Appunti Ingegneria del software A (rel.13/11/12)

1. Che cos'è l'ingegneria del software?

L'Ingegneria del software è un insieme di teorie, metodi e strumenti, sia di tipo tecnologico che organizzativo, per lo sviluppo, l'operatività, la manutenzione e l'uscita di scena del software di qualità, il tutto entro costi e tempi preventivati.

2. SD: Dare la definizione di processo

Un processo definisce chi fa cosa, quando farlo e come raggiungere un certo risultato.

3. SD: Che cos'è un Processo di sviluppo software?

Un processo di sviluppo software è un insieme strutturato di attività richieste per sviluppare un sistema software. E' un processo intellettuale e creativo.

4. SD/ Modello a cascata: elencare i punti che lo contraddistinguono

- Definizione e analisi e dei requisiti
  Si esegue lo studio di fattibilità e, se superato, l'analisi dei requisiti
- Progettazione del sistema e del software
  Si progetta il sistema e poi si procede alla progettazione della struttura del software (in modo che possa essere facilmente tradotta in moduli eseguibili)
- Programmazione e test dei moduli
  Si scrivono i componenti software ex-novo o riciclando codice, poi si testa che siano aderenti alle specifiche
- Integrazione e test del sistema
  In questa fase si esegue l'integrazione dei componenti, si eseguono test di integrazione, si ricontrollano i requisiti del sistema dopo di che si può consegnare al cliente
- Installazione e manutenzione
  In questa fase c'è l'installazione presso il cliente, la manutenzione (debug) la gestione della possibile evoluzione derivata da requisiti consequenziali e il ritiro (phase-out).

Il principale difetto del modello a cascata è la difficoltà ad apportare modifiche al processo una volta avviato. Ogni fase deve essere completata prima di passare alla successiva. Le specifiche devono essere ben redatte e condivise perché il modello abbia successo.

5. SD: Che cosa si intende per Agile Software Development?

E' un metodo di progettazione del software che promuove e privilegia l'uso di iterazioni di sviluppo. Il progetto del software è infatti basato su unità chiamate iterazioni che durano solitamente da 1 a 4 settimane.

Privilegia la comunicazione faccia a faccia a quella scritta e la collaborazione e comunicazione tra le persone dei vari team di sviluppo.
Viene enfatizzato il software come metodo primario di misurazione dello stato di avanzamento dei lavori, producendo poca documentazione rispetto ad altri processi. Ogni iterazione è un intero progetto software che include pianificazione, analisi delle specifiche, progettazione, programmazione, collaudo e documentazione e che tende a produrre una release funzionante, anche se incompleta.
Al termine di ogni iterazione il team di sviluppo rivaluta le priorità del progetto.

I capisaldi dell'Agile manifesto sono:

- Soddisfazione del cliente tramite veloci e continue emissioni di software utile (emissioni settimanali piuttosto che mensili)
- Il software funzionante è il metodo principale di misura del progresso
- Anche i cambiamenti di specifica dell'ultima ora sono i benvenuti
- Intima e giornaliera cooperazione tra commerciali e sviluppatori
- Quella "Face-to-Face" è la migliore forma di comunicazione
- I progetti sono costruiti attorno ad individui motivati di cui si dovrebbe avere fiducia
- Continua attenzione all'eccellenza tecnica e alla buona progettazione
- Semplicità
- Team auto-organizzanti
- Costante adattamento a circostanze che cambiano (flessibilità)

6. SD: Che cosa si intende per sviluppo iterativo e quali sono pro e contro?

E' un procedimento che ha come obiettivo ottenere qualcosa di funzionante nel minor tempo possibile. L'implementazione iniziale è perfezionata finché il sistema non è completo. Le tecniche sono:

- Vaporware: es. implementare una interfaccia utente "finta"
- Componenti usa e getta.
- Moduli dummy.
- Prototipazione rapida tramite RAD (Rapid Development) tools
- Rifinitura incrementale.

I pro dello sviluppo iterativo sono:

- Successo nello sviluppo di sistemi interattivi di piccola e media grandezza.
- Successo nello sviluppo di parti per grandi sistemi (es. interfaccia utente).
- Successo nello sviluppo di sistemi a breve ciclo di vita.

I contro dello sviluppo iterativo sono:

- Mancanza di visibilità sul processo
- I sistemi sono spessi mal strutturati
- Skill speciali sono spesso richiesti (es. per la rapida prototipazione)

7. SD: Descrivere Extreme Programming (XP)

E' forse il modello agile più conosciuto e utilizzato: i vari scenari sono rappresentati da storie degli utenti suddivise ed implementate come una serie di azioni (task). I

programmatori lavorano in coppie e sviluppano test per ogni task prima di scriverne il codice (verifica continua del programma durante lo sviluppo per mezzo di programmi di test) e tutti i test vengono rieseguiti quando del nuovo codice è integrato nel sistema.

Frequente reingegnerizzazione del software, di solito in piccoli passi incrementali, senza dover rispettare fasi di sviluppo particolari.

E' basato su quattro attività:

a. Pianificazione.
b. Progettazione.
c. Sviluppo.
d. Testing.

Approfondendo ulteriormente il tema (forse oltre gli obiettivi dell'esame): le dodici regole (o pratiche) che sono alla base di Extreme Programming possono essere raggruppate in quattro aree:

- Feedback a scala fine

    1. Pair Programming - tutto il codice viene prodotto da due programmatori che lavorano insieme su una sola workstation.

    2. Planning Game - è una riunione di pianificazione che avviene una volta per iterazione, tipicamente una volta a settimana.

    3. Test Driven Development - i test unitari vengono scritti prima di scrivere il codice. Test funzionali e unitari.

    4. Whole Team - in XP, il "cliente" non è colui che paga il conto, ma la persona che realmente utilizza il sistema. Il cliente deve essere presente e disponibile a verificare (sono consigliate riunioni settimanali o Jour fixe).

- Processo continuo

    Continuous Integration - Integrare continuamente i cambiamenti al codice eviterà ritardi più avanti nel ciclo del progetto, causati da problemi d'integrazione.

    Refactoring o Design Improvement - riscrivere il codice senza alterarne le funzionalità esterne, cambiando l'architettura, in modo da renderlo più semplice e generico.

    Small Releases - consegna del software avviene tramite frequenti rilasci di funzionalità che creano del valore concreto.

- Comprensione condivisa

    Coding Standards - Scegliere ed utilizzare un preciso standard di scrittura del codice. Questo rappresenta un insieme di regole concordate, che l'intero team di sviluppo accetta di rispettare nel corso del progetto.

Collective Code Ownership - significa che ognuno è responsabile di tutto il codice; quindi contribuisce alla stesura chiunque sia coinvolto nel progetto.

Simple Design - i programmatori dovrebbero utilizzare un approccio del tipo "semplice è meglio" alla progettazione software. Progettare al minimo e con il cliente.

System Metaphor - descrivere il sistema con una metafora, anche per la descrizione formale. Questa può essere considerata come una storia che ognuno - clienti, programmatori, e manager - può raccontare circa il funzionamento del sistema.

- Benessere dei programmatori

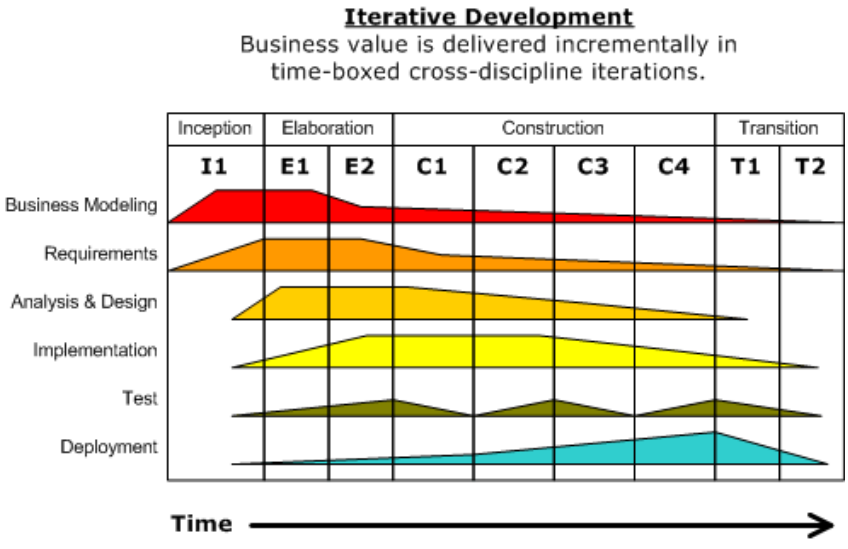
Sustainable Pace - il concetto è che i programmatori o gli sviluppatori software non dovrebbero lavorare più di 40 ore di lavoro settimanali.

8. SD: Descrivere Unified Process (UP)

E' più di un semplice processo, è una struttura estensibile che dovrebbe essere personalizzata all'azienda o al progetto. Le caratteristiche principali sono:

- Iterative and incremental : Le fasi principali (tranne inception) sono divise in iterazioni temporali ; ogni iterazione risulta in un incremento
- Use case driven : bastato su Use Cases e scenari di utilizzo
- Architecture centric: focalizzato sull'architettura del sistema o un insieme di esse
- Risk Focused: tende ad anticipare l'analisi degli aspetti più critici, tipicamente durante le prime iterazione della fase di elaborazione.

9. SD: Descrivere le fasi di Unified Process



- Inception Phase
  L'obiettivo di questa fase è sviluppare una visione approssimata del sistema, identificare le specifiche e gli use cases, delimitare il progetto e gli ambiti e produrre una bozza di cronoprogramma e una stima grossolana dei costi.

- Elaboration Phase
  Affrontare i fattori di rischio, stabilire e validare l'architettura del sistema, produrre use case diagrams, class diagrams e package diagrams.
  Al termine dell'elaborazione l'architettura del sistema deve essersi stabilizzata e deve dimostrare che le funzioni chiave del sistema sono supportate con le performance richieste. Al termine della fase di elaborazione deve essere chiaro come procedere con la Construction Phase.

- Construction Phase
  Durante questa fase, la più corposa del progetto, il resto del sistema è costruito seguendo quando prodotto dalla fase precedente. Il tutto è suddiviso in iterazioni e il risultato di ognuna è una release eseguibile del software. Durante questa fase si usano diagrammi UML come Activity, Sequence, Collaboration, State (Transition) and Interaction Overview.

- Transition Phase
  Durante questa fase il sistema è consegnato agli utenti. I feedback da essi sono inseriti nel sistema tramite iterazioni del processo di transizione. In questa fase è previsto anche il training del cliente e altre operazioni di contorno.

10. SD: *Descrivere il Model Driven Development*

L'obiettivo di MDD è "programmare" ad un livello più alto di astrazione : si riferisce infatti all'uso sistematico di modelli come artefatti ingegneristici primari per tutta la durata della progettazione. Si basa sulla trasformazione di un modello in un programma eseguibile.

I difetti principali di questo modello sono:
- Necessità di skill speciali e formazione supplementare
- Difficoltà nello specificare formalmente l'interfaccia utente

Il vantaggio di questo modello è che:
- E' adatto laddove sono necessarie particolari validazioni di sicurezza prima della messa in servizio

11. SD: *Descrivere il Model Driven Architecture (MDA)*

Pubblicato da OMG, è un approccio "agile" allo sviluppo del software appartenente alla categoria del Domain Engineering.

Secondo questo modello esiste una fase preliminare in cui le specifiche vengono prodotte principalmente a livello testuale, dopo di che inizia il processo iterazionale con:
- L'analisi, che produce un modello CIM/PIM (Computational Independent Model /Platform Indipendent Model)

- Il progetto a basso livello che partendo da CIM/PIM produce il PSM (Platform Specific Model)
- La codifica
- Il testing
- La messa in servizio

12. SD/RE: Cos'è un requisito

Un requisito può spaziare da una dichiarazione astratta ad alto livello di un servizio o di un vincolo di sistema alla dettagliata specifica funzionale di una funzione matematica.
I requisiti possono servire come:

- Base per fare un'offerta (ove devono essere aperti ad interpretazioni).
- Base per il progetto vero e proprio (ove devono essere definiti in dettaglio)

13. SD/RE: Cos'è l'ingegneria dei requisiti

L'ingegneria dei requisiti è un procedimento atto a stabilire e a delimitare i servizi che il cliente richiede ad un sistema e i vincoli sotto cui esso deve operare ed essere sviluppato. In altre parole, ha come obiettivo raccogliere e analizzare la descrizione dei servizi e dei vincoli di un sistema.

14. SD/RE: Elencare le categorie di requisiti

Basati sulle caratteristiche del sistema:

- Requisiti funzionali
- Requisiti non funzionali
- Requisiti di dominio

Basati sulle natura statica/dinamica del sistema:

- Requisiti durevoli
- Requisiti volatili

Basati sulla tipologia di pubblico destinatario dei requisiti

- Requisiti dell'utente
- Requisiti del sistema

15. SD/RE: Parlare dei requisiti funzionali

Requisiti funzionali: rappresentano il comportamento atteso dal sistema. Descrivono come il sistema deve reagire a particolari dati di ingresso e come il sistema dovrebbe comportarsi in situazioni particolari.

16. SD/RE: Parlare dei requisiti non funzionali

Requisiti non funzionali: vincoli sui servizi o sulle funzioni offerte dal sistema. Specificano i criteri che possono essere usati per valutare l'operato del sistema, piuttosto che specifici comportamenti specifici. Vincolano la progettazione e l'implementazione del sistema, e non descrivono un servizio che il sistema dovrebbe fornire.
Sono suddivisi in tre sottotipi:

- Requisiti di prodotto: requisiti che specificano il comportamento del prodotto finale (es. velocità di esecuzione, affidabilità, efficienza, portabilità...).

- Requisiti organizzativi : vincoli derivati da politiche e procedure organizzative dell'azienda (requisiti di consegna, implementazione, standard).

- Requisiti esterni : requisiti che derivano da fattori esterni al sistema e al suo processo di sviluppo (requisiti etici, legislativi, di interoperabilità...).

17. SD/RE: Parlare dei requisiti di dominio

Requisiti di dominio : requisiti derivanti dal dominio di applicazione del sistema e che riflettono le caratteristiche di quel dominio. Possono essere funzionali o non funzionali. Es. Deve esserci un'interfaccia utente uniforme per tutti i database basata sullo standard X

18. SD/RE: Parlare dei requisiti durevoli

Requisiti durevoli : requisiti stabili derivati dall'attività principale dell'azienda del cliente. (Es. un ospedale avrà sempre dei dottori) .
Possono derivare da modelli di dominio.

