PROCESSO E SVILUPPO DEL SOFTWARE

RIPASSO METODI AGILI

I metodi agili rispondono alla esigenza di sviluppare dei software i quali requisiti sono in continuo cambiamento. I classici metodi a cascata che supponevano una prima fase di definizione di requisiti per poi successivamente passare alla fase di programmazione, risentivano troppo di questi cambiamenti dei requisiti, perciò, vennero introdotti i metodi agili. Questi metodi non presuppongono più che i requisiti vengano definiti una sola volta all'inizio e poi basta, ma che questi cambino durante lo sviluppo del software. I metodi agili tendono ad avere una organizzazione ciclica che utilizza delle iterazioni che vanno a sviluppare parte dei requisiti. Lo sviluppo, quindi, prevede che ad ogni iterazione il gruppo di requisiti di priorità maggiore venga sviluppato per primo e, tramite delle iterazioni che sviluppano piccole parti di questo blocco dei requisiti, si completa lo sviluppo di questo gruppo producendo così un working system. La produzione su piccole parti di sistema consente di reagire senza problemi ad eventuali modifiche dei requisiti, causate dal feedback ottenuto dal risultato delle singole iterazioni, senza perdere troppo lavoro.

I processi agili hanno delle caratteristiche comuni sull'implementazione dei processi:

1- Enfasi sulla capacità del team e sulla selezione dello stesso
2- Si evita la produzione di documentazione inutile che diventa obsoleta velocemente
3- Diretta comunicazione tra i membri del team con riunioni varie
4- Nulla è definitivo, non si deve inseguire la perfezione in quanto col passare del tempo il design evolve e si migliora con il cambiare dei requisiti
5- Utilizzo di processi iterativi per il controllo della qualità come delle fasi di testing continue applicate ad ogni iterazione

Il metodo agile più diffuso è il metodo SCRUM che prevede che una iterazione denominata Sprint duri circa 4 settimane, i requisiti sono raccolti all'interno di un product backlog e prioritizzati sulla base delle richieste del committente, infine i backlog vengono sviluppati durante la fase di sprint. Una fase molto importante dello SCRUM process è il Daily Scrum Meeting che prevede delle riunioni giornaliere per mantenere alto il livello di comunicazione e collaborazione; questo meeting prevede che ogni team member dica a turno quello che ha fatto, quello che farà quel giorno e quali sono le criticità incontrate. Nel processo SCRUM ci sono tre ruoli principali:

1- Il team che lavora allo sviluppo delle funzionalità
2- Il product owner che è il cliente che partecipa alla definizione dei requisiti e alla revisione dei risultati ottenuti dai processi di sprint
3- Lo Scrum Master che si occupa di controllare che tutte le pratiche del processo di scrum siano svolte correttamente

Infine, nello SCRUM vi sono altri due meeting: uno sprint planning meeting eseguito prima della fase di spring dove si pianifica il lavoro da fare e uno sprint review meeting dove si fanno i conti con quello che si è prodotto e sulle criticità incontrate.

INTRODUZIONE ALLE DEVOPS

Tra i vari modelli di DevOps il più classico è il modello a cascata che si basa sulla assunzione che i requisiti siano stabili e chiari durante lo sviluppo del prodotto. I requisiti però cambiano durante l'attività di sviluppo e quindi vennero introdotti i metodi agili che prevedono che i requisiti cambino durante la fase di sviluppo e che questi cambiamenti siano gestibili a livello operativo. I DevOps introducono un nuovo livelli di adattabilità, ovvero rendono agile anche la fase di software operation cioè quella parte di rilascio delle release una volta completata una iterazione nello sviluppo.

Il DevOps, dunque, risulta utile in quanto risolve il problema di disconnessione tra il team di sviluppo e quello di rilascio dell'applicativo, risulta inoltre utile a risolvere problemi legati alla grande porzione di tempo spesa durante le fasi di testing, deploying e designing.

Il DevOps è un metodo di lavoro che promuove una stretta collaborazione tra il development team e il operations team tale da andare a definire un unico team che si occupa di entrambe le fasi. DevOps al suo interno possiede numerosi processi che implementano numerose automazioni durante tutto il ciclo di vita di sviluppo tra cui:

- Continuo sviluppo
- Continua integrazione
- Continuo testing
- Continuo rilascio del prodotto
- Continuo monitoraggio

I ruoli principali dei membri di un team che lavora secondo DevOps sono:

- Automation expert
- Security engineer
- Quality assurance che verifica la qualità del prodotto rispetto ai requisiti
- Code release manager che si occupa sulla parte di deployment di una release
- DevOps evangelist che sostituisce lo SCRUM Master e che quindi verifica che tutte le pratiche di DevOps siano eseguite in maniera corretta

I principali principi sui cui si basa DevOps sono:

1- Svolgere azioni Customer-Centric: avendo dei rilasci continui è necessario ponderare le proprie azioni pensando all'impatto che esse avranno sui customer
2- Responsabilità End-to-End: il team gestisce interamente il prodotto, sia nella sua fase di progettazione e sviluppo, sia nella attività di mantenimento del prodotto
3- Continui miglioramenti del prodotto e dei processi e di minimizzare le perdite
4- Automatizzare il maggior numero di processi possibili
5- Lavorare come un unico team
6- Monitorare e testare continuamente i propri prodotti per identificare al più presto eventuali problemi delle release e conseguentemente porre rimedio alla svelta

BUILD, TEST, RELEASE

Le fasi di build, test e release sono fondamentali all'interno dei metodi DevOps e tendenzialmente vengono automatizzati. Il sistema di version control più utilizzato è Git che prevede che ogni

utente abbia una propria working copy sulla quale lavora e una propria copia di repository ed esegue dei commit ad una repository condivisa tra tutti gli utenti.

Lo sviluppo su Git avviene su Multi-Branch, ovvero si progetta separatamente su più rami e successivamente li si uniscono. Ciascuna feature quando viene sviluppata la si sviluppa su un ramo dedicato che cessa di esistere una volta completato il suo sviluppo e trasferito il suo risultato all'interno del ramo di develop.

Lo sviluppo su molteplici branch consente anche a delle automatizzazioni delle operazioni di verifica del codice (test) tramite delle pipeline che vengono attivate quando una componente viene aggiornata (component phase), quando si assembla un intero sistema (system phase) o un sottosistema (subsystem phase) ed infine quando si esegue la produzione di un software (production phase). I controlli tramite pipeline spesso vengono utilizzati come quality gates, ovvero ogni qualvolta si vuole eseguire una operazione di push il codice deve soddisfare tutti i requisiti preposti dalle pipeline di controllo altrimenti non viene accettato all'interno del branch.

Nella component phase si pone attenzione sulla più piccola unità testabile che viene aggiornata; i tipi di test che si possono fare sono le revisioni del codice da parte di altri membri, i test di unità, analisi statica del codice.

Nella subsystem phase abbiamo un cambiamento sulle più piccole unità che possono essere rilasciate ed eseguite e i suoi principali controlli sono: test funzionali che testano l'unità rispetto ai requisiti, i test delle performance e check sulla sicurezza. In molti casi alcune componenti di questo tipo per essere eseguite devono interagire con altri porzioni di software; per evitare ciò si fa uso di mocks e stubs che creano delle alternative a questi servizi dai quali dipende la porzione di software in analisi.

Nella system phase vengono eseguiti dei controlli sull'intero sistema e i tipi di test eseguibili sono: test di integrazione, test di performance, stress test, load test e test di sicurezza.

Infine nella production phase che prevede il deployment i test che possono essere eseguiti sono dei test con obiettivo principale di fare dei rapidi check per verificare che i file compilati siano stati creati correttamente. I principali test sono il deployment incrementale e il zero downtime.

SCHEMI DI AGGIORNAMENTO AUTOMATICO

Ma come si può gestire la costante evoluzione automatica del software? Bisogna prestare estrema attenzione quando si migra da una versione ad una nuova di un software in quanto questo processo comporta dei grandi rischi di instabilità. Vengono quindi introdotti degli schemi di aggiornamento sofisticati legati ad un concetto di evoluzione incrementale. Questi schemi sono:

- Schemi incrementali degli utenti: incrementare lentamente il numero di utenti esposti alla nuova versione del software
- Incrementali delle richieste: incrementare gradualmente la percentuale di utenti esposti alla nuova versione della feature quando ne fanno richiesta d'uso
- Incrementali sulle componenti del software che vengono aggiornate
- Non incrementali con dei backup

Un famoso schema incrementale per gli utenti è il dark launching, introdotto da Facebook, questo schema prevede che il prodotto aggiornato venga esposto solo ad una porzione degli utenti per

verificarne il funzionamento ed individuare eventuali criticità prima di esporlo a tutta l'utenza evitando così figure di merda globali. Una volta corrette le criticità della nuova feature introdotta si incrementa la porzione di utenti esposta all'aggiornamento e così via finchè tutti gli utenti avranno visione della nuova feature. Questo tipo di schema è strettamente legato alle Canary Releases che ha lo stesso funzionamento del Dark Launching; l'unica differenza è che le Canary generalmente vengono utilizzate per degli update a livello backend, mentre le Dark vengono utilizzate per delle feature frontend.

Lo schema User Experimentation si pone come obiettivo quello di verificare se la nuova versione del software sia migliore di quella precedente. Per fare ciò si espone ad un sottoinsieme degli utenti selezionato accuratamente alla nuova versione. Tramite l'utilizzo di funzioni di monitoring delle azioni degli utenti si traggono informazioni statistiche riguardanti le due feature e dunque definire quale delle due è la migliore.

