Laurea Magistrale in Informatica A.A. 2020/2021
Università degli Studi di Milano-Bicocca

Appunti Progetto e Sviluppo del Software

Pelusi Marta
@Marta629

Indice

1 Sofware process **4**
1.1 Metodi di sviluppo agile . . . 4
1.2 DevOps . . . 6
1.2.1 Rainbow deployment . . . 10
1.3 Deployable units . . . 10
1.4 Monitor . . . 13
1.5 Risk management . . . 13
1.5.1 Cause dei rischi . . . 14
1.5.2 Risk assessment (valutazione del rischio) . . . 15
1.5.3 Risk control . . . 16
1.6 CMMI - Capability Maturity Model (I) . . . 18
1.6.1 Process area . . . 19

2 Requirements engineering **22**
2.0.1 Tipi di requisiti . . . 24
2.0.2 Categorie dei requisiti . . . 25
2.0.3 Processo di ingegnerizzazione dei requisiti . . . 26
2.0.4 Selezione degli stakeholder . . . 27
2.1 Artefact-driven elicitation techniques . . . 29
2.1.1 Background study . . . 29
2.1.2 Data collection . . . 30
2.1.3 Questionnaires . . . 30
2.1.4 Storyboards . . . 30
2.1.5 Scenari . . . 31
2.1.6 Prototipi e mock-ups . . . 31
2.2 Stakeholder-driven elicitation techniques . . . 32
2.2.1 Interviews . . . 32
2.2.2 Osservazioni . . . 33
2.2.3 Group session . . . 34
2.3 Requirements documentation . . . 34
2.3.1 Organizzazione globale di un documento di specifica dei requisiti . . . 36
2.3.2 Esempi diagrammi . . . 38
2.3.3 Formal notations . . . 40
2.4 Requirements quality assurance and evolution (garanzia ed evoluzione dei requisiti) . . . 42
2.4.1 Cross referencing . . . 45
2.4.2 Traceability matrices . . . 45
2.4.3 Feature diagrams . . . 46
2.5 Requirements quality assurance (garanzia di qualità dei requisiti) 47
2.5.1 Checking decision tables . . . 49

3 Software design and developement **52**
3.1 Model View Controller - MVC 52
3.1.1 Design patterns 53
3.1.2 Page controller 54
3.1.3 Front controller 55
3.1.4 Template view 57
3.1.5 Transformation view 57
3.2 Object relational mapping - design patterns 60
3.2.1 Active record 60
3.2.2 Data mapper 61
3.2.3 Identity map (pattern) 63
3.2.4 Identity field (pattern) 63
3.3 Java JPA 2.0 64
3.3.1 Jpa application 64
3.3.2 Entity manager 64
3.3.3 Relazione tra entità 66
3.3.4 Ereditarietà 66
3.4 Enterprise application - component-based system 67
3.4.1 Inversion of control o dependency injection 67
3.4.2 Service locator o dependency lookup 67
3.4.3 EJB Applications 68
3.4.4 Session bean 69
3.4.5 Message-driven bean 69
3.4.6 Instance pooling 71
3.4.7 Isolation and database locking 74

1 Sofware process

1.1 Metodi di sviluppo agile

I **metodi agili**, che hanno uno sviluppo a spirale, sono metodi nati per la gestione dei requisiti.
Questi presuppongono che i requisiti possano cambiare durante lo sviluppo di un progetto software. Il loro scopo è essere agili ed elastici e plasmare le esigenze del programmatore sulla base dello sviluppo del progetto.

Lo sviluppo si articola in **iterazioni** e produce in output parte del prodotto finale. Il progresso maggiore è rappresentato dal numero di features che sono state dimostrate. Lo sviluppo in iterazioni consente al team di sviluppo di adattarsi rapidamente alle mutevoli esigenze.
Ad ogni iterazione si seleziona il gruppo di requisiti più urgenti da utilizzare, li si analizza ed infine entrano nelle iterazioni; in output si ha un pezzo di working system testato ed integrato. Quest'ultimo, poi, può essere messo in produzione.
Questo è un procedimento che si fa per ogni blocco di requisiti.
Ad ogni rilascio si raccolgono feedback per chiarire e/o modificare l'insieme dei requisiti.

Come viene implementato un metodo agile

1. c'è molta enfasi ui membri del gruppo e sul processo di selezione del team
2. l'idea è quella di evitare di produrre documentazione costosa da mantenere
3. diventa importante la comunicazione diretta attraverso meeting e riunioni (non più attraverso lettere o documenti scritti)
4. nulla è definitivo: non si deve seguire la perfezioni ma si deve entrare nell'ottica che tutti gli elementi si miglioreranno e perfezioneranno nel tempo. Anche il design evolve mano a mano che vengono definiti i requisiti
5. il processo di sviluppo è un processo iterativo che include pratiche per controllare la qualità del design e del software
6. continuo testing: i testi sono eseguiti ad ogni modifica del sotware in automatico

Processo scrum

I **ruoli** sono:

- team: costruisce il prodotto che il cliente utilizzerà: l'applicazione o il sito web, ad esempio. Il team è interfunzionale e include tutte le competenze necessario per consegnare ciascuno il prodotto. È anche auto-organizzato (autogestito), con un alto grado di autonomia e responsabilità.

- product owner: responsabile dell'ottimizzazione del guadagno identificando le caratteristiche del prodotto, traducendo questi in un elenco di funzionalità prioritarie, decidendo quale dovrebbe essere in all'inizio dell'elenco per il prossimo Sprint, e ridefinisce continuamente le priorità e perfezionare l'elenco, quindi conosce i requisiti.

- scrum master: aiuta il team ad apprendere e ad applicare i prodotti per ottenere valore aziendale. Lo ScrumMaster fa tutto ciò che è in suo potere per aiutare la squadra ad avere successo. È il coordinatore che controlla che le pratiche dello scrum vengono svolte correttamente.

Nella fase di analisi dei requisiti sono attivi:

- product owner

- team

Nella fase di lavoro è attivo il team che produce un output.
Dietro le quinte c'è lo scrum master che supervisiona.

1. **Product Backlog**: un progetto Scrum è guidato da una visione del prodotto compilata da Product Owner, ed espresso nel Product Backlog. Il product backlog è un elenco prioritario di ciò che è richiesto, classificato in ordine di valore per il cliente o l'azienda, con gli articoli di valore più alto a in cima alla lista. Il Product Backlog si evolve nel corso della vita di progetto e gli elementi vengono continuamente aggiunti, rimossi o ridefiniti. Il product backlog dev'essere visibile ai partecipanti dello scrum process, lo sprint backlog è estratto dal product backlog e durante lo sprint lo sprint backlog è modificato dal team.

2. **Sprint**: scrum struttura lo sviluppo del prodotto in cicli di lavoro chiamati Sprint, iterazioni di lavoro che in genere durano da 1 a 4 settimane, ovvero 30 giorno in cui il team può chiedere consigli a persone anche esterne al team. Gli Sprint hanno una durata fissa e terminano in una data specifica, sia che il lavoro è stato completato o meno; non vengono mai estesi.

3. **Sprint Planning**: all'inizio di ogni Sprint, si svolge lo Sprint Planning Meeting. Il Product Owner e lo Scrum Team (con la facilitazione di lo ScrumMaster) rivedere il Product Backlog, discutere gli obiettivi e contesto per gli elementi e lo Scrum Team seleziona gli elementi dal file Product Backlog da completare entro la fine dello Sprint, a partire dalla parte superiore del Product Backlog.
   Ogni elemento selezionato dal Product Backlog viene progettato e poi suddiviso in una serie di attività individuali. L'elenco delle attività viene registrato in un documento chiamato Sprint Backlog.

4. **Daily Scrum Meeting**: una volta che lo Sprint è iniziato, lo Scrum Team si impegna in un altro di le pratiche chiave di Scrum: il Daily Stand-Up Meeting. Questo è un riunione breve (15 minuti) che si tiene ogni giorno

lavorativo presso un tempo stabilito. Tutti i membri del team partecipano. In questo incontro, vengono presentate le informazioni necessarie per controllare i progressi. Questo le informazioni possono comportare ripianificazione e ulteriori discussioni subito dopo il Daily Scrum. Lo scrum è un modello di processo basato su scrum in scala per progetti e team di grandi dimensioni.

5. **Sprint Review e Retrospective**: al termine dello Sprint, c'è lo Sprint Review, dove il team e gli stakeholder controllano cosa è stato fatto durante lo Sprint, discuterne e capire cosa fare dopo. Presenti a questa riunione sono il Product Owner, i membri del team e ScrumMaster, più clienti, stakeholder, esperti, dirigenti e chiunque altro interessato.
   Dopo la Sprint Review, il team si riunisce per lo Sprint Retrospective che è un'opportunità per il team di discutere di ciò che è funziona e cosa non funziona e accetta le modifiche da provare.
   L'obiettivo dello sprint review è raccogliere feedback dagli stakeholders.

Extrem programming

Il metodo **extrem programming** ha la stessa struttura dei metodi agili, ovvero ha un insieme di requisiti chiamati "stories".
Il requisito è definito come un attore che cerca di fare qualcosa (da qui il nome "storia").

Durante l'iterazione vengono implementate le storie e c'è una continua revisione del codice prodotto; alla fine c'è una release dimostrata. Come nei metodi agili, sono sempre raccolti feedback.

1.2 DevOps

Processo a cascata: i requisiti del sistema devono essere chiari e stabili durante lo sviluppo del prodotto.
Questa assunzione, però, è difficilmente verificata perché i requisiti cambiano e cambiano anche le necessità del sistema durante lo sviluppo. Per questo sono stati introdotti i metodi agili, che sono in grado di lavorare con requisiti che cambiano frequentemente.

DevOps è un metodo di sviluppo che considera che anche la parte di *operation* di un sistema software dev'essere agile (oltre alla parte di sviluppo *develop*), perché anch'essa produce upgrades e aggiornamenti.

- la parte di sviluppo e operation sono due team indipendenti che comunicano tra loro, ma questa disconnessione tra i due team crea inefficienza (anche in riferimento alle tempistiche/ ritardi) che invece DevOps cerca di risolvere
- DevOps vuole diminuire il costo di test e deploying
- si possono commettere, inoltre, facilmente errori nel codice di deployment, cosa che DevOps vuole evitare

DevOps è un processo che promuove la collaborazione tra team che si preoccupano di sviluppo e operation. Vuole creare un unico team che cura la parte di sviluppo e operation del software e questo rende più efficace la produzione e il deployment del software.
Include numerosi processi che mirano a ripetere operazioni in modo continuo e automatico.
Ciò che viene sviluppato dev'essere anche integrato, monitorato e testato in modo costante.
In altre parole DevOps mette insieme due gruppi di lavoro distinti (operations e developers) riducendo la disconnessione tra i due; inoltre riduce i costi di deployment e test e riduce il rischio di commettere errori nel codice.

Confronto tra DevOps e Agile

Feature	**DevOps**	**Agile**
Agility	In both developement and operation	In developement
Process & Practices	Focus on processes: CD, CT, CI, CM	Focus on pratices: CP, Scrum,
Source of feedback	Feedback from monitoring tools	Feedback drom customers
Scope	Agility and Automation	Agility and partially automati

Con DevOps l'automazione diventa centrale: dal momento in cui viene prodotto il cambiamento scatenano una serie di operazioni automatiche per trasferire l'update direttamente sul campo.

Ruoli:

- automation expert
- security engineer
- software developer/tester
- code release manager
- quality assurance
- DevOps evangelist $\approx$ scrum master

Principi:

- customer-centric action, per una pianificazione delle azioni pensando a un team unico
- end-to-end responsibility, perché il team gestisce interamente il prodotto
- continuous improvement, ovvero cercare continuamente il miglioramento dei propri prodotti e dei propri processi
- automate everything, per cercare di rendere tutti i processi automatici
- work as one team, facendo di tutto per avere un unico team che lavora
- monitor and test everything

Build, test, release

Git è un ottimo sistema di version control (distribuito), più popolare e con molte funzionalità.

Multi-branch development: ogni branch deve avere la propria definizione e dev'essere creato per un obiettivo specifico. Quando un branch viene unito ad un altro branch bisogna essere consapevoli del motivo.

Pipeline: azioni attivate ad ogni cambiamento del software, in pratica workflow composti da job attivate dal sistema di version control a seguito di eventi specifici (es push di nuovo codice). Le pipeline possono essere attivate automaticamente o manualmente e ci permettono di verificare il nostro codice a seguito di una serie di criteri. Vengono attivate quando un componente viene attivato, quando si assembla il sistema, etc...
Le pipeline vengono usate come *quality gate*, ovvero il mio codice deve verificare e superare tutti i controlli all'interno di una pipeline prima di passare al livello successivo.
Le pipeline possono anche essere attivate in modo regolare (es test che richiedono più tempo e risorse).

Component phase: il focus è sulla più piccola unità che viene aggiornata, ad esempio un aggiornamento locale.

Subsystem phase: un cambiamento può essere testato o cambiato nel sottosistema. Si possono fare test funzionali più sofisticati, come test di performance, sicurezza, etc... questa unità, però, potrebbe dipendere da altri servizi e quindi non può essere eseguito singolarmente. Per testare il servizio isolato, quindi, potrebbe essere necessario creare mock o artefatti.

System phase: possiamo testare il sistema intero integrando sia test che performance che sicurezza. I test sono di costo crescente.

Production phase: legata alla necessità di creare artefatti che andranno direttamente sul campo, ad esempio se voglio automatizzare il deployment. Posso fare test per fare rapidi check per valutare la correttezza della produzione degli artefatti.

Il punto principale è come gestire l'evoluzione automatica del software attivata da una procedura automatica che gestisce gli aggiornamenti.
Passare da una versione 1.1 a una versione 1.2 può essere rischioso se si commettono errori.

Gli schemi degli aggiornamenti sono **incrementali** rispetto a:

- utenti (dark lauching, canary releases, user experimentation)
- richieste (gradual upgrades/rollout)

- parti di software/componenti (rolling upgrade)
- backup (green/blu deployment, rainbow deployment)

Process Dark Lauching

Il **processo dark lauching** arriva dal mondo Facebook che l'ha introdotto a partire dal 2011. Prima del 2011 gli aggiornamenti impattavano su scala mondiale ed erano esposti a vari rischi.
Dopo il 2011 la nuova feature è stata resa accessibile solamente ad alcuni utenti. I malfunzionamenti e i fallimenti vengono tradotti in miglioramenti del software, cioè uno sviluppo continuo che continua a riparare il software. Quando la nuova feature si rivela stabile per il gruppo ristretto di utenti si incrementa il numero di utenti che hanno accesso alla nuova feature fino a far accedere alla feature tutti gli utenti.

Dark lauching (front-end) è simile a Canary Releases (backend) e spesso i due termini sono usati in modo intercambiabile.

User experimentation

Domanda: A è migliore di B?

L'idea è quella di studiare diverse varianti del sistema, ovvero diversi modi di fare la stessa cosa e il loro impatto, esponendo gli utenti a queste varianti, misurandone l'impatto per vedere qual è il modo migliore di fare qualcosa.
Anche in questo caso le due versioni vengono esposte ad utenti diverse, quindi sussistono due versioni contemporaneamente. Questo aiuta anche a fare una valutazione su base statistica.

Inizialmente il nuovo servizio riceverà una fetta molto piccola delle richieste. Tutto questo è continuamente monitorato. Se le cose funzionano bene il load balancer sposta tutto il traffico dal vecchio al nuovo servizio.
In questo modo se dovessero esserci malfunzionamenti del nuovo servizio il load balancer può ridirigere il traffico.

Rolling upgrade

Il **rolling upgrade** è un update che riguarda tante componenti di un sistema distribuito.
Attraverso tecniche di monitoring si verifica la correttezza dei vari servizi prima di farne l'upgrade.
Perchè questo schema sia attuabile le nuove versioni dei servizi devono essere compatibili con i nuovi scenari, e devono essere perfettamente funzionanti.

Blu/green deployment

Lo switch da una versione all'altra è immediata e si hanno due versioni identiche dell'intera struttura, ma una ospita la versione nuova e l'altra ospita la versione vecchia.

Viene fatto deployment in un ambiente clone della versione vecchia e quando si è soddisfatti si fa uno switch immediato alla nuova. Questo può essere rischioso, ma se ci fossero problemi si può fare direttamente il rollback alla vecchia versione risolvendoli poi. Viene anche garantito, quindi, un backup.

1.2.1 Rainbow deployment

C'è lo switch come per il blu/green deployment, ma il momento di coesistenza delle varie versioni è prolungato. Questo accade perchè talvolta processare una richiesta può richiedere un tempo molto lungo, quindi in questo modo si evita di interrompere la vecchia versione e lasciare in stand by il tutto per ore prima di passare alla nuova versione, quindi per un lasso di tempo le due versioni coesisteranno.

1.3 Deployable units

Virtualizzazione

Tendiamo a creare risorse virtuali che possono essere simulate e usate come se fossero vere tramite un hardware che riesce a virtualizzare diversi sistemi operativi.

Deployment del sistema software:

- cloud (VMs)
- cloud (containerS)
- bare metal
- dedicated server

VMs

Cosa sono le VM?

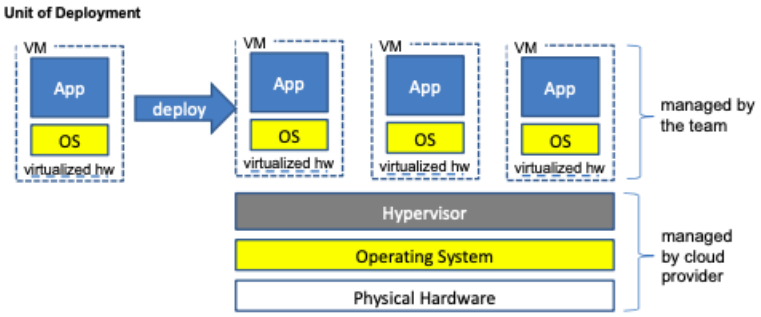
Una macchina virtuale (VM) è un'emulazione di un sistema informatico. In parole povere, rende possibile eseguire quelli che sembrano essere molti computer separati su hardware che in realtà è un computer.

