no
- 1 : raramente
- 2 : Sì, abbastanza
- 3 : Sì

4) Che è la percentuale di strutture gerarchiche (ogni programma contiene un numero comparabile di moduli)
0 : meno del 70%
1 : entro 70% e 80%
2 : entro 80% e 90%
3 : più di 90%

PUNTEGGIO PER LE DOMANDE DI COESIONE(COHESION) E ACCOPPIAMENTO(COUPLING)

$$V_{\text{answer}_1} = \frac{(0 \times \%1A) + (1 \times \%1B) + (2 \times \%1C) + (3 \times \%1D)}{50}$$

$$V_{\text{answer}_2} = \frac{(0 \times \%2A) + (1 \times \%2B) + (2 \times \%2C) + (3 \times \%2D)}{50}$$

sistema

$$V_{\text{Modularity}} = \frac{V_{\text{answer}_1} + V_{\text{answer}_2} + V_{\text{answer}_3} + V_{\text{answer}_4}}{\max(V_{\text{answer}_1}) + \max(V_{\text{answer}_2}) + \max(V_{\text{answer}_3}) + \max(V_{\text{answer}_4})}$$

dove %X è la percentuale di moduli (per la risposta 1) o di coppie di moduli (per la risposta 2) di tipo X rispetto all'insieme di moduli/coppie del

PROCEDURA PER IL PUNTEGGIO DEGLI ATTRIBUTI

a) La documentazione deve essere analizzata in modo approfondito per ottenere una risposta sufficientemente precisa a ciascuna domanda contenuta nell'elenco di controllo. Ogni domanda prevede quattro possibili risposte e ognuna di esse è associata a un punteggio.

b) Ogni revisore appartenente al gruppo di valutazione deve rispondere alle domande della lista di controllo in modo indipendente. Il recensore non deve conoscere la risposta degli altri prima.

Le domande identificate come non applicabili devono essere contrassegnate in questo modo esplicitamente.

Quando una domanda è stata identificata come non valida, viene analizzata in modo più dettagliato per verificare se è applicabile o meno. Se alla fine dell'analisi la domanda è considerata applicabile, il punteggio più basso tra quelli possibili viene assegnato alla domanda stessa. Questo evento dà anche un'indicazione di non conformità della documentazione esaminata.

- c) Quando tutti i membri del gruppo di valutazione danno risposte diverse a una determinata domanda, è necessario che raggiungano una risposta comune concordata.
- d) Al termine della discussione sarà compilato un elenco di controllo ufficiale unico con le risposte concordate dai membri del gruppo di valutazione.
- e) Calcolare il punteggio del sottoattributo in base alla formula di calcolo.

TEMPLATE PUNTEGGIO DEGLI ATTRIBUTI

Attribute:			
Question #	N/A	Not Valuable	Score
Total Score (see formula on slide 12)			

TEMPLATE VALUTAZIONE DELL'INDICE

Index:		
Attribute	Weight (+/-)	Score
Evaluation (see formula on slide 13)		
Quality level (see acceptance scale on slide 14)		

GARANZIA DELLA QUALITA' DEL SOFTWARE (SOFTWARE QUALITY ASSURANCE)

La garanzia della qualità del software (SQA) è un approccio pianificato e sistematico per garantire che sia il processo software che il prodotto software siano conformi a standard, processi e procedure.

Gli obiettivi di SQA sono migliorare la qualità del software monitorando sia il software che il processo di sviluppo per garantire la piena conformità agli standard e alle procedure stabilite. Passi per stabilire un programma SQA:

1. Ottenere l'accordo del top management sul suo obiettivo e supporto (senza il supporto della gestione, SQA non può essere efficace)(necessario perché molto costoso)
2. Identificare le questioni SQA, redigere un piano SQA, stabilire norme e procedure SQA, attuare il piano SQA e valutare il programma SQA

SQA ROLE (RUOLO DEL TEAM DI SOFTWARE QUALITY ASSURANCE)

Il ruolo di SQA è quello di dare alla direzione l'assicurazione che il processo ufficialmente stabilito è effettivamente attuato. SQA garantisce che:

- Esiste una metodologia di sviluppo adeguata
- I progetti utilizzano norme e procedure nel loro lavoro
- Vengono effettuati controlli e verifiche
- Vengono prodotti documenti a sostegno della manutenzione e del miglioramento
- La gestione della configurazione del software è impostata per controllare le modifiche.
- Esecuzione e superamento delle prove
- Le carenze e le deviazioni sono identificate, documentate e portate all'attenzione della direzione

SQA GOALS (OBIETTIVI DEL TEAM DI SQA)

Gli obiettivi di SQA è quello di ridurre i rischi da:

- Monitoraggio adeguato del software e del processo di sviluppo;
- Garantire il pieno rispetto delle norme e delle procedure in materia di software e di processo;
- Garantire che le inadeguatezze del prodotto, del processo o delle norme siano portate all'attenzione della direzione in modo che possano essere fissate.

SQA non è responsabile della produzione di prodotti di qualità. È responsabile della revisione contabile (cioè, guardando il prodotto in profondità, confrontandolo con gli standard e le procedure stabiliti) le azioni di qualità e per avvisare la gestione di eventuali deviazioni

SQA STANDARDS E PROCEDURE

Gli standard sono i criteri a cui i prodotti software sono confrontati (cioè, gli standard definiscono ciò che dovrebbe essere fatto). I requisiti minimi per le norme comprendono:

- Standard di documentazione: specificare forma e contenuto per la pianificazione, il controllo e la documentazione del prodotto e fornire coerenza durante l'intero progetto.
- Standard di progettazione: specificare la forma e il contenuto del prodotto di progettazione. Forniscono regole per tradurre i requisiti software nella progettazione del software e per rappresentarli nella documentazione di progettazione.
- Code Standards: specifica la lingua in cui il codice deve essere scritto e definisce eventuali restrizioni all'uso delle caratteristiche della lingua. Definiscono la struttura del linguaggio legale, le convenzioni di stile, le regole per le strutture di dati e la documentazione interna del codice.

Le procedure sono i criteri ai quali vengono confrontati i processi di sviluppo e di controllo (cioè, le procedure definiscono come il lavoro deve effettivamente essere fatto, da chi, quando e cosa viene fatto con i risultati).

PIANO DI SOFTWARE QUALITY ASSURANCE (SQA PLAN)

Il piano SQA specifica gli obiettivi SQA, i compiti da eseguire e gli standard e le procedure rispetto ai quali il lavoro di sviluppo deve essere misurato.

Lo standard IEEE per la preparazione del piano SQA (730-1998) include: Direzione; Documentazione; Norme, prassi e convenzioni; Revisioni e verifiche; Gestione della configurazione del software; Segnalazione dei problemi e azioni correttive; Strumenti, tecniche e metodologie; Codice di controllo; Controllo dei media; Controllo del fornitore; Raccolta, manutenzione e conservazione delle registrazioni.

LEZ38 – 08/05/2024

VERIFICATION E VALIDATION (VERIFICA E CONVALIDA)

Garantire che un sistema software sia conforme alle sue specifiche e soddisfi le esigenze dell'utente.

Verifica = il software deve essere conforme alle sue specifiche:

"Are we building the product right?"

Convalida = il software dovrebbe fare ciò che l'utente richiede realmente:

"Are we building the right product?"