Per gli schemi incrementali delle richieste uno schema tipicamente usato è quello degli aggiornamenti graduali (gradual upgrade). Questo schema dunque, tramite l'utilizzo di un load balancer, consente a due versioni del software di coesistere e inizialmente gli utenti esposti alla feature aggiornate saranno una percentuale molto bassa, ma successivamente questa verrà incrementata gradualmente fino a passare tutte le nuove richieste alla feature aggiornata nel caso in cui non ci fossero problemi. Se ci sono problemi con la nuova versione, il load balancer può immediatamente ridirigere tutte le nuove richieste sulla versione precedente del software ritenuta stabile consentendo così una manutenzione del nuovo servizio senza perdere l'operatività dell'intero sistema.

Un ulteriore schema incrementale delle componenti è il rolling upgrade. Questo schema a differenza del gradual update che si occupa di un aggiornamento di una sola feature, si occupa della introduzione di diverse componenti che mettono a disposizione diverse feature. Le nuove compoenti vengono quindi introdotte gradualmente una dopo l'altra e per ogni nuova feature introdotta si eseguono operazioni di monitoring per verificarne l'efficacia.

Un altro schema di upgrade è il cosiddetto blue/green deployment. Questo schema è uno schema non incrementale e di backup. Questo schema prevede lo switch da una versione del software ad un'altra sia immediato. Per fare ciò si rilascia e si testa la nuova versione all'interno di una replica isolata di hardware e software del sistema e, una volta pronti per la fase di commit, si esegue lo switch alla nuova versione riconfigurando un router di indirizzamento e trasformando la versione precedente del sistema in una copia di quello nuovo da utilizzare nei casi di disaster recovery.

Lo schema appena trattato può essere generalizzato all'interno dello schema chiamato rainbow deployment. Questo schema prevede sempre l'utilizzo di uno switch, ma il momento di coesistenza delle due versioni del software è prolungato a tal punto da poter avere multiple versioni di sistema coesistenti allo stesso momento.

Tutti gli schemi appena presentati necessitano di una implementazione certosina, in particolare per quanto riguarda i possibili problemi di compatibilità causati dalla differenza delle versioni nuove e vecchie.

DEPLOYABLE UNITS

Le componenti che effettivamente vengono aggiornate, tipicamente sfruttano il mondo cloud che consente la virtualizzazione di queste nuove componenti. Nel contesto di virtualizzazione si creano appunto delle versioni virtuali di risorse che possono essere utilizzate come se fossero vere. I tipi di deployment che possiamo fare sono:

- Cloud (virtual machines): vi è una infrastruttura gestita dal cloud provider (oltre a gestire ovviamente le componenti hardware fisiche e i sistemi operativi) chiamata Hypervisor che consente di eseguire diverse macchine virtuali. Ogni macchina virtuale mette a disposizione un sistema operativo che lavora con un hardware virtualizzato. L'unità di deployment diventa dunque una di queste macchine virtuali ed andare ad eseguire un deployment di un aggiornamento consiste nel andare ad aggiornare il sistema all'interno di una di queste VM.
- Cloud (containers): creare e modificare delle VM può essere molto oneroso, quindi vengono introdotti i Container Engine al posto dell'Hypervisor. La differenza sostanziale è che all'interno di un container non vi è un intero sistema operativo, ma bensì solo quelle librerie necessarie all'applicazione presente all'interno del container di funzionare.
- Bare Metal: questo metodo prevede che i cloud provider ci forniscano accesso diretto alle componenti hardware. Non vi è più quindi la virtualizzazione ma l'accesso diretto ad una macchina con delle risorse stabilite su cui fare il deployment delle applicazioni. Il vantaggio di questa soluzione è di ottenere un notevole incremento della velocità ed efficienza dell'ambiente.
- Dedicated server: soluzione vintage che si occupa del trasferimento delle nuove funzioni all'interno di un server dedicato che ha le stesse funzioni di un bare metal ma con la differenza che questo deve essere gestito da noi e non dal cloud provider.

Un altro metodo di deployment è il Non-service software deployment. Questo tipo di deployment si applica spesso per le applicazioni mobile dove sono gli utenti a decidere se aggiornare o meno il proprio applicativo.

Per quanto riguarda il Monitoring delle nuove funzionalità introdotte dagli aggiornamenti, una tipica piattaforma per il controllo è la ELK (ElasticSearch, Logstash, Kibana). Il suo funzionamento prevede la presenza di sonde che raccolgono dati, questi dati passano per un BUS chiamato Logstash e arrivano in un DB di dati temporali chiamato ElasticSearch e infine possono essere visualizzati con una Dashboard chiamata Kibana.

RISK MANAGMENT E RISK ANALYSIS

Durante il ciclo di vita di un software development possono avvenire dei fallimenti nel progetto come la mancanza di uno sviluppatore specializzato in una funzionalità che comporta il rallentamento nello sviluppo e dunque ad una mancata consegna del prodotto finito nei tempi prestabiliti.

Il Risk Management è quella disciplina che ha come obiettivo quello di identificare ed eliminare i rischi prima che questi possano diventare una minaccia per il progetto. Data questa definizione risulta necessario definire il termine rischio; un rischio è la possibilità che ci sia un danno o una perdita. Per calcolare la grandezza di questi rischi è possibile utilizzare una formula definita Risk Exposure: RE= P(UO)* L(UO) ovvero la probabilità che si verifichi un danno * l'entità del danno

stesso. Una volta identificato un rischio e la sua grandezza è necessario individuare i Risk trigger, ovvero le cause che portano al verificarsi di un rischio.

Esistono due classificazioni principali per i rischi:

- Rischi relativi al processo: problemi nel processo di sviluppo software quali costi, risorse, tempo, personale, ecc…
- Rischi relativi al prodotto: problemi del prodotto software quali sicurezza, affidabilità, performance, ecc…

Ma come si possono gestire questi rischi? Il Risk Management per far fronte ai rischi appena descritti generalmente si struttura in due step: la fase di Risk Assessment e la fase di Risk Control. Nella fase di Risk Assessment si cerca di individuare i rischi più rilevanti e potenzialmente dannosi lavorando in 3 fasi:

1- Risk identification: fase in cui si identificano i rischi
2- Risk Analysis: fase in cui si analizzano i rischi
3- Risk Prioritization: fase in cui si ordinano i rischi analizzati per pericolosità

Nella fase di Risk Control invece si sviluppa un piano per gestire i rischi ricevuti dalla fase precedente. Si sviluppano dei piani che sono i management e i contingency plans per gestire i rischi, i quali saranno costantemente monitorati e, se si dovessero verificare, gestiti secondo le procedure stabilite dai piani.

Vediamo ora nel dettaglio le varie fasi del Risk Management. Iniziamo con la Risk Identification; individuare i rischi veramente rilevanti è estremamente complesso ma esistono delle strategie per individuarli. Una strategia è quella di utilizzare delle checklist che includono una serie di rischi comuni a tutti i progetti e un analista scorre questa lista individuando quali di questi rischi potrebbe presentarsi nel suo caso particolare. Un altro metodo molto utilizzato per individuare i rischi ricorre all'utilizzo di riunioni quali brainstorming atti a mettere in luce potenziali rischi.

La fase di Risk Analysis si occupa di stime eseguite sui rischi, quali i danni potenziali. I principali metodi usati per l'analisi dei rischi consiste nello sfruttare i modelli di stima dei costi, delle simulazioni e prototipi o ancora sfruttare delle checklist. Questa fase può portare a dover fare delle decisioni riguardanti le modalità di sviluppo del prodotto atte a minimizzare i rischi; per fare questo tipo di decisioni ci si può aiutare con gli alberi di decisione. Grazie agli alberi è possibile avere una rappresentazione di tutte le possibilità con le relative probabilità e potenziali costi in danno per poter scegliere la meno peggio delle ipotesi.

Andando a ragionare su quali possano essere le cause dei rischi. Per poter individuare tutte le combinazioni di potenziali cause dei rischi si utilizzano i Risk Trees; questi alberi alla radice hanno il rischio e ogni nodo rappresenta un evento che si può verificare che a sua volta può essere scomposto in eventi minori e così via fino ad arrivare ad eventi foglia i rami che collegano i nodi identificano relazioni di tipo AND o OR. Da qui è possibile identificare tutte le combinazioni di eventi atomici in grado di portare al verificarsi di un rischio; l'operazione utilzzata per individuare tali elementi atomici è la cut-set tree derivation.

L'ultimo step riguarda la Risk Prioritization e la prima domanda che ci si deve porre è da quali rischi partire. Per dare una priorità ai rischi si utilizza la Risk Exposure: RE= P(UO)* L(UO). Per dare una

priorità ci si basa sia sul valore di RE sia su quello di P(UO) sia di L(UO); infatti vi sono degli intervalli di valori ai quali possono appartenere i valori di questi e in base all'intervallo di appartenenza si ha un rischio più o meno serio.

Andiamo ora ad analizzare nel dettaglio gli step del Risk Control. Iniziando dal Risk Management-Planning che riceve in ingresso una lista di rischi risultato degli step precedenti e sviluppa un piano di controllo e di gestione dei rischi. Viene definito un documento che pianifica diversi aspetti di controllo e gestione dei rischi quali: obiettivi, responsabilità, approcci, risorse, ecc… Anche l'attività di gestione dei rischi è legata a delle checklist che possono essere consultate per i rischi più comuni ed ottenere delle direttive operative sulla gestione di tali rischi.

In generale quando si va a valutare un rischio ci sono degli aspetti generali da tenere in conto come ad esempio la probabilità che tale rischio si verifichi; il lavoro che si esegue sotto questo aspetto è quello di andare a ridurre la probabilità che il rischio si verifichi oppure, se possibile, andare ad evitare completamente che il rischio si verifichi. Oltre alla probabilità è possibile andare a lavorare sul danno che un rischio comporterebbe e quindi in questo caso si va a cercare di minimizzare i possibili danni oppure, se possibile, evitare completamente dei danni nel caso in cui questo rischio si verifichi.