I sistemi operativi (OS) e le relative applicazioni condividono le risorse hardware da un singolo server host o da un pool di server host. Ogni VM richiede il proprio sistema operativo sottostante e l'hardware è virtualizzato. Un hypervisor o un monitor di macchina virtuale è software, firmware o hardware che crea ed

esegue VM. Si trova tra l'hardware e la macchina virtuale ed è necessario per virtualizzare il server.

Dall'avvento della tecnologia di virtualizzazione a prezzi accessibili e dei servizi di cloud computing, i reparti IT grandi e piccoli hanno adottato le macchine virtuali (VM) come un modo per ridurre i costi e aumentare l'efficienza.

Ogni macchina virtuale ha un SO (=sistema operativo) che lavora con l'hypervisor che mette a disposizione diverse risorse.



Fare un update consiste nell'andare ad aggiornare le varie macchine virtuali. Possono esserci utenti differenti che accedono alle risorse sottostanti nel cloud avviando l'applicazione in modo differente; questo viene detto **multi-tenant**.

ContainerS

Cosa sono i containers?

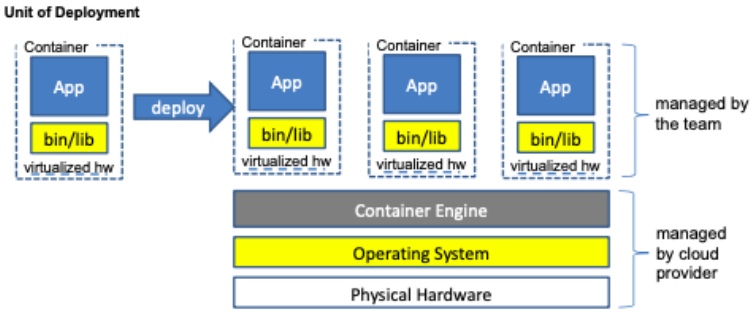
Con i containers, invece di virtualizzare il computer sottostante come una macchina virtuale (VM), viene virtualizzato solo il sistema operativo.

I containers si trovano sopra un server fisico e il loro sistema operativo host, in genere è Linux o Windows. Ogni containers condivide il kernel del sistema operativo host e, di solito, anche i binari e le librerie. I componenti condivisi sono di sola lettura. La condivisione delle risorse del sistema operativo come le librerie riduce notevolmente la necessità di riprodurre il codice del sistema operativo e significa che un server può eseguire più carichi di lavoro con una singola installazione del sistema operativo. I containers sono quindi eccezionalmente leggeri: misurano solo megabyte e richiedono solo pochi secondi per avviarsi. Rispetto ai container, l'esecuzione delle VM richiede pochi minuti e sono un ordine di grandezza maggiore di un container equivalente.

A differenza delle VM, tutto ciò che un container richiede è sufficiente da un sistema operativo, da programmi e librerie di supporto e da risorse di sistema per eseguire un programma specifico. Ciò significa in pratica che puoi mettere da due a tre volte più applicazioni su un singolo server con containers rispetto a quanto puoi con una VM. Inoltre, con i containers è possibile creare un ambiente

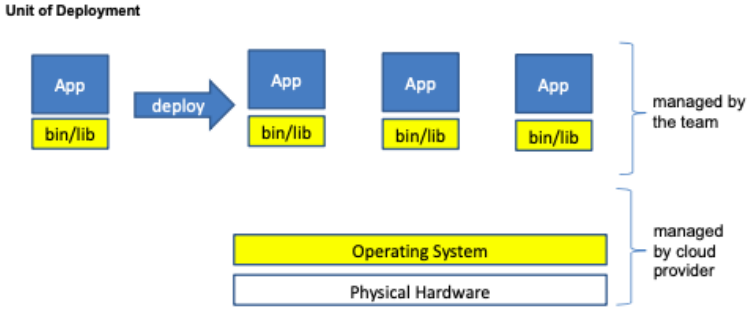
operativo portatile e coerente per lo sviluppo, il test e la distribuzione.

I container diventano delle mini macchine virtuali perchè al loro interno non hanno più il SO ma hanno solo l'applicazione che devono eseguire, così cloud riesce a spostarla senza costi elevati; aggiornarli o distruggerli è poco costoso.
Anche questo è un contesto **multi-tenant**.



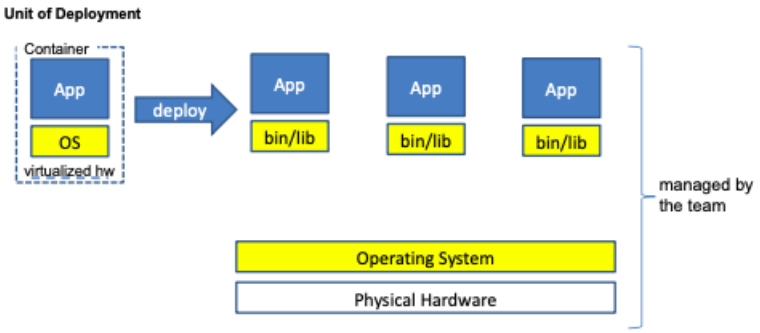
Bare metal

I **bare metal** offrono risorse hardware gestite dal cloud provider.
In questo caso sparisce l'elemento virtualizzazione ma ho una macchina virtuale su cui fare deployment dell'applicazione che voglio eseguire guadagnando l'accesso diretto alla macchina.
Questa è una soluzione **single-tenant**, cioè non c'è più la condivisione delle risorse.
Aumentano velocità, efficienza e costi dell'utilizzo del servizio.



Dedicated server

È come il bare metal ma il programmatore deve gestire il server e l'hardware.



Applicazioni mobile

Se l'app mobile ha servizi back-end si applica quanto detto fino ad ora.
Per quanto riguarda l'app sugli smartphone abbiamo che è l'utente a decidere se aggiornare l'applicazione oppure no.
L'elemento di automazione finisce con la parte di release.

1.4 Monitor

In ambiente cloud ci sono tante soluzioni che permettono di fare **monitoring** in modo efficace, come ad esempio lo stack ELK.

Il **monitor** è fatto da 3 componenti principali:

- elasticsearch
- ???
- kibana

Kibana è una dashboard che include tante visualizzazioni di dati raccolti.

ELK può definire anche altri meccanismi di allerta e quindi creare meccanismi di monitoring.

Tools

Uno degli elementi chiave di DevOps è l'automazione, rendendo automatici più procedimenti possibili dello sviluppo del software fino ad automatizzare anche gli update.

1.5 Risk management

Il fallimento di progetto di sviluppo software rallenta lo sviluppo di progetto e decreta il fallimento del progetto perchè non è stato messo sul mercato in tempo.
È necessario identificare e gestire/eliminare i rischi prima che questi diventino una minaccia per il successo del progetto.

Rischio: possibilità che ci sia un danno/infortunio.
Pericolosità del rischio: $RE = P(UO) * L(UO)$ probabilità di produrre il danno per l'entità del danno stesso: tanto più un rischio è probabile quanto più esso è serio.

Esempio: un progetto di sviluppo software fallisce perchè il signor Smith, che è l'unico ad avere conoscenze del modulo, lascia l'organizzazione.

Le aree di rischio possono produrre aree non soddisfacenti per:

- utenti
- sviluppatori
- manutentori

1.5.1 Cause dei rischi

Triggers: eventi che abilitano il rischio.
Per prevenire un rischio possiamo lavorare sui suoi triggers:

$$\text{triggers} > \text{rischio} > \text{unsatisfactory outcome}$$

È importante capire quali sono gli eventi accaduti prima del rischio su cui si sarebbe potuto lavorare e prevenire il rischio.

Esempio: il signor Smith è l'unico ad avere la conoscenza del modulo.

Classi di rischio:

- processo (problemi di sviluppo)
- prodotto (problemi di sicurezza, affidabilità, performance, etc...)

Cosa posso fare per gestire i rischi?

Risk assessment:

$$\text{Risk identification} > \text{Risk analysis} > \text{Risk prioritization}$$

Risk control:

$$\text{Risk management} - \text{planning} > \text{Risk monitoring} \leftrightarrow \text{Risk resolution}$$

Queste due fasi sono ciclicamente ripetute perchè durante lo sviluppo del software i rischi cambiano.

1.5.2 Risk assessment (valutazione del rischio)

Risk identification: come faccio ad identificare i rischi?

L'identificare i rischi è un processo legato all'esperienza dello sviluppatore perchè è molto difficile.

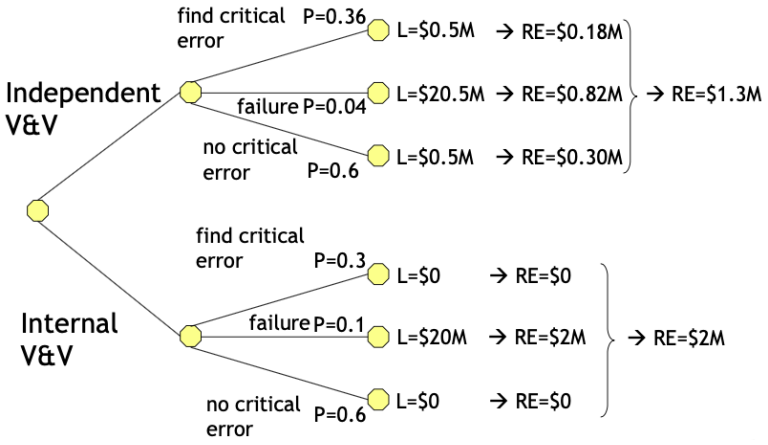
Un modo di farlo è quello di utilizzare delle *check-list* che includono un insieme di rischi plausibili e comuni e si scorre la lista chiedendosi se il rischio si applica o no al proprio progetto.
È importante chiedersi perchè quel dato rischio potrebbe essere un rischio specifico per il mio progetto. Un rischio è preso in considerazione solamente se c'è un motivo specifico per considerarlo.

Un altro modo di farlo sono riunioni di confronto tra gli sviluppatori.

Risk analysis

Attività legata all'esperienza.
Ci si può aiutare con modelli di stima dei costi, simulazioni di prototipi per capire con che probabilità accadono certi eventi, uso di check-list, analogie con altri progetti, etc...

Decision analysis: analisi volta a prendere decisioni; queste sono decisioni che mirano a minimizzare certi rischi.
Per questo è utilizzato un **decision tree**:



In questo caso è preferibile la prima ipotesi perchè ha una *RE* (pericolosità del rischio) inferiore.

Esempio: le porte del treno possono aprirsi mentre il treno è in movimento. Per fare in modo che questo non si verifichi mai si deve ragionare sulle cause del rischio.

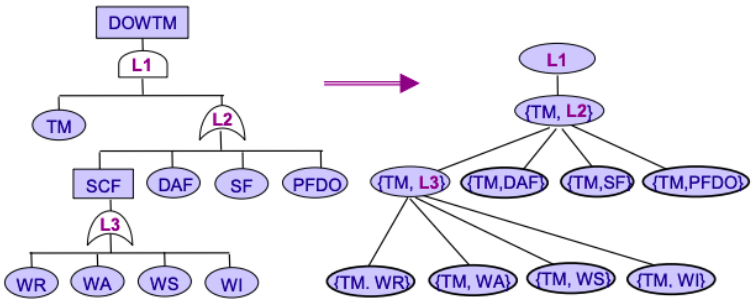
Si utilizza uno strumento chiamato **risk tree**:

- alla radice c'è il rischio

- failure node: ogni nodo rappresenta una condizione che si può verificare

La scomposizione delle condizioni è guidata da link di tipo AND oppure OR.

Si vuole fare in modo che nessuna combinazione si verifica; per farlo si esegue un'operazione chiamata *cut-set tree derivation*: scendendo verso il basso si riportano le combinazioni di eventi che possono produrre il fallimento e successivamente si vanno a calcolare le combinazione di eventi foglia. Il cut-set tree rappresenta, quindi, combinazioni di AND minimali di eventi di guasto non scomponibili, sufficienti a causare il rischio.
Esempio: cut-set tree derivation



Risk prioritization: da quali rischi partire?

Si stabilisce da dove partire utilizzando la RE. Se non si è certi della stima si possono fare delle stime in intervalli.
Quando abbiamo tutte le stime possiamo calcolare la RE e plottare i valori trovati in un grafico, con la probabilità sull'asse delle x e il danno sull'asse delle y.

1.5.3 Risk control

Risk management-planning: cosa facciamo per gestire un rischio?

Il **risk control** è un'attività legata a cataloghi e check-list.

Strategie:

- lavorare per ridurre la probabilità di un rischio: bisogna ragionare sul rischio stesso e sulle cause per capire come ridurne la probabilità
- eliminazione del rischio (ridurre a 0 la probabilità del rischio), ad es: porte del treno aperte solamente dal software
- ridurre il danno provocato dal rischio: ridurre la probabilità che il danno si verifica quando il rischio accade, ad es: aggiungere allarme della porta che si apre quando si aprono accidentalmente le porte del treno evitando che la gente cada
- ridurre la conseguenza del rischio

- mitigare conseguenze/danno del rischio

Come confronto le contromisure e quali di esse attivo?

Risk-reduction leverage

Calcolo di quanto una certa contromisura può ridurre un certo rischio. La riduzione di un rischio aggiunta atraverso contromisure è stabilita in base al costo delle contromisure.
Questo calcolo è data dalla seguente formula:

$$RRL(r, cm) = (RE(r) - RE(r/cm))/cost(cm)$$

con cm=contromisura, r=rischio

Si seleziona la contromisura con migliore RRL.

Defect detection prevention

Confronta le varie contromisure in modo quantitativo facendo un confronto indiretto. Posso lavorare sulle singole contromisure e i singoli rischi, ma alla fine si ragiona contemporaneamente sulle varie contromisure.

1. produzione della risk impact matrix - impatto del rischio sugli obiettivi del progetto.
   impact(r,obj):

   - 0 no impact
   - 1 totale non raggiungimento dell'obiettivo indicato

   Faccio una tabella in cui per ogni rischio stimo l'impatto su ogni obiettivo.
   Si può anche calcolare la criticità di un rischio rispetto a tutti gli obiettivi indicati.
   La criticità sale se sale l'impatto e se sale la probabilità del rischio.
   Si può pure calcolare la perdita di raggiungimento degli obiettivi qualora tutti i rischi si verificassero.

   $$\text{criticality(r)} = \text{likelihood(r)} \times \Sigma_{\text{obj}}(\text{impact(r, obj)} \times \text{weight(obj)})$$

   $$\text{loss(obj)} = \text{weight(obj)} \times \Sigma_{\text{r}}(\text{impact(r, obj)} \times \text{likelihood(r)})$$

2. calcolo della criticità del rischio per stabilire le varie contromisure da adottare (countermeasure effectinevess matrix)
   Reduction(cm,r): quanto le contromisure riducono il rischio

   - 0 rischio non ridotto
   - 1 rischio eliminato

   Si possono calcolare due grandezze:

- quanto ogni rischio è ridotto se tutte le contromisure sono attivate - *combineReduction(r)*
- effetto di ogni contromisura sulla'insieme dei rischi considerato - *overallEffect(cm)*

$$\text{combinedReduction}(\text{r}) = 1 - \Pi_{\text{cm}}(1 - \text{reduction}(\text{cm}, \text{r}))$$

$$\text{overallEffect}(\text{cm}) = \Sigma_{\text{r}}(\text{reduction}(\text{cm}, \text{r}) \times \text{criticaly}(\text{r}))$$

3. rapporto tra l'effetto di ciascuna contromisura e il suo costo e vado a selezionare quelle con rapporto migliore

Risk monitoring and resolution

Management plan: descrive quali contromisure attivare e come in relazione ai rischi osservati.
Contingency plan: cosa fare nel momento in cui il rischio accade.

Risk monitoring

Piano per monitorare continuamente i vari rischi.
Queste attività sono costose, quindi c'è una priorità da seguire di top-10 risk items.

Recap:

1.6 CMMI - Capability Maturity Model (I)

Maturità dei processi

Probabilità di portare a termine con successo un progetto di cui una buona parte dipende dalla maturità del processo, ovvero dal grado di controllo delle azioni che vado a svolgere per realizzare il mio progetto.

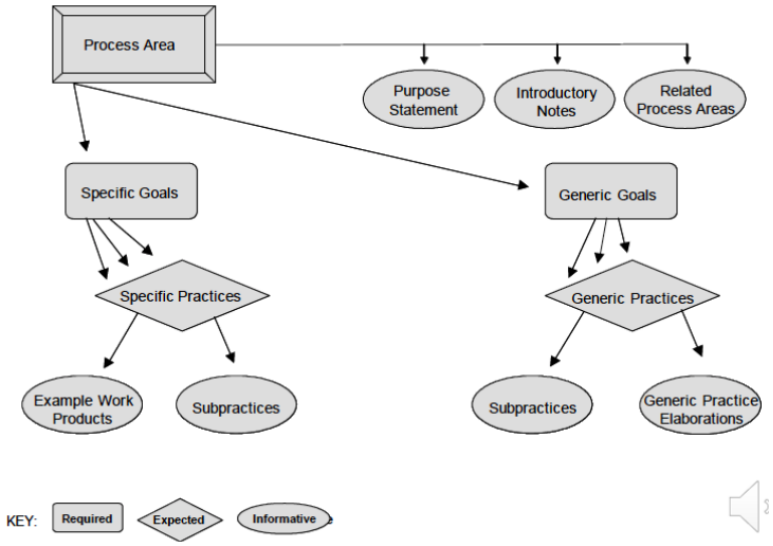
- progetto immaturo:

  - azioni indefinite
  - azioni incontrollate

- progetto maturo:
  - tutte le attività sono ben definite
  - tutte le attività sono controllate

CMMI: un migliore processo porterà ad un prodotto migliore, e lo fa definendo una serie di azioni e pratiche standard che ci permettono di governare tutti gli aspetti relativi ad un processo software.

Organizzazione dello standard



1.6.1 Process area

Tutti i modelli CMMI contengono 16 aree di processo principali. Queste aree di processo coprono concetti di base che sono fondamentali per il miglioramento del processo in qualsiasi area di interesse (ovvero acquisizione, sviluppo, servizi). Del materiale nelle process area può essere adattato per affrontare una specifica area di interesse. Di conseguenza, il materiale nelle aree centrali del processo potrebbe non essere esattamente lo stesso per tutte le process area.