TECNICHE V&V

- Ispezioni del software: Riguarda l'analisi della rappresentazione del sistema statico per scoprire problemi (tecniche statiche)
 - o Può essere integrato da documenti e analisi dei codici basati su strumenti
- Test del software: Riguarda l'esercizio e l'osservazione del comportamento del prodotto (tecniche dinamiche)
 - o Il sistema è eseguito con dati di prova e il suo comportamento operativo è osservato

TIPI DI TESTING

1. Validation testing

- Prove volte a dimostrare che un sistema soddisfa le esigenze degli utenti
- Una prova di convalida di successo richiede che il sistema funzioni correttamente utilizzando determinati casi di prova di accettazione

2. Defect testing (Controllo dei difetti)

- Prove destinate a rilevare difetti del sistema
- Una prova del difetto è una prova che rivela la presenza di difetti in un sistema

3. Statistical testing (Test statistici)

- Prove destinate a riflettere la frequenza degli input degli utenti
- utilizzati per la stima dell'affidabilità

Ci focalizziamo ora sul Defect Testing.

DEFECT TESTING

L'obiettivo del test dei difetti è scoprire i difetti nei programmi.

Ciò contrasta con i validation testing(test di convalida) che mirano a dimostrare che un sistema soddisfa i requisiti degli utenti.

I validation test richiedono che il sistema esegua correttamente utilizzando determinati casi di test di accettazione per essere un test riuscito, al contrario un test di guasto riuscito è un test che provoca un comportamento anomalo del programma.

Nota: i test dimostrano la presenza di difetti ma non la loro assenza.

FASI DEL DEFECT TESTING



Component testing - Test componenti (testing di unità e modulo)

- Collaudo dei singoli componenti del programma
- Di solito la responsabilità è dello sviluppatore di componenti (tranne talvolta per i sistemi critici)
- Le prove derivano dall'esperienza del committente

Integration testing - Test di integrazione (test del sottosistema e del sistema)

- Prove di gruppi di componenti integrati per creare un sistema o un sottosistema
- La responsabilità è di un gruppo di collaudatori indipendente
- Le prove si basano su una specifica di sistema

L'User testing (convalida o collaudo di accettazione) non fa parte del processo di verifica dei difetti.

POLITICHE DI Sperimentazione

Solo test esaustivi possono dimostrare che un programma è privo di difetti. Tuttavia, fare test esaustivi è impossibile. I test quindi devono essere basati su un sottoinsieme di possibili casi di test, secondo politiche che dovrebbero essere elaborate dal team V&V (e non dal team di sviluppo).

I test dovrebbero esercitare le capacità di un sistema piuttosto che le sue componenti. Testare situazioni tipiche è più importante dei casi con valori limite.

CASI E DATI DI PROVA PER TESTING

Test cases (test di prova):

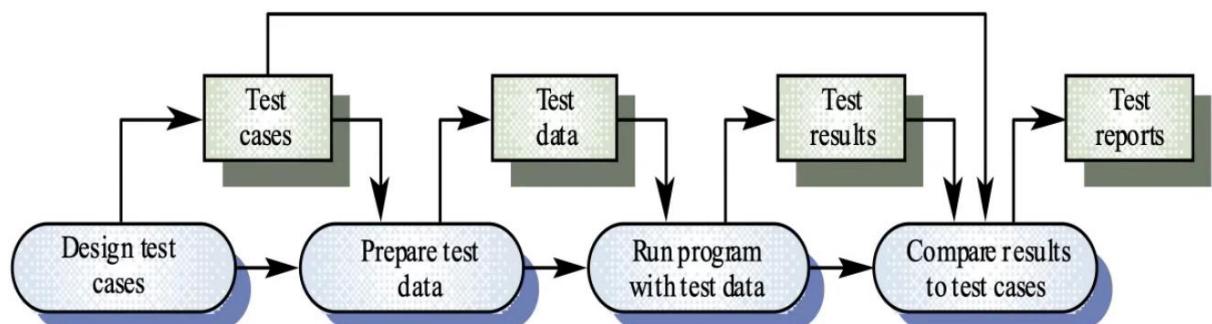
- Input per testare il sistema e le uscite previste da questi input se il sistema funziona secondo le sue specifiche
- I casi di prova sono generalmente generati manualmente (perché non è facile ricavare automaticamente l'output del test da specifiche informali)

Test data (dati di prova):

- Input concepiti per testare il sistema
- I dati di prova possono essere generati automaticamente

IL PROCESSO DI DEFECT TESTING

Processo generale:



(i rettangoli stondati azzurri rappresentiamo le attività, i rettangoli quadrati verdi rappresentano gli output prodotti utilizzati poi come input da altre attività, la freccia rappresenta sia il flusso dati che il flusso di controllo tra una attività all'altra)

BLACK-BOX TESTING (TEST A SCATOLA NERA)

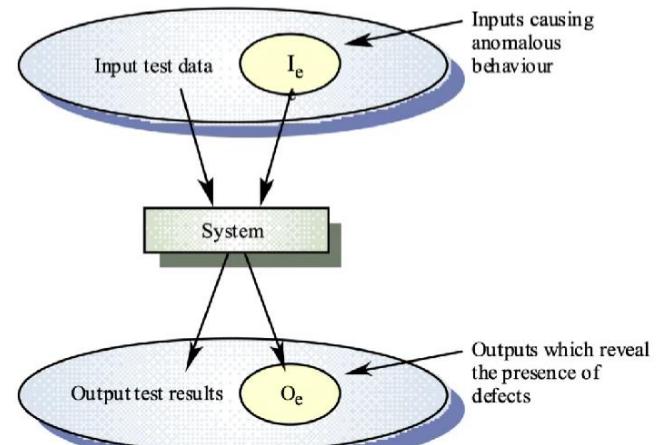
Un approccio al test in cui il programma è considerato come una scatola nera.

I casi di test del programma sono derivati dalle specifiche del sistema.

Il tester presenta gli input al componente o al sistema ed esamina l'output corrispondente.

Se gli output non sono quelli specificati, il test ha rilevato con successo un problema con il software.

Chiamato anche test funzionale perché il tester si occupa solo della funzionalità e non dell'implementazione del software.



EQUIVALENCE PARTITIONING (PARTIZIONE PER EQUIVALENZA)

I dati di input e i risultati di output spesso rientrano in classi diverse in cui tutti i membri di una classe sono correlati.

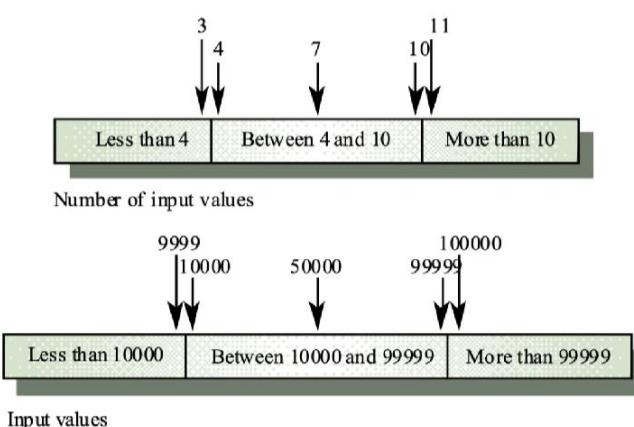
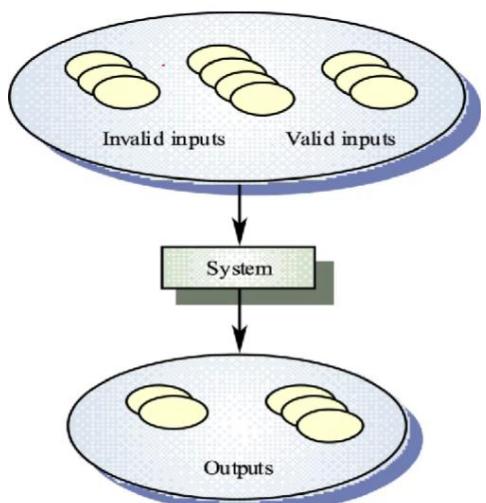
Ciascuna di queste classi è una partizione di equivalenza in cui il programma si comporta in modo equivalente per ogni membro della classe.