Una volta che si è a conoscenza delle varie contromisure da adottare per ridurre probabilità e danno di un rischio ci si deve porre il problema di confrontarle tra di loro e scegliere quale utilizzare. Vengono quindi utilizzati due metodi quantitativi che sono il Risk-Reduction Levarage e il Defect Detection Prevention. Il primo si occupa di calcolare quanto una certa contromisura può ridurre un certo rischio con la seguente formula: RISK EXPOSURE – (RE SEA PPLICATA CONTROMISURA) / COSTO CONTROMISURA. Il secondo metodo invece permette di confrontare le varie contromisure tramite un confronto indiretto con delle matrici che si articola in tre passi:

1- Produzione di una Risk Impact matrix ottenendo informazioni riguardanti la criticità, likelyhood e perdita per ogni rischio
2- Produzione di una Countermeasure Effectiveness matrix che individua dati quali la riduzione dei danni, la riduzione combinata delle varie contromisure ed infine l'effetto complessivo delle contromisure
3- Determinare una soluzione che bilanci la riduzione del rischio e il costo della contromisura

Le due fasi successive del Risk Control sono le Risk Monitoring e Risk Resolution; queste sono strettamente legate tra loro e collaborano in quanto mentre si sta risolvendo un rischio è necessario mantenere attivo il monitoraggio di tale rischio e viceversa. Monitorare i rischi è molto costoso quindi si cerca generalmente di mantenere controllati i primi 10 nella lista della priorità.

CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

La probabilità di portare a termine con successo un processo dipende in gran parte dalla maturità del processo. La maturità di un processo dipende da un grado di controllo che si ha delle azioni svolte per realizzare il progetto. Un progetto poco maturo non ha ben definite le attività di sviluppo e non ha controllo sulla gestione del progetto. Il CMMI è uno strumento che si occupa di controllare un processo per renderlo più maturo andando a definire delle azioni e pratiche

standard per governare al meglio un progetto. Il CMMI è strutturato da una Process area che si suddivide a sua volta in dei sotto processi che seguono in ordine di grandezza:

1- Process Area: collezione di obiettivi che riguardano delle particolari aree del processo di sviluppo software. Ciascuna process area ha una descrizione dell'obiettivo finale, delle note introduttive riguardanti l'ambito trattato e infine definisce le process area ad essa correlate.
2- Specific Goals e Generic Goals: gli obiettivi generici sono comuni a tutte le process area, mentre gli obiettivi specifici caratterizzano una process area e sono unici per quella specifica process area.
3- Pratiche specifiche e pratiche generiche: pratiche specifiche per il raggiungimento di un obiettivo specifico e pratiche generiche per raggiungere un obiettivo generico. Le pratiche possono essere organizzate in delle sottopratiche e alla fine producono un workproduct.

Tutto quello appena descritto va a creare un mapping che definisce le azioni da svolgere per il conseguimento degli obiettivi di un progetto. Lo standard CMMI introduce dunque dei livelli di maturità ben specifici che un progetto può raggiungere. Vi sono due linee di miglioramento che possono essere prese in considerazione per lo standard CMMI:

1- Capability levels: rappresenta quanto bene si sta gestendo una process area. I livelli vanno da 0 a 3
2- Maturity levels: rappresenta quanto bene si sta lavorando tutte le process area attivate. I livelli vanno da 1 a 5

Andiamo a definire brevemente i generic goals del CMMI:

1- GG1: svolgimento delle pratiche specifiche
2- GG2: svolgimento delle pratiche di planning, allocazione delle risorse, responsabiltà, ecc..
3- GG3: svolgimento delle azioni di tailoring come definire i processi e raccogliere feedback sulle esperienze legate al loro sviluppo

In base al raggiungimento dei GG abbiamo un Capability level differente; GG1 corrisponde ad un capability level 1 e così via.

I Maturity levels in breve corrispondono a:

1- Progetto caotico e disorganizzato
2- Azioni pianificate ed eseguite secondo delle policy
3- Organizzazzione standard aziendale
4- Misura quantitativa sull'andamento del processo
5- Continua ottimizzazione del processo in itinere

Se tutte le process area appartenenti ad 1 dei 5 livelli sono svolte con un CL pari a 3 allora possiamo dire che il livello di maturità di un processo è pari a quel livello.

REQUIREMENT ENGINEERING

Quando si sviluppa un software bisogna farlo in modo tale che esso sia in grado di risolvere correttamente dei problemi e dunque è necessario definire e comprendere al meglio quali problemi il nostro software deve risolvere e il contesto nel quale si pongono questi problemi.

Quindi possiamo dire che i requisiti di ingegnerizzazione si focalizzano non su come sviluppare un software ma nel definire al meglio quali problemi deve risolvere; quindi, l'attenzione non sarà verso la macchina\codice\funzioni da sviluppare, ma bensì sul mondo esterno, sui suoi fenomeni e sui fenomeni eseguiti dalla macchina nel mondo esterno.

Quando si parla di requirement engineering normalmente quando si cerca di risolvere un problema abbiamo già un sistema esistente (system as-is) che risolve il nostro problema ma in modo non del tutto soddisfacente o nel modo in cui volevamo noi. Noi però vogliamo creare un nuovo sistema che andremo a realizzare (system to-be) che implementa la soluzione in maniera diversa, oppure più completa.

La RE consiste in una serie di attività che hanno come obiettivo quello di esplorare, valutare, riadattare, visualizzare l'obiettivo, le capacità e le qualità rispetto ad un software system to-be. Per fare ciò si sfrutta il system as-is. L'output è un documento di specifica dei requisiti che documenta tutte le qualità e capacità che il nostro software system to-be deve avere.

Il RE è una delle prime attività che vengono svolte durante lo sviluppo di un progetto che, a differenza delle attività successive di sviluppo software, si pone come obiettivo quello di definire un sistema corretto e che faccia quello che ci serve.

Lavorare sul RE è difficile in quanto:

1- Sappiamo che il sistema sarà in continua evoluzione quindi bisogna programmare con lungimiranza
2- L'ambiente di lavoro è ibrido e si può incombere non solo in problemi tecnici di sviluppo ma anche in problemi organizzativi tra sviluppatori
3- Bisogna trattare molti aspetti, tra cui gli aspetti qualitativi e funzionali
4- Bisogna lavorare su diversi livelli di astrazione come, ad esempio, i macro-obiettivi strategici e gli obiettivi operativi
5- Abbiamo tanti stakeholders (quelli che hanno interesse nel progetto) con diverse richieste
6- Considerazioni tecniche sullo sviluppo del software stesso

Vi sono diversi tipi di requisiti sui quali si lavora durante la fase di RE, tra cui i requisiti non negoziabili (descriptive statements) e i requisiti negoziabili (prescriptive statements). I requisiti possono differire tra loro per gli elementi che prendo in considerazione che possono essere i requisiti del software, le proprietà del mondo esterno con il quale il sistema deve interagire e, infine, le assunzioni iniziali riguardanti gli ambienti in cui si deve sviluppare il software.

Un modello utilizzato per rappresentare l'interazione tra software e ambiente esterno è il Parnas95. L'interazione in questo modello viene schematizzata da 4 elementi:

1- Input device: sensori che aiutano il software a percepire l'ambiente esterno
2- Software to be
3- Output device: attuatori che agiscono sull'ambiente esterno
4- Ambiente esterno

I requisiti del software possono essere suddivisi in due gruppi:

- Requisiti funzionali: indicano quali funzionalità un sistema deve avere

- Requisiti non funzionali: vincoli di qualità sui requisiti funzionali, ovvero come i requisiti funzionali devono esser attuati e quali regole devono rispettare.

I requisiti non funzionali sono moltissimi e non sempre sono utili al nostro progetto, quindi esistono delle tassonomie che raccolgono gran parte dei requisiti non funzionali e dei loro obiettivi che possiamo consultare per individuare quali di questi siano utili al nostro progetto.

REQUIREMENT QUALITIES

I requisiti hanno delle qualità che devono essere soddisfatte:

- Completezza nella descrizione dei requisiti (quasi impossibile perché se ne aggiungono sempre dei nuovi)
- Consistenza tra i requisiti che quindi non si devono contraddire
- Non ambiguità
- Misurabilità
- Fattibilità
- Comprensibilità
- Buona struttura
- Modificabilità
- Tracciabilità, ovvero tenere traccia di tutti gli artefatti che hanno portato alla creazione di un requisito

Gli errori tipici che possono essere commessi quando si va a creare un documento dei requisiti sono:

- Omissioni: non identificare per tempo alcuni requisiti
- Contraddizioni: definire requisiti contradditori tra di loro
- Inadeguatezza: requisiti non definiti in modo chiaro
- Ambiguità: requisiti descritti in maniera ambigua che lascia a diverse interpretazioni
- Non misurabilità: requisiti difficilmente testabili a livello di qualità
- Troppa specificità: definire dei requisiti che identificano delle feature a livello di sviluppo e non a livello di prodotto finale
- Requirements impossibili
- Requirements non comprensibili
- Scarsa modificabilità: i cambi di requirement devono essere propagati

REQUIREMENT ENGINEERING PROCESS

Il processo di requirement engineering si sviluppa in 4 fasi in un modello a spirale:

1- Domain understanding e Requirement elicitation: in questa fase si studia il dominio applicativo della applicazione. Si comprende il dominio applicativo, l'organizzazione e le caratteristiche del system as-is. In questa fase si identificano anche gli stakeholders che sono gli acquirenti del nostro prodotto. Per quanto riguarda la elicitation si cerca di capire al meglio come strutturare il progetto, quali sono i vincoli attuativi, quali sono i requisiti software e quali interazioni il nostro prodotto deve avere con l'ambiente esterno. Per fare ciò si interagisce molto con gli stakeholders per ottenere queste informazioni.