19. SD/RE: Parlare dei requisiti volatili

Requisiti volatili : requisiti che cambiano durante lo sviluppo o quando il sistema è in funzione. Possono essere divisi in quattro sottotipi:

- Requisiti mutabili: requisiti che cambiano a causa del cambiamento dell'ambiente esterno in cui l'azienda opera. Es. cambia l'ammontare del finanziamento che l'ospedale riceve e può essere necessario cambiare i trattamenti di cura da apportare ai pazienti.

- Requisiti emergenti: requisiti che emergono dallo sviluppo della comprensione del sistema da parte del cliente nel corso dello sviluppo del sistema.

- Requisiti consequenziali: requisiti derivanti dall'introduzione nell'azienda del sistema di computer in oggetto . La novità può generare nuove visioni e idee sul modo di operare dell'azienda stessa.

- Requisiti di compatibilità : requisiti che dipendono dai sistemi o dai modelli di business particolari all'interno dell'azienda che, cambiando o evolvendosi si devono riflettere sui sistemi commissionati o consegnati.

20. SD/RE: Parlare dei requisiti dell'utente

Requisiti dell'utente : descrizione dei servizi del sistema e dei suoi vincoli operazionali espressa in maniera comprensibile anche dagli utenti del sistema che non hanno particolari conoscenze tecniche. Sono definiti usando linguaggio naturale, tabelle e grafici.

21. SD/RE: Parlare dei requisiti del sistema

Requisiti del sistema : specifica dei requisiti degli utenti espressa in modo più dettagliato, rivolta al cliente e agli sviluppatori. Può essere usata come contratto tra cliente e fornitore.

22. SD/RE: In che cosa consiste il documento dei requisiti?

E' un documento che specifica chiaramente e senza ambiguità i requisiti del sistema identificati.
Deve descrivere che cosa il sistema deve fare e non in che modo lo deve fare (ma non è un documento di progettazione).

Se riguarda sia l'hardware che il software si chiama System Specification.
Se riguarda solo il software si chiama Software Requirements Specification (SRS) e può essere conforme allo standard IEEE830 (esiste anche un template IEEE830).

I requisiti dovrebbero essere:

- **Completi** cioè devono includere la descrizione di tutte le funzioni richieste.
- **Consistenti** cioè non devono esserci conflitti o contraddizioni nella descrizione delle funzioni.
- **Comprensibili** cioè comprensibile ai più il che è molto difficile perché gli ingegneri interpretano in un modo mentre gli utenti in un altro.
- **Espliciti** cioè non devono dare nulla per scontato poiché clienti e sviluppatori tendono a dare scontate cose differenti

In pratica è molto difficile o impossibile produrre un documento con queste caratteristiche.

23. SD/RE: Parlare dei conflitti nel documento dei requisiti

Conflitti tra specifiche non funzionali sono comuni nei sistemi complessi: es. in una astronave devo avere pochi chip elettronici per risparmiare peso e averli a basso consumo per risparmiare energia, ma la cosa è controproducente perché poi me ne servono di più vista la bassa potenza di ognuno.

24. SD/RE: Parlare dei problemi del linguaggio naturale

I problemi del linguaggio naturale sono:

- Mancanza di chiarezza
- Confusione nel descrivere i requisiti (requisiti funzionali e non funzionali tendono a mischiarsi tra di loro)
- Amalgamazione (requisiti diversi espressi tutti insieme in modo pasticciato)
- Ambiguità
- Eccessiva flessibilità (ci sono molti modi di dire la stessa cosa)
- Mancanza di modularità

La soluzione è l'adozione di un linguaggio standard, che utilizza parole chiave per i vincoli (shall/should/can) , le azioni dell'utente (must/may) , l'ambiente-environment (will/might) , e il rischio di incontrare modifiche (expected to/could) .

25. SD/RE: Elencare le parti di cui è composto il processo di ingegneria dei requisiti

- Estrazione
- Analisi
- Validazione
- Gestione

26. SD/RE: Descrivere il processo di estrazione (elicitation)

Identifica le sorgenti più rilevanti di requisiti, determina quali informazioni sono necessarie, le elabora e le sintetizza in una frase appropriata. Permette al cliente di capire meglio di quali requisiti ha bisogno e allo sviluppatore di affrontare il problema corretto. Le tecniche di estrazione sono quattro:

- Campionamento → raccoglie esempi di documentazione, registrazioni, moduli e li sceglie o a caso o con un criterio ben preciso (stratificazione).

- Osservazione dell'ambiente di lavoro → osserva le persone al lavoro per capire come funziona il sistema. Occorre essere cauti e cercare di non farsi notare (le persone non è detto che lavorino allo stesso modo se osservate).

- Questionario → documento che permette all'analista di raccogliere informazioni e opinioni dai partecipanti. Può essere di due tipi:
  - Free format → domande a risposta aperta
  - Fixed format → domande a risposta multipla

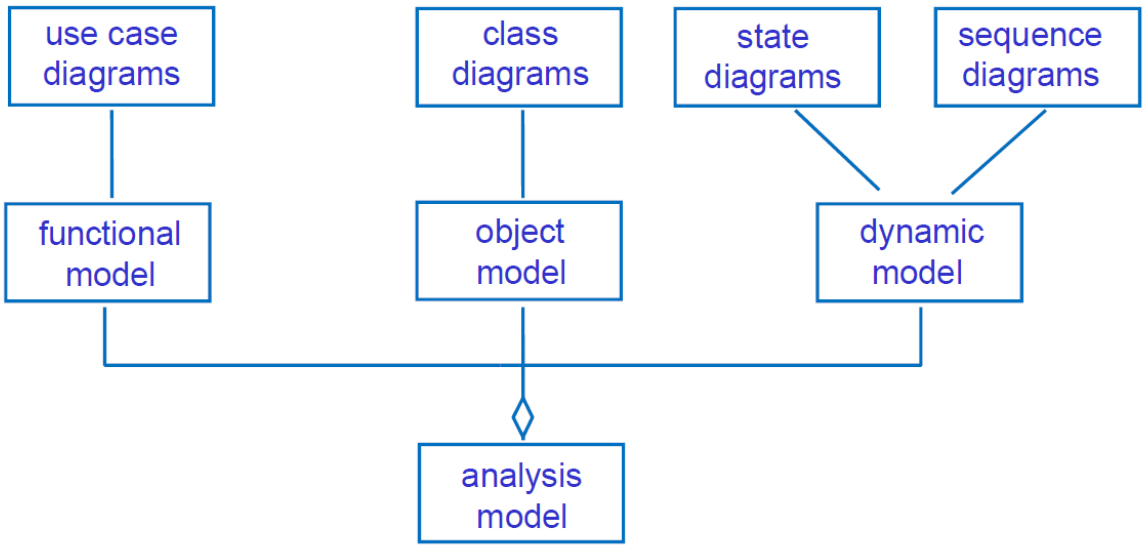
- Intervista → tecnica in cui l'analista colleziona informazioni da un colloquio individuale con i partecipanti. Può essere di due tipi:
  - Non strutturata → poche domande, si lascia all'intervistato la possibilità di dirigere la conversazione.
  - Strutturata → insieme di domande precise da porre all'intervistato.
    Le domande possono essere di due tipi:

- Domande aperte → permettono all'intervistato di rispondere nella maniera che gli sembra più appropriata.
- Domande chiuse → necessitano di risposte specifiche, brevi e dirette.

27. SD/RE: Descrivere il processo di analisi dei requisiti (e l'Analysis Model)

Raffina e struttura i requisiti per facilitarne la comprensione agli sviluppatori, il riuso e la manutenzione.
Porta ad una più precisa e formale specifica dei requisiti e definisce il modello di analisi, tramite la rifinitura del linguaggio informale dei requisiti e la conversione degli stessi in diagrammi di flusso.

- Un modello strutturato di analisi è basato sulle funzioni che devono essere realizzate dal sistema.
- Un modello strutturato orientato agli oggetti è basato su use case e sulle rappresentazioni degli oggetti reali e i concetti del dominio del sistema visibile in figura).



28. SD/RE: Descrivere il processo di analisi: Use Cases

Gli Use cases, detta semplicemente, permettono la descrizione di sequenze di eventi che, presi assieme, portano il sistema a fare qualcosa di utile.
Uno use case è semplicemente una descrizione (una storia) di utenti e sistemi che interagiscono fra di loro in modo da descrivere un processo di business o un processo automatizzato. Sono identificati individuando gli attori e le interazioni tra gli attori e il sistema.
Descrivono l'interazione tra un attore primario, l'iniziatore dell'interazione, e il sistema stesso, attraverso una sequenza di semplici passi.

Sono rappresentati da una combinazione tra use case diagrams, descrizioni testuali e diagrammi di interazione (opzionali).
Ad ogni specifica deve corrispondere almeno uno use case (ma possono essercene più di uno).

29. SD/RE: Descrivere il processo di analisi: linee guida per la costruzione degli Use Cases

Linee guida per l'identificazione degli attori:

- Identificare attori e sistemi che dipendono dalle funzionalità primarie e secondarie del sistema
- Identificare come attori entità esterne con obbiettivi comuni e diretta interazione col sistema
- Identificare gli attori con nomi

Che cosa deve includere uno use case:

- Stato di partenza e precondizioni
- Quando lo use case comiuncia
- Ordine delle attività nel flusso principale degli eventi
- Qualsiasi flusso alternativo degli eventi
- Qualsiasi flusso eccezionale degli eventi
- Post condizioni e stato finale
- Attori coinvolti e use cases inclusi o estesi
- Diagrammi di interazione
- Problemi di progettazione (in un appendice)

Tipi di use cases

- Business/Domani use cases
  Sono le interazioni degli utenti col dominio/business
- System Use Cases
  Sono le interazioni tra il sistema e gli utenti
- Un Business use case contiene un insieme di system use cases

Identificazione dello scenario

- Compiti che devono essere eseguiti dagli utenti e dal sistema
- Condizioni di partenza e gli stati ove finisce
- Flusso informazioni verso l'utente e il sistema
- Eventi convogliati dagli utenti e dal sistema
- Normale flusso di eventi
- Cosa può andare storto
- Altre attività concorrenti

Linee guida per un buon use case:

- Cercare di descrivere gli use cases senza pensare a come verranno implementati
- Essere più esplicativi e narrativi possibile
- Introdurre tutti i possibili scenari di uno use case

- Nominare uno use case partendo con un verbo in maniera da enfatizzare che è un processo (es. comprare articoli, introdurre un ordine , Ridurre l'inventario ecc.ecc.)
- Documentare exception handling o branching (es. "quando l'acquisto articolo fallisce allora…. Oppure quando l'approvazione della carta di credtito fallisce allora ….)
- Non rappresentare step individuali come use cases (es evitare di definire uno use case per "stampare una ricevuta")
- Pochi step (max 5) che portano ad un risultato utile per gli utenti finali
- Descrive interazioni e meccanismi, non politiche
- Nasconde i dettagli implementativi di interfacce utente e di sistema
- Poche pagine(5-10)
- Un solo iniziatore
- Include le eccezioni principali e la loro gestione

30. SD/RE: Descrivere cosa sono le Classi di Analisi

Le classi di analisi sono un'astrazione di una o più classi reali o sottosistemi da realizzare. Le classi di analisi gestiscono i requisiti funzionali.
Le classi di analisi sono di solito rappresentate con diagrammi di classe attraverso stereotipi Boundary, Control ed Entity (le semantiche di questi stereotipi fornisce un metodo coerente per trovare ed analizzare queste classi).
Una buona Classe di analisi fornisce una chiara astrazione di quanto estratto dal dominio del problema, incorpora un piccolo insieme di responsabilità ben definite e ben eseguite, fornisce una chiara separazione tra astrazione, specifiche e implementazione.

Le tecniche per individuare le classi di analisi sono cinque:

- Analisi nome-verbo → i nomi possono identificare classi, attributi e istanze, i verbi operazioni, relazioni e vincoli.
- Guidato dagli use case → similare all'analisi nome-verbo ma centrato sui casi d'uso, identifica gli oggetti e le loro responsabilità e come questi oggetti collaborano tra di loro , si ottiene analizzando i diversi scenari descritti dai casi d'uso.
- Common class patterns → deriva le classi dalla generica teoria di classificazione degli oggetti (si suddividono le classi in base alla tipologia dell'oggetto rappresentato seguendo schemi predeterminati, metodo inefficiente)
- Carte CRC → basata su carte rappresentanti le classi, ognuna suddivisa in tre parti: nome, responsabilità e collaboratori della classe . Una sessione di brainstorming animato aiuta una definizione più precisa delle classi
- Approccio misto: usando tutte o alcune delle tecniche sopra viste

31. SD/RE: Descrivere il processo di validazione

Processo per stabilire e giustificare la credenza che i requisiti documentati corrispondono alle volontà del cliente, degli utenti e di altre parti interessate. Si può fare con tre tecniche:

- Ricontrollo dei requisiti → ricontrolli regolari dovrebbero essere svolti durante la definizione dei requisiti; coinvolge sia il cliente che lo sviluppatore.
- Prototipazione → utilizza dei prototipi (eseguibili) del sistema , creati solo per qusto scopo, per testare la veridicità dei requisiti.

- Generazione di test-case → sviluppo di test per ogni requisito che ne controllano la verificabilità.

32. SD/RE: Descrivere il processo di gestione

E' il processo di gestione del cambiamento dei requisiti durante il processo di ingegnerizzazione dei requisiti e lo sviluppo del sistema (i requisiti infatti sono inevitabilmente incompleti e inconsistenti).

E' responsabilità di questa fase la gestione di:

- Mantenere i costi nel budget
- Stimare il costo del progetto in base ai requisiti
- Gestire la volatilità dei requisiti
- Gestire la configurazione dei requisiti del sistema
- Negoziare il cambiamento dei requisiti e ristimare il costo del sistema in seguito ad esso.

Quando una modifica ai requisiti è richiesta, dovrebbero essere eseguite le seguenti tre azioni:

- Analisi del problema : discutere i problemi derivanti dai requisiti e proporre cambiamenti.
- Cambiamento di analisi e costi : valutare gli effetti dei cambiamenti sugli altri requisiti.
- Cambiamento dell'implementazione : modifica il documento dei requisiti e altri documenti per applicare il cambiamento.