La **process area** racchiude al suo interno una collezione di pratiche organizzate secondo obiettivi e riguarda una particolare area del processo di sviluppo software.
Tutte le **process area** sono descritte in modo omogeneo; tale descrizione descrive l'obiettivo della data process area.

Ogni process area ha obiettivi specifici e generici:

- generici: comuni a tutte le process area, ovvero rappresentano quanto sia ben organizzata e definita nel corso del processo
- specifici: caratterizzano la process area, ovvero al suo interno si trova una lista di pratiche, cioè azioni che se svolte ci permettono di raggiungere quell'obiettivo specifico.

Le pratiche possono essere suddivise in sottopratiche.
Anche gli obiettivi generici possono essere scomposti in sottopratiche.

CMMI possiamo usarlo come ispirazione per le azioni da svolgere o per confrontare il nostro processo con lo standard CMMI per valutarne il grado di maturità.

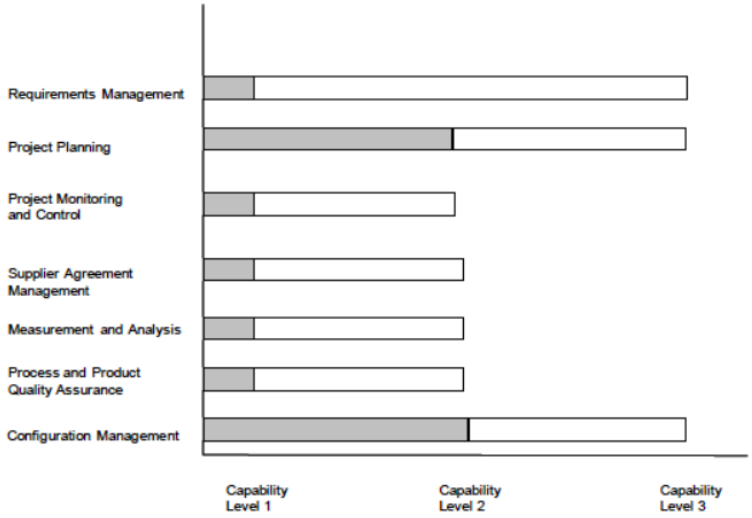
Ci sono due linee di miglioramento rispetto allo standard.

- capability levels (0-3): stabilisce quanto bene si sta gestendo una process area
- maturity levels (1-5): cattura il livello raggiunto dall'intero processo di sviluppo ragionando su tutte le process area attivate

Gli obiettivi generici impongono controllo e documentazione delle pratiche svolte.

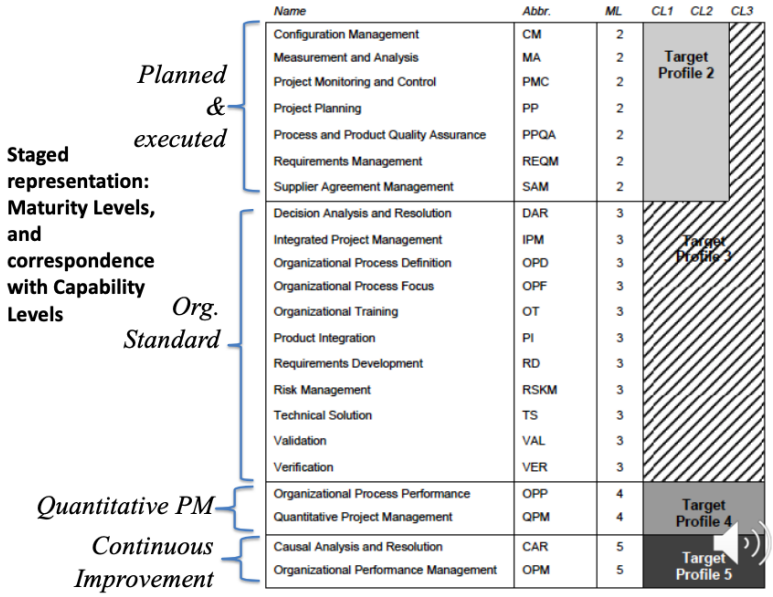
Obiettivi generici

1. pratiche specifiche
2. svolgimento di pratiche che hanno a che vedere con locazione di risorse, plan del team, etc...
3. versioni di tailoring



Level	Maturity Levels	Description
Level 1	Initial	Ambiente ad hoc e caotico, non stabile che supporti il ??processo
Level 2	Managed	Pianificato ed eseguito in conformità con le politiche; persone qualificate; uscite controllate, che vengono monitorate, controllate e riviste; pratiche mantenute anche in caso di stress; stato dei prodotti di lavoro visibili al management in punti definiti
Level 3	Defined	Standard organizzativo, adattamento degli standard sui progetti specifici sulla base di linee guida di sartoria; descrizioni più rigorose
Level 4	Quantitatively Managed	Stabilire obiettivi quantitativi per la qualità e le prestazioni del processo e utilizzarli come criteri nella gestione dei progetti
Level 5	Optimizing	Miglioramento continuo delle prestazioni del processo attraverso processi incrementali e innovativi e miglioramento tecnologico. L'effetto dei miglioramenti viene valutato quantitativamente

Nell'immagine successiva, per passare dal livello 2 al livello 3 devo aver attivato tutte le capability level del secondo livello. A questo punto posso dire che il livello di maturità del mio processo è 3.
Se tutte le process area del mio processo sono 3 allora il maturity level sarà 5. Se tutte le process area *org. standard* sono 3 il ML sarà 3, se le process area *quantitative PM* sono 3 il ML sarà 4, se tutte le process area *continuous improvement* sono 3 il ML sarà 5.



2 Requirements engineering

Ingegnerizzazione dei requisiti: comprendere come la soluzione software si deve comportare per risolvere il problema che stiamo affrontando. Per soddisfare i requisiti di sistema si devono rispettare i requisiti del software, le assunzioni e le proprietà di dominio.

Ci concentriamo su due elementi:

- problema
- contesto in cui il problema si verifica

The problem world and the machine solution

World: per risolvere un problema c'è l'esigenza nel mondo di produrre il sistema che vogliamo andare a risolvere.

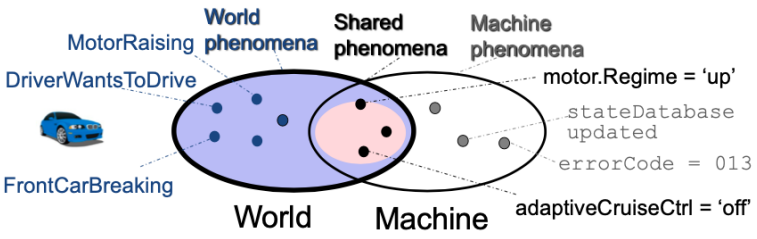
- esseri umani
- componenti fisici

contribuiscono alla definizione del problema.

Machine: macchina che andiamo a realizzare (software+hardware)
Ingegnerizzazione dei requisiti: capire qual è il problema e qual è l'effetto che la macchina che andiamo a realizzare ha sul problema. Capire anche quali sono le assunzioni e le proprietà da prendere in considerazione.
A noi interessa comprendere il mondo, il problema e l'interazione che la macchina che andiamo a realizzare ha col problema.

Come vedremo nell'immagine seguente:

- l'ambiente include le persone che usano il sistema
- l'ambiente è parte del sistema, quindi ha requisiti del sistema ma non ha i requisiti del software
- i requisiti del software sono un sottoinsieme dei requisiti del sistema



- **system as-is**: sistema che esiste già e che risolve già il problema (in modo non soddisfacente)
- **system to-be**: sistema che andremo a realizzare, ovvero è il system-as-is quando la nuova macchina descritta nei requisiti sarà operativa

Il System as-is ci dà tantissime informazioni sul problema che vogliamo andare a risolvere sviluppando il nostro System to-be.

Def ingegnerizzazione dei requisiti: attività che hanno come obiettivo quello di esplorare, valutare, documentare, consolidare, rivisitare e adattare l'obiettivo, le capacità, qualità, vincoli e assunzioni su un software che andiamo a realizzare (system to-be) sulla base del system as-is. L'output di attività di requirements engineering può essere rappresentato da un documento di specifica di requisiti in cui si documentano tutti i vincoli e le assunzioni che il sistema deve realizzare, oppure un insieme/elenco di requisiti che descrive come si deve comportare il sistema che andiamo a realizzare.

1. requirements engineering: vogliamo capire qual è il sistema che dev'essere sviluppato e cosa deve fare. Con le seguenti attività cerchiamo di sviluppare il sistema nel modo migliore
2. software design
3. software implementation
4. software evolution

- bisogna ragionare su diverse versioni del sistema che voglio andare a realizzare (as-is, to-be) e devo cercare di prevedere il comportamento del sistema to-be-next. L'ambiente in cui si lavora è un ambiente ibrido (con esseri umani, leggi, regole, etc...) per sviluppare il sistema nel modo più corretto devo comprendere il contesto

- ci sono diversi aspetti che devono essere trattati, quali funzionali e qualitativi (non funzionali), cioè con quali qualità le funzionalità devono essere erogate (efficienza, sicurezza, performance, etc...) richieste
- tante attività tecniche tra loro collegate: rischi, quality assurance, priorità nello sviluppo delle
- ci sono molti livelli di astrazione, come ad esempio obiettivi strategici o aspetti operazionali
- ci sono più stakeholder (=persone interessate al progetto) e noi dobbiamo risolvere le loro funzionalità, etc...

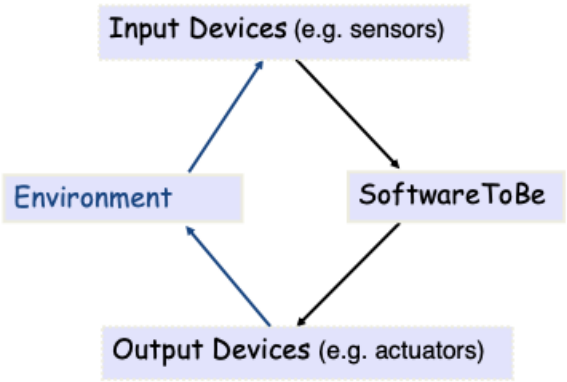
2.0.1 Tipi di requisiti

Descriptive statements: requisiti che di fatto sono non negoziabili/modificabili, ovvero rappresentano dei comportamenti che vengono dalle leggi del mondo (dominio) su cui la macchina andrà a lavorare.
Prescriptive statements: hanno una negoziabilità, ovvero riguardano comportamenti che il sistema deve avere, ma il requisito è modificabile perchè è lo sviluppatore che lo decide (es posso decidere io il limite di libri da prendere in prestito).

I requisiti che andiamo ad indicare possono differire anche per gli elementi che tengono in considerazione, ad esempio gli elementi di un sistema software e come questo interagisce col mondo:

- come si comporta l'ambiente per capirne il funzionamento
- non dicono mai come si comporta internamente il sistema software che sarà definito in una fase successiva
- proprietà dell'ambiente/dominio applicativo che non può essere modificabile
- assunzioni, ovvero chi realizza il software fa delle assunzioni su com'è fatto l'ambiente (occhio a non fare assunzioni troppo restrittive)

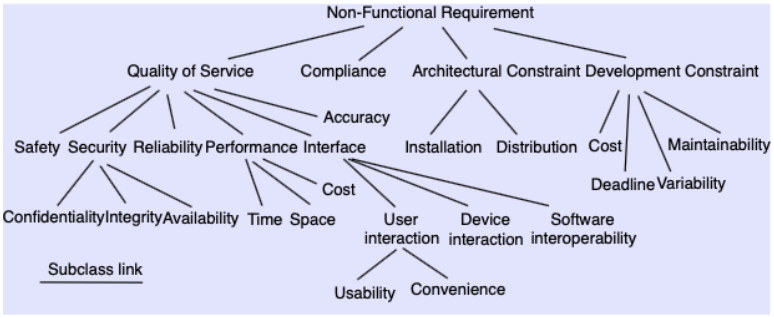
Esempio: Parnas95



2.0.2 Categorie dei requisiti

- **requisiti funzionali**: indicano le funzionalità che un sistema deve implementare, ovvero cosa dev'essere in grado di fare (ogni sistema ha le sue)
- **requisiti non funzionali**: indicano delle qualità o dei vincoli sulle funzionalità, quindi sul requisito funzionale

L'insieme di qualità su cui ragionare quanto dobbiamo descrivere i requisiti non funzionali, questi requisiti tendono ad essere gli stessi, quindi esistono delle tassonomie, ad esempio:



Se qui vengono trovati delle caratteristiche interessanti per il nostro progetto li andremo ad aggiungere ai nostri requisiti non funzionali.

Requirements qualities

Qualità che l'insieme dei requisiti deve soddisfare:

- completezza di ciò che descriviamo
- consistenza, cioè requisiti tra loro consistenti, ovvero non devono essere requisiti che entrano in contraddizione tra loro
- non ambigui, cioè non basato su interpretazioni
- misurabile
- fattibili
- comprensibile
- modificabile
- mantenibile nel tempo
- tracciabile, cioè capire quali sono stati gli artefatti come conseguenza dell'interpretazione di un requisito

Errori nella scelta dei requisiti

- omissioni: non siamo riusciti ad identificare tutti i requisiti
- contraddizioni: requisiti in contrasto tra loro
- inadeguatezza: non sono indicati nel modo adeguato
- ambiguità: requisiti che danno informazioni che si prestano a diversi tipi di interpretazioni
- non misurabili: possono essere problematici da gestire e riguardano anche i requisiti non funzionali (user-friendly)

Difetti nella scelta dei requisiti

- troppa specificità: descrivo comportamenti interni del mio software (cosa che non devo fare perchè io nei requisiti devo dire QUALE funzionalità dev'essere realizzata, non come)
- non implementabili per vincoli temporali o di budget
- non leggibili/comprensibili: creare acronimi troppo complessi
- struttura troppo lunga: struttura troppo complessa da leggere (es testi in prosa)
- evitare di fare riferimento a requisiti che si descrivono successivamente
- evitare di non aver definito nel momento giusto determinati concetti: parlo di un concetto che non è mai stato definito
- struttura non modificabile: la struttura dei requisiti deve essere facilmente modificabile
- razionalità: dev'essere chiaro perchè quel requisito dev'essere incluso e il lettore uò mettere in discussione l'insieme dei requisiti definiti; si perderebbe tempo a rifare i ragionamenti per arrivare al motivo che ci ha portato ad includere quel requisito

2.0.3 Processo di ingegnerizzazione dei requisiti

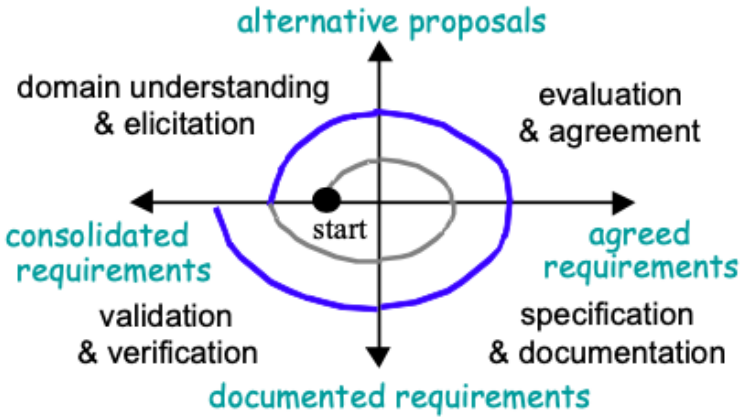
1 fase - Domain understanding & elicitation: si studia e si comprende il dominio applicativo dell'applicazione che si sta andando a realizzare, bisogna comprendere anche l'organizzazione del sistema che si sta andando a realizzare. In questa prima fase si identificano gli stakeholder (=individui/organizzazione che ha interessi rispetto al sistema che sto andando a realizzare) - bisognerà poi interagire con gli stakeholder.
Elicitation: si vuole capire nel dettaglio l'obiettivo del progetto, i vincoli, gli obiettivi software e le assunzioni sull'ambiente - queste informazioni si capiscono interagendo con gli stakeholder.

2 fase - evaluation & agreement: si identificano le richieste in conflitto tra loro e si valutano le ipotesi alternative che risolvono questi conflitti. Si identificano i rischi del prodotto modificando i requisiti in base ad un ragionamento basato sui rischi. Si valuta quali requisiti vanno sviluppati prima di altri.

3 fase - specification & documentation: traduciamo i requisiti in un documento/archivio con la loro definizione precisa (di requisiti, componenti software, proprietà, idee future, assunzioni, etc...)

4 fase - validation & verification: verifica della correttezza dell'insieme dei requisiti raccolti. Ci sono delle attività che si possono svolgere per individuare eventuali errori.

Questo diventa un processo iterativo a spirale:



2.0.4 Selezione degli stakeholder

Tecniche di elicitation: tecniche che permettono di scoprire i requisiti che un progetto deve soddisfare. Per scoprire questi requisiti buona parte delle azioni da svolgere hanno a che vedere con gli stakeholder del progetto. Conoscendo chi nutre interesse rispetto al progetto si potranno prendere in considerazione gli interessi e quindi realizzare un progetto che realizza tutti gli interessi degli stakeholder.

Aspetti rilevanti per la selezione:

- organizzazione che ci ha chiesto di sviluppare il progetto
- posizione che lo stakeholder occupa all'interno dell'organizzazione
- livello di esperienza del dominio applicativo
- livello di esposizione che il sistema deve risolvere
- influenza che avrà il progetto
- obiettivi personali degli stakeholder e conflitti di interesse

Individuazione degli stakeholder: fatta attraverso attività di brain storming che puntano a dare risposta ad una serie di domande che ci permettono di individuare gli stakeholder, come per esempio:

- who is affected positively and negatively by the project?
- who has the power to make it succeed (or fail)?
- who makes the decisions about money?
- who are the suppliers?
- who are the end users?
- who has influence over other stakeholders?
- who could solve potential problems with che project?
- who is in charge of assigning or procuring resources or facilities?
- who has specialist skills which are crucial to the project?

Se dovessimo dimenticare uno stakeholder significa che noi sviluppiamo il progetto senza il suo punto di vista e se questo si presenterà tardi questo può significare spenderci tanti soldi per riprogettare tutto. Questo è un rischio maggiore rispetto a quello di non trovare dei requisiti.
Con gli stakeholder si va anche ad interagire per avviare le attività di richiesta/-scoperta di requisiti.