2- Evalutation & agreement: in questa seconda fase dopo aver raccolto tutte le informazioni e richieste da parte degli stakeholders bisogna verificare che i requisiti definiti da questi non siano conflittuali ed eventualmente individuare delle alternative per soddisfare i requisiti in conflitto. Inoltre si identificano i rischi del nostro sistema e si stilla una priorità dei requisiti e su quali implementare per prima.
3- Specification & documentation: definizione precisa tramite documentazione dei requisiti software, dei requisiti non funzionali, del dominio e delle possibili evoluzioni del sistema nel cosiddetto requirements document
4- Validation & verification: fase in cui si verifica la correttezza dei requisiti validandoli, verificandoli e sistemando eventuali errori al fine di consolidare il nostro requirements document

Una volta giunti al punto 4 si reitera il ciclo in maniera molto simile a quella dei metodi agili al fine di identificare nuovi requisiti, nuovi stakeholders e di migliorare complessivamente il documento RD.

ELICITATION TECHNIQUES

Ora analizzeremo con precisione la prima fase di RE, ovvero la fase di domain understanding e requirement elicitation. La selezione degli stakeholder è una delle prime operazioni da eseguire in quanto grazie ad essi si può avere una migliore conoscenza riguardo i requisiti della clientela interessata al nostro progetto. Gli stakeholder vengono coinvolti nel processo e i requirement fatti da essi tendenzialmente devono essere sempre soddisfatti. La selezione di questi dunque diventa di fondamentale importanza e in generale si selezionano in base alle seguenti caratteristiche:

- Conoscenza del dominio di lavoro
- Posizione nella organizzazione
- Ruolo decisionale
- Obiettivi personali e conflitti di interesse

Per identificare quali tra i vari candidati nel ruolo di stakehold sono i migliori in genere si esegue una sessione di brainstorming tra gli analisti del team dove ci si pongono domande del tipo “Chi ha soldi? Chi ha più probabilità di avere un prodotto di successo? Chi ha conoscenze specifiche cruciali per il progetto?Ecc”

Una volta identificati gli stakeholder bisogna confrontarsi con questi nella cosiddetta fase di elicitation. Il dialogo con gli stakeholder è complesso in quanto vi è una disparità di conoscenza di dominio, vi sono diversi ostacoli comunicativi, difficoltà nell’accesso alle risorse ecc.. Di fronte a questi problemi è evidente che vi sia una necessita di comunicare con della terminologia adatta e di descrivere al meglio i punti chiave del progetto.

Una componente molto importante della interazione con gli stakeholders è quella della Knowledge Reformulation, ovvero riformulare le informazioni acquisite allo stakeholder per verificare che la nostra comprensione sia corretta.

Talvolta gli stakeholders sono molti e quindi è necessario fare una distinzione tra i diversi stakeholders e comportarsi in maniera differente con essi. I fattori di maggior influenza che

distinguono uno stakeholder da un altro sono il potere decisionale e l'interesse sul progetto. Tanto più alto è l'interesse e il potere decisionale tanto più importanti sono gli stakeholders.

ARTEFACT-DRIVEN ELICITATION TECHNIQUES

Le tecniche di elicitazione utilizzate per scoprire i requisiti del progetto si dividono in due grandi famiglie:

- Artefact driven: tecniche che fanno uso di artefatti per dedurre i requisiti.
- Stakeholder driven: tecniche che sfruttano la conoscenza e le richieste degli stakeholder per definire i requisiti

Un primo tipo di tecnica artefact driven è il background study che si occupa di studiare le organizzazioni interessate nel progetto, il dominio applicativo e, infine, il system as-is per estrarre quante più informazioni possibili. Questo tipo di ricerca ci consente di ottenere numerose informazioni senza coinvolgere lo stakeholder che può essere costoso e inutile per ottenere questo tipo di informazioni, tuttavia il numero di informazioni reperibili è limitato in quanto il numero di documenti da consultare potrebbe essere troppo grande.

Una seconda tecnica artefact driven è l'utilizzo di questionari. Si possono ottenere informazioni considerando una ampia popolazione di stakeholders sfruttando un questionario ponendo domande a scelta multipla. Le domande devono essere chiare e le risposte devono essere precise. Per le domande a risposta chiusa è possibile porre delle domande che richiedono di indicare un grado di soddisfazione/accordo o disaccordo riguardante delle affermazioni utilizzando ad esempio la scala di Liker.

Un altro artefatto utilizzabile sono gli storyboard. Questa tecnica consiste nell'utilizzare degli esempi per creare delle storie tramite l'utilizzo di slide/immagini ecc al fine di identificare se la nostra comprensione del system as-is o system to-be sia corretta. Le storyboards possono essere creati in maniera attiva o passiva. I metodi passivi prevedono che la storyboard sia creata e poi proposta agli stakeholders, mentre i metodi attivi prevedono che gli stakeholders contribuiscano alla creazione della storyboard. Si possono utilizzare anche degli scenari per descrivere le tipiche sequenze di interazione del sistema:

- Scenari positivi: rappresentano come il sistema si dovrebbe comportare
- Scenari negativi: rappresentano come il sistema non dovrebbe comportarsi
- Scenari normali: identifica delle situazioni ricorrenti nel sistema
- Scenari anormali: identifica situazioni eccezionali che si possono verificare

Gli scenari sono comunque uno strumento limitato in quanto possono fornire informazioni parziali, possono incombere in delle informazioni troppo specifiche, possono contenere dettagli poco utili o infine essere incompatibili con le conoscenze e richieste di tutti gli stakeholders.

Esistono altre tecniche artifact driven che fanno uso di prototipi e mock-ups con obiettivo di controllare l'adeguatezza di un requisito producendo dei piccoli esempi di software in azione. L'obiettivo è quello di chiarire se quello che si sta costruendo è coerente con gli interessi e richieste degli stakeholders. Il focus durante la creazione di questi prototipi può essere quello di verificare se si sta implementando correttamente una specifica funzionalità, oppure un focus legato alla interfaccia grafica e usabilità del prodotto. La distinzione tra prototipo e mockup è che

se l'artefatto creato ha come unico fine quello di essere esposto una sola volta agli stakeholders allora abbiamo un mock-up, mentre se ciò che creiamo sarà parte del nostro prodotto finale si parla di prototipo. Questa tecnica è molto utile per mostrare concretamente come sarà il software, ma può portare ad alte aspettative.

STAKEHOLDER-DRIVEN ELICITATION TECHNIQUES

Passiamo ora a definire quali sono le tecniche utilizzate coinvolgendo direttamente gli stakeholders nel processo di elicitazione.

La principale tecnica utilizzata è l'intervista. Questa tecnica consiste nel selezionare un solo stakeholder dal quale si vogliono acquisire informazioni, organizzare una riunione, fare delle domande e registrare le risposte e fare anche dei report che sarà successivamente sottoposto all'intervistato per una validazione. È possibile intervistare più stakeholders ma il contatto diventa più debole e le informazioni ottenute saranno minori. Questo approccio è molto costoso dunque è necessaria una fase preparatoria adeguata. Vi sono due tipologie di interviste:

- Interviste strutturate: l'intervista consta in una serie di domande preparate e specifiche per identificare i requisiti
- Interviste non strutturate: l'intervista consiste in una discussione aperta su degli argomenti importanti sul system as-is e to-be per definire i requisiti e dare spazio alle considerazioni degli stakeholders

La preparazione delle domande strutturate deve essere fatta in maniera meticolosa e specifica per quel determinato stakeholder; bisogna inoltre mettere subito al centro dell'intervista le problematiche e considerazioni dell'intervistato seppur mantenendo sempre un buon controllo dell'intervista senza divagare.

Una seconda tecnica stakeholder driven è quella basata sull'osservazione e gli studi etnografici. Spesso capita che l'utilizzo di parole per descrivere dei concetti risulta molto complicato per gli stakeholders e quindi si può utilizzare questo tipo di approccio che consta nell'osservare gli stakeholder che svolgono dei task per identificare come queste azioni vengono svolte e dunque trarre informazioni utili del system as-is. Queste osservazioni possono essere di due tipi:

- Attive: invece di osservare si partecipa al task svolgendolo personalmente per comprenderne il funzionamento e le criticità. Questo consente di individuare dei problemi nascosti di cui gli stakeholders non si rendono conto.
- Passive: si osserva come da esterni le operazioni degli stakeholder e il loro metodo (protocol analysis) oppure si osservano dei task per un lungo periodo per identificare delle proprietà particolari (ethnographic study)

Un'ultima tecnica è quella della group session. Questa tecnica è molto utile per la risoluzione di conflitti in quanto si sceglie un gruppo di partecipanti e si innesca una discussione al fine di individuare caratteristiche del system to-be e risolvere i conflitti tra richieste da diversi stakeholder. Le group session possono essere di due tipi:

- Strutturate: ogni partecipante ha un ruolo come quello del moderatore, manager, developer ecc.. Questo contribuisce ad identificare i requisiti di alto livello comuni a tutte le figure.
- Non strutturate: definite anche sessioni di brainstorming dove i partecipanti non hanno alcun ruolo e dove si generano inizialmente delle idee per risolvere i conflitti e successivamente le si analizzano e si sceglie la migliore e quale approccio utilizzare.

REQUIREMENTS & DOCUMENTATION

Abbiamo appena descritto nel dettaglio la prima fase di RE ora passiamo alla terza dato che le tecniche di valutazione dei requisiti tra cui diverse riunioni e tecniche di evaluation le lascia fare a te perché il cazzo di corso è blended.