I casi di test devono essere scelti da ciascuna partizione.

Nota: non si prendono in considerazione soltanto gli input validi ma anche quelli non validi.

Partizionare ingressi e uscite del sistema in "partizioni di equivalenza": se l'input è un intero di 5 cifre compreso tra 10.000 e 99.999, le partizioni di equivalenza sono < 10.000 | 10.000 - 99.999 | > 99.999

Scegliere casi di test al confine di queste partizioni più casi vicino al punto medio delle partizioni con input validi 09999, 10000, 50000, 99999, 100000.



Esempio applicato ad una funzione di ricerca:

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;
    Found : in out BOOLEAN; L: in out ELEM_INDEX) ;
```

Pre-condition

-- the array has at least one element
T'FIRST <= T'LAST

Post-condition

-- the element is found and is referenced by L
(Found and T (L) = Key)

or

-- the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Input partitions:

- Input conformi alle condizioni preliminari
- Input in cui non è soddisfatta una precondizione
- Ingressi in cui l'elemento chiave è un membro dell'array
- Ingressi in cui l'elemento chiave non è un membro dell'array

LINEE GUIDA TESTING (TESTING DI SEQUENZE)

- Software di prova con sequenze che hanno un solo valore
- Utilizzare sequenze di diverse dimensioni in diversi test
- Deriva i test in modo che il primo, medio e ultimo elemento della sequenza siano accessibili
- Prova con sequenze di lunghezza zero

Esempio input partitions e test cases su funzione di ricerca:

Array	Element
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

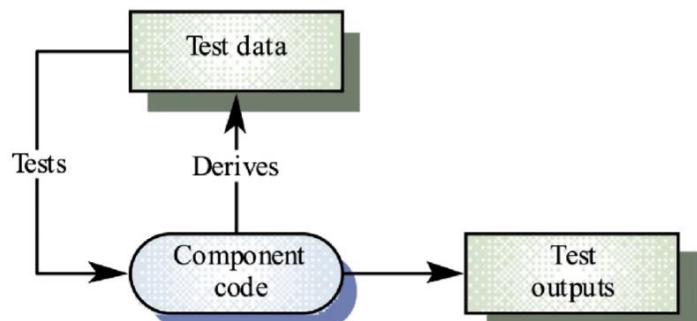
Input sequence (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, ??

STRUCTURAL TESTING (TEST STRUTTURALI)

A volte chiamato test scatola bianca.

I casi di test sono derivati dalla struttura del programma.

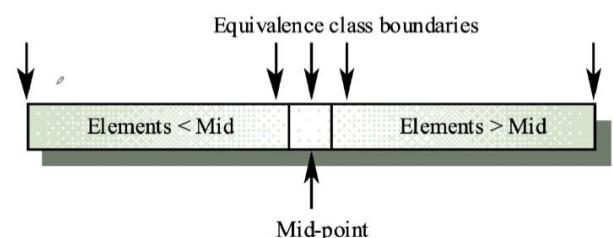
La conoscenza del programma viene utilizzata per identificare ulteriori casi di test. L'obiettivo è esercitare tutte le istruzioni del programma (non tutte le combinazioni di percorso).



ESEMPIO DI EQUIVALENCE PARTITIONING SU FUNZIONE DI RICERCA BINARIA

Casi possibili partizioni di equivalenza su funzione di ricerca binaria:

- Pre-condizioni soddisfatte, elemento chiave in array
- Pre-condizioni soddisfatte, elemento chiave non nell'array
- Pre-condizioni non soddisfatte, elemento chiave non presente nell'array
- L'array di input ha un unico valore
- L'array di input ha un numero pari di valori
- L'array di input ha un numero dispari di valori

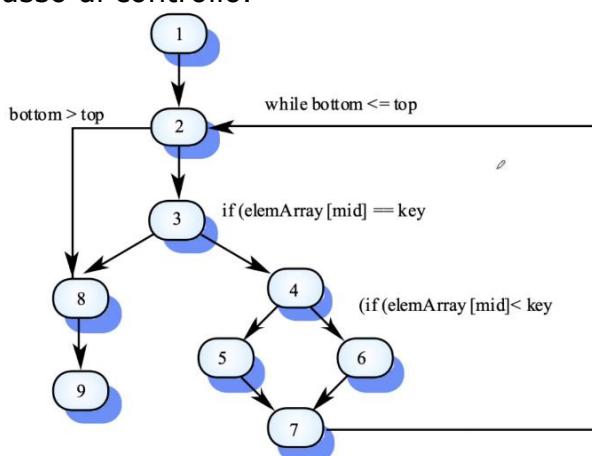


Input array (T)	Key (Key)	Output (Found, L)
17	17	true, 1
17	0	false, ??
17, 21, 23, 29	17	true, 1
9, 16, 18, 30, 31, 41, 45	45	true, 7
17, 18, 21, 23, 29, 38, 41	23	true, 4
17, 18, 21, 23, 29, 33, 38	21	true, 3
12, 18, 21, 23, 32	23	true, 4
21, 23, 29, 33, 38	25	false, ??

PATH TESTING (TEST DEI PERCORSI)

L'obiettivo del test dei percorsi è garantire che l'insieme dei casi di test sia tale che ogni percorso attraverso il programma (e quindi ogni istruzione) venga eseguito almeno una volta. Il numero di percorsi attraverso un programma è di solito proporzionale alla sua dimensione. Poiché i moduli sono integrati nei sistemi, diventa impossibile utilizzare tecniche di test del percorso, che sono quindi utilizzate principalmente in fase di test di unità o modulo.

Il punto di partenza per il test del percorso è un diagramma di flusso del programma che mostra i nodi che rappresentano le decisioni del programma e gli archi che rappresentano il flusso di controllo.



<- Esempio path testing in ricerca binaria

1, 2, 3, 8, 9
1, 2, 3, 4, 6, 7, 2
1, 2, 3, 4, 5, 7, 2
1, 2, 3, 4, 6, 7, 2, 8, 9

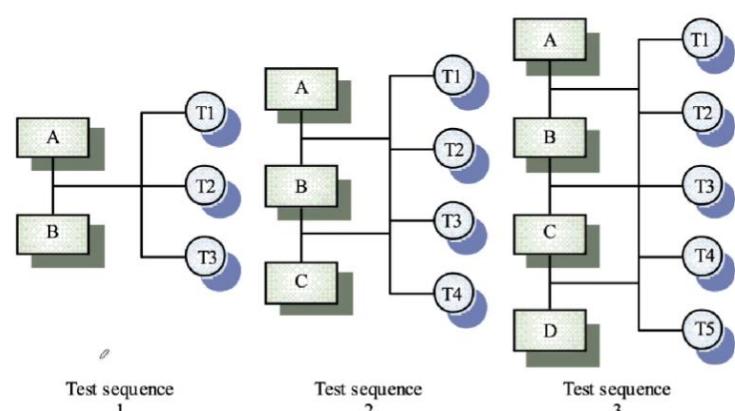
I casi di test dovrebbero essere ricavati in modo che tutti questi percorsi siano eseguiti. Gli analizzatori di programmi dinamici o i profiler possono essere utilizzati per verificare che i percorsi siano stati eseguiti.

INTEGRATION TESTING

Test di sistemi o sottosistemi completi composti da componenti integrati.

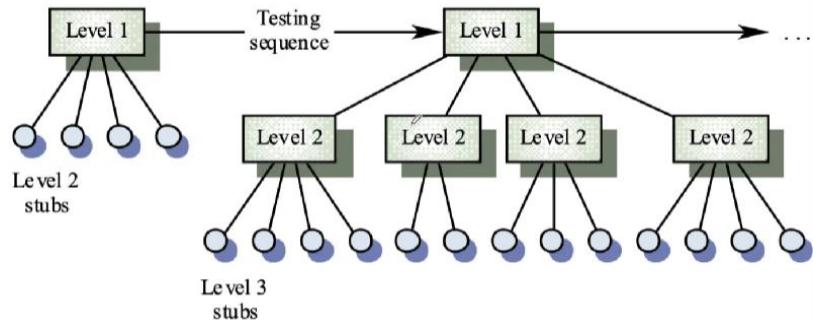
Le prove di integrazione dovrebbero essere prove in scatola nera con casi di prova derivati dalle specifiche.

La difficoltà principale è la localizzazione degli errori.



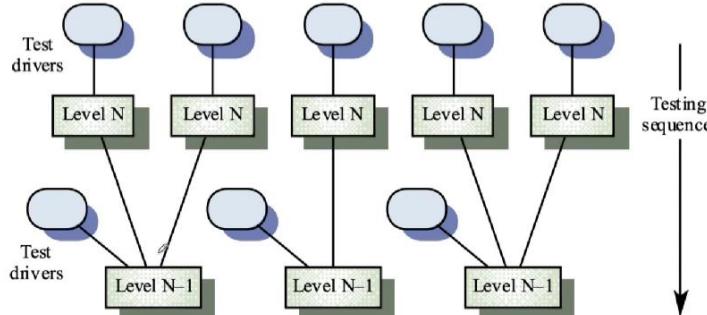
Approcci di integrazione testing:

1. Test top-down: iniziare con un sistema di alto livello e integrare dall'alto verso il basso rimpiazzando singoli componenti con stub, dove appropriato.



2. Test bottom-up: integrare i singoli componenti in livelli fino alla creazione del sistema completo.

In pratica, la maggior parte del l'integrazione comporta una combinazione di queste strategie.



Metodi di testing:

- Architectural validation: integration testing di tipo top-down consentono di individuare meglio gli errori nell'architettura di sistema
- System demonstration: integration testing di tipo top-down consentono una dimostrazione limitata in una fase iniziale dello sviluppo;
- Test implementation: spesso più facile con integration testing di tipo bottom-up
- Test observation: si hanno problemi con entrambi gli approcci. Può essere richiesto un codice supplementare per osservare le prove.

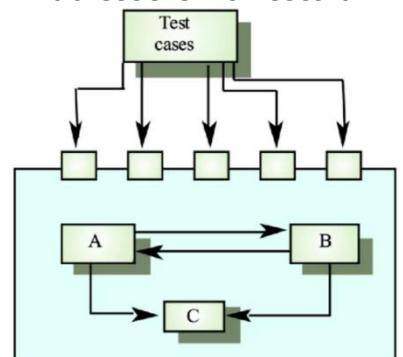
INTERFACE TESTING

Avviene quando moduli o sottosistemi sono integrati per creare sistemi più grandi.