33. SD/RE: Quale è lo scopo dello studio di fattibilità?

Lo studio di fattibilità serve per decidere se il sistema proposto è realizzabile o no. Controlla che il sistema contribuisca agli obiettivi aziendali, sia sviluppabile usando la tecnologia corrente e il budget assegnato e che possa essere integrato con altri sistemi utilizzati.

34. SD/RE: Descrivere i rischi dell'ingegneria dei requisiti

- Difficoltà nell'identificare precisamente i requisiti.
- Incomprensione del dominio o del vero problema
- Difficoltà nel risolvere i conflitti tra requisiti.
- Cambiamento rapido dei requisiti.
- Tendenza a strafare.

35. UML: Cos'è UML?

Unified Modeling Language, linguaggio standard per la visualizzazione, la specifica, la costruzione e la documentazione degli artefatti di un sistema software complesso. Fornisce

diagrammi multipli che consentono di visualizzare diversi tipi di dettaglio dell'architettura. E':

- Semplice, perché richiede solo pochi concetti e simboli
- Espressivo, in quanto è applicabile a un largo spettro di sistemi e metodi.
- Utile, perché si focalizza solo sugli elementi necessari all'ingegneria del software.
- Consistente, perché lo stesso concetto/simbolo deve essere applicato sempre allo stesso modo.
- Estensibile, perché gli utenti e gli sviluppatori hanno una certa libertà di estendere la notazione.

36. UML: Parlare delle specifiche dell'UML (su che cosa è basato UML)

UML è basato su:

- Infrastruttura: composta da costrutti linguistici fondamentali.
- Superstruttura: costrutti a livello utente (e.g. diagrammi UML)
- Object Constraint Language (OCL) : linguaggio formale usato per descrivere con precisione requisiti (in particolare vincoli) che i modelli grafici UML non riescono ad esprimere.
- Interscambio di diagrammi : facilità e trasparenza nella condivisione di documenti che siano UML compliant.

37. UML: Quali sono i diagrammi UML?

Sono diagrammi che permettono di analizzare il sistema da diversi punti di vista:

1) Use Case View
2) Structural View
3) Behavioral View
4) Implementation View
5) Environment View

38. UML/Use Case View: Parlare della Use Case View

Gli Use Case Diagram, i più importanti di UML, sono diagrammi dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso. Descrivono un comportamento dinamico del sistema.

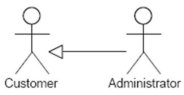
Uno use case descrive la proposta funzionalità di un sistema e rappresenta una unità discreta di interazione tra un utente (umano o macchina) e il sistema. Questa interazione è una unità singola di lavoro utile e può includere una complessa interazione tra le parti.

L'approccio alla modellazione a usecases è di tipo black box; non è importante come i casi d'uso sono implementati nel sistema o come il sistema lavora internamente ma piuttosto capire quali sono le funzionalità che si vogliono dal sistema stesso.
Inoltre si agevola l'individuazione dei requisiti, la scelta delle tecniche di modellazione adeguate e la progettazione dei test del sistema.

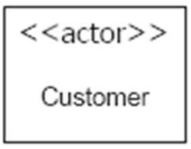
Il diagramma è composto da:

- Il sistema, che è rappresentato da un rettangolo. Il perimetro del rettangolo divide cosa è dentro o fuori al sistema (divide quindi gli attori dagli use case)

- Gli attori sono entità esterne al sistema (utenti o altri sistemi).
  Un attore può generalizzare un altro attore con l'apposita freccia come nel class diagram.



  Una rappresentazione alternativa degli attori può essere fatta con un quadrato come in figura:



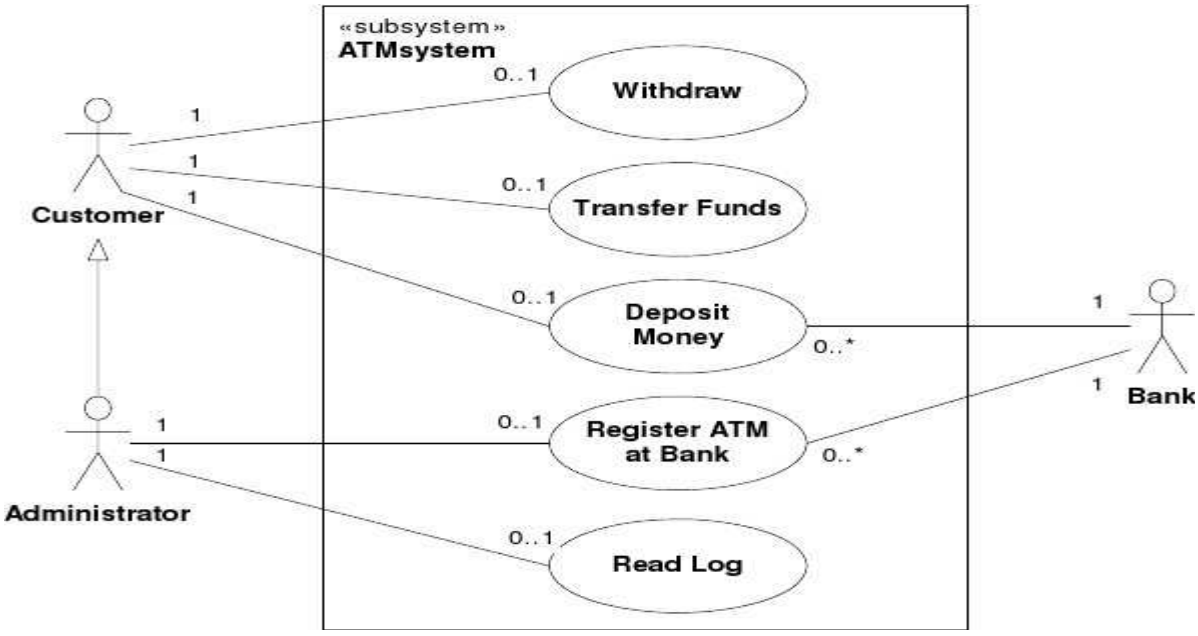
- Gli use case (racchiusi in ovali) . Un use case può racchiudere la funzionalità di un altro use case tramite una linea tratteggiata e la keyword <<includes>>
  In questo caso la funzionalità della classe base necessita dell'altra. Un po' come in una subroutine richiamata dal programma principale.
  In questo esempio viene specificata meglio l'operazione di inclusione

The [...] figure illustrates an e-commerce application that provides customers with the option of checking the status of their orders. This behavior is modeled with a base use case called CheckOrderStatus that has an inclusion use case called LogIn. The LogIn use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. An include relationship points from the CheckOrderStatus use case to the LogIn use case to indicate that the CheckOrderStatus use case always includes the behaviors in the LogIn use case.

Con la parola chiave <<extends>> invece si estende la funzionalità di un use case con un altro use case. In questo caso la funzionalità della classe base viene estesa dall'altra.

You are developing an e-commerce system in which you have a base use case called Place Online Order that has an extending use case called Specify Shipping Instructions. An extend relationship points from the Specify Shipping Instructions use case to the Place Online Order use case to indicate that the behaviors in the Specify Shipping Instructions use case are optional and only occur in certain circumstances.

- Relazioni tra attori e use cases, con la loro molteplicità.



39. UML/Structural View: Da quali diagrammi è composta la Structural View

E' rappresentata da diagrammi statici:

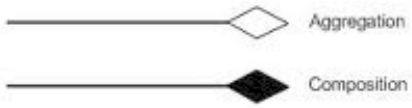
1. Class Diagram
2. Object Diagram
3. Composite Structural Diagram
4. Package Diagram

40. UML/Structural View: Cos'è il Class Diagram

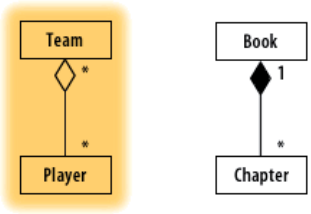
Class diagram è un diagramma che illustra classi e relazioni tra classi. Rappresenta una visione statica del modello descrivendo quali attributi e comportamenti la compongono piuttosto che dettagliarne i metodi.

Mostra i blocchi (classi) che costruiscono un sistema orientato agli oggetti. É composto da classi (rettangoli) o interfacce (<<interface>> prima del nome), suddivise in compartimenti: nome della classe, attributi della classe (+ : visibilità pubblica, - : visibilità privata,# : visibilità protetta, ~ : visibilità package) nella forma "<+/-> <nome> : <tipo>", e dai collegamenti tra le classi, che possono essere:

- **Composizione e Aggregazione** : usati per rappresentare elementi composti a loro volta da componenti più piccoli (paradigma "whole-part", "part-of", "has a" cioè realizzazione di una relazione di contenimento). Ma l'intensità della relazione è più forte nella composizione, ove la terminazione dell'oggetto padre termina anche i figli, cosa che non avviene nel caso dell'aggregazione (la vita degli aggregati non dipende da quella dell'aggregante)

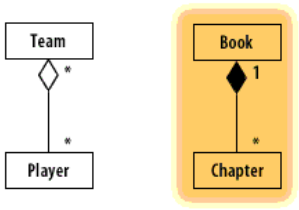


Weak aggregation



Players may particpate in many teams at the same time. For example, they can play for a city league and a company team. When a team is disbanded the players still exist and can be reassigned to other teams.

Strong aggregation, or composition

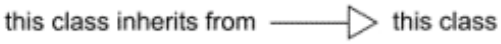


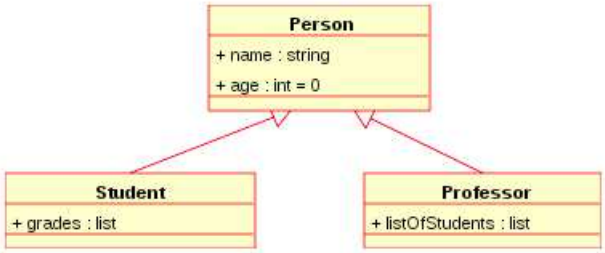
Chapters belong to a specific book. Without the book the chapters have no context and lose their meaning. If the book is deleted the chapters are deleted along with it. It does no good to keep them since they cannot be reassigned or reused in another book.

- **Associazione** : implica una relazione tra due elementi (che di solito è implementata tramite una lista nella classe padre) ma la differenza rispetto all'aggregazione è che qui non è implementato il paradigma "whole/part" . Il simbolo è una riga continua.

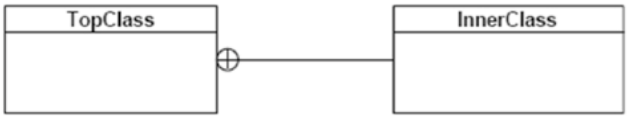
La freccia (che può comparire anche in una aggregazione) indica <u>navigazione</u> (l'oggetto puntato dalla freccia non sa dell'esistenza dell'altro). Es.

- **Generalizzazione** : usata per indicare ereditarietà





- **Nidificazione** : Indica classi interne, ha un simbolo di pallino crociato dalla parte della classe ospitante.



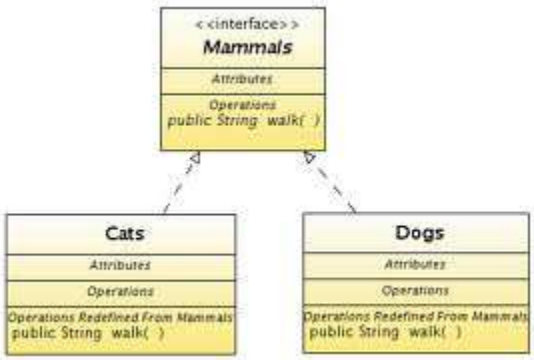
- **Dipendenza** : Talvolta la relazione tra classi può essere molto debole; è una relazione tra un oggetto "cliente" e un oggetto "fornitore".I cambiamenti all'oggetto fornitore impattano anche l'oggetto cliente, da qui la dipendenza.

this class is dependent upon - - - > this class

- **Realizzazione** : E' usata per indicare che una classe implementa un'interfaccia. E' disegnato come una generalizzazione ma con una riga tratteggiata.

this class is a realisation of - - - ▷ this class

Esempio:



41. UML/Structural View: Cos'è l' Object Diagram

Object diagram: Un object diagram è un'istanza di un class diagram, saranno quindi presenti i valori reali degli attributi che nel class diagram non erano specificati. Lo si può paragonare ad una fotografia ad un certo instante temporale del nostro sistema.
Quindi, laddove un class diagram descrive tutte le possibili situazioni, l'Object model descrive una particolare situazione.
Gli oggetti sono rappresentati da rettangoli con intestazione "Object :Class".
Sono poco utilizzati in UML se non per documentare casi di test.

42. UML/Structural View: Cos'è il Composite Structure Diagram

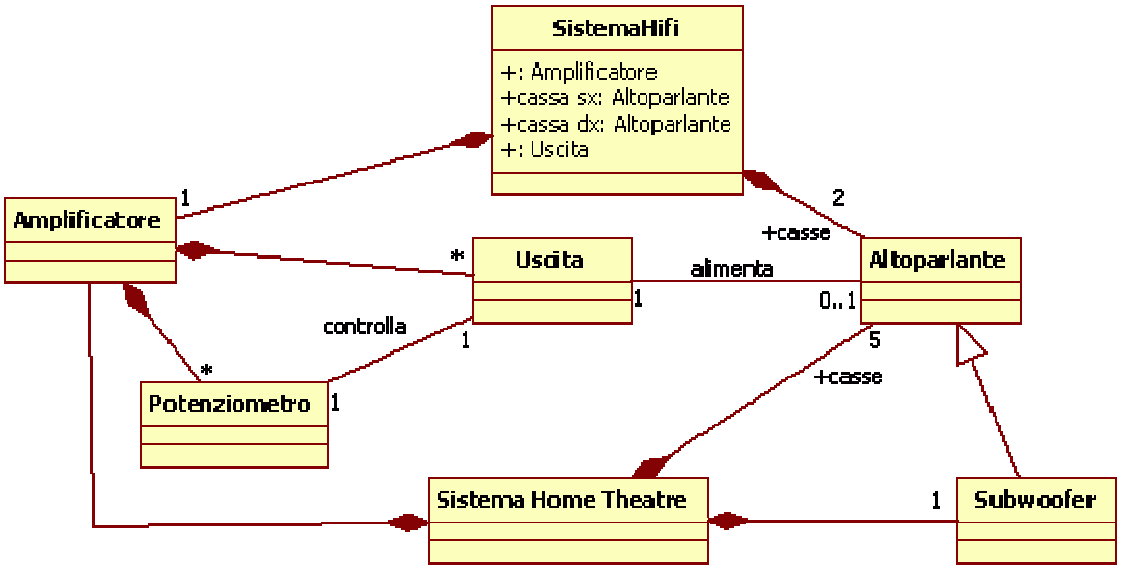
Composite structural diagram : mostra la struttura interna di un "classificatore" che è un elemento UML descritto da parti, porte e connettori ed ha come obiettivo quello di descrivere una funzionalità.