L'attività di specification & documentation definiamo in maniera chiara e decisa i concetti e assunzioni che fanno parte nel nostro software. Il tutto viene organizzato all'interno di un documento chiamato Requirements Document.

La domanda che sorge spontanea è come documentare i requisiti. Vi sono diverse tecniche per farlo e ne seguiranno alcune.

La prima tecnica è quella di utilizzare il linguaggio naturale per descrivere i requisiti. Il linguaggio naturale ha una ampia espressività, ma al tempo stesso questo nasconde delle ambiguità. Si può utilizzare un linguaggio naturale strutturato per risolvere le problematiche appena esposte; questo linguaggio prevede l'utilizzo di due regole generali ovvero le regole locali e le regole globali. Le prime definiscono come andiamo a descrivere un singolo requisito, mentre le seconde riguardano l'organizzazione dell'intero documento RD. Prima di vedere queste tipologie di regole dobbiamo identificare delle regole generali applicati a tutti i documenti RD:

- Identificare chi legge i requisiti e scriverli in maniera adeguata (più o meno tecnica)
- Spiegare cosa si sta per fare prima di farlo
- Motivare le scelte e sintetizzarle
- Definire ciascun concetto prima di utilizzarlo
- Identificare cosa è rilevante e cosa no
- Usare esempi e diagrammi

Per quanto riguarda le regole locali abbiamo ad esempio l'utilizzo di un template per la specifica dei requisiti. I requisiti dunque saranno definiti in maniera uguale dove vengono specificati quali campi devono essere compilati ogni volta che si vuole descrivere un requisito come ad esempio:

- Un identificatore unico per il requisito che ne ricorda la funzione
- Categoria di apparenza del requisito come assunzione, definizione, requisito funzionale, ecc..
- Specifica formulazione del requisito
- Fit criterion che equivale al test di accettazione dei metodi agili per capire se tale requisito è stato implementato bene. È importante dare dei valori misurabili ai requisiti per poterli valutare
- Priorità del requisito
- Stabilità del requisito

Per quanto riguarda le regole globali abbiamo il documento IEEE Std-830 che fornisce un template organizzato in 3 sezioni principali: introduzione, descrizione generale, requisiti specifici.

La parte introduttiva introduce il documento e il sistema e specifica:

1- Qual è l'obiettivo del RD e che tipo di informazioni troviamo e a chi sono rivolte le varie sezioni di questo documento
2- Descrizione breve del prodotto, del suo dominio e del suo obiettivo
3- Descrizione delle definizioni più importanti e acronimi
4- Riferimenti alle risorse
5- Overview del documento

La sezione di descrizione generale si occupa di dare una definizione del confine tra il prodotto che si va a realizzare e l'ambiente con il quale interagisce per capire bene cosa possiamo controllare e cosa no. Si suddivide in 6 step che sono:

1- Definizione della prospettiva del prodotto rispetto all'ambiente esterno
2- Definizione delle funzionalità del prodotto
3- Definizione degli utenti che useranno il prodotto
4- Definizioni dei vincoli generali sul prodotto
5- Definizione delle dipendenze e eventuali assunzioni
6- Indicare quali sono i requisiti principali e quali possono essere fatti dopo

L'ultima sezione del documento IEEE std-830 è quella dei requisiti specifici, dove i requisiti sono definiti con il massimo dettaglio possibile utilizzando ad esempio il template visto nelle regole locali. Questa fase si articola in 2 step:

1- Distinzione dei requisiti funzionali che vanno suddivisi in maniera specifica (GUARDA APPROFONDIMENTI)
2- Definizione dei requisiti non funzionali rilevanti e creare uno step per ciascuno di questi

REQUIREMENTS SPECIFICATION DIAGRAMS

Il linguaggio naturale non è l'unica opzione possibile per andare a specificare i requisiti, una di queste è quella dell'utilizzo dei diagrammi. I diagrammi hanno una notazione semi formale, in quanto se da una parte i grafici hanno delle regole per la loro creazione, dall'altra parte i diagrammi possono essere interpretati ambiguamente.

I diagrammi sono molti utili per:

- Complementare ciò che viene definito con linguaggio naturale
- Andare a descrivere specifici aspetti del sistema as-is e to-be
- Ha una comunicazione di tipo grafica

Alcuni dei diagrammi che possono rappresentare come un sistema si deve comportare, dunque descrivere i requisiti, sono i problem diagrams. Questi diagrammi ci permettono di dare una descrizione dei requisiti a livello di sistema. In questi diagrammi vengono rappresentati gli elementi del sistema, gli elementi del mondo esterno con i quali essi interagiscono e i requisiti collegati in maniera diretta agli aspetti del mondo esterno di cui vincolano il comportamento.

Altri diagrammi utilizzati sono i frame diagrams. Questi diagrammi hanno l'obiettivo di mettere in luce dei pattern frequenti di risoluzione di un requisito. A differenza del problem diagram in questo caso abbiamo per ogni elemento del diagramma un tipo su eventi e componenti. I tipi che possono assumere le componenti sono: causal (comportamento di causa-effetto ricevo un comando e produco un risultato), biddable (comportamento non predicibile) e lexical (non ha comportamento ma definisce un artefatto). I tipi che possono assumere gli eventi sono: causal (eventi che sono diretta conseguenza di un altro evento), event (eventi veri e propri), lexical (dati che vengono utilizzati o salvati).

SU MOODLE CI SONO MOLTI FRAMES UTILIZZATI PER IMPLEMENTARE DEI REQUISITI

Altri diagrammi utilizzabili sono i diagrammi entità-relazione che ci consentono di specificare quali sono le entità, come sono fatte, che relazioni hanno con altre entità e infine quali attributi utilizzano.

Un altro tipo di diagramma utilizzato è il SADT diagrams. Questi diagrammi permettono di specificare il comportamento di alcune attività che un software deve eseguire, scomporre il comportamento in sottoattività e aggiungere alcune informazioni riguardanti le attività stesse. Questi diagrammi possono essere di due tipi:

- Actigram: l'elemento centrale che viene documentato è una attività e mostrano le attività relazionate tra di loro da link di dipendenza
- Datagram: l'elemento centrale che viene documentato è un dato e mostrano le dipendenze di controllo

Nel caso del actigram le attività sono rappresentate in dei box. Le frecce che confluiscono nei box da sinistra sono gli input, le frecce che escono dal box e vanno verso destra sono gli output, le frecce che confluiscono nei box dall'alto identificano dati che influenzano il comportamento dell'attività, infine le frecce che confluiscono dal basso indicano quali sono le unità che svolgeranno quella specifica attività.

I datagram invece pongono all'interno dei box i dati e le frecce identificano: da sinistra attività che produce il dato, verso destra attività che consuma tale dato, da sopra attività di controllo della correttezza del dato, da sotto le repository o risorse dove va a finire il dato.

Un diagramma già visto in UML è il use case diagram che mostra i requisiti identificati e quali attori partecipano a ciascun requisito. Altro diagramma conosciuto è il sequence diagram che identifica gli elementi che partecipano in una esecuzione e l'interazione che essi hanno tra di loro durante l'esecuzione. Altro diagramma conosciuto è la macchina a stati che rappresenta gli stati nei quali un elemento del nostro sistema si può trovare e quali sono gli eventi che implicano un cambio di stato.

Quando produciamo dei diagrammi dobbiamo stare attenti in quanto i diagrammi tra di loro devono essere coerenti in quanto spesso si vanno ad intersecare tra loro. Lavorando coi diagrammi abbiamo a disposizione alcune regole di consistenza che ci aiutano ma non ci dicono se ci sono delle incoerenze tra diagrammi. I diagrammi spesso sono raggruppati all'interno di alcuni standard come ad esempio UML che contiene in se i class diagram, use case diagram, sequence diagram e state diagram.

FORMAL NOTATIONS

Le notazioni formali sono molto utili per complementare sia i documenti scritti col linguaggio naturale, sia i diagrammi, specialmente quando si tratta di descrivere aspetti critici. La sintassi e semantica sono definite, quindi il calcolatore può processare le notazioni interamente. Dato che non vi sono ambiguità nell'interpretazione delle note formali, è possibile che il calcolatore faccia delle analisi di coerenza tra i requisiti scritti con questa notazione.

Un esempio di notazione formale è quello della logica proposizionale. La sintassi in questo caso definisce delle regole che consentono di collegare tra di loro delle proposizioni atomiche (true false) con dei connettivi logici (and, or, if, then). Per quanto riguarda la semantica possiamo dire che uno statement può essere interpretato sulla base delle regole di interpretazione semantica che la logica definisce (è vero se sono vere entrambe per &, è vero se anche solo una è vera per |) e che il suo significato è definito (true false).

Si può avere un approccio formale non solo tramite la logica, ma anche usando le specifiche basate sullo stato. Con questo approccio si definisce in maniera formale gli stati che un sistema può attraversare e come delle operazioni che possono essere eseguite modificano lo stato del sistema. Uno stato viene descritto definendo quali sono le informazioni che caratterizzano uno stato e quali sono le regole che tale stato deve soddisfare. Per quanto riguarda le operazioni abbiamo delle precodinzioni che definiscono in che condizioni la operazione può essere applicata e delle postcondizioni che ne descrivono l'effetto.