Gli obiettivi sono di rilevare guasti dovuti a errori di interfaccia o ipotesi non valide sulle interfacce. Particolarmente importante per lo sviluppo orientato agli oggetti in quanto gli oggetti sono definiti dalle loro interfacce.

Tipi di interfaccia:

- Interfacce di parametri: trasmissione dei dati da una procedura all'altra
- Interfacce di memoria condivisa: il blocco di memoria è condiviso tra procedure
- Interfacce procedurali: il sottosistema comprende un insieme di procedure che devono essere invocate da altri sottosistemi
- Interfacce per il passaggio dei messaggi: sottosistemi che richiedono servizi da altri sottosistemi



Errori di interfaccia:

- Uso improprio dell'interfaccia: un componente chiamante chiama un altro componente e fa un errore nel suo uso della sua interfaccia, ad es. parametri nell'ordine sbagliato, di tipo sbagliato o nel numero sbagliato.
- Incomprensione dell'interfaccia: un componente chiamante incorpora ipotesi sul comportamento del componente chiamato che sono errate.

- Errori di temporizzazione: il componente chiamato e il componente chiamante operano a velocità diverse e si accede a informazioni scadute.

Linee guida per i test di interfaccia:

- Test di progettazione in modo che i parametri di una procedura chiamata siano alle estremità estreme dei loro intervalli
- Prova sempre i parametri del puntatore con puntatori nulli
- Prove di progettazione che causano il guasto del componente
- Utilizzare prove di stress (stress testing) nei sistemi di passaggio dei messaggi
- Nei sistemi di memoria condivisa, variare l'ordine di attivazione dei componenti

STRESS TESTING

Esercita il sistema oltre il suo carico massimo di progettazione. Stressare il sistema spesso fa venire alla luce difetti. Stressare il sistema porta a guasto del sistema:

- i sistemi non devono fallire in modo catastrofico,
- il guasto non deve causare una perdita di servizio o di dati inaccettabile.

Particolarmente rilevante per i sistemi distribuiti che possono presentare grave degrado come una rete diventa sovraccarico.

OBJECT-ORIENTED TESTING

I componenti da testare sono classi di oggetti che vengono istanziate come oggetti. Unità di codice di dimensione superiore rispetto a funzione singola, quindi dobbiamo estendere approcci di tipo white-box .

Un sistema object oriented è una ragnatela di oggetti che collaborano, quindi non gerarchico pertanto non adatto a approcci top-down e bottom-up.

Approccio di testing a livelli mediante metodi di testing associati agli oggetti, classi di testing object, test cluster di oggetti che hanno collaborato, testing del sistema OO completo.

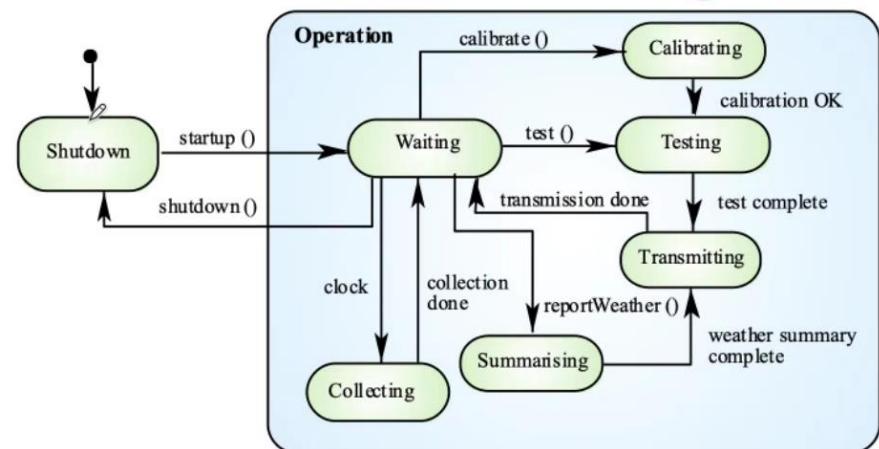
Cosa significa fare testing di classi? Il testing completo di una classe comprende:

- Verifica di tutte le operazioni associate a un oggetto
- Impostazione e interrogazione di tutti gli attributi oggetto
- Esercitare l'oggetto in tutti gli stati possibili

L'ereditarietà rende più difficile progettare test di classe oggetto in quanto le informazioni da testare non sono localizzate.

ESEMPIO TESTING DI UNA CLASSE

WeatherStation
identifier
reportWeather ()
calibrate (instruments)
test ()
startup (instruments)
shutdown (instruments)



I test case sono necessari per tutte le operazioni.

Utilizzare state diagram per identificare le transizioni di stato per il test.

Esempi di sequenze di prova:

`Shutdown` -> `Waiting` -> `Shutdown`

`Waiting` -> `Calibrating` -> `Testing` -> `Transmitting` -> `Waiting`

`Waiting` -> `Collecting` -> `Waiting` -> `Summarising` -> `Transmitting` -> `Waiting`

INTEGRAZIONE TRA OGGETTI

I livelli di integrazione sono meno distinti nei sistemi orientati agli oggetti.

Il test del cluster (cluster testing) riguarda l'integrazione e il test di cluster di oggetti cooperanti. Identificare i cluster utilizzando la conoscenza del funzionamento degli oggetti e delle funzionalità di sistema implementate da questi cluster.

APPROCCI PER IL CLUSTER TESTING

1. Test use-case o scenario

- le prove si basano sulle interazioni degli utenti con il sistema
- ha il vantaggio di testare le caratteristiche del sistema sperimentate dagli utenti

2. Thread testing

- Verifica la risposta dei sistemi agli eventi durante l'elaborazione tramite il sistema

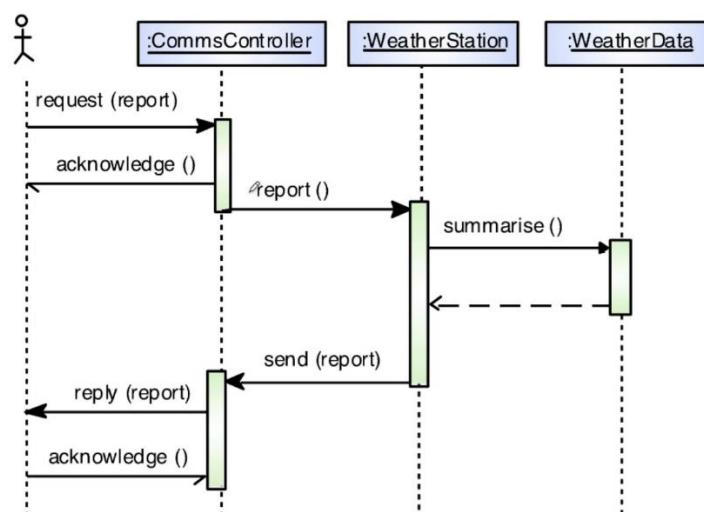
3. Object interaction testing (test di integrazione degli oggetti)

- Test di sequenze di interazioni di oggetti che si fermano quando un'operazione oggetto non richiede servizi da un altro oggetto

TESTING BASATO SU SCENARIO

Identificare scenari da casi d'uso e integrare questi con diagrammi di interazione che mostrano gli oggetti coinvolti nello scenario (per esempio sequence diagrams).

Si consideri lo scenario nel sistema delle stazioni meteorologiche in cui viene generato un report:



TESTING WORKBENCHES (BANCHI DI PROVA)

I test sono una fase di processo costosa. I banchi di prova forniscono una gamma di strumenti per ridurre i tempi e i costi totali delle prove. La maggior parte dei banchi di prova sono sistemi aperti perché le esigenze di prova sono specifiche dell'organizzazione.

LEZ40 – 29/05/2024 (BPM Business Process Management)

BACKGROUND E QUELLO DI CUI ABBIAMO BISOGNO