Questi diagrammi sono simili ai Class Diagram eccetto che modellano un'uso specifico della struttura di classe.

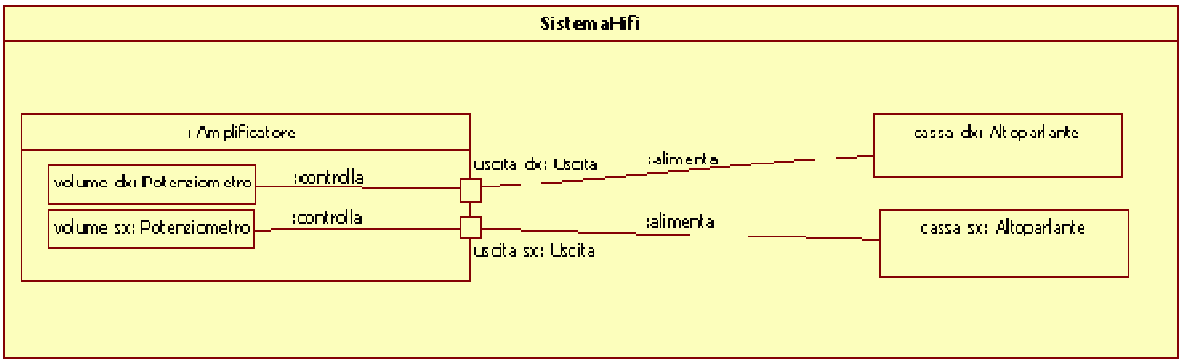
É composto da :

- Un classificatore (rettangolo) cioè una classe o una classe astratta o in generale un componente che può essere descritto da una interazione tra parti
- Parti di una classe, ovvero ruoli giocati a runtime da un'istanza di una classe (rettangoli)
- Porte, ovvero punti di interazione per collegare parti tra loro o con l'ambiente esterno (quadratini)
- interfacce fornite (linee con cerchio verso l'esterno)

- interfacce richieste (linee con mezzaluna verso l'esterno).



Class diagram di un sistema Hifi



Composite Structure Diagram che descrive una funzionalità

43. UML/Structural View: Cos'è il Package Diagram

Package Diagram : è un meccanismo per organizzare elementi in gruppi e serve ad organizzare porzioni di sistema evidenziandone le relazioni e le dipendenze .
Può essere usato in due modi:

- Nell'organizzazione delle fasi di sviluppo, in cui ogni package contiene artefatti corrispondenti ad una fase (es. per organizzare use case diagrams)
- Nell'organizzazione funzionale, in cui ogni package raccoglie un insieme di classi la cui interazione è una funzionalità del sistema.

44. UML/Behavioral View: elencare i diagrammi più importanti

Rappresenta le interazioni dinamiche tra i componenti del sistema per implementare i requisiti. Mostra come sono distribuiti i compiti. É rappresentata dai diagrammi dinamici:

1. Sequence Diagram
2. Communication (Collaboration) Diagram
3. *Robustness Diagram*
4. Activity Diagram
5. State Diagram
6. *Interaction Overview Diagram*
7. Timing Diagram

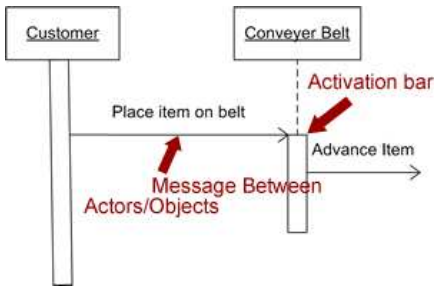
45. UML/Behavioral View: cos'è il Sequence diagram

Sequence diagram : Descrive, per mezzo di un insieme sequenziale di interazioni, come un processo è interpretato da un gruppo di oggetti .

Lo scopo principale è definire sequenze di eventi che risultano in qualche risultato desiderato.
L'attenzione e più sull'ordine nel quale messaggi si susseguono piuttosto che sui messaggi stessi. Il diagramma comunica questa informazione lungo le dimensioni verticali e orizzontali: la dimensione verticale mostra dall'alto verso il basso, la sequenza temporale dei messaggi e delle chiamate mentre la dimensione orizzontale mostra, da sinistra a destra, le istanze degli oggetti ai quali sono indirizzati i messaggi.

Oggetti grafici utilizzati:

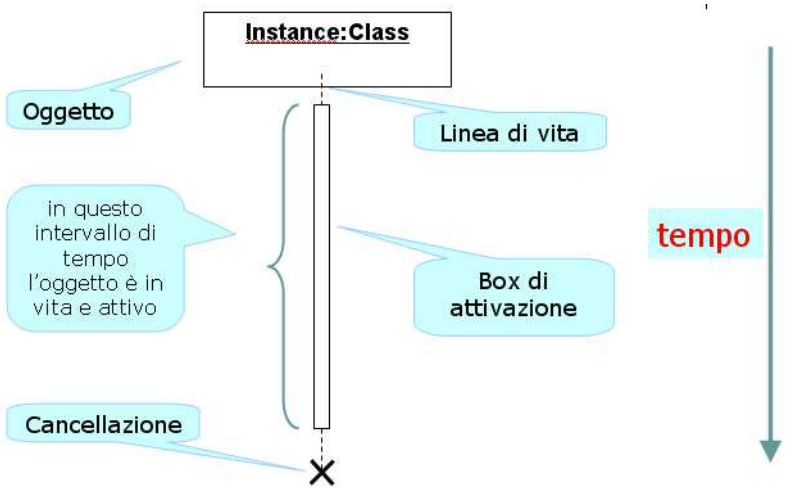
- Ruoli: sono quelli che gli oggetti possono interpretare all'interno dell'interazione (rettangoli in cima alla lifeline)
- Linee di vita (lifelines) : rappresentano l'esistenza di un oggetto all'interno di un arco temporale (linee tratteggiate)
- Attivazioni : rappresentano il tempo durante il quale un oggetto sta interpretando un'operazione (rettangoli stretti verticali)



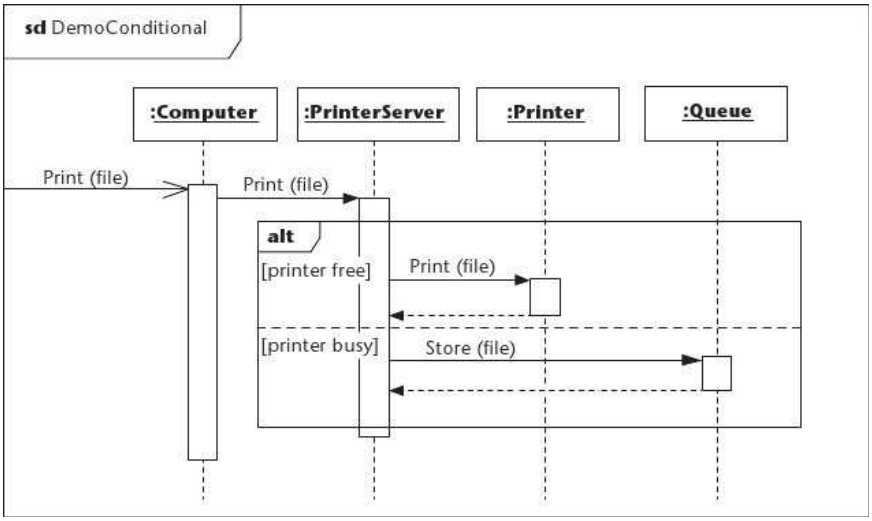
Attivazioni

- Messaggi: comunicazione tra oggetti (chiamate a metodo)
- Fragments
  Un frammento permette la rappresentazione di una complessa procedura logica dentro un sequence diagram. Sono dei rettangoli annegati dentro al diagramma e presentano un etichetta che indica la tipologia di operazione:
    - Alt : Alternative fragment for mutual exclusion conditional logic expressed in the guards.

- Loop : Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as loop(i, 1, 10)
- Opt : Optional fragment that executes if guard is true.
- Par : Parallel fragments that execute in parallel.
- Region: Critical region within which only one thread can run.



Esempio: un computer, ricevuto un comando di stampa lo inoltra al print server. Se la stampante non è occupata, il print server le inoltra il documento da stampare. Altrimenti salva il documento sulla coda di stampa.



Esempio
Parte Centrale del Bubble Sort

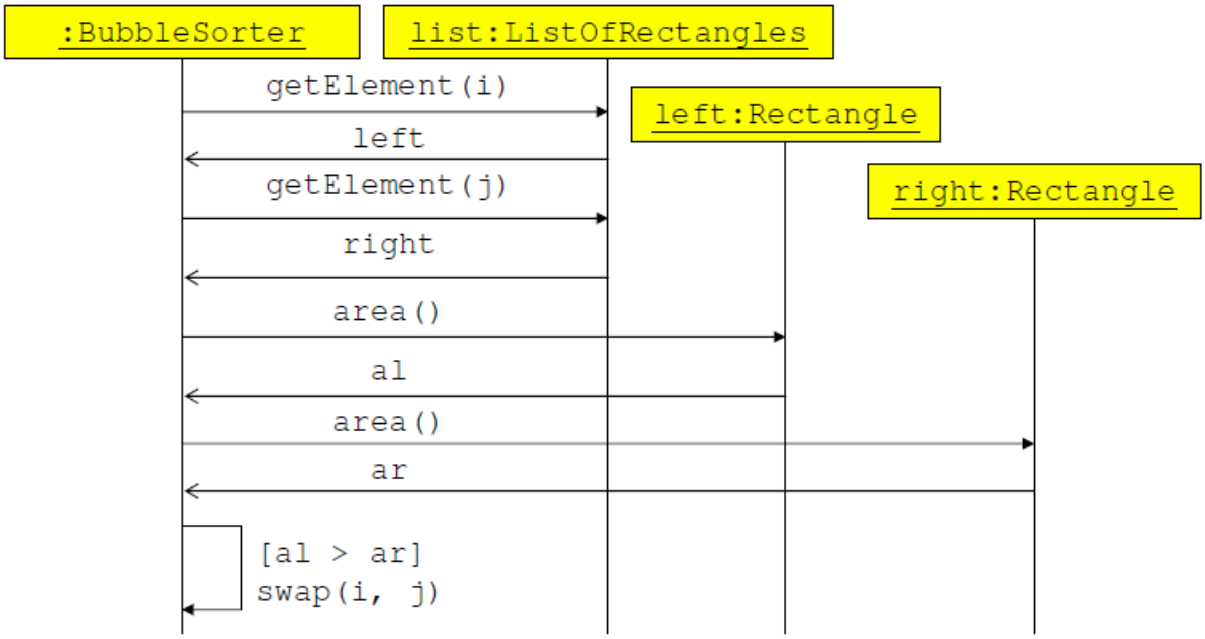
- In Java

```
Rectangle left  = list.getElement(i);
Rectangle right = list.getElement(j);

if(left.area() > right.area()) swap(i, j);
```

- Con uno pseudo-codice

```
Rectangle left  = i-esimo elemento di list;
Rectangle right = j-esimo elemento di list;

int al = area di left;
int ar = area di right;

if (al > ar) swap(i, j);
```

Sequence Diagram



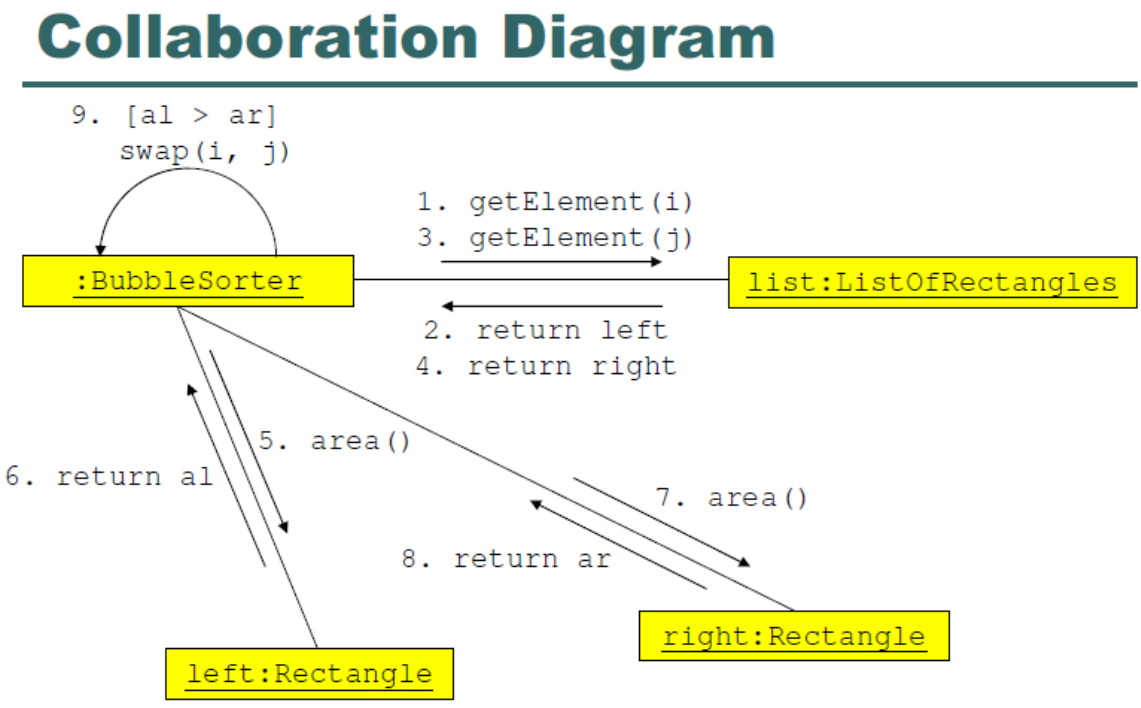
46. UML/Behavioral View: cos'è il Communication (o Collaboration) Diagram

Communication diagram : è un diagramma di interazioni che mostra informazioni similari al Sequence Diagram ma più focalizzato sulle relazioni tra gli oggetti. NON modellano infatti il flusso temporale dei messaggi MA il flusso di dati e le quantità di messaggi scambiati

- Gli oggetti sono mostrati con "connettori" di associazione tra di loro
- I messaggi sono aggiunti alle associazioni e mostrati come una corta freccia che punta nella direzione del flusso di messaggi
- La sequenza di messaggi è mostrata attraverso uno schema di numerazione

Forniscono una rappresentazione alternativa al Sequence Diagram in un formato basato sulla struttura piuttosto che sul tempo.