In conclusione a tutti questi capitoli che partono dall'utilizzo del linguaggio naturale non strutturato fino ad arrivare alle note formali possiamo dire che:

- Il linguaggio naturale non strutturato non è una opzione valida e fa schifo al cazzo
- Il linguaggio naturale strutturato è la opzione principale da utilizzare in particolare per le regole locali e globali
- I diagrammi sono molto importanti per complementare le descrizioni del linguaggio naturale, in particolare quando i concetti esposti dal linguaggio naturale sono più facili da comprendere tramite una comunicazione grafica
- Le note formali sono particolarmente efficaci per trovare dei problemi all'interno dell'insieme dei requisiti, ma che per il loro costo e complessità sono utilizzate solo in rare circostanze

REQUIREMENTS QUALITY ASSURANCE AND EVOLUTION

La quality assurance e evoluzione dei requisiti è lo step finale della RE. La domanda spontanea che ci si pone è quella di come validare i requisiti; dare una risposta a questa domanda è molto difficile e ci sono pochi approcci.

Il primo approccio che è il più applicabile e anche il più costoso consiste nell'eseguire manualmente una ispezione e revisione dei requisiti. Un team di revisori dunque si occuperà di effettuare dei controlli per individuare degli errori specifici manualmente.

Se i requisiti sono memorizzati in un DB ben strutturato, alcuni controlli possono essere automatizzati tramite l'utilizzo di queries.

Se i requisiti sono degli artefatti eseguibili, come ad esempio dei sequence diagram, è possibile fare delle simulazioni atte ad individuare dei bug nelle specifiche dei requisiti.

Un ultima tecnica è quella del model checking dove se abbiamo dei requisiti descritti in modo formale, allora è possibile analizzarli automaticamente rilevando incoerenze, contraddizioni, non soddisfazione di alcune proprietà.

Una volta completata la quality assurance abbiamo terminato una iterazione dei 4 step della RE. Le iterazioni però non terminano con un solo ciclo in quanto abbiamo una evoluzione dei requisiti determinata dai 4 step appena conclusi e quindi si rende necessaria una nuova iterazione di elicitation, evaluation, documentation e quality assurance.

I cambiamenti devono essere gestiti e per farlo è necessario essere consapevoli e preparati del possibile cambiamento dei requisiti e possibilmente anticiparli. Valutare l'impatto di un cambiamento, individuare su quali elementi agire ed essere pronti a tracciare dei legami tra i requisiti sono delle azioni fondamentali da eseguire per gestire al meglio i cambiamenti dei requisiti.

Prepararsi ai cambiamenti significa anche dover gestire in maniera differente i requisiti, ovvero saper distinguere quali requisiti sono predisposti al cambiamento e quali sono più stabili e trattarli in maniera differente. Per fare ciò i requisiti possono essere distinti in dei livelli in generale basso, medio, alto; per individuare a quali livello di stabilità appartiene un requisito possiamo usare alcune indicazioni generali:

- I requisiti che possiamo immaginare come presenti in delle versioni contratte del sistema o anche in delle versioni estese del sistema verosimilmente saranno quelli più stabili
- Gli aspetti concettuali tendenzialmente sono più stabili degli aspetti operazionali ad esempio il fatto che deve essere inviata una notifica è più stabile rispetto a come questa debba essere inviata
- Le funzionalità chiave del sistema sono generalmente più stabili di quelle non funzionali
- I requisiti ottenuti da una analisi di diverse opzioni, come ad esempio la risoluzione di conflitti o a delle contromisure a dei rischi, tendono ad essere poco stabili

Al fine di gestire al meglio l'evoluzione dei requisiti un fattore di vitale importanza risulta essere la tracciabilità. Si può dire che un elemento è tracciabile se è possibile dire da dove esso arriva, il significato della sua esistenza, come viene usato e per quale obiettivo. Essere a conoscenza di queste informazioni rende più semplice la gestione dei cambiamenti di quel preciso requisito in quanto possiamo valutare l'impatto di un possibile cambiamento e individuare quali altre componenti siano influenzate da questo cambiamento.

Per rispondere a queste domande è necessario essere in grado di stabilire dei legami tra elementi di diverso tipo che vanno dal codice sorgente ai manuali dell'utente. Vi sono dunque diversi tipi di link che collegano gli elementi di un software tra di loro:

- Link forward e backward: link che vanno dalla source al target e viceversa, ovvero se un requisito motiva la realizzazione di un pezzo di architettura e viceversa
- Link orizzontali e verticali: orizzontali identificano dipendenze tra artefatti dello stesso tipo come un requisito che dipende da un altro requisito, verticali invece quando gli artefatti dipendenti tra loro appartengono a fasi diverse dello sviluppo

Percorrere questi link ci consente di individuare le implicazioni causate da un cambiamento di un determinato elemento. La parte più difficile delle tracciabilità è quella di tracciare in maniera efficiente i link che collegano tra di loro gli elementi del sistema. Alcune tecniche di traceability sono:

- Cross referencing: tecnica che si occupa di assegnare un identificatore univoco ad ogni elemento e che questo identificatore può essere utilizzato all'interno di altri elementi per creare un link. Dopo aver creato questi link è possibile utilizzare dei browser di ricerca per individuare le correlazioni tra vari elementi del sistema. Questa tecnica è inizialmente poco costosa e veloce da implementare, ma in caso di cambiamenti diventa costosa perché si devono aggiornare tutti gli identificatori.
- Traceability matrices: questa tecnica va a creare delle matrici di tracciabilità che sono delle matrici le cui righe e colone rappresentano gli elementi del sistema e i valori nelle celle sono 1 se c'è una dipendenza tra gli elementi e 0 se non c'è. Nel caso di cambiamenti risulta più semplice in questo caso consultare le dipendenze dell'elemento, tuttavia risultano difficili da gestire le modifiche alla matrice.
- Feature diagrams: questo tipo di diagramma permette di rappresentare il concetto di variante di un sistema, ovvero un sistema che può essere distribuito in versioni con differenti feature. Alla radice di questo diagramma vi sarà il progetto e i nodi identificano le feature che fanno parte del sistema. I nodi che identificano le feature possono essere obbligatori o opzionali in quanto in alcune versioni del software alcune feature possono non esserci.

CI SONO ALTRE TECNICHE CHE CI SONO NELLE DISPENSE

Con questo si conclude tutta la parte relativa al processo di Requirement Engineering.

DEVELOPMENT OF ENTERPRISE APPLICATIONS – SEZIONE MVC

In questo capitolo ci si occuperà di sviluppo su framework per applicazioni enterprise. I tre contenuti principali di questo capitolo sono: MVC (Model View Controller), ORM (Object Relational Mapping), Component-based (enterprise) systems.

Il Model View Controller è una soluzione architetturale che a livello macroscopico si compone in tre parti:

- Model: identifica le classi che realizzano la logica applicativa dell'applicazione (backend) e i dati. Si occupa di esporre lo stato dei dati e le funzionalità della applicazione
- View: identifica ciò che viene mostrato all'utente. Ha la responsabilità di rappresentare il contenuto informativo del Model con delle finestre di tipo grafico.
- Controller: controlla l'interazione tra Model e View. Controlla gli input degli utenti che riceve la view (user gestures) e li mappa in operazioni eseguibili dal model (ad esempio la richiesta di login) per poi andare anche a selezionare la view di risposta da dare all'utente dopo la sua azione

Vediamo ora alcuni design patterns per comprendere meglio come articolare ciascuna di queste tre componenti. I design pattern del model li abbiamo già visti (in un passato a me sconosciuto)

quindi ci concentriamo sui pattern messi a disposizione per il controller e la view. I design pattern che andremo a vedere sono:

Controller design:

- Page controller
- Front controller
- Intercepting filter (da fare da dispense)
- Application controller (da fare da dispense)

View design:

- Template view
- Transformation view
- Two steps view (da fare da dispense)

Sappiamo che il controller orchestra l'esecuzione ricevendo una richiesta dell'utente (user gestures) proveniente dalla view, la processa e la traduce in operazioni da eseguire sul model e poi seleziona la view successiva da restituire all'utente.

Le principali responsabilità del page controller sono di ricevere la richiesta, controllare i parametri ed estrarli dalla richiesta, invocare la logica di business basandosi sui parametri della richiesta, selezionare la view successiva e preparare i dati per la presentazione da passare alla view. Questo tipo di pattern ha l'idea di implementare un controller per ciascuna pagina (ad esempio letteralmente tutte le componenti di una pagina web) che esegue le operazioni appena menzionate. In questo modo al crescere della applicazione, delle sue funzionalità e view non si avrà una componente che cresce in egual misura ma si avrà un numero maggiore di page controller che gestiscono piccole parti di sistema. Una semplice implementazione di page controller consiste in un albero dove partendo dal basso abbiamo le classi che eseguono le operazioni del model, nella sezione centrale abbiamo due controller che si occupano di gestire le operazioni definite dalle classi del model, infine nella sezione superiore abbiamo le view collegate ai controller.

Il pattern successivo che analizziamo è il front controller. A differenza del page controller, il front controller è un componente singolo, ovvero che l'applicazione ha un unico front controller che riceve le richieste da tutte le componenti della view. Ogni volta che il front controller riceve una richiesta in ingresso si avvale di una serie di classi che rappresentano i comandi che possono essere richiesti dall'utente. I dati in ingresso vengono gestiti in primis dall'handler, successivamente viene istanziato il comando da eseguire con i relativi parametri e, infine, viene eseguito tale comando che interagirà con la logica applicativa per eseguire la sua funzione. Anche questa soluzione è scalabile in quanto, al crescere dell'applicazione, ciò che cresce non è la dimensione dell'handler che riceve le richieste ma cresce il numero di classi e comandi che vengono aggiunti per implementare nuove funzionalità. Le operazioni eseguire dal front controller dunque sono:

Handler:

1- Riceve la richiesta dalla view

2- Esegue le operazioni generali
3- Decide quale comando eseguire in base alla richiesta
4- Alloca l'istanza di comando (concrete command) e ne delega l'esecuzione

Concrete command:

1- Inizializza i parametri della richiesta
2- Invoca il metodo che esegue la logica del comando
3- Determina la view successiva in base all'esito del metodo
4- Restituisce il controllo alla view

Un esempio di implementazione di questo schema consiste sempre in un albero molto simile a quello precedentemente descritto con delle differenze a livello del controller. Alla base abbiamo le classi gestite dal model, in mezzo abbiamo lo strato di controllo caratterizzato da un front controller che si configura con tutte le view e al di sotto di questo la solita gerarchia di comandi collegata direttamente alle classi del livello model. I vantaggi apportati dal front controller rispetto al page controller sono che si evita il codice duplicato nei diversi controller dei page controller in quanto si utilizza un solo controller, facilita la configurazione in quanto abbiamo un unico entry point e, infine, consente una facile implementazione dei nuovi comandi da applicare per le nuove richieste.