Le organizzazioni svolgono attività per fornire prodotti e servizi ai propri clienti. Ogni prodotto o servizio è il risultato di diverse attività che possono essere svolte dai dipendenti manualmente, dai dipendenti con l'ausilio di sistemi informativi, dai sistemi informativi in modo automatico.

I prodotti e i servizi sono forniti in base agli obiettivi aziendali, come ad esempio aumentare la soddisfazione del cliente, ridurre i costi, migliorare l'efficienza, ridurre i tempi di esecuzione, ridurre i tassi di errore.

Per raggiungere gli obiettivi aziendali, ci si aspetta che le risorse dell'impresa (dipendenti, sistemi informativi) lavorino insieme.

Per raggiungere gli obiettivi di business dobbiamo organizzare le attività dell'impresa e comprendere come le attività interagiscono tra loro.

La soluzione consiste nel definire, analizzare, attuare e monitorare i cosiddetti processi aziendali. I processi aziendali facilitano la collaborazione tra le risorse aziendali in modo efficiente ed efficace. I processi aziendali possono essere gestiti in modo efficace se sono descritti e documentati da modelli di processo.

BUSINESS PROCESS (PROCESSO AZIENDALE)

Processo aziendale (BP):

- Un BP è un insieme di attività logicamente correlate che vengono eseguite in modo coordinato
- Queste attività eseguono congiuntamente un obiettivo aziendale
- Un BP è costituito da eventi, attività e punti decisionali
- Il risultato ha valore per almeno un cliente di BP

Flusso di lavoro o processo aziendale automatizzato:

- Un processo automatizzato in tutto o in parte da un sistema software
- Il sistema software trasferisce le informazioni da un partecipante all'altro in base alle dipendenze temporali e logiche nel modello di processo

BUSINESS PROCESS INTERACTIONS (INTERAZIONI CON I PROCESSI AZIENDALI)

Ogni BP è eseguito da una singola organizzazione, ma può richiedere un qualche tipo di interazione con BP di altre organizzazioni.

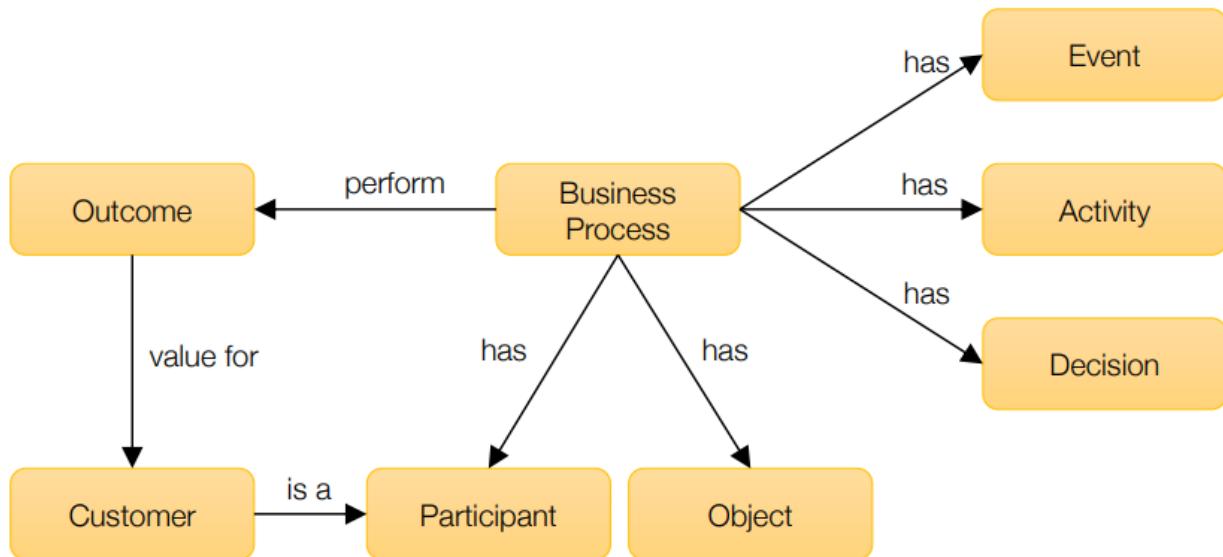
Orchestrazione dei processi aziendali (business process orchestration)

- I BP eseguiti all'interno dell'organizzazione possono essere controllati in modo centralizzato
- Simile a un'orchestra con il suo direttore d'orchestra

Coreografia dei processi aziendali (business process choreography)

- BP che richiede l'interazione con BP di altre organizzazioni
- Sincronizzazione tramite passaggio di messaggi
- Simile ai ballerini che devono eseguire una propria danza secondo una coreografia comune

CONCETTI CHIAVE DEL BUSINESS PROCESS



BPM(BUSINESS PROCESS MANAGEMENT)

Una migliore comprensione delle operazioni dell'organizzazione e delle loro relazioni può essere attuata dalla rappresentazione esplicita dei processi aziendali.

Il BPM è un approccio incentrato sui processi per migliorare le prestazioni che combina le tecnologie dell'informazione con le metodologie di processo e di governance.

Il BPM include concetti, metodi, tecniche, strumenti per supportare tutte le fasi del ciclo di vita della BP.

Altre discipline riguardano il miglioramento delle prestazioni organizzative:

- Total Quality Management (TQM)
 - o Migliorare e sostenere continuamente la qualità dei prodotti e dei servizi
 - o Concentrarsi sul prodotto e sui servizi, non sui processi

- Lean Production
 - o Eliminazione delle attività di spreco che non aggiungono valore al cliente
 - o Basso utilizzo della tecnologia dell'informazione
- Six Sigma
 - o Minimizzazione dei difetti (errori) con forza utilizzando la misurazione dell'output (soprattutto in termini di qualità)

BPM ottiene il continuo approccio di miglioramento del TQM, utilizza i principi e le tecniche del Lean e del Six Sigma e li combina sfruttando le capacità dell'information technology.

STAKEHOLDERS

Con il Business Process Management vengono prodotti diversi artefatti

- Ogni artefatto ha un diverso ambito e livello di astrazione in base agli stakeholder specifici

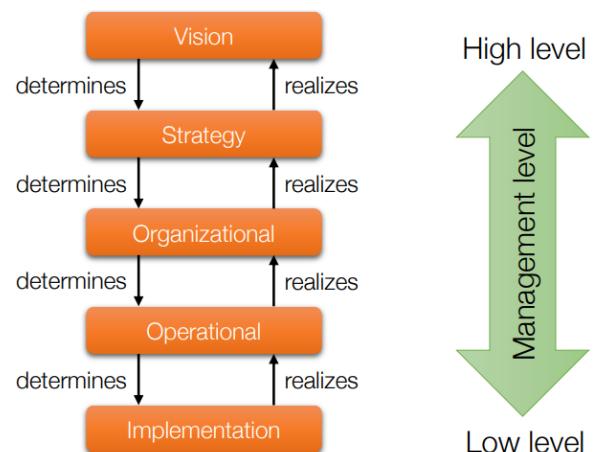
Gli stakeholder possono essere raggruppati in ruoli diversi.

Ruoli degli stakeholders:

- Chief Process Officer (Responsabile del processo)
 - o Ruolo dirigenziale di alto livello
 - o Gestisce a livello globale i BP dell'azienda
- Business Engineer (Ingegnere aziendale)
 - o Esperto di dominio
 - o Definisce gli obiettivi strategici dei BP
- Process Designer (progettista di processi)
 - o Modelli BP che interagiscono con altri stakeholder
- Responsabile del processo
 - o Ha il compito di garantire la corretta esecuzione del BP
- Architetto di sistema
 - o Imposta i sistemi informativi per l'attuazione del BP
- Sviluppatori
 - o Professionisti IT che implementano il BP nei sistemi
- Partecipanti e lavoratori del processo
 - o Eseguire le attività assegnate nell'istanza BP

BUSINESS PROCESS AND MANAGEMENT

Esistono diversi tipi di processi aziendali a seconda del livello di gestione coinvolto.



BUSINESS PROCESS LEVELS

Processi aziendali organizzativi

- Processi di alto livello
- Definire le funzionalità aziendali senza dettagli tecnici
- Generalmente definito in forma testuale con l'uso di tecniche semi-formali (semplici diagrammi)

Processi operativi aziendali

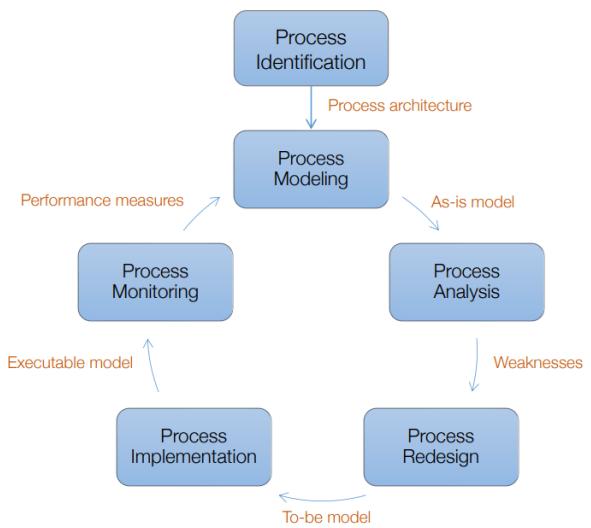
- Definizione dei modelli BP
- Specificazione delle attività e delle relazioni

Processi aziendali implementati

- Includere informazioni tecniche per mettere in atto il BP su una piattaforma specifica