Ecco lo stesso esempio del Bubble Sort espresso come Collaboration Diagram



47. UML/Behavioral View: cos'è il Robustness diagram

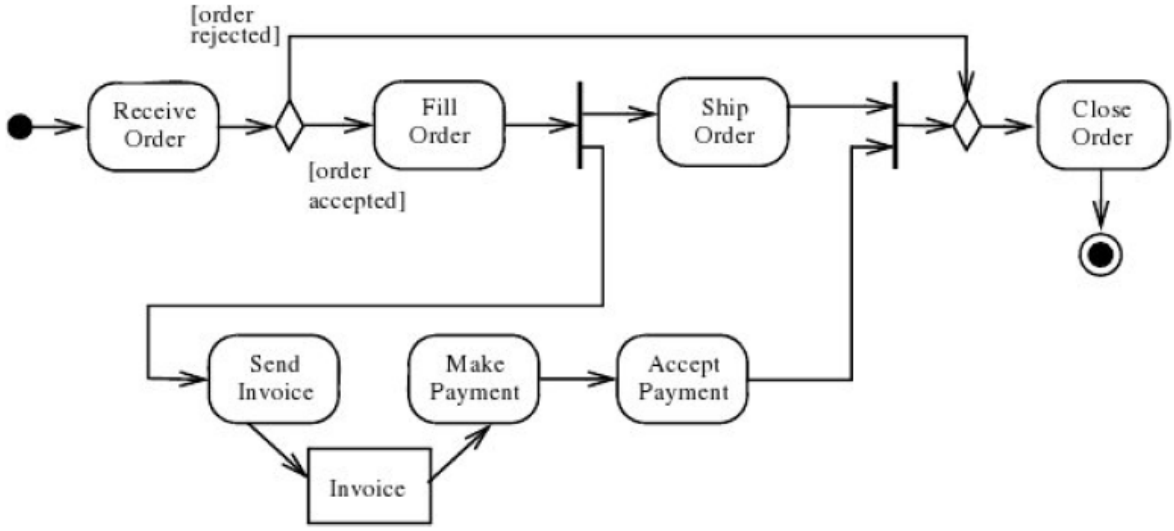
Robustness diagram è un Communication Diagram o un Sequence Diagram semplificato, ha come scopo quello di ottimizzare gli Use Cases :

- Controllare la loro correttezza
- Determinare se vanno a coprire anche tutti i casi di azione alternativi
- Identificare tutti gli oggetti necessari allo sviluppo

É rappresentato da cerchi, elementi di controllo, interfacce e entità.

48. UML/Behavioral View: cos'è l' Activity Diagram

Activity diagram mostra la sequenza delle attività, e lo fa da un punto iniziale a uno finale, dettagliando i molti cammini decisionali esistenti nella progressione degli eventi contenuti nell'attività. E' una specie di diagramma di flusso, ove si aprono cammini indicanti una "fork" cioè l'inizio di un processo parallelo e "join" l'attesa che ambedue i processi finiscano per poter proseguire . Inoltre ci sono le losanghe "in apertura" per indicare un processo decisionale, in "chiusura" per congiungere più cammini.

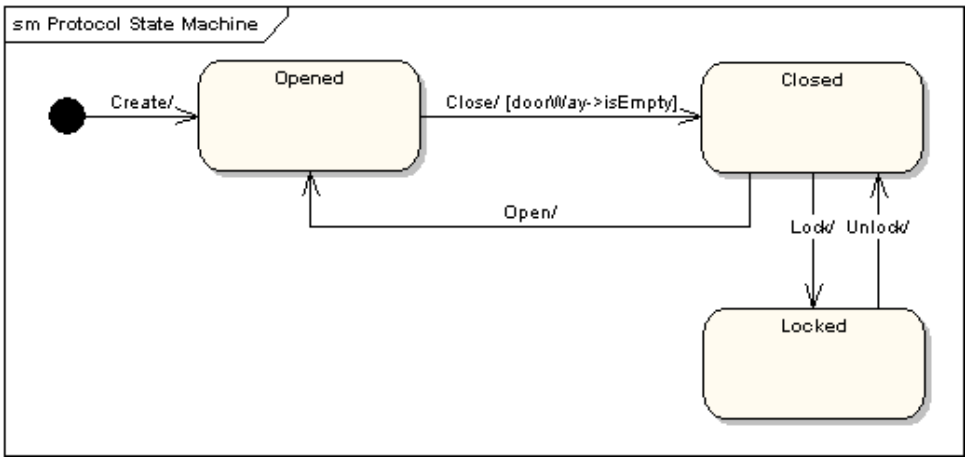


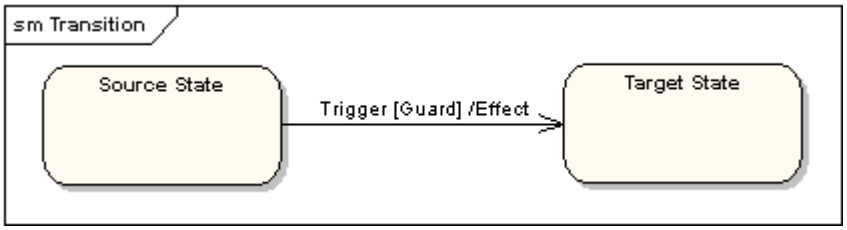
49. UML/Behavioral View: cos'è lo State Machine Diagram

Lo State Machine Diagram modella il comportamento di un singolo oggetto, ovvero specifica la sequenza di stati che un oggetto attraversa durante il suo ciclo di vita in risposta a stimoli dell'ambiente. Somiglia ai grafi delle macchine a stati di Mealy e di Moore ma con diverse integrazioni grafiche che lo rendono più espressivo.

É composto da:

- Stati (rettangoli arrotondati con il nome dello stato all'interno)
- Transizioni da uno stato all'altro (linea con una freccia)
- Punto iniziale (pallino nero)
- Punto finale (cerchio con punto al centro)
- Scelte (rombo)
- Giunzioni pseudo stato (pallino nero a cui arrivano e partono tante frecce).
- Composizioni (rettangoli arrotondati che contengono intere parti del diagramma)





Da notare la notazione per la transizione Trigger [Guard] / Effect

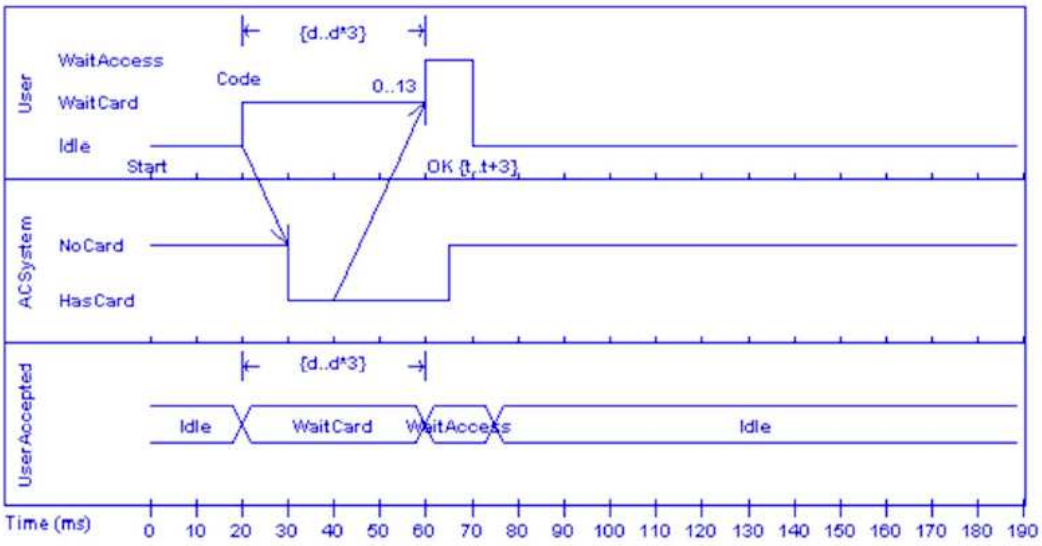
- Trigger è la causa della transizione (segnale/evento/cambiamento di qualcosa/passaggio del tempo)
- Guard è una condizione che deve essere vera affinché il trigger scateni la transizione
- Effect è l'azione invocata come conseguenza della transizione (Mealy)

50. UML/Behavioral View: cos'è l'Interaction overview diagram

E' una specie di activity diagram in cui i nodi possono essere Sequence Diagrams, Communication Diagarams, Interaction Overview Diagrams e Timing Diagrams).
Le notazioni sono quasi tutte le stesse dell'Activity Diagram (fork/join) con eccezioni di due nuovi elementi: le occorrenze di interazione e gli elementi di interazione.

51. UML/Behavioral View: cos'è il Timing diagram

Usato per visualizzare il cambiamento di stato o di valore di uno o più elementi nel tempo. Può anche mostrare le interazioni tra eventi programmati e vincoli temporali e di durata che li governano. Somiglia ai diagrammi temporali che si trovano nei datasheet.



52. UML/Implementation View: Descrivere la Implementation View

Descrive artefatti implementativi dei sottosistemi logici definiti nella Structural View.
Possono includere artefatti intermedi usati nella costruzione del sistema (files di codice, file dati, librerie).
Definisce le dipendenze tra componenti implementati e le loro connessioni, usando interfacce fornite o richieste.
Si rappresenta con il **Component diagram** , che illustra le parti di software che formano il sistema. Ha un livello di astrazione più alto di un class diagram.

53. UML/Environment View: Parlare della Environment View

Rappresenta la topologia hardware del sistema. Fornisce informazioni per l'installazione e la configurazione del sistema e descrive come i componenti software sono distriobuiti sui nodi hardware, è utile per analizzare requisiti non funzionali (affidabilità, scalabilità, sicurezza…).
Fornisce informazioni sull'installazione e la configurazione del sistema.
Si rappresentata con il **Deployment Diagram** , che modella l'architettura d'esecuzione di un sistema. Descrive la configurazione hardware in un sistema in termini di nodi e connessioni, la relazione fisica tra hardware e software e visualizza e mostra come gli artefatti sono stati installati e si muovono nel sistema.

54. UML/OCL: Cos'è OCL

Object Constraint Language : I diagrammi UML non sono abbastanza accurati per descrivere completamente un requisito, per integrarli è quindi nato OCL che è un linguaggio che permette di descrivere le espressioni e i vincoli in un sistema orientato agli oggetti.
É un linguaggio formale tipizzato con una precisa sintassi e semantica ma che non è un linguaggio di programmazione.
Permette di descrivere il sistema in modo approfondito e formale ma allo stesso tempo mantenendo facilità di lettura/scrittura da parte delle aziende e degli sviluppatori.
OCL è basato su vincoli.

55. UML/OCL: Cos'è un vincolo

Un vincolo è una restrizione su uno o più valori di un modello/sistema orientato agli oggetti.
I vincoli sono dichiarativi, cioè specificano qualcosa che deve essere vero e non qualcosa che deve essere fatto.
Non hanno effetti collaterali cioè non alterano lo stato del sistema.
Sintassi e semantica sono formalizzate impedendo quindi interpretazioni ambigue.

56. UML/OCL: Cos'è un vincolo invariante

Un vincolo invariante è un vincolo connesso a un elemento del modello.
Deve valere per tutte le istanze e deve essere vero per tutto il tempo in cui l'istanza è a riposo (un'istanza non è a riposo quando è in esecuzione).

57. UML/OCL: Cos'è una precondizione

Una precondizione è un vincolo che deve essere vero quando è invocata un'operazione. E' una responsabilità del chiamante accertarsi che sia vera, e il chiamato può rilevare errori di programmazione controllando che lo sia.

58. UML/OCL: Cos'è una postcondizione

Una postcondizione è vincolo che deve essere vero dopo il completamento di un'operazione.
E' una condizione duale alla precondizione.

59. UML/OCL: Cos'è un vincolo di guardia

Un vincolo di guardia è un vincolo che deve essere vero prima dell'avvenimento di una transizione (in maniera analoga alla precondizione). Si adatta in particolare agli State e Activity Diagrams.

60. UML/OCL: Tipi di dati

- Tipi di Base : reali, interi stringhe e booleani
- Tipi Collezione: sono cioè liste che si estrapolano navigando attraverso le associazioni in un modello UML
- Tipi user defined che sono le classi, i tipi, le interfacce e gli enumerativi di un modello UML

61. UML/OCL: Cos'è una collezione

La collezione è un tipo astratto predefinito di OCL. E' diviso in quattro sottotipi:

- Insieme (Set) è un insieme matematico e come tale non contiene duplicati.
- Insieme ordinato (OrderedSet) → insieme con elementi ordinati.
- Borsa (Bag) è come un Set ma può contenere duplicati

- Sequenza (Sequence) è come una Bag ma con gli elementi ordinati.

62. OOP: concetto di information hiding

E' una tecnica di modularizzazione che ha come obiettivo quella di ridurre la complessità del software, nascondendo i dettagli implementativi di un oggetto in modo da impedirne o regolarne, tramite metodi o precise politiche di esportazione, l'accesso.
Inoltre agevola un rapido cambiamento ed aggiornamento del software in quanto i dettagli progettuali dei componenti (oggetti) rimangono nascosti (e come tali non impattano quindi sul resto del programma).

63. OOP: concetto di incapsulamento

L'oggetto è gestibile come una scatola nera in cui l'attenzione è ai servizi che esso offre e non a come essi vengono implementati.

64. OOP: concetto di polimorfismo

Un metodo della classe base può essere ridefinito nelle classi derivate, tramite l'overloading, in modo da reimplementare un metodo "ad hoc" che va a sovrascrivere quello della super classe .

65. OOP: Che cos'è un'interfaccia

E' un insieme di metodi pubblici di una classe senza la loro implementazione. La classe che eredita l'interfaccia ha l'obbligo di implementare i metodi. E' un elemento fondamentale per realizzare l'ereditarietà in un linguaggio OOP.