Ostacoli per la conoscenza di informazioni e interazione con stakeholder:

- la conoscenza di un progetto sarà distribuita tra tutti gli stakeholder
- ostacolo di comunicazione e conoscenze tra noi informatici e stakeholder
- conoscenza che non viene comunicata esplicitamente
- difficoltà di accesso a informazioni sensibili/segrete utili per il progetto
- fiducia con gli stakeholder
- le condizioni cambiano

Interazione con gli stakeholder

Capacità comunicativa importante

- utilizzare la giusta terminologia
- abilità di andare diretti al punto e analizzare i casi/situazioni in modo accurato
- instaurare le giuste relazioni personali di fiducia che aiuterà nella scoperta dei requisiti

Knowledge reformulation: pratica che ci consente di verificare se abbiamo compreso nel modo corretto il dominio; è bene riformulare e ripresentare l'informazione acquisita allo stakeholder.
Talvolta può essere complicato gestire tutti gli stakeholder perchè sono davvero tanti.
È bene, quindi, considerare due variabili principali:

- potere decisionale dello stakeholder (power)
- livello di interesse dello stakeholder (interest)

Strategia:

Power	**Interests**	**Strategy**
High	High	Fully engage: engaged regulary and managed closely most effort to keep them satisfied
High	Low	Keep satisfied: keep informed and satisfied, without overloading them with information
Low	High	Keep satisfied: keep informed and regulary consult them to gather detailed information and also make sure they do not have major issues with the project
Low	Low	Minimum effort: inform them with general information. Monitor them in case their power or interest increase

2.1 Artefact-driven elicitation techniques

Tecniche che possiamo sfruttare per scoprire i requisiti del progetto.

2.1.1 Background study

Attività che produce le conoscenze necessarie per interfacciarsi con gli stakeholders, ovvero significa collezionare, leggere e sintetizzare una conoscenza che viene da documenti che riguardano (senza coinvolgere lo stakeholder):

- organizzazione interessate al progetto
- dominio applicativo
- funzionamento system as-is

Non si fanno domande allo stakeholder riguardo ad informazioni che potremmo imparare in altro modo.

2.1.2 Data collection

Possiamo ottenere alcune informazioni non attraverso documenti ma svolgendo un'attività che consiste nel raccogliere nuovi dati che ci permettono di capire qualcosa sul sistema as-is o to-be.

C'è un limite alla scalabilità che questa attività di background study può avere perchè può essere complicato e costoso da fare. È importante mettere in pratica l'utilizzo della *meta-knowledge*, ovvero selezionare parti di documenti rilevanti per leggere solo la parte di documentazione utile evitando di spendere tempo sul resto.

2.1.3 Questionnaires

Utili per ottenere informazioni considerando da una popolazione ampia di stakeholder preparando un questionario di domande che devono essere scritte e formulate in modo molto chiaro:

- domande a scelta multipla (tante domande chiuse e poche domande aperte perchè a rispondere alle domande aperte ci si mette molto di più e nessuno ha voglia/tempo)
- pensare bene alle domande
- discriminare i vari livelli di risposta affinché sia chiaro differenziare l'importanza delle varie opzioni
- non mettere troppe opzioni intermedie per non creare confusione, chiedere preferenze
- non condizionare le risposte attraverso la domanda (biased) (es: "ti piace questa schifosa interfaccia?")
- è importante inserire domande che sono una la negazione dell'altra per capire se chi sta rispondendo è attento oppure no (ad una certa distanza le une dalle altre)

Voler sottomettere un questionario non implica per forza poterlo sottomettere, infatti alcune volte può essere difficile somministrarlo alla popolazione.

2.1.4 Storyboards

Narrazioni/storie di utilizzo del sistema, fatte attraverso degli esempi, formate dalla sequenza di snapshots (=sequenze di immagini, sketch, etc...) al fine di guidare la scoperta di requisiti.

Possono essere create in modo:

- **passivo**: facciamo la narrazione/storyboard che presentiamo allo stakeholder per verificare che la nostra comprensione è corretta

- **attivo**: lo stakeholder contribuisce alla costruzione della narrazione

Hanno come obiettivo il capire se si è compreso correttamente il system as-is e to-be.

2.1.5 Scenari

Descrivono attraverso una sequenza di interazioni degli scenari di utilizzo del sistema as-is o to- be.
L'utilizzo di esempi è un modo semplice di comunicare e concordare il comportamento che dovrebbe avere un sistema.

Gli scenari possono essere:

- **positivi**: come il sistema si dovrebbe comportare
- **negativi**: cosa il sistema non dovrebbe fare

Gli scenari positivi si suddividono in:

- **normali**: situazione attesa che accade più di frequente
- **anormali**: situazioni eccezionali che si possono verificare

Esempi e controesempi è un modo utilizzassimo per comunicare con lo stakeholder e capire il comportamento del sistema.

Esistono dei limiti nell'utilizzo degli scenari:

- hanno una visione parziale
- esplosione combinatoria
- gli esempi possono deviare la comprensione del comportamento del sistema
- dettagli irrilevanti
- scenari difficili da mettere assieme/difficili da collegarli (attraverso la comunicazione con gli stakeholder)

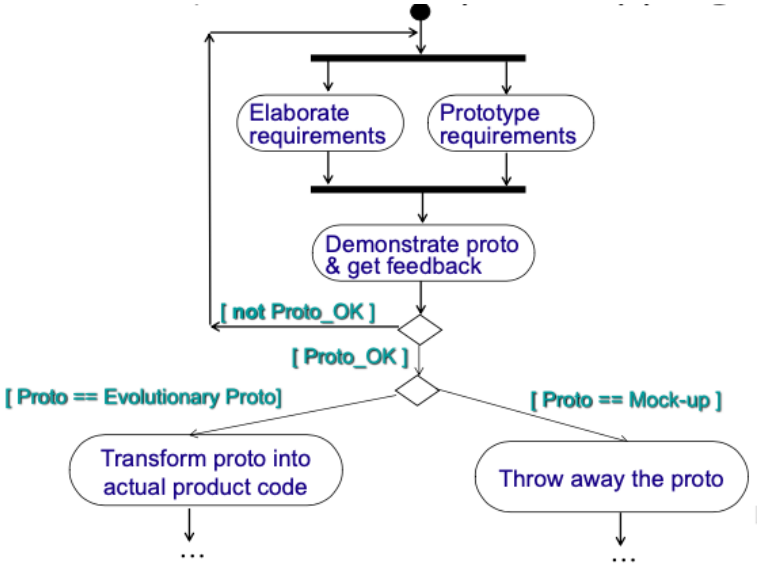
2.1.6 Prototipi e mock-ups

L'obiettivo è quello di controllare l'adeguatezza di un requisito che può essere presentato visualizzando direttamente il risultato che esso porterebbe sull'applicazione presentando degli esempi del software in azione (l'app non è ancora implementata, sono solo degli esempi!).

Perchè costruire mock-ups?

- capire se abbiamo compreso correttamente determinate funzionalità
- usabilità della UI

La differenza tra prototipo e mock-up è che, sebbene entrambi vengano utilizzati per ottenere un feedback dalle parti interessate, il prototipo viene poi trasformato nell'applicazione, mentre il mock-up viene buttato via.



Pro:

- sensazione di concretezza visuale attraverso l'utilizzo di mock-up e dimostrazioni pratiche trasmessa allo stakeholder

Contro:

- costo di produzione di un mock-up
- siccome non c'è logica applicativa ed è solo una simulazione, alcune funzionalità potrebbero creare aspettative troppo alte (ad es non rispecchiano i tempi di esecuzione della realtà)

2.2 Stakeholder-driven elicitation techniques

C'è interazione diretta con lo stakeholder.

2.2.1 Interviews

Selezionare lo stakeholder dal quale vogliamo acquisire informazioni (dev'essere chiaro perchè abbiamo selezionato proprio lui, dev'essere chiaro l'obiettivo dell'intervista e quali informazioni vogliamo estrarre), bisogna organizzare la riunione, fare l'intervista registrando/scrivendo le risposte, produrre il report delle risposte e infine sottomettere tale report all'intervistato per avere conferma di aver scritto bene le risposte.
Può essere fatta ad uno o più stakeholder.
Questa è una tecnica costosa, quindi possiamo intervistare poche volte.

Tipi di intervista:

- **strutturata**: abbiamo predeterminato tutte le domande da sottoporre allo stakeholder per raggiungere il nostro obiettivo predeterminato
- **non strutturata**: discussione libera con lo stakeholder su system as-is, problematiche e possibili soluzioni

Normalmente le interviste svolgono entrambe le parti (strutturata/non strutturata).

Giuda sulla preparazione delle interviste

- arrivare preparati costruendo un bel background study
- programmare domande apposta per lo stakeholder intervistato tenendo in considerazione anche il ruolo e l'attività che l'intervistato svolge (sto intervistando qualcuno perchè sono interessata al suo punto di vista)
- mantenere sempre il controllo dell'intervista non divagando
- mantenere un buon rapporto mettendo l'intervistato a proprio agio rompendo il ghiaccio con chiacchiere informali e domande semplici
- cercare di apparire come una persona affidabile con la quale l'intervistato può confidarsi svelando i problemi che poi dovremo risolvere
- chiedere "perchè" (è molto importante conoscere la motivazione razionale del perché qualcuno mi sta chiedendo di fare qualcosa)

2.2.2 Osservazioni

È più importante ed immediato descrivere procedure piuttosto che spiegarle a parole.
Esistono procedure basate sull'osservazione (scoperta del system as-is) e osservando impariamo e capiamo cogliendo i problemi e le parti più efficaci. Queste osservazioni possiamo utilizzarle per influenzare il system to-be.

Le osservazioni possono essere:

- **attive**: divento un team member svolgendo il task che svolge lo stakeholder per comprenderne meglio il comportamento
- **passive**: noi guardiamo e non produciamo interferenze (osservatori esterni)
  - protocol analysis
  - ethnographic studies

Pro:

- rivelare un insieme di requisiti che altrimenti rimarrebbe ignoto

Contro:

- operazione lenta perchè dobbiamo fare attività sul luogo, come scrivere, osservare, etc...
- inaccurato (non possiamo osservare senza informare, quindi questo potrebbe cambiare il proprio modo di lavorare sapendo di essere osservati)
- scoperta del funzionamento del system as-is che non ci aiuta col system to-be

2.2.3 Group session

Tecniche utili per la risoluzione di conflitti. Consiste in un workshop distribuito in uno o più giorni in cui c'è una discussione tra più partecipanti scelti che innescano una discussione che permette di comprendere meglio come dovrà essere il system to-be.

- strutturate: ogni partecipante ha un ruolo ben determinato e ognuno contribuisce in funzione del proprio ruolo portando il proprio punto di vista. Questo serve per ragionare sui requisiti di più alto livello perchè sono comuni a tutte le figure e ragionare su conflitti e discordanze
- non strutturate (brainstorming): i partecipanti non hanno un ruolo stabilito. Questo è sviluppato in due fasi:
  - generazione delle idee: tutti i partecipanti espongono le proprie idee su come risolvere il conflitto
  - valutazione: vengono analizzate tutte le idee prodotte e valutate per costo, fattibilità, etc... così da arrivare ad una visione condivisa degli approcci da utilizzare

Pro:

- facile risoluzione di conflitti
- sfruttare inventiva dei partecipanti

Contro:

- aspetti gestiti con attenzione
- è difficile coinvolgere figure spesso occupate
- difficile gestire partecipanti con indole più da leader
- la discussione potrebbe divergere e non convergere

2.3 Requirements documentation

Definisco obiettivi, concetti, assunzioni, responsabilità, razionale e indicazioni delle varianti e delle evoluzioni previste per il mio software.
Si deve definire una struttura coerente, ovvero un documento, contenente tutto ciò.

Come documentare i requisiti

Utilizzo del linguaggio naturale in modo non ristretto. Utilizziamo la prosa per descrivere i nostri requisiti. Il linguaggio naturale ha un'espressività illimitata per descrivere qualsiasi requisito. Ovviamente questo è fatto con delle regole sulla struttura del documento.
Bisogna stare attenti all'utilizzo del linguaggio naturale perchè nasconde ambiguità, ovvero il lettore potrebbe comprendere qualcosa di diverso rispetto a quello che intendevamo dire noi.

Struttura linguaggio naturale

- **regole locali**: come vado a descrivere il singolo requisito
- **regole generali**: organizzazione dell'intero documento

Nota: il termine *terminology clash* è usato per indicare dei sinonimi, ovvero usa parole diverse per esprimere lo stesso concetto. Il termine *designation clash*, invece, è utilizzato per esprimere omonimie, ovvero utilizza un solo termine per esprimere più concetti.

Regole stilistiche

Quando specifichiamo dei requisiti dobbiamo:

- identificare chi leggerà i nostri requisiti. Quello che scriviamo potrebbe essere destinato a stakeholder privi di conoscenza tecnica oppure a sviluppatori con conoscenza tecnica
- spiegare cosa sto per descrivere, fare, specificare prima ancora di descriverlo, farlo o specificarlo (utile per contestualizzare)
- motivare (prima di farle) le scelte che si fanno e poi fare una sintesi delle scelte e dei requisiti facendo il punto di come il sistema deve comportarsi
- mi assicuro di aver definito qualsiasi concetto da me utilizzato
- mi assicuro che ciò che sto scrivendo sia comprensibile e rilevante per un eventuale lettore
- per ogni frase/elemento/requisiti devo indicare un solo requisito/assunzione/proprietà di dominio non mescolando tra loro più concetti. Questo può rendere difficoltosa la lettura e la comprensione del documento
- distinguo ciò che è "obbligatorio" da ciò che è "desiderabile" in termini di requisiti
- evito utilizzo eccessivo di acronimi e gergo informatico
- uso degli esempi per chiarire dei concetti (funzionano sempre e sono sempre utili)

- quando possibile utilizzare diagrammi e illustrazioni visuali/grafiche per spiegare comportamenti del software (molto utili)

Notare che tutte queste regole si applicano ad un documento generico, non solo specifica di requisiti!!

Regole locali

Utilizzo di un template per la specifica di requisiti. Bisogna descrivere tutti i requisiti nello stesso modo - non descrivere un requisito in modo diverso dall'altro. Il template va definito per l'organizzazione del progetto.

Un esempio di template può essere una tabella che riempie tutti i seguenti campi:

- identifier (identificativo del requisito)
- category (funzionale/non funzionale/dominio, etc...)
- specification (formulazione del requisito)
- fit criterion (comprendere se il requisito è stato implementato bene)
- source (fonti di eliminazione che hanno portato alla def di quel requisito)
- rationale (ragione per cui quel requisito esiste)
- interaction (interazione con altri requisiti)
- priority (priorità di realizzazione del requisito)
- stability, commonality (mutazione del requisito in futuro)

Fit criterion

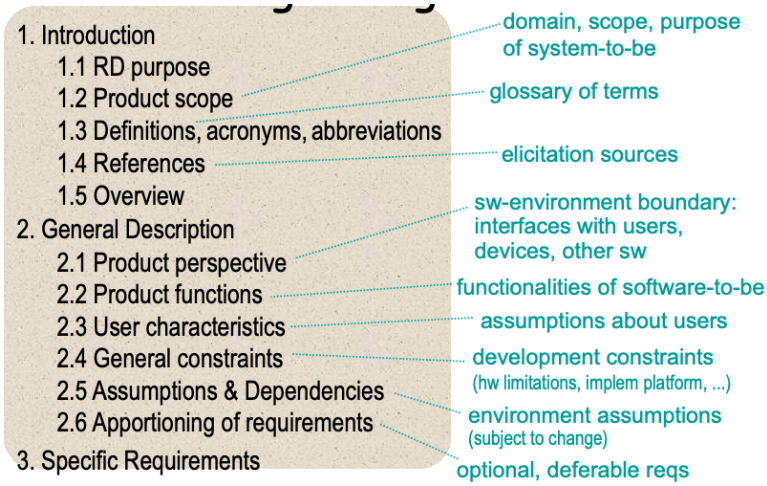
Il **fit-criterion** specifica come soddisfare quantitativamente un requisito, ovvero serve a stabilire in modo oggettivo se quel requisito è stato rispettato oppure no. Questo è critico per requisiti non funzionali per cui la misura può non essere rispettata.

Esempio: the schedule meeting dates shall be convenient to partecipanti. **fit criterion**: scheduled dates should fit the diary constraints of at least 90% of invited participants in at least 80% of cases.

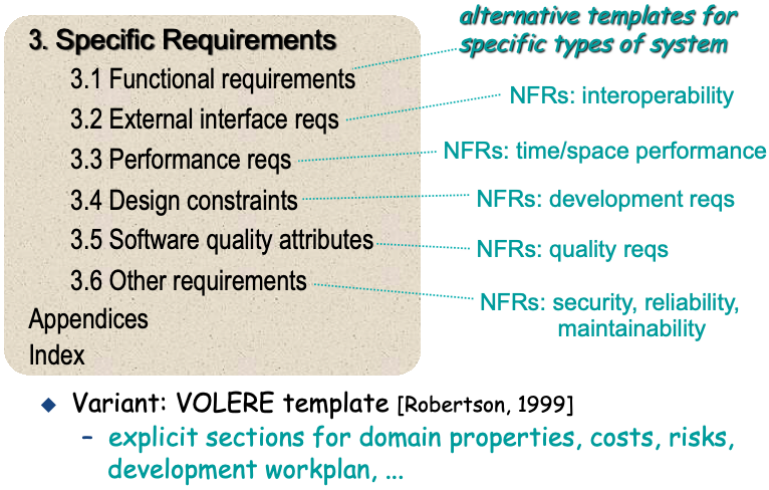
2.3.1 Organizzazione globale di un documento di specifica dei requisiti

Standard IEEE std-830

3 sezioni:



2.1: definisco il confine tra ciò che posso controllare e ciò su cui posso solo fare assunzioni o analisi.
3: requisiti indicati nel dettaglio secondo vari template, ad esempio quello visto prima. Non si può immaginare la descrizione dei requisiti esplicitata come una lista telefonica.



(doc standard su e-learning) Non ha senso organizzare i requisiti per utente. Nell'appendice vanno specificati eventuali altri materiali utili.