Passiamo ora ai pattern dedicati alla View Design. Il primo pattern che andiamo ad analizzare è la template view. In questo pattern la parte di rendering viene ottenuta definendo parte di contenuto in maniera statica e parte in maniera dinamica il cui valore dipende dal modello e dal suo stato attuale. Quando si ultizza template view spesso capita di utlizzare delle classi helper che consentono alla pagina non faccia diretto accesso al modello ma utilizzano come tramite questi helper. Gli helper quindi permettono di eseguire delle elaborazione intermedie dei dati per facilitare il rendering alle view.

Un secondo pattern abbastanza specifico è il transform view. In questo caso non abbiamo una pagina template con porzioni renderizzate in maniera statica e porzioni dinamiche, ma la pagina viene creata interamente in modo dinamico a partire dai dati che devono essere presentati. Il componente principale che ha il compito di costruire la pagina è il transformer. Un modo di implementare tale pattern può essere quello di arricchire il model in modo da produrre dei dati favorevoli alla costruzione dinamica delle pagine, come ad esempio creare dei dati in format XML per avere una facile conversione in HTML. A livello di albero dunque abbiamo nel livello model le classi che hanno a disposizione funzione per convertire in XML una i dati processati e nel livello controller abbiamo una classe chiamata trasformer che si occupa di trasformare i dati XML in HTML prima di passarli al controller. Il peggior difetto di questo approccio è che risulta molto difficile includere implementazioni logiche all'interno della view, quindi l'approccio non può essere utilizzato per renderizzare una intera pagina ma al massimo una porzione di questa.

SPRING MVC

Il framework Spring MVC implementa molteplici design pattern di cui ne fa uso tra cui:

- MVC
- Front controller

- Intercepring Filter
- Context object
- View helper

Vediamo ora il funzionamento di Spring MVC. Il sistema riceve una richiesta in input, questa richiesta viene intercettata dal dispatcher server che a sua volta distribuisce i dati all'handler mapping, il quale seleziona un controller adatto per servire la richiesta ricevuta; i dati successivamente sono trasmessi all'handler adapter che si occupa di invocare la business logic del controller appena selezionato. Queste tre componenti appena menzionate sono implementate automaticamente da spring MVC. A questo punto entra in gioco il controller che deve essere implementato dagli sviluppatori per definire in che modo esso interagisce con la componente model e, una volta interagito con essa, sceglie quale view restituire all'handler adapter che a sua volta lo inoltra a view resolver che seleziona la view e la restituisce a view.

DEVELOPMENT OF ENTERPRISE APPLICATIONS – SEZIONE ORM

L'Object Relational Mapping si occupa di instaurare una mappatura tra la rappresentazione degli oggetti in memoria e il modo in cui sono persistiti i dati. Quindi il problema che vuole risolvere ORM è, data un'entità nell'applicazione, quello di individuare in che modo trasformarla per andarla a persistere in maniera coerente con il modello utilizzato per la memorizzazione dei dati.

La soluzione proposta da ORM dal punto di vista architetturale prevede la presenza di un client che ha dei dati che devono essere persistiti su una risorsa; per collegare tra di loro questi due elementi viene introdotto un API Gateway che si occupa di effettuare le traduzioni dei dati in entrambe le direzioni.

Esistono diversi pattern in grado di realizzare il gateway in questione e sono:

- Table data gateway (NON LO SPIEGA)
- Row data gateway (NON LO SPIEGA)
- Active record
- Data mapper

Iniziamo ad approfondire Active Record. Questo pattern consente di implementare il processo di traduzione degli oggetti in memoria in record per il nostro DB andando ad arricchire le classi che implementano i dati che devono essere persistenti con dei metodi che servono per l'interazione con il DB. I metodi in questione possono essere dei metodi finder che vengono utilizzati per individuare dei dati all'interno del DB; questi metodi sono metodi statici in quanto essi non restituiscono dei record del DB ma degli oggetti del dominio dell'applicazione. Altri metodi utilizzati da questo pattern sono i metodi gateway che sono non statici e che ci consentono di lavorare con le istanze. Alcuni metodi che possono essere creati sono:

- Load: consentono di creare una istanza di un oggetto a partire dal contenuto del DB
- Constructor: consentono di creare un oggetto in memoria che poi dovrà essere persistito in un DB
- Finder statici: consentono di eseguire delle query sul DB che restituisce una collezione di oggetti

- Write: metodi che consentono di eseguire le operazioni di update, delete, insert di entità nel DB

I difetti di questo tipo di pattern sono che costringe a mescolare la logica di business con metodi legati alla persistenza e che tende a forzare le corrispondenze tra la logica di mapping del DB e del software.

Passiamo ora al pattern Data Mapper. Questo pattern si può considerare una evoluzione del Active record in quanto i metodi per la gestione della persistenza vengono estratti dalla classe. In sostanza quindi avremo una classe con la sua business logic e avremo anche una classe mapper che conterrà al suo interno i metodi utili alla persistenza degli oggetti di quella determinata classe.

Come costruire il mapping e come implementare le classi e i metodi sono solo una parte del ORM. Vi sono diversi altri pattern che svolgono funzioni diverse ma altrettanto importanti sempre sulla persistenza dei dati. Alcuni di questi pattern sono:

Pattern di comportamento:

- Unit of work
- Identity map
- Lazy load

Pattern strutturali:

- Value Holder
- Identity field
- Embedded value
- LOB

Iniziamo analizzando la Unit of Work; questo pattern si occupa di individuare il momento ideale in cui le modifiche fatte in locale vengono sincronizzate con il DB. La soluzione più adatta sarebbe quella che prevede che l'utente esegua le sue operazioni e il sfotware mantiene in memoria tutte le operazioni eseguite in modo tale che al momento del commit da parte dell'utente viene avviata la transazione a livello di sistema che esegue le operazioni sul DB per poi chiudere la transazione. L'oggetto che si occupa di mantenere in memoria tutte le azioni di un utente si chiama Unit of Work; essa implementa una serie di metodi chiamati register che tengono traccia degli oggetti creati, modificati o cancellati. La Unit of work lavora in collaborazione con le strategie di Lock dato che è possibile che vi siano diverse transazioni aperte in contemporanea e per questo devono essere gestite con dei lock. Le strategie di lock possono essere di due tipi:

- Locking ottimistici: questo tipo di lock assume che i diversi utenti lavorino su dati differenti e che quindi sia improbabile che vi siano dei commit che vadano ad operare sugli stessi dati. Questo comporta che i record non siano bloccati ma in essi viene aggiunto un numero di versione che viene incrementato ad ogni modifica eseguita su quel dato, in modo tale che se il numero di versione attuale è diverso da quello ottenuto in fase di modifica del dato viene lanciato un rollback
- Locking pessimistici: questo tipo di lock non consente agli utenti di accedere ai record sui quali altri utenti stanno già eseguendo delle operazioni. Questa soluzione risulta più lenta ma evita il rischio di avere dei rollback causati dalla concorrenza.

L'ultimo aspetto su cui ragionare ora è quello di individuare chi deve notificare alla Unit of Work che è avvenuto un cambiamento su un dato del DB. Ci sono due modi di fare ciò:

- Caller registration: chi modifica il dato notifica che ha modificato il dato. Il metodo invocato dall'utente sulla unit of work in questo caso è il registerdirty() e viene eseguito prima del commit
- Object registration: l'oggetto modificato informa la unit of work di essere stato modificato.

Il secondo pattern che andiamo ad analizzare è identity map. Quando carichiamo degli oggetti provenienti dal DB in memoria è possibile che vengano caricate più copie dello stesso oggetto; se successivamente dovessimo andare a modificare una delle due copie si avrebbe un problema di inconsistenza dei dati. Per evitare questo problema basta caricare in memoria solo gli oggetti che non sono già presenti e per fare ciò si utilizza identity map. Questo pattern svolge una funzione similare a quella della cache che sta in mezzo alla comunicazione tra applicazioni e DB; prima di andare ad estrarre dei dati dal DB si controlla che essi non siano già presenti in memoria. Per svolgere questo compito identity map mantiene i riferimenti a tutti gli oggetti in memoria.

Altro pattern di cui è utile parlare è il lazy load. Caricare degli oggetti dal DB può avere dei problemi di efficienza, specialmente quando si vanno a caricare oggetti di grandi dimensioni. Il lazy load si occupa di risolvere le situazioni in cui da oggetti contenenti numerosi attributi se ne inizializzano solo la parte di cui si è interessati ed eventualmente su richiesta futura da metodi che fanno uso degli attributi mancanti si caricano gli attributi restanti.