66. Java: che cos'è una classe astratta?

Una classe astratta non viene creata per derivarne istanze ma per derivarne altre classi.
Le classi astratte non possono essere istanziate, sono simili concettualmente alle interfacce, eccettuato che, mentre la classe astratta definisce anche il metodo, l'interfaccia lo dichiara solamente.
Una classe che eredita una classe astratta ne eredita anche i metodi e quindi costruisce un legame ("contratto") più forte; viceversa l'interfaccia obbliga la classe derivata ad implementare i metodi elencati, metodi che possono essere completamente diversi da sottoclasse a sottoclasse.

67. Java: Differenze tra classe astratta e classe java

Una classe astratta è una classe che non è istanziabile, a differenza di una classe "java" classica.

68. Java: Spiegare cos'è la dipendenza tra classi

Quando l'esistenza di una classe A con certe caratteristiche è necessaria per l'esistenza di una classe B, si dice che B dipende da A.

69. Java: Che cos'è il byte code?

Un compilatore Java genera byte code (contenuto in un file ".class" o ".jar") un particolare codice eseguibile indipendente dall'hardware che può essere successivamente tradotto in codice macchina da una implementazione locale di una Java Virtual Machines.
Esistono JVM per una moltitudine di piattaforme hardware rendendo di fatto massima la portabilità del byte code.

70. Java: che cos'è il "Garbage Collector"?

E' un servizio offerto dalla JVM per distruggere gli oggetti inutilizzati e liberare memoria, in modo automatico

71. Java: Cos'è un Package

E' un insieme di classi coese e interdipendenti che sono logicamente viste come un unico gruppo.

72. Java: Libreria

Una libreria è un insieme di package che implementano funzionalità comuni.

73. Java: Classi annidate (statiche, interne, locali, anonime)

Sono classi definite all'interno di altre classi. Possono essere di due tipi:

- Statiche: Non hanno accesso agli altri membri della classe ospitante.Questo è comprensibile perché dell'oggetto statico ne esiste in memoria una sola copia, mentre le variabili al di fuori appartengono alle tante istanze della classe ospitante. Quale valore dovrebbero quindi avere? Per questo l'accesso non è consentito. Nel caso di classi statiche non è necessario istanziare la classe ospitante (cosa invece normalmente necessaria), mentre si istanzia comunque la classe (annidata) statica.
- Interne o non-statiche: hanno accesso agli altri membri della classe ospitante, anche se questi sono stati dichiarati privati.
  Sono usate per tre ragioni principali:
  - Raggruppamento logico di classi: se una classe serve solamente ad un altra classe allora è logico innestarla in essa.
  - Maggiore incapsulamento: una classe interna può accedere ai dati della classe contenitore senza che questi debbano essere definiti pubblici.
  - Maggiore leggibilità del codice : le classi minori rimangono vicine a quelle principali ove sono usate.

Esistono inoltre altri due tipi di classi interne :

- Locali: Classi dichiarate all'interno del corpo di un metodo
- Anonime: Classi dichiarate all'interno del corpo di un metodo senza nome e che devono o estendere una classe oppure implementare un'interfaccia .
  - Es.1 **new className(optional argument list){classBody}**
    In cui viene sottinteso "extends" (sarebbe new <anonymous class> extends className ….)
  - Es.2 **new interfaceName(){classBody}**
    In cui viene sottinteso "implements" (sarebbe new <anonymous class> implements interfaceName ….)

74. Java: Differenza tra variabili public, private e protected

- Public : variabili visibili a tutte le classi.
- Protected : variabili visibili nella classe di appartenenza e nelle sottoclassi.
- Private : variabili visibili solo nella classe di appartenenza

75. Java: Eccezioni

Un' eccezione è un'evento verificato durante l'esecuzione di un programma che interrompe il suo normale flusso di istruzioni. Sono rappresentati da Exception Objects , i quali contengono informazioni sull'errore e sono creati dai metodi del programma applicativo e successivamente passati al sistema. Le eccezioni devono essere trattate da un exception handler. In generale se un metodo non le può gestire le passa ad un livello di chiamata superiore dato dallo stack delle chiamate, tramite la parola chiave "throws".

Possono essere:

- Checked exceptions: rappresentano condizioni non valide in aree al di fuori dell'immediato controllo del programma (input utente errato, problemi al database, problemi di rete, file assenti).
  Un metodo è obbligato a stabilire una politica di gestione per tutte le checked exceptions "lanciate" da una sua implementazione (o passarla alla chiamata superiore con "thrown" o gestirla in qualche modo con "try,catch,finally")
  - Try: include il codice "normale" che "potrebbe" lanciare un'eccezione
  - Catch: E' l'handler di un eccezione. Ci possono essere tanti catch, uno per ogni possibile eccezione.
  - Finally: Codice sempre eseguito sia che l'eccezione venga generata e sia no, ove solitamente si mettono operazioni di cleanup.

  Se non si intende catturare un'eccezione, allora lo si specifica con throws nell'intestazione del metodo, insieme al nome dell'eccezione da lanciare al chiamante superiore.

- Unchecked Exceptions: rappresentano condizioni che generalmente riflettono bugs nella logica del programma e che non si possono ragionevolmente gestire a runtime. Un metodo non è obbligato a stabilire una politica di gestione per una unchecked exception "lanciata" dalla sua implementazione (e quasi sempre infatti non sono gestite). Se lo si vuole fare si usa la keyword "throws".

- Runtime Errors: sono unchecked exceptions.

76. Java: Errori

Errori : eventi eccezionali esterni all'applicazione (es guasto hardware), che un'applicazione spesso non può prevedere o risolvere. Di solito, se avvengono, il programma termina.

77. Java: Generics

I Generics (o tipi generici) sono un servizio aggiunto a Java nel 2004 con la versione J2SE 5.0 e permettono al programmatore di operare su oggetti di vario tipo mantenendo la sicurezza a tempo di compilazione. Questo si usa in particolar modo con le Collections le quali, potendo contenere oggetti di vario tipo, causerebbero eventuali errori di cast solo a runtime. Tramite i Generics invece, il programmatore specifica *a priori* la natura degli elementi che compongono una collezione, riuscendo così ad intercettare gli eventuali errori a compile time.
Un "tipo variabile" (type variable) è un identificatore non qualificato. I Type variables sono usati da dichiarazioni di classi generiche , dichiarazione di interface generiche, dichiarazione di metodi generici e da dichiarazione di costruttori generici.
Esempio, nella definizione:

```
public class List<T> {
// ... fields ...
public void add(T t) {
// ... code ...
}
public T get(int index) {
// ... code ...
}
// ... other methods ...
```

<T> viene sostituito col tipo specificato (se specificato, perché non è obbligatorio, in caso non lo sia a fronte di errori di casting si avrà un runtime error invece di un compile error).
Ricorda in qualche modo il preprocessore C e le direttive #define .

78. Java: Collections e Framework Collections

Sono oggetti usati per memorizzare, prelevare, manipolare e comunicare dati aggregati.
In java sono presenti le cosiddette Java Framework Collections, che sono architetture unificate concepite per rappresentare e manipolare collezioni. Sono basate su interfacce, algoritmi ed implementazioni . JCF è la framework collection di java; favorisce la scrittura di codice riutilizzabile e facilita in generale la programmazione .
Le collection di JCF sono Liste, Set, Map e Queue.

79. Java Collections : Liste

List : questa struttura dati è usata per contenere una lista di elementi ove tra di essi ci possano anche essere dei doppioni. Ci sono due implementazioni principali:

- ArrayList, ottimale per accesso casuale e inserzione/cancellazione solamente alla fine
- LinkedList, quando si deve frequentemente aggiungere/rimuovere elementi anche a metà della lista e con accesso alla lista sempre sequenziale.

80. Java Collections : Set

I Set sono collezioni che contengono solo un istanza per ogni articolo (quindi senza doppioni). Ci sono tre implementazioni principali:

- HashSet è ottimale quando è necessario inserire, cancellare e localizzare elementi. Non dà alcuna garanzia sull'ordine di iterazione del set: in particolare non garantisce che l'ordine rimarrà costante nel tempo.
- TreeSet garantisce l'attraversamento e l'accesso ordinato ma l'accesso è più lento.
- LinkedHashSet preserva esattamente l'ordine di inserzione

81. Java Collections : Map

Una Map è un insieme di coppie, chiave e valore ove ogni coppia rappresenta una mappa unidirezionale dalla chiave al valore. Conoscendo la chiave, si può puntare univocamente al valore. Ci sono tre implementazioni principali:

- HashMap è ottimale quando è necessario inserire, cancellare e localizzare elementi
- TreeMap è ottimale quando è necessario attraversare le chiavi in modo ordinato
- LinkedHashMap preserva esattamente l'ordine di inserzione

82. Java Collections : Queue

Una Queue è una lista che usualmente fornisce un accesso FIFO ai suoi elementi.
Ci sono due implementazioni principali:

- LinkedList , ottimale quando è usato il normale ordinamento degli elementi di una FIFO
- PriorityQueue, ottimale quando è necessario ordinare gli elementi sulla base di una comparazione

83. Java: ANT

ANT è un interprete di comandi che viene utilizzato principalmente per automatizzare le operazioni (compilazione, creazione file .jar, esecuzione...) in ambiente Java.
Legge un file .xml per capire cosa fare.

84. Java: Programmazione concorrente

Per la sincronizzazione si usa il :

- multiprocessing
- multitasking
- multithreading

Ci sono due entità che riguardano il meccanismo della concorrenza:

- il processo che è un programma eseguibile caricato in memoria
- il thread che è una unità più leggera che esegue un set di istruzioni, può condividere un spazio di memoria con un altro thread ma ha un suo contesto privato di esecuzione.

85. Java: Descrivere gli stati le politiche e i problemi riguardanti il thread

Può essere nello stato di Ready, Blocked, Running, Waiting.
E' lo scheduler che gestisce l'esecuzione/sospensione dei thread in base a vari algoritmi di priorità che possono essere cooperativi o preventivi.
Si possono verificare vari problemi :

- Starvation in caso un thread non venga mai servito dallo scheduler
- Data race in caso
  - più thread modificano gli stessi dati
  - i risultati dipendono dall'ordine dell'esecuzione

86. Java: Programmazione concorrente: implementazione in Java

I thread in Java si creano:

- Estendendo la classe *thread* e sovrascrivendo il metodo *run*
- Implementando l'interfaccia *Runnable* e in particolare il metodo *run*

87. Java: Tecniche di sincronizzazione

La sincronizzazione riguarda la programmazione concorrente ed è implementabile tramite:

- Uso di metodi sincronizzati
- Uso di blocchi sincronizzati
- Uso di oggetti "atomici"
- Uso di wait e signal

88. Java: Cos'è Javadoc?

E' un programma per generare la documentazione del programma Java; l'input è dato dai sorgenti Java e l'output avviene in HTML (e altri file di package e overview).

89. Principi fondamentali della progettazione : Astrazione (Abstraction)

L'astrazione permette di focalizzarsi sugli aspetti importanti di un problema ad un certo livello e senza l'ostacolo di dettagli non necessari o irrilevanti

90. Principi fondamentali della progettazione : Raffinazione (Refinement)

E' un processo top-down in cui si parte con una descrizione ad alto livello di come il problema può essere risolto; poi si procede a dividere il problema in alcuni

sottoproblemi più piccoli, e via così fino a che il sottoproblema giunge a risoluzione immediata.

91. Principi fondamentali della progettazione : Modularità (Modularity)

Divedere il software in componenti separati che, una volta integrati, soddisfano i requisiti del problema. Permette di ridurre la complessità, facilitare le modifiche, facilitare l'implementazione e lo sviluppo parallelo.

Un metodo di progettazione si può definire modulare se supporta la decomposizione in moduli, la ricomponibilità (cioè riutilizzo in altri progetti) , la comprensione (un modulo deve essere comprensibile senza conoscere gli altri) la continuità (una modifica impatta solo su quel modulo o al più in pochi altri) e la protezione (un baco rimane confinato all'interno e non si propaga ad altri moduli).

E' inoltre basata su :

- unità linguistiche modulari (intesa come coerenza nell'usare lo stesso linguaggio per tutta la libreria)
- poche interfacce (ogni modulo dovrebbe comunicare il meno possibile con gli altri)
- quantità di moduli (progettarne il meno possibile)
- ogni interfaccia con meno dati possibile,
- chiarezza (cioè deve essere immediatamente chiaro se devono comunicare con qualche altro modulo).

Il numero di moduli progettati deve essere giusto : nè troppo pochi (alti costi per scarsa modularità) ma nemmeno troppi (alti costi di integrazione).

92. Principi fondamentali della progettazione : Coesione (Cohesion)

Misura della coerenza tra elementi di un componente o tra i moduli di un sistema.
La coerenza semplifica la correzione, le modifiche e le estensioni, riduce il debug e promuove il riutilizzo . La coesione può essere scarsa (coincidente,logica o temporale) o alta (funzionale o di oggetto).

La coesione deve essere alta mentre l'accoppiamento deve essere il più basso possibile.

Esempio: in una classe Studente il metodo Studia ha una alta coesione mentre il metodo Cucina una bassa coesione perchè non è molto correlato con la classe studente (in parole povere il metodo Cucina non ha molto a che fare con una classe studente).
Quindi più i moduli sono coesi più è probabile modificare un solo modulo in caso di modifica di comportamento. Riprendendo l'esempio della classe Studente, se si vogliono modificare alcuni aspetti di uno studente bisognerà solamente modificare un unica classe.

93. Principi fondamentali della progettazione : Accoppiamento/Coupling

Misura l'interconnessione tra i moduli; se è forte e stretta, anche la dipendenza è alta il che è un male per la modularità; se l'interconnessione è lasca, i moduli sono largamente

indipendenti . Si può avere basso accoppiamento (dati passati come argomenti di un modulo) o forte accoppiamento (dati globali usati all'interno del modulo).
L'accoppiamento deve essere il più basso possibile perchè rende difficile testare il proprio codice e lo rende meno riutilizzabile (meno modulare).

Un esempio può essere una classe A che utilizza le classi B,C,D ed E. Se voglio utilizzare in un altro programma la classe A devo "portarmi dietro" tutte le altre classi ed inoltre se voglio testare la classe A dovrò testare anche le classi B,C,D,E in quanto utilizzate da quest'ultima.