Regole di specifiche dei requisiti semi-formal - diagrammatic notations

Sono notazioni semi-formali, ovvero un diagramma ha una sintassi formale (archi-nodi) ma può essere ambiguo.

Utili per:

- complementare ciò che scriviamo in linguaggio naturale (es descrizione stati che un componente può attraversare)
- dedicati a specifici aspetti del sistema: aiutano a dare una descrizione migliore

- efficace nella comunicazione perchè è grafico
- hanno una sintassi formale, più precisamente semi-formale - porta con sè una certa ambiguità

2.3.2 Esempi diagrammi

NOTA: vedere esempi grafici di diagrammi sulle slide!

Problem diagrams: ci permette di dare una descrizione dei requisiti a livello si sistema. Sono mostrate le componenti del sistema (unica componente o più componenti). Il sistema comunica con diversi elementi dell'ambiente (rettangoli). Ci permette di specificare un requisito (ovale tratteggiato) nella sua forma testuale e riferirlo agli elementi dell'ambiente citati nel requisito. Stiamo dando un contesto al requisito che prima era solo in forma testuale.

Frame diagrams: questa rappresentazione dei requisiti può prendere la forma di un pattern. Risolvono il problema della ripetizione di uno stesso requisito. Abbiamo un tipo su eventi e componenti e abbiamo il nome del pattern.
È usato per indicare il requisito di un utente che dà un comando su uno strumento.

Possiamo distinguere tra eventi causali (causati da altri eventi), eventi veri e propri ed eventi lessicali (dati utilizzati).
Ci si trova con dei pattern che vengono istanziati in modi specifici prendendo un nome.

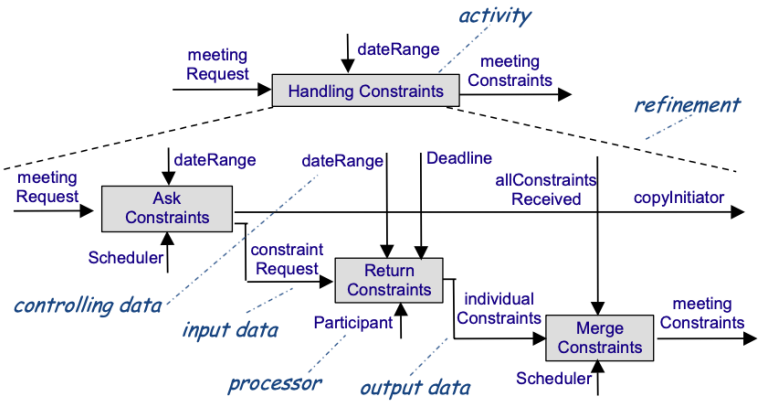
Entity-relationship diagrams: ci permettono di specificare quali sono le entità, come sono in relazione tra loro, quali attributi li caratterizzano, etc...

SADT diagrams: diagrammi che permettono di specificare il comportamento di alcune attività che vanno eseguite dal software e scomporre questo comportamento in sotto-attività e aggiungere una serie di informazioni per ciascuna attività.
Possono essere di due tipi e devono essere coerenti tra loro:

- **activity driven** = actigram: si concentra sulle dipendenze delle attività e le mostra in termini di dati
- **data driven** = datagram: si concentra sulle dipendenze tra i dati e le mostra in termini di azioni (logica di controllo)

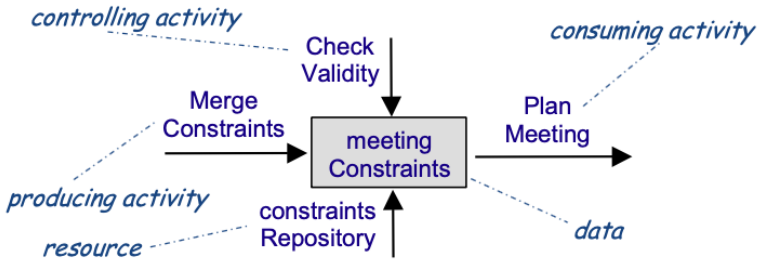
Le attività/dati sono indicate in rettangoli con una serie di frecce; a seconda della freccia cambia il significato.

Esempio: actigram



Da sinistra a destra sono input e le frecce a destra indicano gli output.
Frecce dall'alto: dati che controllano ed influenzano il comportamento dell'attività (ad es aspetti di configurazione ...); non sono input
Frecce dal basso: qual è l'unità che svolgerà quella determinata attività

Esempio: datagram



Al centro c'è il dato che dev'essere usato dal sistema e le frecce ne fanno capire il contesto. Possiamo eventualmente concatenare diversi box.

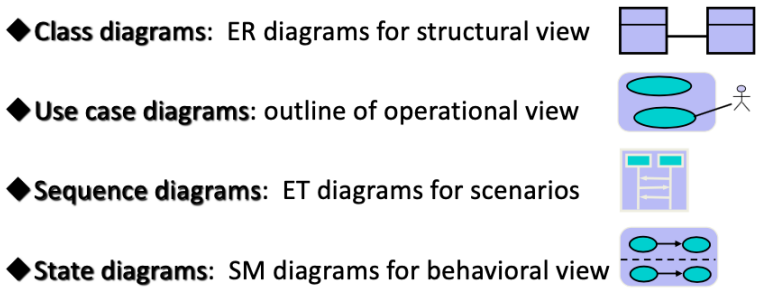
Use case diagrams: ci permette di mostrare quali requisiti sono stati individuati e quale attore partecipa a ciascun requisito.

Event trace diagram - diagrammi di sequenza: il tempo scorre dal basso verso il basso.

State machine diagrams: mostra in quali stati un particolare elemento si può trovare e mostrare quali eventi implicano un cambiamento di stato tramite le transizioni. **R-net diagram**: diagramma ad albero che indica come deve reagire il sistema a seguito di un semplice schema.

Attenzione a quando si producono diagrammi tra loro complementari ma che hanno tra loro intersezioni con quello che noi descriviamo nel testo. Dobbiamo stare attenti a non introdurre inconsistenze nel comportamento del nostro sistema.
Esistono delle regole di consistenza ma dobbiamo essere noi attenti a controllare ciò che esplicitiamo.

Multi-vite specification in UML



Riassumendo: i diagrammi sono ottimi per illustrare elementi importanti che caratterizzano requisiti, utili per la comunicazione con non esperti, etc... l'elemento debole, però, rimane quello della semantica ambigua che ci limita nelle analisi. Lo svolgimento di analisi a partire da questi diagrammi è limitata.
Il passo successivo è avere sintassi e semantica formale. Questo è usato in sistemi ad es mission-critical in quanto sono molto costosi e bisogna giustificare la loro adozione anche in termini di investimento economico.

2.3.3 Formal notations

Sia la sintassi che la semantica sono formalmente definite e questo permette al calcolatore di processarle interamente garantendo un certo livello di precisione senza ambiguità. Da qui, quindi, si possono fare analisi di coerenza tra i requisiti.

Un esempio di notazione formale è quello dato dalla logica formale.

Regole di interpretazione sintattica

$\langle atomicProposition \rangle ::= true | false | \langle propositionSymbol \rangle$
$\langle statement \rangle ::= \langle atomicProposition \rangle | (\neg \langle statement \rangle)$
$| (\langle statement \rangle \wedge \langle statement \rangle) | (\langle statement \rangle \vee \langle statement \rangle)$
$(\langle statement \rangle \Rightarrow \langle statement \rangle) | (\langle statement \rangle \Leftrightarrow \langle statement \rangle)$

Esempio: trainStopped AND Emergency =¿ doorsOpen

Regole di interpretazione semantica

$VAL_i(true) = T;\ VAL_i(false) = F;$
$VAL_i(atomProp) = val_i\ (atomProp)$
$VAL_i(\neg S) = T\ if\ VAL_i(S) = F;\ F\ otherwise$
$VAL_i(s_1 \wedge s_2) = T\ if\ VAL_i(s_1) = T\ and\ VAL_i(s_2) = T;\ F\ otherwise$
$VAL_i(s_1 \vee s_2) = T\ if\ VAL_i(s_1) = T\ or\ VAL_i(s_2) = T;\ F\ otherwise$
$VAL_i(s_1 \Rightarrow s_2) = T\ if\ VAL_i(s_1) = T\ and\ VAL_i(s_2) = T;\ F\ otherwise$
$VAL_i(s_1 \Leftrightarrow s_2) = T\ if\ VAL_i(s_1) = VAL_i(s_2);\ F\ otherwise$

Si può avere un approccio formale anche su specifiche basate su stato descrivendo in modo rigoroso quali sono gli stati che il sistema può attraversare, quali

operazioni possono modificare lo stato, quali regole deve soddisfare, quali operazioni lo rappresentano, etc...

Esempio: Z data schema

- Specify state space fragments by
  - declaring aggregations of coupled state variables
  - stating invariants on these

LibraryAgents
OrdinaryPatron: $\mathbb{P}$ Person
Staff: $\mathbb{P}$ Person

OrdinaryPatron ∩ Staff = ∅

- Declarations of form *stateVar : Type*
  - meaning "the values of *stateVar* are members of set *Type* "
- Variables may be
  - simple e.g. *bc:* BookCopy, *switch:* {off, on}
  - structured e.g. *date:* Day X Month value is a **tuple**
    Available: $\mathbb{P}$ BookCopy value is a **set of** bookcopies
    whichBook: BookCopy ↔ Book value is a **relation**

Le prime due righe dello schema indicano gli attributi di stato.
Sotto la linea orizzontale si indicano degli invarianti, ovvero la relazione tra i due attributi.

Gli stati da soli non bastano perchè ci serve definire come questi stati possono essere modificati per poi fare dei ragionamenti formali.

Esempio: Z operation schema

- Specify operations by
  - declaring input, output variables
  - importing variables from other data schemas
  - stating pre- and postconditions on those (implicitly)

CheckOut
Δ Library, Ξ *LibraryAgents*
p?: Person, bc?: BookCopy
---
bc? ∈ Available ∧ p? ∈ OrdinaryPatron ∪ Staff
OnLoan' = OnLoan ∪ {bc?} ∧ Available' = Available \ {bc?} ∧ ...

- Variables get decorated ...
  stateVar? : *Type* input variable
  stateVar' : *Type* output variable (yielding states)
  stateVar! : *Type* output variable (external, not yielding states)
  Δ *Schema* variables imported from *Schema* are changed
  Ξ *Schema* variables imported from *Schema* are not changed

Nella parte alta sono indicati gli elementi coinvolti nelle operazioni ed è esplicitato su cosa lavorano.
Simbolo triangolo indica che lo stato Library viene modificato; l'altro simbolo

indica che LibraryAgent non viene modificato.
? Indica che p è input e il tipo dell'input è indicato dopo i ":".

Infine c'è una definizione completa di qual è l'effetto di questa funzionalità: la prima riga è una precondizione, mentre l'ultima riga definisce come cambia l'informazione di stato (tramite la teoria degli insiemi).

2.4 Requirements quality assurance and evolution (garanzia ed evoluzione dei requisiti)

La maggior parte di questa parte ha appunti presi in lingua inglese.

How to validate requirements? - Come faccio a sapere se i requisiti che ho scoperto sono soddisfacenti?

- requirements inspections and reviews - analisi manuale per trovare degli errori, ovvero attività estremamente costosa e lenta (revisione dei requisiti)
- queries on a specification database, ovvero archiviazione dei requisiti in db con controlli automatizzati tramite query cercando ad es conflitti, ovvero attività parziale e semplice
- specification animation, ovvero requisiti specificati tramite diagrammi
- formal checking, ovvero analisi formale dei requisiti

Nota: 1: conflitto, 0: no overlap, 1000: overlap ma non conflitto

Requirements changes must be managed

Dobbiamo essere pronti ad anticipare il cambiamento (ad es conoscere legami e dipendenze tra requisiti, gestire conseguenze dei cambiamenti, etc...) per gestirlo nel modo più rapido possibile.

- the problem world keeps changing
- information management problem

Requirements (change) management = process of anticipating, evaluating, agreeing on, propagating changes in RD items.

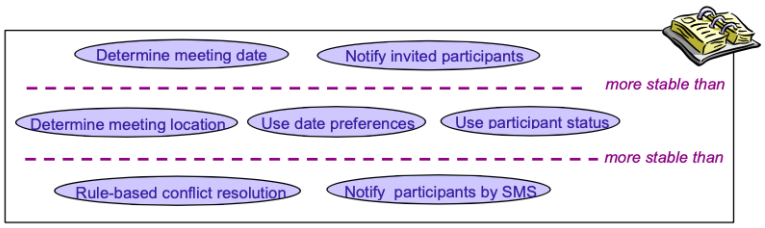
How to prepare for changes? Come ci si prepara al cambiamento?

Saper distinguere requisiti che più probabilmente cambieranno nel tempo e quali saranno più stabili e trattarli in modo diverso.
Bisogna distinguere diversi livelli di stabilità:

- identify likely changes, assess likelihood (valutare la probabilità), document them stability

- by associating levels of stability with groups of statements defining features

I requisiti possono essere distinti in diversi livelli di stabilità per differenziare i vari requisiti:



Analyzing likely changes: heuristic rules - distinguere i livelli di stabilità

- **highest stability level**: features to be found in any contraction, extension, or variant of system.
  - gli aspetti che sono più concettuali/intenzionali legati di più ad obiettivi di funzionamento tendono ad essere più stabili rispetto ad obiettivi operazionali.
  - gli aspetti funzionali sono più stabili rispetto agli aspetti non funzionali perchè cambiano più facilmente.
- choices among alternative options are less stable. I requisiti derivanti da analisi sono poco stabili perchè legati ad assunzioni che potrebbero rivelarsi sbagliati e analisi del momento che successivamente possono cambiare.

Le features possono essere correlate ad aspetti relativi all'ambiente oppure possono essere un insieme di requisiti funzionali.

Evolution support requires traceability management

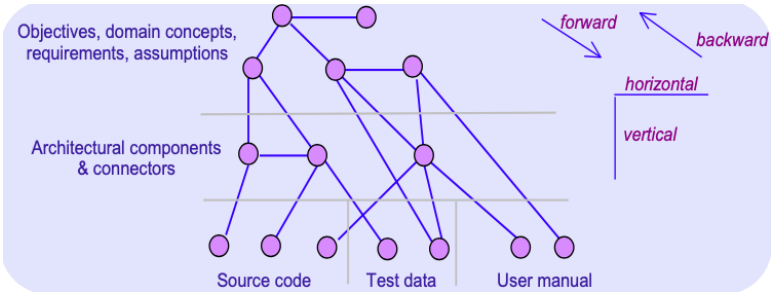
Un elemento è tacciabile se siamo in grado di dire da dove arriva, perchè esiste, come viene usato e per quale obiettivo. Ci interessa la tracciabilità perchè possiamo facilmente gestire i cambiamenti in quanto abbiamo tutte le informazioni necessarie sul cambiamento, e quindi gestirlo in modo efficace.

- an item traceable if we can fully figure out (risolvere)
- traceability management (TM), roughy
- objectives of RE-specific traceability

TM fa affidamento sulla tracciabilità tra gli elementi

- to be identified, recorded, retrieved
- bidirectional: for accessibility from

  - source to target: forward traceability (in avanti)
  - target to source: backward traceability (all'indietro)
- within same phase (horizontal) or among phases (vertical)



Traceability chains support multiple analyses

- backward traceability
- forward traceability

⇒ Localize and assess impact of changes along horizontal (tra artefatti dello stesso tipo)/vertical (artefatti da fasi diverse dello sviluppo) links.

Una volta ottenuti i link possiamo chiederci perchè quell'elemento esiste, da dove arriva, perchè è usato, quali sono le implicazioni, etc... (ricorsivamente).

Traceability management techniques: a wide palette - come facciamo a registrare questi link?

Questo è un problema aperto e complesso...
Alcune tecniche semplici e costose da gestire e mantenere nel tempo:

- cross referencing
- traceability matrices
- feature diagrams
- traceability databases
- traceability model databases
- specification-based traceability management

2.4.1 Cross referencing

Ogni elemento che può essere tracciato deve avere un identificatore e questo può essere utilizzato all'interno di altri elementi per creare un link (ad es "vedi sezione 2.2")

- select items to be traces, assign unique name
- define index/tagging scheme for linking these lexically
- configure standard search/browse engine to this scheme
- retrieve items by following cross-reference chains (=catene)

(Sono poco costosi all'inizio ma elevato costo per il loro mantenimento, gestiti informalmente, è complicato dare una semantica ai link, molto costoso mantenere uno schema del genere; applicabile ad un insieme ristretto di elementi)

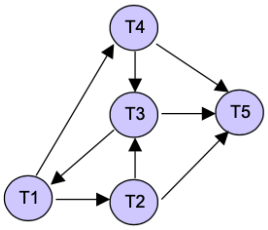
- pro:
  - lightweight (=leggera), readily available
  - any level of granularity
- contro:
  - single, semantics-free link type (lexical reference)
  - hidden traceability info, cost of maintaining indexing scheme
- $\Rightarrow$ limited control and analysis

2.4.2 Traceability matrices

Matrici in cui gli elementi da tracciare compaiono come righe e colonne.
1=c'è dipendenza, 0=non c'è dipendenza

Esempio: matrix representation of single-relation traceability graph. In questo grafo T3 dipende da T4.

Traceable item	T1	T2	T3	T4	T5
T1	0	1	0	1	0
T2	0	0	1	0	1
T3	1	0	0	0	1
T4	0	0	1	0	1
T5	0	0	0	0	0



(è molto costoso da gestire, diventa facile commettere errori; è attuabile su insiemi ristretti di elementi ed è ancora un problema aperto)

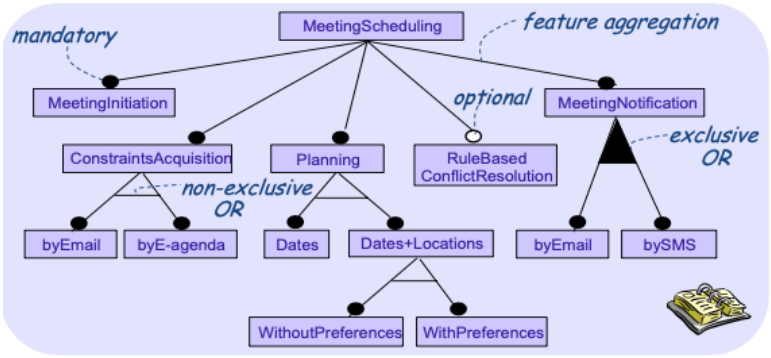
- pro:

  - forward, backward navigation
  - simple forms of analysis

- contro:
  - unmanageable, error-prone (=soggetto a errori) for large graph; single relation only

2.4.3 Feature diagrams

For variant link type: graphical representation of commonalities (=punti in comune) and variations in system family.