Ultimo pattern trattato è identity field. Questo pattern si occupa della sincronizzazione degli oggetti in memoria con i record del DB. Per identificare questo mapping identity field identifica un campo ID aggiunto all'interno delle classi che rappresentano oggetti persistenti sia nel DB che nella classe. Vi sono due modi per generare questo ID; o lo si fa generare dal DB, oppure lo si fa generare dalla applicazione con table scan che esegue una query per identificare il prossimo valore chiave oppure key table che consiste in una tabella contenente tutte le chiavi e il loro valore assegnato e si assegna un valore successivo all'ultimo creato.

JPA

Lo standard Java Persistent Api (JPA) viene utilizzato per applicare l'ORM in quanto dispone di tutti i meccanismi per farlo e quindi ci basta solo dire quali vogliamo utilizzare e come si deve comportare il framework. JPA possiede un sistema di annotazioni che ci permette di istruire il framework su come gestire il mapping e dei servizi che gestiscono le operazioni di persistenza dei dati. Una applicazione JPA è una applicazione java che possiede delle classi che implementano la buiness logic e dei particolari tipi di classi chiamati entity class che sono le classi che contengono i dati che si mappano nel DB. Le classi dell'applicazione interagiscono con un servizio chiamato entity manager che implementa la maggior parte delle funzioni del ORM.

Iniziamo a vedere nel dettaglio il funzionamento di entity manager la cui funzione principale è quella di gestire le classi che rappresentano i dati che devono essere persistiti (entity class). Una entità all'interno di una applicazione può essere in due stati:

- Managed o attacched: identifica una entità sotto la gestione dell'entity manager. Questo implica che possano essere utilizzati i pattern di Identity Map o Unit of Work che

consentono di tracciare in automatico le modifiche apportate sui dati a patto che sia stato dichiarato quali classi siano nello stato managed.
- Unmanaged o detached: identifica una entità che non è gestita dall'entity manager. L'oggetto quindi è presente in memoria ma per l'entity manager è come se non esistesse.

Le entity clas devono essere definite in modo tale che sia stabilita una connessione al DB nel quale devono essere persistiti i lori dati; nel caso in cui si volessero utilizzare più DB gruppi di entity class vengono suddivisi in persistent unit che possono essere assegnate a diversi DB. Allo stesso modo le classi presenti in memoria sono suddivise in gruppi uguali chiamati persistent context.

A livello di codice rendere una classe una entity class è facilissimo basta mettere @entity prima della definizione della classe e assegnare un id con @id. Implementare un entity manager è altrettanto semplice basta usare la funzione createEntityManager() che riceve come argomento il nome della persistent unit. Una volta implementato l'entity manager è possibile eseguire diverse operazioni tra cui:

- Persistere i dati: usando la funzione persist alla quale si passa come argomento l'oggetto da persistere. Attenzione che il comando mette in schedule questa operazione per eseguirla non appena viene terminata la transazione e non la esegue istantaneamente
- Caricare degli oggetti: usando la funzione find() o getReference() che riceve come argomento l'id dell'oggetto interessato e il nome della classe di quell'attributo.
- Modificare le entità: tutte le modifiche eseguite su classi managed sono tracciate e vengono applicate automaticamente alla conclusione di una transizione. Se volgiamo far diventare una classe managed dobbiamo usare merge() al quale si passa come attributo l'entità e restituisce una nuova entità merged.
- Rimuovere entità: usando la funzione remove(). L'entità rimossa diventa unmanaged

Per quanto riguarda il Mapping, basta configurare correttamente il framework per ottenere un mapping adeguato. Bisogna definire le primary keys delle entità (generalmente un valore long) per definire correttamente il mapping di una entità nel DB. Nel caso in cui si volessero generare due tabelle nel DB a partire da una sola entità, allora alla notazione @table() utilizzata per definire una tabella da persistere segue la notazione @SecondaryTable() alla quale si passa la porzione di tabella da persistere come tabella a se stante. Le due tabelle condividono la primary key in modo tale da rendere possibile il join tra le due tabelle.

Le relazioni tra le entità all'interno di JPA possono essere di vario tipo:

1- Relazioni uno ad uno: a loro volta possono essere monodirezionali e bidirezionali. Per le relazioni monodirezionali si usa @OneToOne sull'attributo di una tabella che fa riferimento al contenuto di un'altra tabella e si setta come contenuto della colonna la primary key della tabella di cui si sta facendo riferimento.
2- Uno a molti / molti a uno: a loro volta possono essere monodirezionali e bidirezionali. Per le relazioni monodirezionali uno a molti si usa la relazione @OneToMany e si usa la funzione @JoinTable() per eseguire un join tra le tabelle della relazione uno a molti. Per le relazioni monodirezionali molti a uno si usa la notazione @ManyToOne prima di definire una colonna della nostra tabella che conterrà il riferimento ad un'altra tabella.

3- Molti a molti: a loro volta possono essere monodirezionali e bidirezionali. Anche qui nel caso monodirezionale basta la notazione @ManyToMany prima della tabella che collega le tabelle da mettere in relazione tra di loro.

Ultima sezione trattata da JPA riguarda il mapping delle eriditarietà. Nei DB non esiste una gerarchia tra oggetti, tuttavia quando si eseguono delle query sul DB è necessario avere una mappatura delle gerarchie al fine di estrarre le sottoclassi degli elementi selezionati. Per fare ciò ci sono tre strategie:

- Single table: si usano le notazioni @Inheritance() al quale si passa un parametro che definisce la strategia di ereditarietà (in questo caso single table), @DiscriminatorColumn che genera le colonne contenenti gli attributi delle classi e @DiscriminatorValue che non so cosa fa, queste annotazioni consentono di visualizzare in un'unica tabella tutte le classi nella stessa gerarchia il che può comportare la assenza di valori per determinati attributi
- One table per concrete class: questa strategia crea una tabella per ogni classe concreta che può essere istanziata. Si utilizza la solita annotazione @Inheritance() con argomento TABLEPERCLASS così che le tabelle ereditano gli attributi della tabella che sta al livello superiore.
- Table per subclass: questa strategia crea delle tabelle con un mapping esatto con le classi della applicazione. In questo caso ogni tabella possiede solo i dati della classe corrispondente ma questa soluzione è lenta in quanto bisogna eseguire delle operazioni di join tra le tabelle per collegarle tra loro

DEVELOPMENT OF ENTERPRISE APPLICATIONS – SEZIONE COMPONENT-BASED SYSTEM

Ultima parte dello sviluppo di applicativi enterprise che tratta dello sviluppo per componenti. Quando parliamo di sviluppo per componenti si parla di uno sviluppo che consente di creare delle unità (componenti) di cui si può fare il deployment in maniera indipendente dagli altri componenti.

L'aspetto chiave per la gestione delle dipendenze tra componenti è il decoupling, ovvero rendere indipendenti le componenti di cui dobbiamo fare il deployment anche se a runtime queste componenti devono interagire con altre.

Vediamo alcuni design pattern che gestiscono dinamicamente le dipendenze tra componenti:

- Inversion of Control: i campi che memorizzano dei riferimenti di altre componenti vengono popolati automaticamente da un componente esterno. L'idea quindi è di introdurre un oggetto esterno chiamato assembler che ha la funzione di individuare il componente che deve essere utilizzato ed iniettare la dipendenza nella classe di cui ha bisogno. In generale la dipendenza viene iniettata quando l'oggetto viene creato. Le dipendenze possono essere iniettate in tre modi: usando il costrutture, usando un setter, usando una interfaccia.
- Service Locator: la classe che ha bisogno di popolare un campo con un riferimento chiede in maniera esplicita al service locator il valore di riferimento. L'idea è quella di avere un registro che conosce qual è l'implementazione che deve essere usata di volta in volta. L'implementazione più semplice consiste in un singleton con coppia chiave valore.

Vediamo ora un modello per sviluppo di componenti java chiamato EJB. Le funzioni di EJB che andremo a vedere si concentrano principalmente sul lato server side delle componenti che hanno le seguenti caratteristiche:

- Implementano la business logic
- Sono componenti distribuiti
- Facili da integrare
- Utilizzano ORM con JPA
- Hanno una comunicazione asincrona
- Sfruttano i servizi web services

EJB è conosciuta come la tecnologia dei Three Beans, ovvero esistono tre componenti chiamate session bean usati per implementare la comunicazioni sincrona e message-driven bean per la comunicazione asincron e ,infine, entity bean che sono gli oggetti persistenti nel DB. Vediamo ora come queste tre componenti vengono implementate.

Il session bean è sostanzialmente una calsse che viene invocata in modo sincrono e che implementa determinate funzioni. I session bean possono essere di due tipi: stateless (non contiene informazioni di stato) o statefull (contiene informazioni di stato). I protocolli utilizzato per invocare le API sono:

- Local interface: il metodo può essere invocato solo localmente
- Remote interface: il metodo può essere invocato con RMI
- Endpoint interface: il metodo può essere invocato con SOAP

I message-driven bean implementano una interfaccia che viene eseguita quando si riceve un messaggio. Alla ricezione di un messaggio vengono estratti i dati contenuti in esso e, in base al loro valore, vengono eseguite delle operazioni.

I bean sono incapsulati da dei container che intercettano tutte le richieste che vengono a fatte ad un bean. Attraverso il container sarà possibile aggiungere una serie di servizi in modo dichiarativo. Ogni vendor mette a disposizione una diversa implementazione di container; le configurazioni del container sono:

- Default behavior definito dal vendor
- Annotations che configurano il comportamento del container e sono definite dallo sviluppatore
- Descrittori che sovrascrivono le annotazioni

I bean hanno un ciclo di vita, questo comporta che i bean implementino dei metodi che verranno chiamati dal container quando ad esempio essi vengono appena creati o stanno per essere distrutti. Ciò comporta la necessità di avere la possibilità di far comunicare il bean con il container e per fare ciò si usa una logica di callback.