94. Principi fondamentali della progettazione OOP: Class Design

I principi che governano un buon design delle classi sono:

- Completezza: la classe non fa niente di meno di quanto richiesto
- Sufficienza: la classe non fa niente di più di quanto il cliente ragionevolmente si aspetta
- Primitività: i servizi dovrebbero esser semplici, atomici e unici
- Alta coesione
- Basso accoppiamento

L'identificazione delle operazioni da eseguire è facile al contrario dell'assegnazione di queste operazioni alle varie classi; a tale scopo si usano delle convenzioni di massima:

- Progettazione Responsability Driven: se un Cient manda un messaggio ad un Server, è responsabilità del server eseguire l'operazione richiesta
- Ereditarietà: se un'operazione è usata da una classe e dalle sottoclassi allora è la classe che la deve implementare
- Polimorfismo e Dynamic Binding : se un'operazione può essere applicata alle istanze di una classe derivata e la classe non ha istanze, allora questa operazione "astratta" va incorporata nella classe stessa.

95. Principi fondamentali della progettazione OOP: Relationship Design

Tutte le relazioni devono avere la navigabilità e la molteplicità specificate e dovrebbero anche avere un nome di ruolo nella estremità "target".

Relazione di aggregazione (whole/part): l'aggregato usa i servizi delle parte, la quale li esegue . L'aggregato è dominante mentre la parte è tendenzialmente più passiva.

Operazione di rifinitura delle associazioni con aggregazione/composizione:

- Aggiungere la molteplicità
- Decidere quale parte è "whole" e quale "part" : se la molteplicità del "whole" è 1 allora è possibile usare la composizione, altrimenti aggregazione
- Aggiungere la navigazione da "Whole" a "Part"
- Considerare anche la possibilità di unire le due classi "whole/part" e farne una unica.

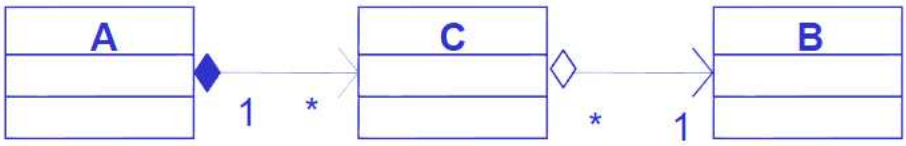
Le relazioni many-to-one non possono essere composizioni bensì aggregazioni.

Le relazioni one-to-many possono essere rappresentate da composizioni attraverso l'uso di classi container o inbuilt arrays.

96. Principi fondamentali della progettazione OOP: Reificazione e relazioni many-to-many

La reificazione è un processo che caratterizza qualche cosa di astratto facendolo diventare un oggetto.

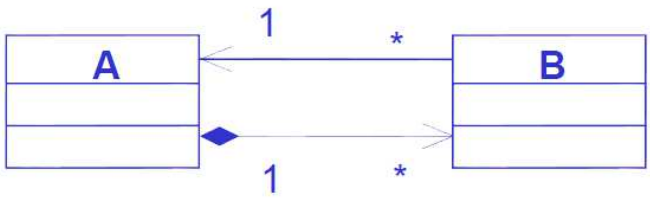
E' un concetto che si usa quando si vogliono realizzare delle relazioni many-to-many , in cui si *reifica* la relazione tra le due classi in una apposita classe



Decidere quale parte è "whole" e usare aggregazione o composizione

97. Principi fondamentali della progettazione OOP: Relazioni bidirezionali

Relazioni bidirezionali non possono essere rappresentate da composizioni o aggregazioni bensì da una composizione o aggregazione whole/part e una associazione o dipendenza da part a whole



98. Principi fondamentali della progettazione OOP: Identificazione delle interfacce

Le interfacce separano la specifica di funzionalità dalla implementazione.
Per individuarle bisogna verificare le associazioni e i messaggi, fattorizzare gruppi di operazioni e attributi, cercare classi che giocano lo stesso ruolo, considerare future espansioni e verificare le dipendenze tra componenti.

99. Principi fondamentali della progettazione OOP: Identificazione delle ereditarietà

E’ la relazione tra due classi che realizza l’accoppiamento più forte , da usare solo quando c’è una chiara e ben identificabile relazione “is-a” .

Ci sono vari tipi di ereditarietà:

- Ereditarietà di implementazione: gli sviluppatori creano sottoclassi nel codice andano a rifinire la classe padre: non è buono per il riutilizzo
- Ereditarietà di Specifica : Classificazione di concetti in gerarchie ti tipo in modo che un oggetto da una classe specifica può essere sostituito da un’oggetto creato da una delle sue sottoclassi
- Specializzazione: Gestisce in maniera differente padre e figlio per eseguire lo stesso task
- Estensione : aggiunge funzionalità del padre aggiungendo dati e comportamenti.

100. Design Pattern: Cosa sono?

I design pattern sono soluzioni riutilizzabili per problemi ricorrenti nella programmazione.
Sono indipendenti dal linguaggio di programmazione, possono essere semplici ma anche molto complessi e, ce utilizzati durante lo sviluppo, sono detti idiomi programmativi. I pattern più noti sono i GoF (Gang Of Four), che sono classificati in Creazionali, strutturali e comportamentali.

101. Design Pattern Creazionali

Gestiscono la creazione dinamica degli oggetti all’interno di un sistema

- Abstract factory: Fornisce una classe/interfaccia per creare famiglie di oggetti imparentati o dipendenti senza specificare le classi concrete
- Factory Method: Definisce una classe/interfaccia per creare un oggetto, ma delega alle sottoclassi quali classi sono da istanziare
- Builder, permette la creazione dinamica di oggetti basandosi su algoritmi facilmente intercambiabili.
- Prototype: Specifica i tipi di oggetti da creare usando un’istanza prototipo , e crea nuovi oggetti copiando questo prototipo
- Singleton, usato per garantire che una sola istanza di un oggetto venga creata nel sistema.

102. Design Pattern Strutturali

Micro-architetture statiche di utilizzo generale, gestiscono il modo in cui classi e oggetti vengono composti per formare architetture complesse

- Adapter, usato per risolvere problemi di incompatibilità tra interfacce.
- Bridge, disaccoppia un’astrazione dalla sua implementazione
- Composite, usato per trattare in modo uniforme oggetti singoli e gruppi di oggetti.
- Façade, usato per fornire un’interfaccia unificata ad un gruppo di oggetti.
- Proxy, fornisce un surrogato o segnaposto di un oggetto per controllarne l’accesso.

103. Design Pattern Comportamentali

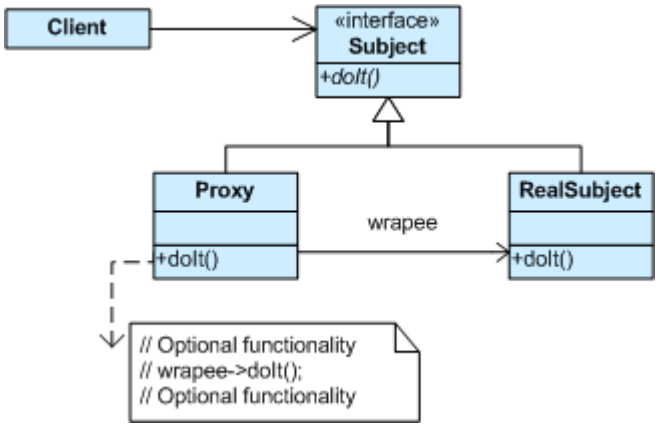
Descrivono il comportamento di un'architettura di oggetti, definiscono collaborazioni tra oggetti (in sostanza comunicazioni tra oggetti).
I principali sono:

- Command: usato per incapsulare l'astrazione di richiesta
- Iterator: fornisce un modo per accedere agli elementi di un oggetto aggregato senza esporre la sua implementazione sottostante
- Observer: definisce una dipendenza uno a molti fra oggetti diversi, in maniera tale che se un oggetto cambia il suo stato, tutti gli oggetti dipendenti vengono notificati del cambiamento avvenuto e possono aggiornarsi

104. Design Pattern/Pattern strutturali: Proxy

Fornisce un surrogato o segnaposto di un oggetto per controllarne l'accesso.
Si usa quando:

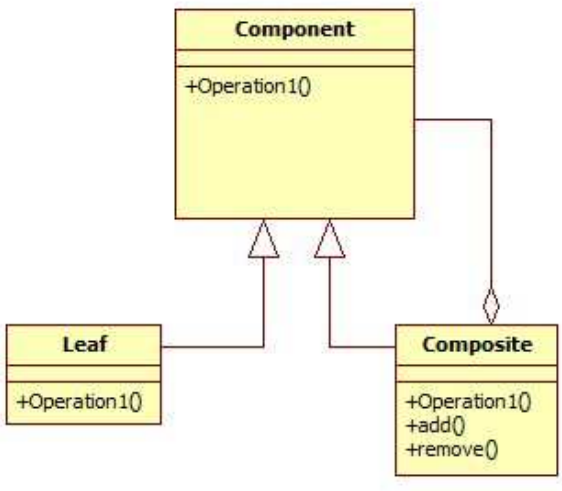
- Un sistema vuole creare oggetti "costosi" (virtual proxy)
- Un sistema vuole mantenere dei risultati temporaneamente (cache proxy)
- Un sistema vuole usare rappresentanti locali per oggetti remoti (remote proxy)
- Un sistema vuole controllare gli accessi ad un oggetto condiviso (protection proxy)
- Un sistema vuole rimpiazzare puntatori con qualcosa che esegue operazioni addizionali quando un oggetto è richiamato (smart reference)



105. Design Pattern/Pattern strutturali: Composite

Consente ad oggetti utilizzatori di creare strutture ad albero in modo che sia possibile trattare in modo uniforme:

- oggetti foglia
- oggetti contenitore.



Un composite viene usato quando

- si vogliono rappresentare gerarchie di oggetti.
- si vuole consentire di ignorare le differenze tra oggetti foglia e oggetti contenitore.

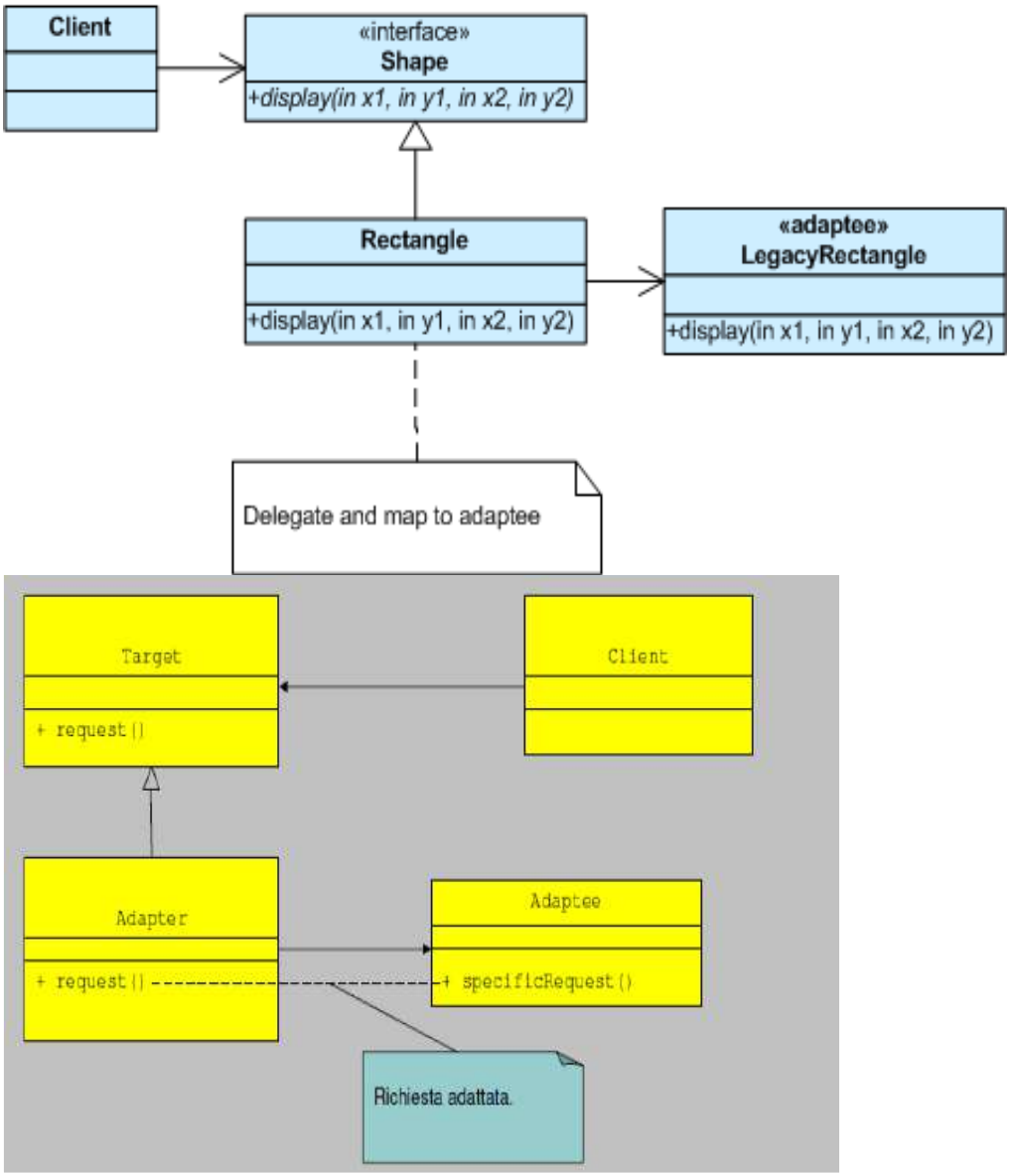
NB Component è l'interfaccia

```
public interface Component {
    public void assemble();
}

public class Leaf implements Component {

    public void assemble() {
        System.out.println("Leaf assembled"+this);
    }
}

public class Composite implements Component {
         // Collection of child groups.
         private List<Component> components = new ArrayList<Component>();

         public void assemble() {
             System.out.println("Assembling composite "+this);
           for (Component component : components) {
              component.assemble();
           }
         }

         // Adds the group to the structure.
         public void add(Component component) {
                components.add(component);
         }

         // Removes the group from the structure.
         public void remove(Component component) {
                components.remove(component);
         }
       }

public class ImplementComposite {
          public static void main(String[] args) {
                 //Initialize three "Leaves"
                 Leaf leaf1 = new Leaf();
                 Leaf leaf2 = new Leaf();
                 Leaf leaf3 = new Leaf();

                 //Initialize three "components"
                 Composite composite1 = new Composite();
                 Composite composite2 = new Composite();
                 Composite composite3 = new Composite();

                 //Put Leaves in the components
                 composite2.add(leaf1);
                 composite2.add(leaf2);

                 composite3.add(leaf3);

                 composite1.add(composite2);
                 composite1.add(composite3);


                 composite1.assemble();
             }
       }
```