BUSINESS PROCESS LIFECYCLE

Il ciclo di vita della BP è costituito da fasi correlate tra loro secondo un modello di sviluppo a spirale. Ogni fase è composta da diverse attività con dipendenze logiche tra loro. Sono possibili anche approcci incrementali ed evolutivi per implementare il ciclo di vita della BP.



IDENTIFICAZIONE DEL PROCESSO

Il primo passo per il team BPM è l'identificazione dei processi rilevanti e delle relazioni tra di essi (architettura di processo).

Per determinare quale processo è rilevante, è necessario misurare il valore fornito da esso. Le misure di performance del processo devono essere chiaramente definite:

- Misure relative ai costi
- Misure relative al tempo
- Misure relative alla qualità (tassi di errore)

L'identificazione del processo si compone di due fasi successive: Designazione e Valutazione. Nessuna di queste fasi include lo sviluppo di modelli di processo dettagliati.

DESIGNATION PHASE (FASE DI DESIGNAZIONE)

Ottenerne una comprensione dei processi dell'organizzazione e delle loro interrelazioni.

Il numero di processi identificati nella fase di designazione deve essere un giusto compromesso:

- Pochi processi: grande portata per ciascuno di essi (ogni processo ha un gran numero di operazioni) quindi una riprogettazione dei processi potrebbe avere un grande impatto sull'efficacia dell'organizzazione, ma la riprogettazione dei processi è più complessa
- Molti processi: poco spazio per ciascuno di essi, l'impatto di una riprogettazione è minore ma è più facile.

EVALUATION PHASE (FASE DI VALUTAZIONE)

I processi dovrebbero essere prioritari perché non sono ugualmente rilevanti: i processi che potrebbero creare perdite o rischi devono essere analizzati per il consolidamento, lo smantellamento o l'eliminazione.

Criteri per la fase di valutazione:

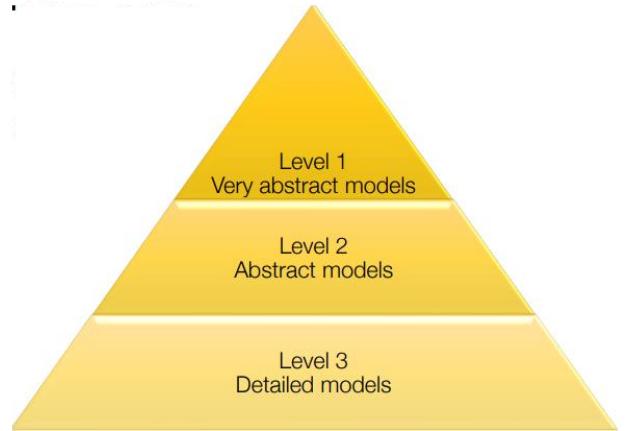
- Importanza: selezionare i processi che hanno il maggiore impatto sugli obiettivi strategici dell'azienda
- Disfunzione: determinare i processi che sono nei guai più seri perché questi processi traggono il massimo profitto dalle iniziative BPM
- Fattibilità: il BPM dovrebbe concentrarsi su processi idonei da cui è ragionevole attendersi benefici

PROCESS ARCHITECTURE (ARCHITETTURA DI PROCESSO)

L'architettura di processo è un modello concettuale che mostra i processi e le relazioni tra i processi.

Le relazioni tipiche sono consumatore-produttore:

- l'output di un processo è l'input per un altro processo
- Un'architettura di processo definisce diversi livelli di dettaglio per le relazioni prodotto-consumatore
- La sfida più importante è catturare i processi al primo livello



PROCESS ARCHITECTURE DEFINITION (DEFINIZIONE DELL'ARCHITETTURA DI SISTEMA)

L'architettura di processo può essere definita considerando i seguenti due aspetti:

1) Tipo di caso (case type):

- Classificazione dei casi gestiti da un'organizzazione
- Un caso può essere un prodotto o un servizio consegnato da un'organizzazione ai propri clienti

2) Funzione

- Scomposizione dell'organizzazione
- Una funzione è qualcosa che l'organizzazione fa (acquisti, produzione, vendite)

Quando vengono identificati casi e funzioni, è possibile comporre una matrice: un segno nella cella indica che la funzione corrispondente può essere eseguita per il tipo di caso corrispondente.

Nella fase finale viene selezionata quale combinazione di funzioni aziendali e tipi di caso formano un processo aziendale.

		Case type			
		C1	C2	C3	C4
Functions	F1	X	X	X	
	F2			X	X
	F3	X	X	X	X

MODELING (MODELLAZIONE)

La modellazione è un modo per gestire la complessità che si basa su:

- Comprendere il comportamento del mondo reale
- Identificare e prevenire i problemi

Un modello è una rappresentazione astratta con l'intento di catturare aspetti specifici rispetto alle seguenti proprietà:

- Mappatura: il modello riflette gli aspetti chiave del mondo reale
- Astrazione: il modello cattura solo gli aspetti rilevanti per l'ambito della modellazione, astraendo da quelli irrilevanti
- Scopo: il modello dipende dal pubblico di destinazione e da due scopi principali per la modellazione dei processi

PROCESS MODELING (MODELLAZIONE DEI PROCESSI)

La modellazione dei processi (o scoperta dei processi o progettazione dei processi) mira a comprendere in dettaglio il processo aziendale. I risultati della modellazione dei processi sono uno o più modelli di processo as-is che documentano lo stato attuale di ogni processo rilevante. I modelli di processo as-is riflettono la comprensione del lavoro che le persone nell'organizzazione hanno. I modelli sono spesso diagrammi (diagrammi di flusso) al fine di ridurre l'ambiguità del testo in formato libero, il diagramma deve utilizzare una notazione che sia compresa da tutte le parti interessate; è consentita una descrizione testuale complementare.

La modellazione dei processi può essere eseguita in quattro fasi:

1. Impostazione del team: assemblare un team in un'azienda responsabile del lavoro sul processo
2. Raccolta di informazioni: costruire una comprensione del processo
3. Modellazione: creare il modello di processo

4. Garantire la qualità: garantire che il modello di processo soddisfi i criteri di qualità

IMPOSTAZIONE DEL TEAM – RUOLI DI MODELLAZIONE DEI PROCESSI

In generale, nella modellazione dei processi sono coinvolti due ruoli:

1) Analisti di processo (process analysts)

- Hanno familiarità con i linguaggi di modellazione e i diagrammi
- Generalmente hanno una comprensione limitata del processo da modellare
- Dipendono dalle informazioni fornite dagli esperti del settore

2) Esperti del dominio (domain experts)

- Avere una profonda conoscenza di come viene eseguito un processo
- Potrebbero essere partecipanti al processo, un proprietario di processo o un manager
- Non hanno familiarità con i linguaggi di modellazione

Gli analisti di processo e gli esperti di dominio hanno ruoli complementari:

Skills	Process Analysts	Domain Expert
Modeling	Hi	Low
Process knowledge	Low	Hi

RACCOLTA DI INFORMAZIONI

È possibile identificare tre metodi di tecniche di scoperta per raccogliere informazioni

- 1) Basato sull'evidenza (Evidence): analisi dei documenti, osservazione o scoperta automatica del processo
- 2) Basato su interviste (Interview): forward o backward
- 3) Basato su workshop (Workshop)

I metodi differiscono l'uno dall'altro in termini di obiettività, ricchezza, consumo di tempo e immediatezza del feedback.

	Evidence	Interview	Workshop
Objectivity	high	medium	high
Richness	medium	high	high
Time Consumption	low	medium	medium
Immediacy of Feedback	low	high	high

METODO DI MODELLAZIONE

Un processo può essere modellato secondo un metodo con cinque fasi

- 1) Identificare i confini del processo: definire l'ambito del processo identificando gli eventi che lo innescano e i possibili esiti del processo
- 2) Identificare le attività e gli eventi: definire, senza ordine, le attività principali e gli eventi intermedi
- 3) Identificare le risorse e i loro passaggi di consegne: definire chi è responsabile delle attività
- 4) Identificare il flusso di controllo: quando e perché vengono eseguite le attività e gli eventi ; definire l'ordine dipendenze, punti decisionali, ripetizioni
- 5) Identificare elementi aggiuntivi: estendere il modello con artefatti (documenti, dati di input, dati di output, database) e gestori di eccezioni

MODELLAZIONE DEI PROCESSI AZIENDALI & MOF

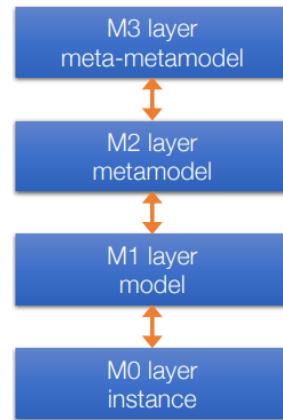
Nella modellazione, la complessità dell'istanza può essere gestita utilizzando diversi livelli di astrazione: al livello inferiore c'è l'istanza ; ai livelli superiori i concetti sono sempre più astratti.

Una delle strutture di astrazione gerarchica più utilizzate è lo standard MOF (Model Object Facility) fornito da Object Management Group (OMG).

PANORAMICA SU MOF

L'architettura di Meta Object Facility si basa su 4 livelli:

- 1) Un metamodello è composto da entità e connessioni
- 2) Una sequenza è una relazione temporale tra due attività
- 3) Il processo è la sequenza dell'attività A e poi dell'attività B
- 4) Processo aziendale reale



MOF & BPMN

Utilizzando lo stesso meta-metamodello M3 (MOF), è possibile definire diversi metamodelli M2.

Secondo il MOF, OMG ha definito il metamodello M2 Business Process Model and Notation (BPMN).

BPMN utilizza una notazione grafica per progettare e documentare i processi aziendali.

ANALISI DEI PROCESSI (PROCESS ANALYSIS)

identificazione e analisi delle problematiche dei processi: Comprendere le principali cause degli esiti negativi permette di identificare il modo migliore per affrontare tale problematica. L'output dell'analisi è una raccolta strutturata di problematiche: le problematiche devono essere quantificate utilizzando misure di performance.

MISURE DI PERFORMANCE (PERFORMANCE MEASURES)

Generalmente le prestazioni di processo sono misurate rispetto ai seguenti tre aspetti: tempo, costo, qualità.

Ogni aspetto di prestazione può essere perfezionato in diverse misure di performance di processo (Key Performance Indicator - KPI). KPI è la quantità che può essere calcolata in modo univoco a partire dalle informazioni di processo.

Il tempo è la misura delle prestazioni utilizzata più di frequente. Il tempo di esecuzione del processo può essere misurato in termini di:

- Tempo di ciclo o tempo di produttività: tempo impiegato per gestire un caso dall'inizio alla fine ; generalmente considerato per la riprogettazione del processo
- Tempo di elaborazione o tempo di servizio: tempo impiegato dalle risorse (ad esempio partecipanti al processo o applicazioni software richiamate dal processo) per la gestione del caso
- Tempo di attesa: tempo trascorso in modalità inattiva ; il tempo di attesa include il tempo di accodamento (attesa di una risorsa), in attesa di sincronizzazione, in attesa di input da parte di un altro partecipante.

MISURA DEI COSTI (COST MEASURE)

Per misurare i costi, questi possono essere scomposti in

- Costi fissi: costi che non sono influenzati dall'intensità del trattamento.
Ad esempio, l'uso di infrastrutture o la manutenzione dei sistemi informativi
- Costi variabili: costi correlati a una quantità variabile.
Ad esempio, il livello delle vendite, il numero di beni acquistati, il numero di nuove assunzioni.

MISURA DI QUALITÀ (QUALITY MEASURE)

La qualità può essere vista dal punto di vista del cliente e del partecipante, quindi la qualità può essere scomposta in:

- Qualità esterna
 - o Soddisfazione del cliente in relazione al prodotto o al processo

- Soddisfazione del prodotto in termini di specifiche o raggiungimento delle aspettative