Permette di rappresentare il concetto di "variante di un sistema", ovvero realizziamo un sistema ma lo vogliamo distribuire con combinazioni diverse di features (ad es free per linux, Mac, JCompact representation of large number of variants windows, oppure un'altra variante etc...). Vogliamo con questo diagramma rappresentare tutte le varianti del nostro progetto tramite i rettangoli (=features del sistema).
Ogni feature può essere

- obbligatoria (pallino nero)
- opzionale (pallino bianco)

Ogni feature può essere composta da altre features e questa composizione può essere fatta in

- AND
- OR (triangolo bianco)
- XOR (or esclusivo)

Calcolando tutte le possibili combinazioni posso tirare fuori tutte le versioni del mio sistema includendo features obbligatorie/opzionali etc...

Pro: compact representation of large number of variants.

2.5 Requirements quality assurance (garanzia di qualità dei requisiti)

Quality assurance QA at RE time

- errors and flaws (difetti) in requirements documents
- **QA task 1**: find as many of these as possible in the RD (=requirements documents) by
  - validation
  - verification
  - checks
- **QA task 2**: report defects, analyze their causes, fix them

Approaches to requirements quality assurance

- requirements inspections and reviews
- queries on a specification database
- requirements validation by specification animation
- formal check

Requirements inspections and reviews

- simple idea:
  - ask selected people to inspect the RD individually
  - meet to agree on list of problems to be fixed
- various forms
  - walkthroughs (procedure dettagliate)
  - more structured
- fairly (abbastanza) common for source code
- empirical studies showed it is effective for requirements too

A structured inspection-based QA process

1. **inpection planning**: determine reviewers, timing, scope fo meetings, report format
2. **individual reviewing**:
   - free mode: no directive on where to find what
   - checklist-based: specific issues, defect types, RD parts

    - process-based: specific role for each reviewer, specific procedure, defect type, focus, analysis technique
    - ...

3. **defect evaluation at review meetings**: collect defects, agree on serious ones, analyze causes, recommend, report
4. **RD consolidation**: revise according to recommendations

Inspection guidelines

- inspection report should be
    - informative, accurate, constructive
    - standard structure with room for free comments, lightweight
- inspectors should be
    - independent from RD authors
    - representative of all stakeholders, different background
- inspection should come not too soon, not too late
- more focus required on critical parts

Inspection checklists

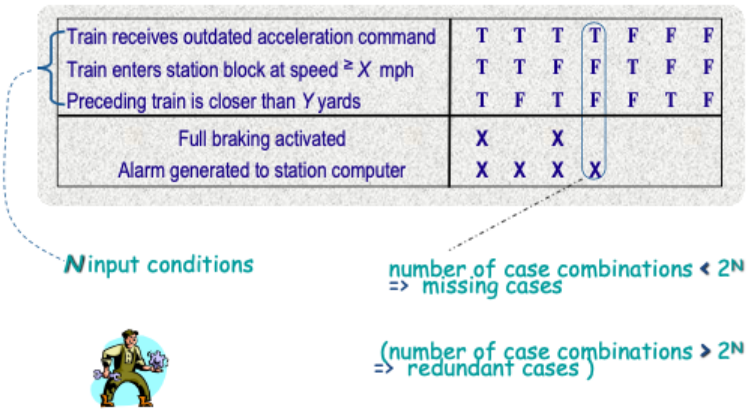
- aim (scopo): guide defect search towards specific issues
- defect-driven checklists
- quality-specific checklists
- domain-specific checklists
- language-based checklists

Esempi: defect-driven checklist (***vedi slide!***)

Esempi: quality-specific checklist (***vedi slide!***)

Esempi: language-specific checklist (***vedi slide!***)

2.5.1 Checking decision tables



Requirements inspections and reviews: strengths and limitations

- pro:
  - experiences to be even more effective than code inspection
  - wide (=larga) applicability
- contro
  - burden and cost of inspection process
  - no guarantee of not missing important defects

Queries on a specification database

- for diagrammatic specs
- specs maintained in requirements database
- queris capture checks for structural consistency intra- or inter-diagrams
- diagram-specific query engine can be generated from ER meta-spec of diagram language (***esempio diagramma su slide!***)

Requirements validation by specification animation

- aim: check requirements adequacy wrt actual needs
- approach 1: show concrete interaction scenarios in action
  - pro: enactment tools can be used on ET diagram
  - contro: scenario coverage problem
- approach 2: use spec animation tool
  1. extract or generate executable model from specs

2. simulate system behavior from this model
3. visualize the simulation while running
4. get user's feedback

- model-based
- many animators available for variety of spec languages

Visualizing the simulation

To submit environment events and watch model's reaction while simulation running:

- textual format
- diagram
- domain scenes

Requirements animation: strength and limitations

- pro:
  - best way to check adequacy wrt real needs, actual environment
  - supports stakeholder involvement
  - extendable to animate counterexamples generated by other tools
  - animation scenarios can be kept for later replay
- contro:
  - coverage?
  - no free lunch: formal spec needed

Powerful, automated, QA techniques require formal specs

- language checks
- language-specific consistency/completeness check
- property verification

Language checks

- syntax checking against grammar rules
- type checking
- static semantics checking
- circularity checking

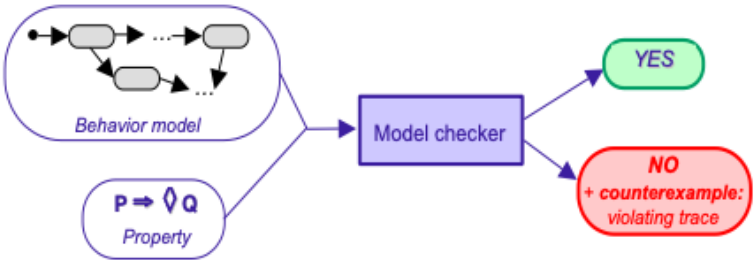
Dedicated consistency checks

When services or behaviors are specified as I/O relations

- is the relation a function?
- is this function total?

Algorithmic property verification: model checking

- to check whether a behavior model satisfies a requirement, domain property, or assumption
- by exhaustive search through model for property violation
- generated counterexample = trace leading (=principale) to violation



Esempio: using a model checker - ***vedi su slide!!***

Checkable properties

- reachability (raggiungibilità): "this situation can/cannot be reached"
- safety (sempre vero che): "the desired condition C always hold"
- liveness (condizionalità): "the desired condition C will eventually hold"

Model checking: strengths and limitations

- pro:
  - fully automated checks
  - esaustive search =¿ flaws cannot be missed
  - counterexample traces may reveal subtle (=dedicata) errors, hard to spot (=individuare) otherwise
  - natural coupling with animators for trace visualization
  - increasingly (=sempre più) used by industry in mission-critical projects
- contro:
  - combinatorial state explosion
  - counterexample may be complex, hard to understand

Deductive property verification: theorem proving

- generates new formal specs
- to show logical consequence of specs, point out inconsistencies
- generally interactive

Deductive verification in Z

Inference rules:

- from first-order predicate logic
- language-specific
$$\frac{prop[s_0],\ \{prop\}op\{propr\} \text{ for every } \triangle\text{−operation op}}{prop[s] \text{ for every state } s}$$
- proof example (***vedi slide!***)
- $\{prop\}return\{prop\}$ proved similarity
- hence the theorem by use of invariance rule

Theorem proving: strengths (punti di forza) and limitation

- pro:
  - soundness (=solidità) and completeness of formal system used
  - usable for showing inconsistent specs, inadequate consequences
  - can be applied to very large system
  - failing proofs may reveal problem causes
- contro:
  - difficult to use
  - no counterexamples produced

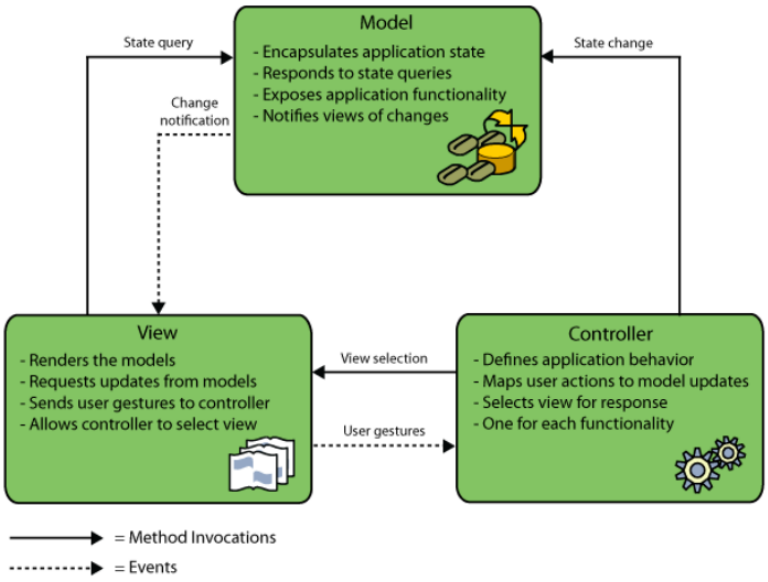
3 Software design and developement

3.1 Model View Controller - MVC

Soluzione architetturale che si compone di tre componenti che svolgono un ruolo ben definito:

- **Model**: classi che realizzano la logica applicativa dell'applicazione + rappresentazione di dati; in base all'esito dell'esecuzione delle operazioni che il controller mappa sul model seleziona la successiva view che farà il rendering dei dati successivi; può notificare alla view cambiamenti di stato

- **View**: ciò che viene mostrato e presentato all'utente (contenuto informativo e stato del model); raccoglie input dell'utente; fa il rendering dei dati e per farlo accede al model per averne le informazioni
- **Controller**: facilitatore/controllore per quella che è l'interazione tra il model e la view; controlla l'applicazione e riceve gli input ricevuti dall'utente e li mappa come operazioni nel model (cambiamento di stato)



Interazione MVC:

1. l'utente interagisce con l'app mandando degli input (interazione con la view) al controller
2. il controller riceve questi input e li mappa nel model come operazioni
3. il model in base all'esito delle operazioni sceglie la view che dovrà fare il rendering dei dati
4. la view scelta raccoglie le informazioni sull'input dell'utente dal model e fa il rendering dei dati mostrando l'interfaccia all'utente

3.1.1 Design patterns

Sono da ispirazione per dare un'articolazione al nostro progetto.
Ci soffermiamo sui design patterns che servono per articolare il comportamento del controller e della view.

Controller design:

- page controller
- front controller

- intercepting filter (dopo aver eseguito una fase di pre-processing un filtro, invoca un metodo del FilterChain per progredire nell'esecuzione dei filtri)
- application controller

View design:

- template view
- transformation view
- two-steps view:
  - la seconda fase dello schema di progettazione in due fasi rende una pagina logica
  - il primo passo del modello two-step view design crea una pagina logica con le informazioni che devono essere presentate nella pagina finale senza applicare un look and feel
  - il two-step view design pattern realizza la separazione logica tra la strutturazione del contenuto della pagina e il rendering della pagina

3.1.2 Page controller

Il controller orchestra l'esecuzione ricevendo un input proveniente dalla view prodotto dall'utente, la processa e la traduce in operazioni da effettuare sul model (stato dell'applicazione) che poi selezionerà un'altra view.

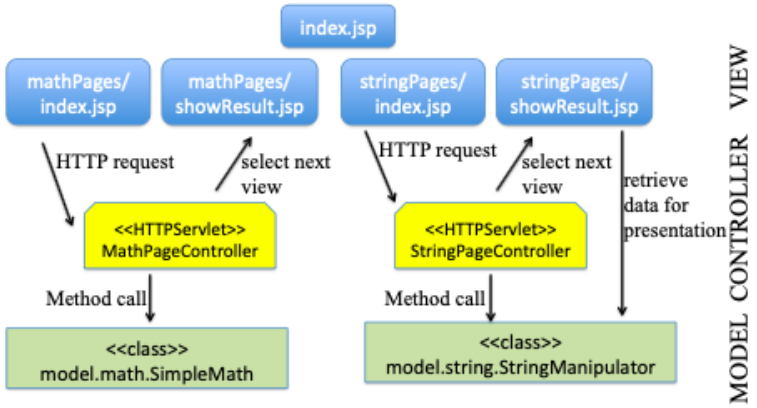
Ruoli del controller:

- riceve la richiesta
- controlla i parametri
- estrae i parametri dalla richiesta
- invoca la logica di business sulla base della richiesta
- selezionare la view successiva
- prepara i dati per la presentazione

Idea: implementare un controller per ciascuna pagina che effettua le operazioni sopra indicate.

Col crescere della dimensione dell'applicazione, delle funzionalità e della view, non c'è un componente che cresce in modo indefinito arrivando ad una applicazione ingestibile, ma aumenta il numero di controller di dimensione ridotte e più gestibili (alta scalabilità).

Esempio:



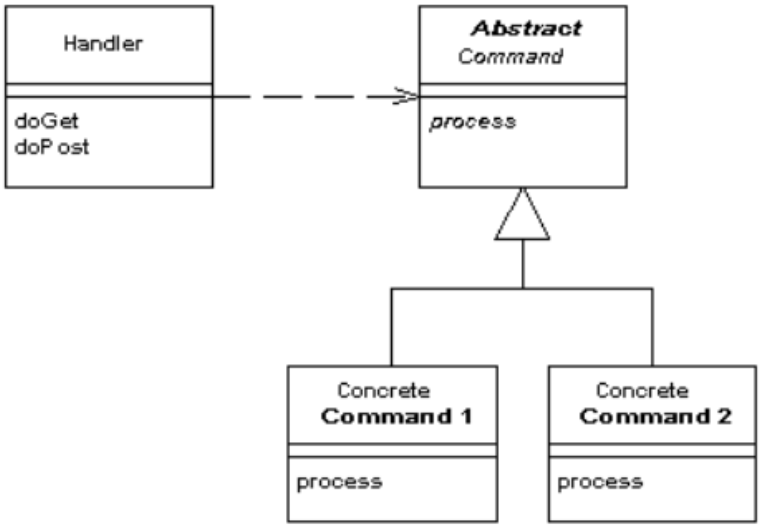
In questo esempio sono implementate:

- 2 classi java per i model
- 2 controller (HTTPServlet java)
- le view sono pagine .jsp che ci permettono di produrre le richieste che poi arrivano al controller

3.1.3 Front controller

È un componente singolo, ovvero normalmente un'app ha un unico front controller che riceve le richieste dalle componenti della view.

Ogni volta che riceve una richiesta si avvale della collaborazione con diverse classe (gerarchia di classi) che rappresentano i comandi richiesti dall'utente attraverso l'interfaccia GUI (es aggiunta elementi, ricerca, login, etc...). Ogni comando che può essere richiesto dalla GUI è implementato come una classe.



Il metodo *process* interagirà con la logica applicativa.
È una soluzione architetturale scalabile perchè al crescere della dimensione del-

l'app cresce la gerarchia di comandi con nuovi comandi man man aggiunti.

Handler:

1. riceve la richiesta
2. esegue le operazioni di tutti i comandi eseguiti
3. identifica il comando che deve essere eseguito
4. alloca l'istanza del comando
5. dedica l'esecuzione del comando

Command:

1. inizializza i parametri
2. invoca il metodo che esegue la logica del comando (process)
3. viene determinata la view successiva in base all'esito del comando
4. passa il controllo alla view

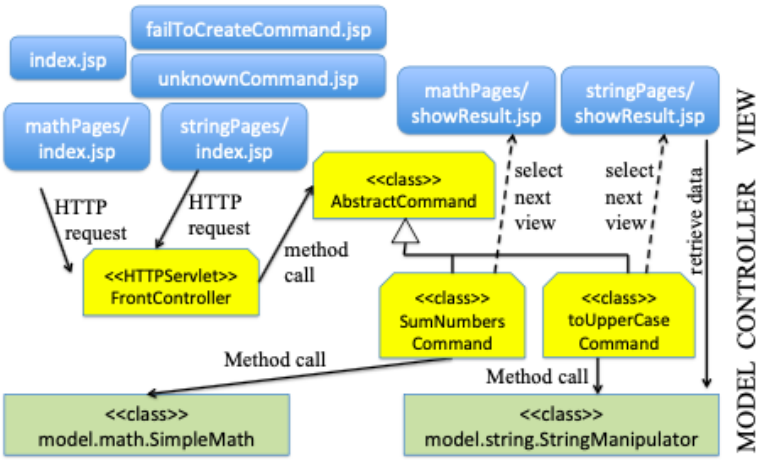
Esempio:

l'handler/controller:

1. riceve la richiesta
2. esegue le operazioni
3. identifica il comando che deve essere eseguito (e lo alloca)
4. esegue il comando chiamando la classe command

command:

1. invoca il metodo progress (esegue la logica del comando da eseguire)
2. sceglie la view successiva in base all'esito del comando
3. passa il controllo alla view



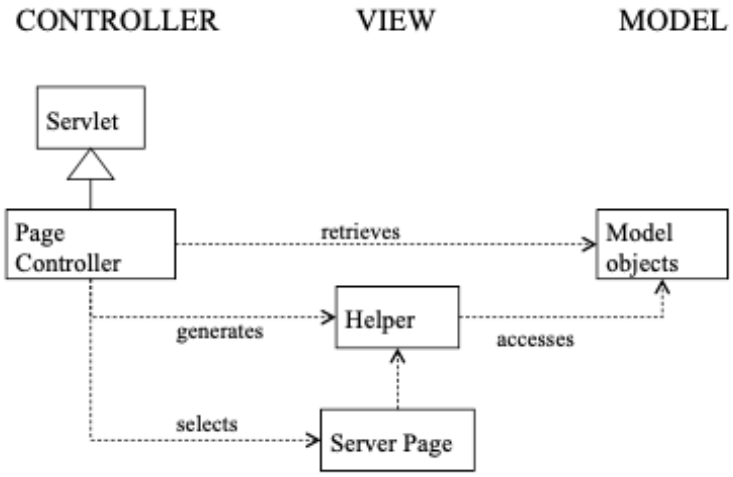
Il front controller è più complesso rispetto al page controller perchè:

- evita codice duplicato
- facilita la vita nell'implementazione perchè ha un entry point unico
- è facile da estendere perchè è singolo ed è facile arricchire la gerarchia dei comandi che possono essere aggiunti in modo abbastanza trasparente

3.1.4 Template view

Questo è il pattern più utilizzato per creare pagine web. La parte di rendering è ottenuta definendo parte di contenuto statico e parte di contenuto dinamico, il suo effettivo valore dipende dal modello e dal suo stato. Riceve i dati e li visualizza in pagine nel formato HTML, il quale è ottenuto da un template con markers.