106. Design Pattern/Pattern strutturali: Adapter

E' usato per risolvere problemi di incompatibilità tra interfacce

- Consente a oggetti diversi di essere utilizzati insieme anche se le rispettive interfacce non sono compatibili
- Aumenta il riutilizzo delle classi perché consente di utilizzarle in situazioni non previste inizialmente senza doverle modificare.
- Serve quando si vuole utilizzare una classe che non offre un'interfaccia corretta, senza modificarla
- Serve per creare una classe che utilizza i servizi di un'altra classe di cui non è ancora nota l'interfaccia



In questo esempio, abbiamo una "vecchia" funzione rectangle che contiene un metodo "display" che si aspetta di ricevere in ingresso le coordinate dell'origine del rettangolo più larghezza ed altezza.
Il client però vuole lavorare con le 4 coordinate separate. Introduciamo un "adapter" sotto forma dell'interfaccia "Shape".
Rinomino quindi Rectangle in LegacyRectangle :

```
class LegacyRectangle
{
    public void display(int x, int y, int w, int h)
    {
        System.out.println("rectangle at (" + x + ',' + y + ") with width " + w
          + " and height " + h);
    }
}
```

e creo “l’adapter “ cioè l’interfaccia Shape:

```
interface Shape
{
  void display(int x1, int y1, int x2, int y2);
}
```

..adesso introduciamo la nuova classe Rectangle

```
class Rectangle implements Shape
{
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee. display (Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 - x1),
          Math.abs(y2 - y1));
    }
}
```

Qui viene creato un nuovo oggetto di nome adaptee (“Adattato”) di tipo LegacyRectangle Viene quindi fatto l’overloading del metodo Dispaly (obbligatorio vista la presenza dell’interfaccia) andando ad adattare le coordinate:

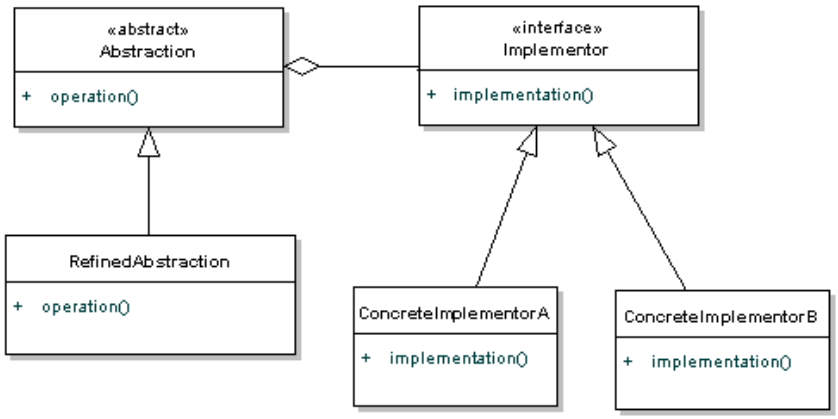
```
class Line implements Shape
{
    private LegacyLine adaptee = new LegacyLine();
    public void display (int x1, int y1, int x2, int y2)
    {
        adaptee.display (x1, y1, x2, y2);
    }
}
```

Ecco il codice main per fare funzionare l’esempio, usando il metodo draw di rectangle non mi accorgo dell’adattamento!

```
public class DpAdapter {

       /**
        * @param args
        */
       public static void main(String[] args) {
              r=new rectangle();

                  int x1 = 10, y1 = 20;
                  int x2 = 30, y2 = 60;
                   r.display (x1, y1, x2, y2);

       }

}
```

107. Design Pattern/Pattern strutturali: Bridge

Disaccoppia un’astrazione dalla sua implementazione. La classe di implementazione e la classe degli utilizzatori possono variare indipendentemente.



Un bridge viene usato quando:

- Si vuole evitare di legare un'astrazione ad una sola implementazione
- Ogni cambiamento di un'implementazione non è influente sugli utilizzatori,
- Per nascondere dettagli implementativi.

Esempio di Bridge

Il Bridge separa una astrazione dalla sua implementazione, cosicchè le due entità possono cambiare indipendentemente. Un interruttore casalingo che comanda luci, ventilatori da soffitto è un esempio di Bridge . Lo scopo dell'interruttore è accendere e spegnere un dispositivo ; l'interuttore effettivo può essere una catenella a tiro, un semplice switch a due posizioni o una varietà di interruttori con variatore

BRIDGE VS ADAPTER

- Gli Adapter fanno funzionare le cose dopo che sono state progettate (vedere la classe rectangle ove l'istanza adaptee risolve l'incompatibilità) ; i Bridge invece le fanno funzionare prima, nel senso che assolvono il loro compito di "disaccoppiamento" in maniera nativa e sena dovere scrivere del codice di adattamento.
- Il Bridge è designato in prima istanza per lasciare variare in modo indipendente sia l'astrazione che l'implementazione. L'Adapter invece è un retrofit per fare funzionare insieme classi non imparentate tra di loro.
- Bridge e (fino ad un certo punto) Adapter hanno strutture similari. Differiscono nel senso che risolvono problemi differenti.

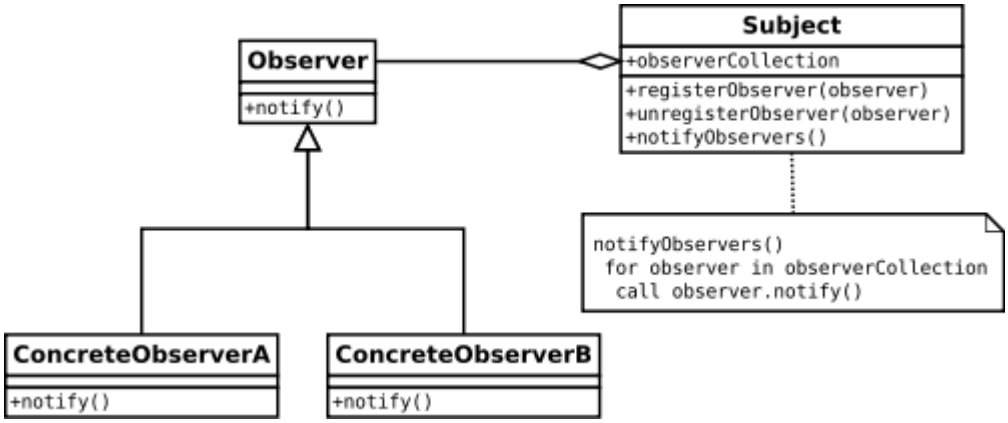
```
//Implementor

public interface TV
{
        public void on();
        public void off();
        public void tuneChannel(int channel);
}

//Concrete Implementor
public class Sony implements TV
{
        public void on()
```

```
        {
                //Sony specific on
        }


        public void off()
        {
                //Sony specific off
        }

        public void tuneChannel(int channel);
        {
                //Sony specific tuneChannel
        }
}

//Concrete Implementor
public class Philips implements TV
{
        public void on()
        {
                //Philips specific on
        }


        public void off()
        {
                //Philips specific off
        }

        public void tuneChannel(int channel);
        {
                //Philips specific tuneChannel
        }
}

//Abstraction
public abstract class RemoteControl
{
  private TV implementor;


  public void on()
  {
    implementor.on();
  }
  public void off()
  {
    implementor.off();
  }

  public void setChannel(int channel)
  {
            implementor.tuneChannel(channel);
  }
}
//Refined abstraction
public class ConcreteRemote extends RemoteControl
{
  private int currentChannel;

  public ConcreteRemote (TV B)
```

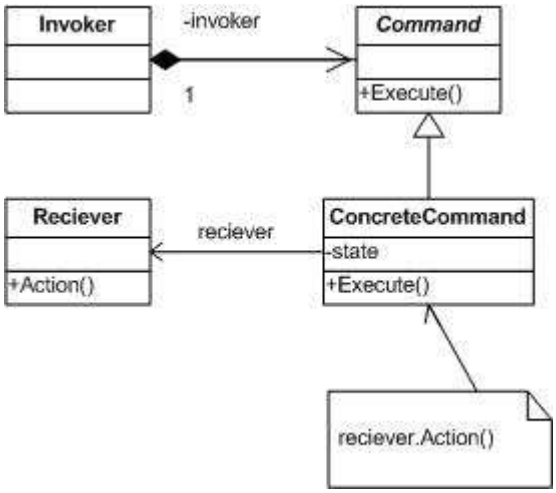
```
        {
                this. implementor =B;
        }

  public void nextChannel()
  {
     currentChannel++;
          setChannel(currentChannel);
  }

  public void prevChannel()
  {
     currentChannel--;
          setChannel(currentChannel);
  }

}

public static void main(String[] args) {

        Sony ConcreteTV1=new Sony ();
        Philips ConcreteTV2=new Philips ();

        ConcreteRemote remote1= new ConcreteRemote (ConcreteTV1);
        button1.on();

        ConcreteRemote remote2= new ConcreteRemote (ConcreteTV2);
        button2.on();
}
```

108. Design Pattern/Pattern Comportamentali : Observer

L'observer pattern è un pattern di progettazione software in cui un oggetto, chiamato "Subject", mantiene un lista di sue dipendenze , chiamati "observers", e li notifica automaticamente di qualsiasi cambio di stato, di solito chiamando uno dei suoi metodi. E' principalmente usato per implementare sistemi distribuiti di gestione di eventi



109. Design Pattern/Pattern Comportamentali : Command



Da quello che capisco, pensiamo ad un telecomando per il controllo di una casa.
Il client crea l'oggetto **receiver** (il destinatario dei comandi)
Il client crea l'oggetto telecomando **che è l'invoker** e che manderà i comandi al receiver.
Il client crea l'oggetto ConcreteCommand che è l'oggetto tramite il quale mandare i comandi. Il costruttore di questo oggetto assegna il comando all'oggetto.

Il telecomando/invoker può così inviare dei comandi (ConcreteCommands che sono a loro volta oggetti) all'oggetto ricevitore . Invoker non sa nulla di come l'oggetto verrà comandato; tutto quello che vuole fare è mandargli dei comandi. In questo modo si realizza un disaccoppiamento che isola completamente l'oggetto invoker dall'oggetto ricevitore.

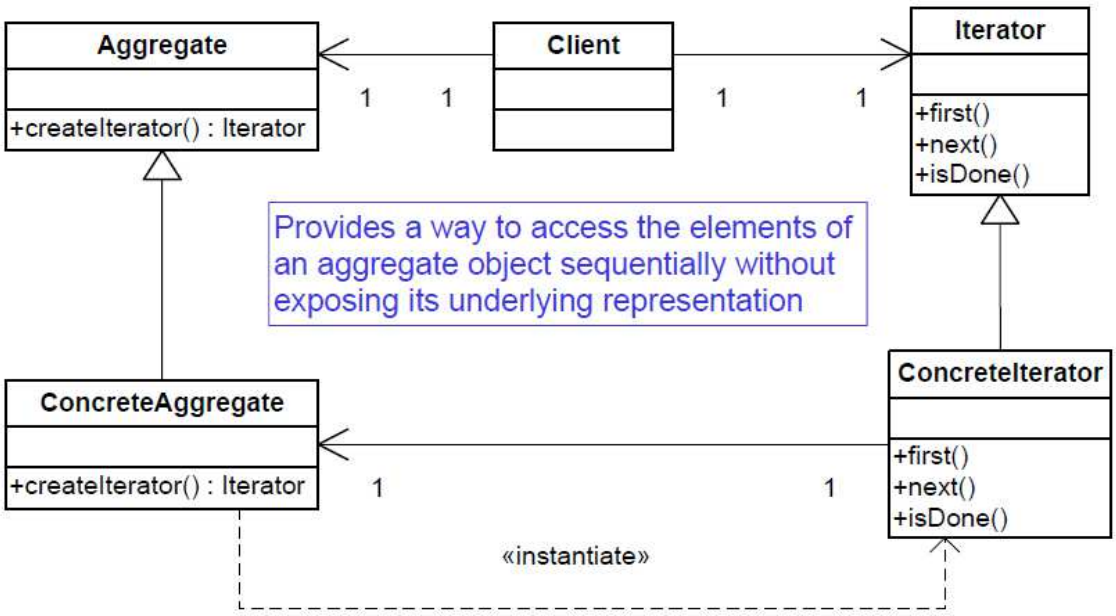
Invoker può essere un aggregatore di comandi e realizzare così un meccanismo di generazione macro, oppure di undo/redo .

Intenti di Command
- Incapsulare una richiesta in un oggetto
- Permettere la parametrizzazione di clienti con differenti richieste
- Permette di salvare le richieste in una coda

110. Design Pattern/Pattern Comportamentali : Iterator

Fornisce un modo per accedere agli elementi di un oggetto aggregato senza esporre la sua implementazione sottostante.

Utile se un sistema vuole supportare attraversamenti multipli di un aggregato di oggetti oppure se un sistema vuole fornire una interfaccia uniforme per l'attraversamento di strutture aggregate differenti (cioè supportare iterazioni polimorfiche) .



NOTA:

- “Definizione” è quando ad es. si costruisce una classe programmandone i metodi ecc.ecc.
- “Dichiarazione” invece è quando, ad es., la si istanzia…

I titoli in corsivo indicano un argomento informativo di cui è più importante cogliere il senso piuttosto che la memorizzazione puntuale.

Esempio Esame #1
-tecniche per individuare i requisiti
-state diagram
-come e perché è importante l'accoppiamento tra moduli software
-differenze tra classe astratta e classe java
-pattern di progettazione "proxy"
-diagramma dei casi d'uso(esercizio da svolgere)
-diagramma di stato(esercizio da svolgere)

Esempio Esame #2
- descrizione delle tecniche per l'individuazione delle classi d'analisi
- descrizione activity diagram
- pattern "observer"
- l'importanza della coesione tra i moduli software
- tecniche di sincronizzazione in java

- esercizio: si parte da un grafico e si ricava il codice java
- esercizio: class diagram per la gestione di un quotidiano

Esempio Esame #3
-Requisiti non funzionali
-Principi della programmazione modulare
-Cosa è OCL
-Descrivi le eccezioni in java
-Pattern iterator
-fare class diagram e definire successive classi java di un esercizio

Esempio Esame #4
-Tecniche di estrazione (elicitation) dei requisiti
-Come si trasforma relazione "molti a molti" nella fase di progetto
-activity diagram
-Caratteristiche delle MapJava
-Pattern di progettazione adapter
-Codice che implementa le classi di un class diagram banale fornito(esercizio)
-class diagram(esercizio)