- Soddisfazione del processo in termini di come il processo viene eseguito
- Qualità interna
 - Qualità correlata al punto di vista del partecipante
 - Livello di controllo in un'attività, quantità di variazioni nel flusso di processo

TECNICHE DI MISURAZIONE DELLE PRESTAZIONI (PERFORMANCE MEASURE TECHNIQUES)

- Analisi dei flussi: permette di stimare le prestazioni complessive di un processo dato lo svolgimento delle sue attività
- Teoria delle code: tecniche matematiche per analizzare i sistemi che hanno contesa di risorse
- Simulazione: la simulazione esegue diverse istanze virtuali di un processo registrando informazioni su ogni fase dell'esecuzione. Produce statistiche relative ai tempi di ciclo, ai tempi medi di attesa e all'utilizzo medio delle risorse

RIPROGETTAZIONE DEL PROCESSO (PROCESS REDESIGN)

Identificazione e analisi dei rimedi per i problemi: sono possibili più opzioni per affrontare un problema ; i possibili rimedi vengono confrontati in termini di misure di performance scelte.

Una volta compresi i problemi e identificati i potenziali rimedi, viene proposta una versione riprogettata (to-be) del processo. Il processo to-be è l'output principale della fase di riprogettazione del processo. L'analisi e la riprogettazione sono strettamente correlate (per ogni opzione di modifica in fase di riprogettazione, deve essere eseguita un'analisi).

SFIDA DI RIPROGETTAZIONE DEI PROCESSI (PROCESS REDESIGN CHALLENGE)

Cambiare un processo non è facile: i lavoratori cambiano con difficoltà perché sono abituati a lavorare in un certo modo, i cambiamenti potrebbero richiedere la modifica dei sistemi informativi con costi associati , i cambiamenti potrebbero interessare anche altre organizzazioni oltre a quella che coordina il processo.

IMPLEMENTAZIONE DEL PROCESSO (PROCESS IMPLEMENTATION)

Nell'implementazione del processo vengono eseguite le modifiche necessarie per passare dal processo as-is al processo to-be.

L'implementazione del processo coinvolge due aspetti:

- Gestione del cambiamento organizzativo: Insieme delle attività necessarie per cambiare il modo di lavorare dei partecipanti al processo
- Automazione dei processi: sviluppo e implementazione di sistemi IT per l'esecuzione del processo to-be

GESTIONE CAMBIAMENTO ORGANIZZATIVO (ORGANIZATIONAL CHANGE MANAGEMENT)

- Spiegare i cambiamenti ai partecipanti al processo: è necessario spiegare quali cambiamenti vengono introdotti e quali sono i benefici.
- Definire un piano di gestione del cambiamento: Quando i cambiamenti avranno effetto e Come gestire la transizione.
- Formazione: insegnare il nuovo modo di lavorare e monitorare i cambiamenti.

MONITORAGGIO DEL PROCESSO (PROCESS MONITORING)

Monitoraggio e analisi dei dati rilevanti raccolti sul processo per identificare possibili aggiustamenti: stato del processo aziendale , eccezioni , tempi di esecuzione , utilizzo delle risorse. La gestione di un processo è uno sforzo continuo.

Potrebbero essere necessari aggiustamenti per soddisfare le aspettative dei processi aziendali in termini di misure di performance e obiettivi di performance: rilevamento di colli di bottiglia, errori ricorrenti o deviazioni dal comportamento previsto ; identificazione dei rimedi.

COMPONENTI DEL LINGUAGGIO DI MODELLAZIONE (MODELING LANGUAGE COMPONENTS)

Un linguaggio di modellazione è costituito da tre parti:

- Sintassi =insieme di elementi di modellazione e regole per combinarli (la sintassi BPMN include attività, eventi, gateway, flussi di sequenza)
- Semantica = associa elementi sintattici con descrizioni testuali a un significato preciso (comportamento degli elementi BPMN)
- Notazione = definisce i simboli grafici per gli elementi

BPMN

Sta per Business Process Model and Notation ed è uno standard OMG, il suo scopo è quello di fornire una notazione per descrivere il processo aziendale che sia comprensibile da analisti BP, sviluppatori IT, lavoratori e manager BP.

Perché BPMN? Le business people sono molto a loro agio con la visualizzazione dei processi aziendali in formato diagramma di flusso. La semantica di esecuzione BPMN è completamente formalizzata (BPMN 2.0 ha una definizione formale (metamodello)) con definizione precisa dei costrutti e delle regole per la creazione dei modelli.

METAMODELLO BPMN (BPMN METAMODEL)

La metamodellazione ha i seguenti vantaggi:

- Formalizzazione di modelli ed entità
- Formalizzazione della relazione tra gli elementi
- Interoperabilità

Non è necessario che il modellatore gestisca il metamodello.

È lo strumento di modellazione che garantisce che il modello sia conforme al metamodello.

SEMANTICA BPMN

Per descrivere come si comportano gli elementi BPMN, viene utilizzato il concetto teorico di token:

- Il comportamento "simulato" degli elementi può essere definito descrivendo come interagiscono con un token
- Il token non fa parte della specifica BPMN
- Gli strumenti di modellazione BPMN non sono necessari per implementare alcuna forma di token

Il token attraverserà i flussi di sequenza e passerà attraverso gli elementi del processo. La sequenza di attraversamento del token scorre istantaneamente, quindi non c'è tempo associato ad essi. Quando arriva a un elemento, il token può continuare istantaneamente o può essere ritardato a seconda . Usiamo questa notazione per i token:



CONTINUA ULTIMO PDF PER EVENTUALE TESI...