Talvolta si utilizzano anche classi *Helper* create dai control per facilitare la costruzione della pagina: essa non fa diretto accesso al modello ma questa e i tag in essa contenuti fanno riferimento all'helper che a sua volta fa riferimento al model. L'helper ci permette di preparare dei dati ad oc per la parte di presentazione. L'helper recupera i dati dal model attraverso i metodi get, possono quindi fare elaborazioni intermedie dei dati che facilitano la parte di rendering.
L'helper facilita la costruzione della pagina, parla col model, recupera i dati direttamente dal model attraverso metodi get e facilita il rendering dei dati.
Esempio: template view + page controller



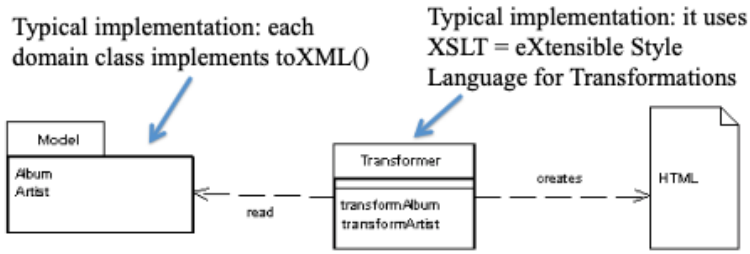
3.1.5 Transformation view

Non si ha una pagina template con parti costanti e dinamiche ma la pagina (HTML) è creata interamente in modo dinamico a partire dai dati che devono essere presentati.
Il transformer è responsabile di produrre in output la pagina sulla base dei soli dati.

Un modo di realizzare questa costruzione delle view è quello di arricchire il model per produrre una rappresentazione dei dati favorevole alla generazione delle pagine HTML da mostrare a schermo, ad esempio aggiungendo un metodo a tutte le classi dell'app che producono un'app XML dei dati (facilita un linguaggio di regole di trasformazione per la rappresentazione XML dei dati che può produrre codice HTML).
Il **transformer** crea la pagina in modo dinamico a partire dai dati e:

1. arricchisce il model
2. il model produce una rappresentazione dei dati favorevole alla creazione di pagine HTML da mostrare a schermo
3. aggiunge a tutte le classi un metodo che produce codice XML traducendo il codice in HTML

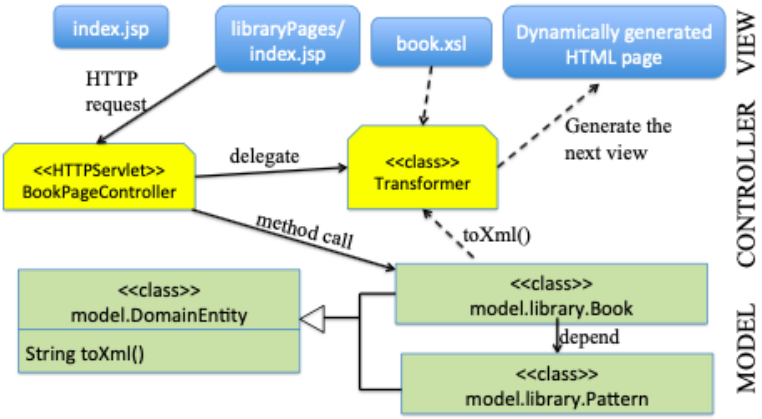
NOTA: ***vedi esempio codice su slide!***



Una sessione mantenuta da un'applicazione web:

- le sessioni utente possono essere tracciate utilizzando i cookie
- le sessioni utente possono essere tracciate utilizzando i campi nascosti nelle pagine web
- una sessione è una serie di richieste logicamente correlate
- i server applicativi di solito favoriscono un modo per memorizzare e recuperare informazioni dalle sessioni

Esempio: page controller + transform view
Il codice HTML è generato automaticamente dai transformer.



È difficile includere logiche applicative nel risultato, ma è facile generare dinamicamente pagine statiche. Per includere logiche applicative template view è più adatto, infatti è il pattern più usato per la realizzazione di app web.
È facile da testare (con trasformazioni XSLT) e da applicare (con rappresentazione XML del dati).

Transform è utile per costruire parti limitate di una pagina ed è difficilmente utilizzato per costruire pagine intere.

Con template view non vediamo ciò che stiamo definendo finchè non lo applichiamo ai dati, mentre con transform view posso sempre visualizzare la pagina che sto creando. (??)

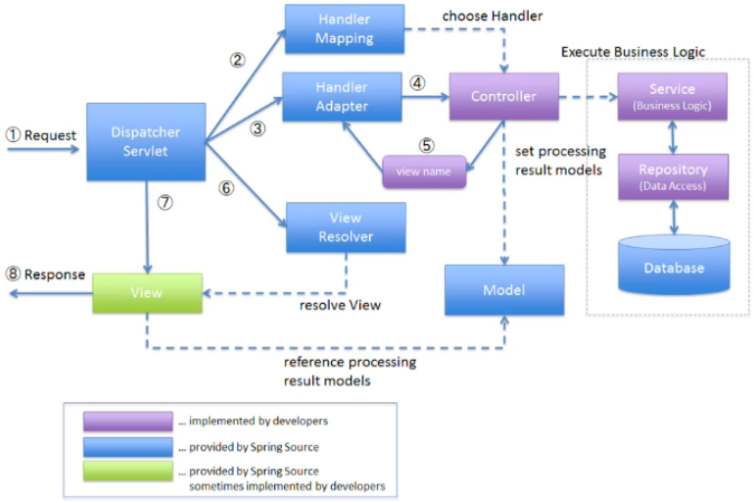
Spring MVC

Framework che ha al suo interno un'implementazione pronta ad essere utilizzata delle diverse componenti:

- MVC
- front controller
- intercepting filter
- context object
- view helper
- ...

Elaborazione di una richiesta con Spring MVC

Ci sono componenti già implementati e componenti che bisogna implementare.



Leggere in ordine le frecce.

Interpretiamo la figura sopra: abbiamo una richiesta (1) che viene intercettata da dispatcher servlet, viene inoltrata a handler mapping che va a selezionare un controller (quello giusto) per servire la richiesta che è stata ricevuta. L'esecuzione passa all'handler adapter che ha la responsabilità di invocare la business logic del controller precedentemente selezionato. (Il controller lo dobbiamo implementare noi!!) Il controller interagisce con tutto ciò che c'è nel model in base alla richiesta ricevuta e sceglie la view che dovrà essere poi visualizzata in base al risultato dell'elaborazione e questa scelta arriva all'adapter che poi la inoltra al view resolver che è in grado di trovare proprio quella view scelta dal controller. La view può accedere al modello Helper per visualizzare il risultato.

3.2 Object relational mapping - design patterns

Cercare di risolvere un problema che esiste tra object model e database relazionali: rappresentare il mismatch tra il modo di rappresentare gli oggetti e il modo in cui sono persistiti i dati in memoria. L'obiettivo è collegare/tradurre il linguaggio ad oggetti in linguaggio compatibile per database.
Il **gateway** nasconde all'utente l'accesso al database, traduce le richieste in operazioni per il database e fornisce una API per l'accesso al database.

Approccio

Il problema è far coesistere client e resource e lo si fa aggiungendo il componente Gateway che funge da traduttore:

$$\text{Client} \iff \text{GATEWAY} \iff \text{Resource}$$

Esistono diversi pattern che ci permettono di realizzare il Gateway.

Le implementazioni del DB gateway strategy sono:

- active record
- table data gateway
- data mapper

3.2.1 Active record

Implementa il processo di traduzione degli oggetti in memoria tramite record per il db. Si arricchiscono le classi con metodi che servono per l'interazione col db, preoccupandosi di gestire la logica di mapping con il db.

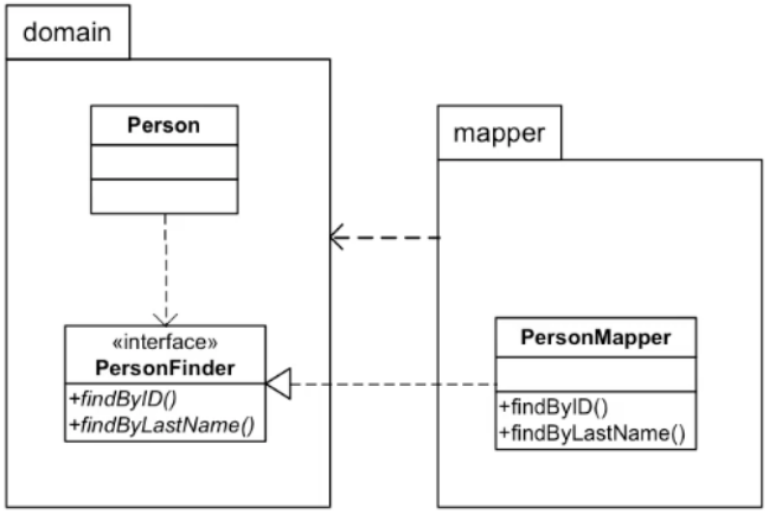
- finder methods: static methods (ha come risultato oggetti/collezioni per il dominio dell'applicazione)
- gateway methods: non-static methods
- load: permettono di creare un'istanza di un oggetto a partire dal db

- constructor: creano un oggetto in memoria che dovrà essere persistito
- finder: metodi statici che incapsulano query sql e ritornano collezioni di oggetti
- write: permettono di aggiornare, inserire, eliminare vari record nel db

Pattern semplice ma efficace.
Ha aspetti criticabili: implementazione che mescola la logica di business con metodi dedicati alla persistenza e tende a forzare la corrispondenza per cui la logica di mapping (logica con cui vengono create le tabelle nel db) si mappa in modo diretto con le classi dell'app.

3.2.2 Data mapper

L'idea è che i metodi per la gestione della persistenza (metodi write) vengono tirati fuori dalla classe.
Questa è un'idea per framework moderni, che mappa i nostri oggetti nel db.
Il metodo *update* è un metodo public che implementa operazioni di persistenza per una data classe di dominio.



Open issues

Problemi da risolvere sulla persistenza dei dati in ORM con anche altri pattern che possono essere presi in considerazione per risolverli.

- aspetti comportamentali
- aspetti strutturali

Vediamo alcuni pattern che risolvono questo tipo di problema.

Unit of work (pattern)

Quando le modifiche in memoria devono essere sincronizzate col db?
Bisogna aggiungere la definizione di system transaction.

Ogni scelta dell'utente è un'interazione col software con una semantica *all-or-nothing*.

Gestire esecuzioni con questo tipo di semantica come devo comportarmi con le transazioni?

- transazioni di business e system transaction coincidono: non rispetta la proprietà di isolamento delle transazioni
- si fa una system transaction per ogni operazione dell'utente: non rispetta la proprietà di atomicità delle transazioni
- l'utente esegue le operazioni, il software mantiene queste operazioni in memoria e solo quando viene chiusa la business transaction viene aperta la system transaction per completare e confermare le operazioni (approccio utilizzato: UNIT OF WORK)

Ha dei metodi che permettono di tenere traccia di tutto ciò che è stato registrato. Implementa un metodo `registerDeleted(...)` che viene utilizzato per tracciare gli oggetti del dominio che vengono cancellati all'interno di una business transaction, raccoglie le operazioni eseguite da una business transaction e le esegue all'interno di un'unica operazione di sistema, tiene traccia delle modifiche agli oggetti del dominio. Ogni volta che viene attivato un metodo la unit of work ne viene informata e si ricorderà che ci sono dei dati da caricare sul db.

Locking

Il concetto di **locking** è introdotto perchè diverse business transaction possono essere aperte nello stesso momento.

- **optimistic locking**: assumiamo che diversi utenti lavorano su dati differenti e quindi raramente due business transaction possono interferire. I record non vengono bloccati, ma a tutti i record viene aggiunto il numero di versione (si incrementa ad ogni modifica). In caso di modifiche concorrenti si deve fare il rollback della transazione e rifarla da capo
- **pessimistic locking**: non si permette alle transazioni di lavorare su dati su cui stanno lavorando altre transazioni. È più lento (si riducono le performance) ma più sicuro

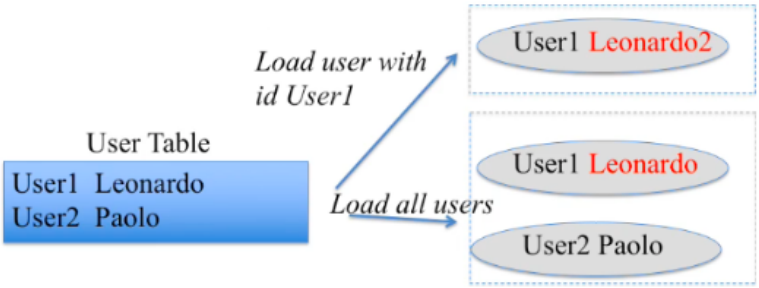
Notification policies

- **caller registration**: chi modifica il dato (il client) deve informare la unit of work della modifica
- **object registration**: oggetto modificato informa la unit of work che lui stesso è stato modificato (soluzione più adottata)

3.2.3 Identity map (pattern)

Cosa succede in memoria quando carichiamo oggetti dal db?

Probelma: lo stesso oggetto è presento in memoria due volte con la stessa entità. Se una di queste due dovesse essere modificata si avrebbe una situazione di inconsistenza. Questa è una situazione da evitare.



Dovremmo avere come risultato un'unica entità per User1 Leonardo.
Questo lo si risolve con identity map, che è una sorta di cache tra l'app e le applicazioni che implicano un'interazione con db. Prima di estrarre i dati dal db si controlla che i dati siano già presenti in memoria, e nel caso lo siano l'identity map ritorna il riferimento all'oggetto in memoria senza crearne delle copie; se l'oggetto non è presente in memoria si interagisce con db e si carica l'oggetto in memoria.

Lazy load

Problema di efficienza nel caricamento di oggetti dal db.
Per risolvere questo problema si usa lazy load, che ha come idea il non caricare tutti i valori per tutte le classi ma caricarli solamente se l'applicazione deve accedere ad essi (solo se servono).
Quando l'applicazione crea l'oggetto, quindi, non lo inizializza da subito. Quando sarà il momento di inizializzarlo, prima di farlo, si deve controllare che non sia stato ancora inizializzato.

3.2.4 Identity field (pattern)

È importante sincronizzare gli oggetti presenti in memoria con i record presenti nel db.
Si potrebbero utilizzare degli attributi della classe come identificatori, ma anche se potrebbe funzionare è sconsigliato di utilizzare gli attributi perchè il software può evolvere.
È bene quindi che si utilizzi un campo ad oc: identity Field, ovvero un id long nel db. Ma come si genera di volta in volta mantenendo la sincronizzazione?

- lo genera il db
- lo definisce l'applicazione come numero incrementale sfruttando le informazioni presenti nel db con:

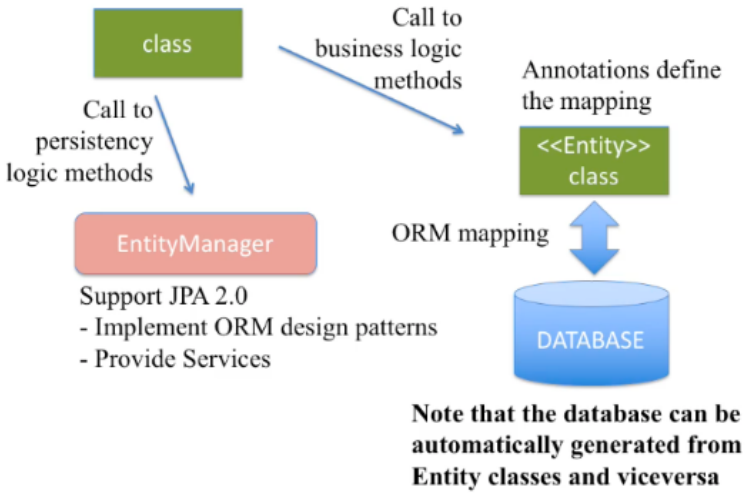
- table scan: si sanno tutti gli id assegnati e ne si assegna uno nuovo
- key table: si pesca l'ultimo valore e lo si incrementa di 1

3.3 Java JPA 2.0

Ci sono meccanismi già implementati che dobbiamo utilizzare dichiarativamente.

Jpa ha un sistema di annotazioni e servizi che usiamo per comandare operazioni che hanno a che fare con la persistenza.

3.3.1 Jpa application



3.3.2 Entity manager

Gestisce le classi che rappresentano dati che devono essere persistiti.

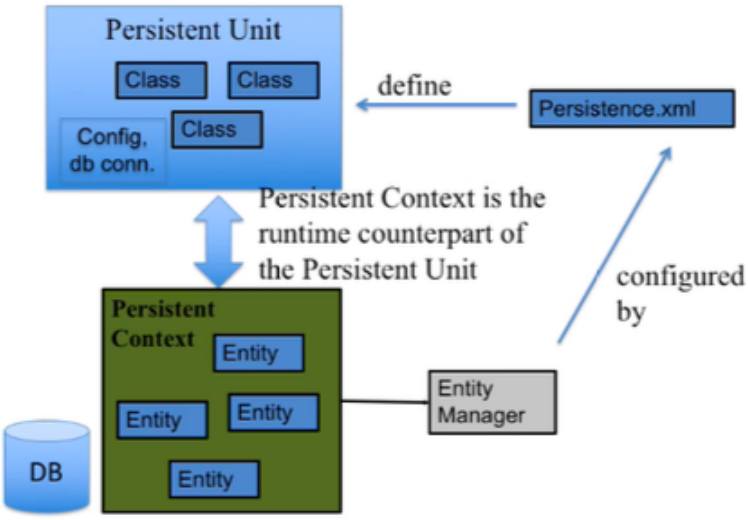
Un'entità può trovarsi in due stati:

- managed: l'entità è sotto la gestione dell'entity manager
- unmanaged: l'entità non è sotto la gestione dell'entity manager

L'entity manager è in grado di utilizzare i vari design pattern (visti nell'altra lezione ORM).
Dal momento in cui un oggetto viene creato vogliamo sempre che sia nello stato managed. Lo stato dell'oggetto può essere modificato in qualsiasi momento.

persistence.xml: ci permette di dire che le classi entity devono essere persistite su un certo db, e quindi dobbiamo considerare la conessione al db. Per fare questo si possono dividere le classi in persistent unit.

Le entità che fanno riferimento allo stesso persistent unit fanno parte dello stesso persistet context.



È necessario, inizialmente, inizializzare l'entity manager:

```
EntityManagerFactory factory;
EntityManager em;

factory = Persistence.createEntityManagerFactory("name of the
    persistent unit as it occurs in persistence.xml");

em = factory.createEntityManager();
```

Metodo persist

La scrittura dell'oggetto viene *schedulata* per la prima occazione unita (la unit of work si ricorda di questa applicazione e l'operazione viene caricata all'apertura-chiusura di una transazione).

Caricare oggetti

Si può fare con due metodi: `find()` e `getReference()`.
Il valore di ritorno è già un'oggetto.
Quando si usa l'entity manager non si useranno mai informazioni del db.

Ciò che scriviamo sono oggetti e ciò che carichiamo sono oggetti.
Quando scriviamo le query ci riferiamo ancora al modello object Oriented, non al db.
Il query language predica sui nomi delle classi e i nomi degli attributi e questo sarà poi tradotto nel db.

Modifiche alle entità

Tutte le modifiche sono tracciate e nel momento in cui una transazione è aperta-chiusa queste vengono scaricate nel db.

Si possono "trasformare" oggetti unmanaged in oggetti managed tramite il metodo merge(): l'oggetto che ritorna è una copia managed dell'oggetto unmanaged.
Se si hanno due oggetti (managed e unmanaged) con lo stesso identificativo e si invoca merge sull'oggetto unmanaged, questo sovrascrive l'oggetto managed nel db.

Eliminare entità

Si rimuove un'entità tramite il metodo `remove()`. Nell'immediato rimane in memoria e la sua distruzione dipende da cosa si fa nell'applicazione.

Esiste anche il metodo `refresh()` per forzare un'aggiornamento delle entità in memoria con i valori presenti nel db.

Primary key: utilizzano un valore long come soluzione più efficace e più semplice.

3.3.3 Relazione tra entità

Possono essere monodirezionali o bidirezionali:

- One-to-one
- Many-to-many
- One-to-many oppure many-to-one

3.3.4 Ereditarietà

È sfruttata dal framework e dal query Language perchè viene navigata la gerarchia di ereditarietà: se faccio una query a persona ritorno anche studente e professore.

Posso mapparla in 3 modi:

- **single table**: efficiente perchè ho un'unica tabella con uno spreco di risorse perchè avrò colonne null
- **one table per concrete class**: una tabella per ogni classe concreta che può essere istanziata.
  Strategia semplice ma meno efficiente della precedente soprattutto per query che mettono insieme oggetti di tipo diverso
- **one table per subclass**: le tabelle hanno un mapping esatto con le classi della nostra app. È particolarmente efficiente ma è abbastanza lento e rallenta il software perchè per ricostruire un oggetto devono essere eseguiti dei join

3.4 Enterprise application - component-based system

NOTA: ***capitolo molto tecnico da approfondire sulle slide per quanto riguarda il codice e i vari esempi grafici!!***

Sviluppo per componenti: sviluppo che permette di creare unità di componenti di cui si può fare il deployment in modo indipendente dagli altri componenti (dipendenze non statiche soddisfatte a run-time).

Handling dependencies

Decoupling: rendere indipendenti le componenti di cui si deve fare il deployment anche se a run- time le componenti devono interagire con le altre (ci sono comunque dipendenze funzionali).
Dipendenza statica: una classe ne invoca un'altra tramite il costruttore. Va modificata e mantenuta staticamente.

Dipendenza dinamica

- inversion of control: i campi che memorizzano riferimenti ad altri componenti vengono popolati automaticamente da un componente esterno
- service locator: basato sul fatto che la classe che ha bisogno di popolare un campo con un riferimento chiede direttamente ad un componente esterno il riferimento

3.4.1 Inversion of control o dependency injection

Idea: introdurre l'assembler che ha la responsabilità di individuare il componente che dev'essere utilizzato e inietta la dipendenza nella classe che ne ha bisogno.

Il modello di progettazione dell'inversione del controllo è nodo anche come **Dependency Injection**; le dipendenze sono solitamente iniettate quando viene creato l'oggetto, sono iniettate da un componente esterno (*assembler*) e possono essere iniettate attraverso un costruttore o attraverso metodi regolari.

L'injection avviene senza che MovieLister faccia nulla.
La dipendenza viene iniettata alla creazione dell'oggetto dall'assembler.
Come programmare l'assembler? Lo si trova già implementato nei framework.

3.4.2 Service locator o dependency lookup

Idea: avere un registro che conosce qual è l'implementazione che dev'essere utilizzata di volta in volta e la classe che deve soddisfare la dipendenza fa l'accesso al service locator (registro) chiedendo che venga ritornato un riferimento al componente che dev'essere utilizzato per la dipendenza.

Le dipendenze dei componenti sono soddisfatte cercando un riferimento adeguato ad un servizio del registro (service locator); il modello di design del service locator è noto anche come modello di design **Dependency Lockup**.

Il service locator è un singleton che memorizza delle coppie chiave-valore (riferimento al componente che dev'essere utilizzato - interface che il componente implementa). Per interrogare il service locator c'è bisogno di un riferimento al servizio del service locator all'interno delle nostre classi e in molti framework questo può essere settato attraverso delle annotazioni (=dependency injection). Le annotazioni permettono anche di creare un livello di astrazione tra componenti.

Component model EJB 3.0

- EJB 3.0 (Glassfish, jboss...)
- Spring

3.4.3 EJB Applications

- server-side components
- componenti distribuiti
- implementano business logic
- facili da integrare
- numerosi servizi di sicurezza, transazioni e persistenza utilizzabili in modo configurabili
- JPA persistency
- comunicazione sincrona/asincrona tra componenti tramite scambio di messaggi
- implementazione di servizi e web services

The three beans

Beans: componenti di EJB

- Entity bean: java JPA (implementa il dominio)
- session beans: comunicazione sincrona (sia stateless che statefull)
- message-driven bean: comunicazione asincrona

3.4.4 Session bean

Rappresenta un pezzo di logica invocato in modo sincrono.
Pezzo di classe che implementa un'applicazione che interagisce con altre Bean.

Si possono abilitare chiamate di vario genere tramite delle allocazioni:

- è una classe che implementa un'interfaccia Locale e/o un'Interfaccia Remota e/o un'Interfaccia Endpoint
- interfaccia locale:
  - contrassegnato con @javax.ejb.Local
  - specificare i metodi che possono essere invocati localmente
- interfaccia remota:
  - contrassegnato con @javax.ejb.Remote
  - specificare i metodi che possono essere invocati a distanza (con RMI)
- interfaccia endpoint:
  - contrassegnato con @javax.jws.WebService
  - specificare i metodi che possono essere invocati a distanza (utilizzando SOAP con JAX-RPC)
  - WSDL generato automaticamente
- session beans could be stateless or statefull
  - contrassegnato con @javax.ejb.Stateful se stateful
  - contrassegnato con @javax.ejb.Apolide se apolide

Esempio: il session bean stateless, ad esempio, potrebbe implementare una funzione che dice all'utente se un numero intero inserito è primo o no.

3.4.5 Message-driven bean

Classi java che implementano un'interfaccia definita, ovvero quella che viene eseguita quando si riceve un messaggio.

- è una classe che implementa un'interfaccia di messaggi attraverso la quale riceve messaggi asincroni
- i messaggi JMS utilizzano l'interfaccia javax.jms.MessageListener
- message-drive bean è contrassegnato con l'annotazione
  - @javax.ejb.MessageDriven

Il messaggio al suo interno ha tipicamente una struttura chiave-valore e il suo contenuto è recuperato attraverso metodi get.

Bean access

Le classi si devono immaginare incapsulate da un container, ovvero un elemento che si sovrappone tra il Bean e il resto del mondo ed intercetta tutte le richieste fatte a quel Bean.

Attraverso il layer che protegge il Bean (stubs) si possono scrivere dichiarativamente una serie di servizi di sicurezza, transazioni, etc... ovvero noi dobbiamo solo richiedere la logica, non implementarla.

Containers

- comportamento di default
- possono fornire ulteriori servizi sui quali i framework competono

Configurare il comportamento:

- comportamento di default per la gestione dei Bean (se non specificato niente)
- le annotazioni per la gestione dei Bean usate per configurare il comportamento del container
- descrittori/file di configurazione esterni che sovrascrivono (hanno priorità) le nostre annotazioni perchè degli specifici deployment potrebbero aver bisogno di configurazioni ad oc senza andare a modificare il codice

In alcuni casi può essere utile che un Bean possa fare un accesso al container e lo si può fare sfruttando dependency injection (*@Resource*).

I bean hanno un ciclo di vita, ovvero vengono creati quando utile e poi distrutti: questo fa sì che ciascun Bean può implementare metodi invocati dal container con call back quando particolari eventi (prima della creazione del bean o prima della sua distruzione ad es per libeare risorse) sono prodotti.

Dependency management

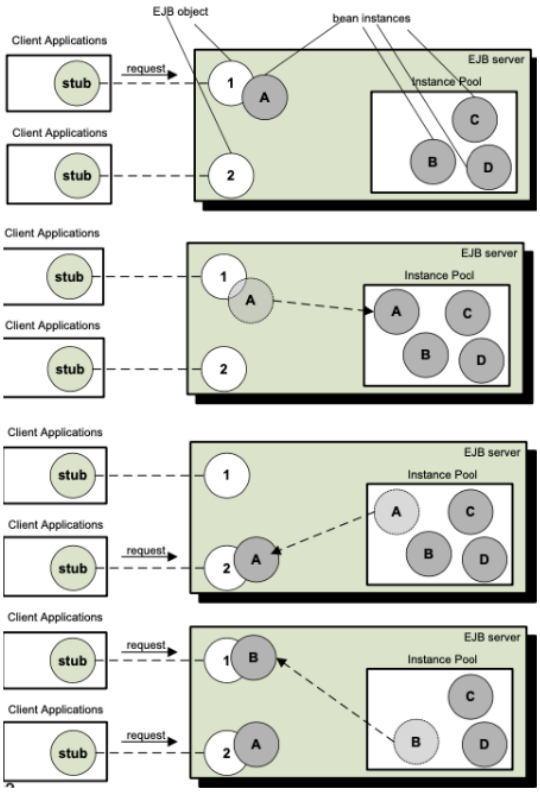
EJB supporta:

- setter/field dependency injection
- serviceLocator
  - JNDI service (Java Naming and Directory Service)
- resource management: gestione efficiente delle risorse
- services: servizi utilizzati all'interno dell'applicazione

3.4.6 Instance pooling

Idea: ridurre il numero di oggetti che devono essere utilizzati per servire richieste ma sfruttare il container per ottimizzare il riuso degli oggetti (per non creare oggetti per ogni richiesta prodotta dal client).
I componenti che lo sfruttano sono gli stateless session bean e i message driven bean.

Bean stateless: può essere riutilizzato per servire richieste differenti



Activation/passivation

Bean statefull: avendo uno stato non può essere usato per rispondere a richieste di client differenti perchè altrimenti si mescolerebbero le informazioni di stato.

Il framework sfrutta un meccanismo di attivazione/passivazione: essendo un ejb object non è necessario mantenere i bean in memoria, soprattutto se non vengono usati.
Idea: se un mean non è utilizzato per tanto tempo viene rimosso dalla memoria e salvato su disco (viene passivizzato) e se dovesse arrivare una richiesta il container che intercetta la richiesta recupera l'oggetto, lo riporta in memoria e lo usa per rispondere alla richiesta (ottimizzazione minore rispetto a stateless).

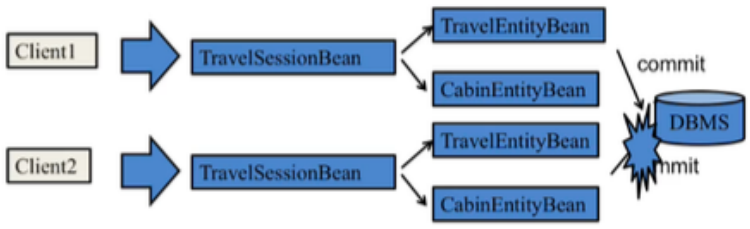
Services

- concurrency

- transaction manager
- persistenza (JPA)
- distribution
- naming (JNDI)
- secutiry
- asynchronous comunication (JMS)
- timer (simple generator of events)

Concurrency

La gestione della concorrenza sta nel fatto che non c'è concorrenza, ovvero è proibito creare nuovi thread all'interno dei bean. Non c'è concorrenza all'interno di un singolo bean. C'è concorrenza su client multipli: ogni client, quando invia una richiesta, accede all'istanza (la propria) dei vari bean. Ogni client caricherà la propria copia dei dati, ovvero ogni client ha la propria visione del mondo. Le richieste dei client sono elaborate in concorrenza.
Ogni utente lavora con una copia privata dei dati (session beans, messge-driven beans ed entity beans), quindi non ci sono problemi di concorrenza fino a quando i dati non sono persistiti.
I conflitti si risolvono con le strategie di locking (locking dei record, ottimistica (numeri di versione), pessimistica).



Distribution: ci sono numerosi protocolli che possono essere usati in modo trasparente per far comunicare tra loro i vari componenti tramite delle annotazioni.

Security

Come per JPA il servizio di java gode di vita propria:

- autenticazione
- autorizzazione

Declarative transactions

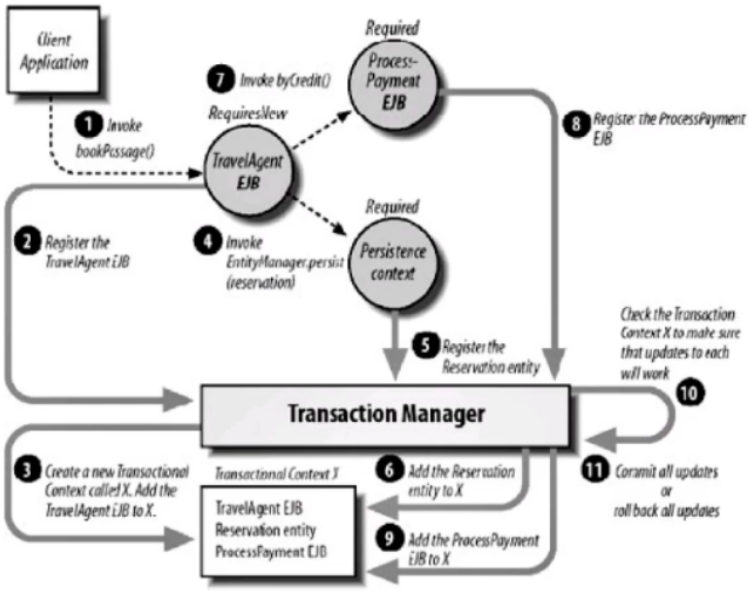
Ci permette di eseguire i metodi dei nostri bean/componenti.
Lo si fa in modo molto semplice aggiungendo delle annotazioni ai metodi.

Annotazioni:

- l'annotazione @TransactionAttribute(...) può essere usata per specificare la semantica transazionale di un metodo
  - diverse opzioni: NotSupported, Supports, Required (default), RequiresNew, Mandatory, Never
    - Required: se viene invcato da un client che non ha avviato alcuna transizione, una transizione è attivata automaticamente
    - Mandandtory: se viene invocato da un client che non ha avviato alcuna transazione, viene rilevata un'eccezione
- per ogni metodo si applica un'annotazione a livello di classe
  - l'annotazione del livello del metodo prevale sull'annotazione del livello della classe
- esempio

```
@Stateless
@TransactionAttribute(NOT_SUPPORTED)
pulic class TravelAgentBean implementa TravelAgentRemote {
  public void seetCustomer(Customer c) {...}
  @TransactionAttribute(REQUIRED)
  pulic TicketDO bookPassage(CreditCardDO card, double price
   ) {...}
}
```

Bisogna distinguere il contesto transazionale del chiamante, ovvero se il chiamante ha aperto la transazione e il tipo di annotazione che il chiamante ha.



3.4.7 Isolation and database locking

Isolamento:

- dirty reads
- unrepetable reads
- ghost reads

Quando si creano transazioni queste non necessariamente sono perfettamente isolate, ovvero che non possono interferire in nessun caso le une con le altre.
L'aspetto dell'isolamento può essere configurato in un db per impedire l'interferenza.

Dirty reads

Il modo in cui le transazioni interferiscono è dato da una transazione che può leggere dei dati che sono stati modificati da una transazione che non è ancora conclusa.

Unrepetable reads

Una transazione 1 (non ancora chiusa) legge dei valori committati (a seguito di operazioni di modifica eseguite dalla transazione 2) da una transazione 2 iniziata dopo l'inizio della transazione 1.

Ghost reads

Una transazione può leggere nuovi record generati/aggiunti da un'altra transazione che si chiude prima.

Isolation levels

Si possono scegliere diversi divelli di isolation:

- read uncommitted
  - letture non ripetibili, letture sporche e letture fantasma
- read committed
  - le letture non ripetibili e le letture fantasma sono possibili
- repeatable read
  - letture fantasma
- serializable
  - isolamento perfetto

Nota:
Alto isolamento ⇒ basse performance (ma tanta sicurezza)
Basso isolamento ⇒ alte prestazioni