Processo e Sviluppo del Software

UniShare

Davide Cozzi
@dlcgold

Indice

Capitolo 1

Introduzione

Questi appunti sono presi a lezione. Per quanto sia stata fatta una revisione è altamente probabile (praticamente certo) che possano contenere errori, sia di stampa che di vero e proprio contenuto. Per eventuali proposte di correzione effettuare una pull request. Link: `https://github.com/dlcgold/Appunti`.
Le immagini presenti in questi appunti sono tratte dalle slides del corso e tutti i diritti delle stesse sono da destinarsi ai docenti del corso stesso.

Capitolo 2

Metodi Agili

I *metodi agili* sono stati definiti per rispondere all'esigenza di dover affrontare lo sviluppo di software in continuo cambiamento. Durante lo sviluppo si hanno vari passaggi:

- comprensione dei prerequisiti
- scoperta di nuovi requisiti o cambiamento dei vecchi

Questa situazione rendeva difficile lo sviluppo secondo il vecchio metodo *waterfall* (a cascata) portando al fallimento di diversi progetti.
I *metodi agili* ammettono che i requisiti cambino in modo "naturale" durante il processo di sviluppo software e per questo assumono un modello di processo *circolare*, con iterazioni della durata di un paio di settimane (figura 2.1). Potenzialmente dopo un'iterazione si può arrivare ad un prodotto che può essere messo "in produzione". Dopo ogni rilascio si raccolgono *feedback* per poter rivalutare i requisiti e migliorare il progetto.
Si hanno quindi aspetti comuni nei metodi agili e nel loro processo:

- enfasi sul team, sulla sua qualità e sulla sua selezione
- il team è *self organizing*, si da importanza ai vari membri del team dato che non esiste un *manager* ma è il team stesso a gestire lo sviluppo
- enfasi al pragmatismo, focalizzandosi su una documentazione efficace evitando di produrre documenti inutili e difficili da mantenere
- enfasi sulla comunicazione diretta, sostituendo i documenti suddetti con meeting e riunioni periodiche

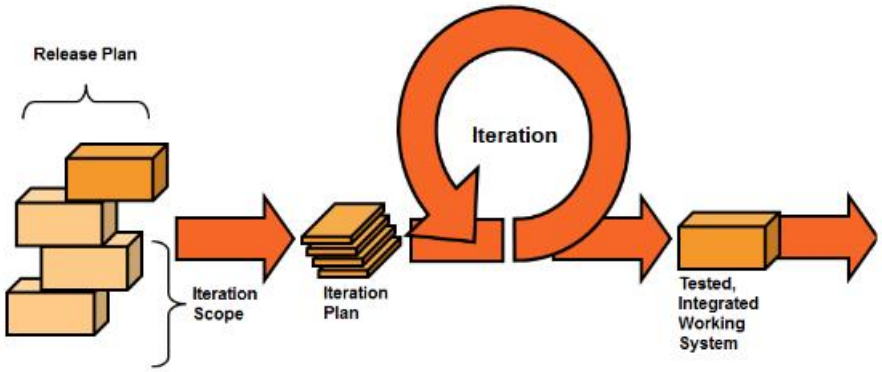


Figura 2.1: Rappresentazione grafica del modello agile. I blocchi a sinistra rappresentano i requisiti (da sviluppare secondo una certa priorità). Lo sviluppo si articola tramite varie iterazioni che porta ogni volta ad aggiungere una parte al prodotto finale (ogni iterazione produce, a partire da un certo requisito, un parte di prodotto di qualità già definitiva, con testing, documentazione etc. . .)

- enfasi sull'idea che nulla sia definitivo: la perfezione non deve essere seguita fin da subito ma saranno gli step a portare al raggiungimento di una perfezione finale (anche dal punto di vista del design)
- enfasi sul controllo costante della qualità del prodotto, anche tramite
  - *continuous testing* grazie al quale un insieme di test viene eseguito in modo automatico dopo ogni modifica
  - *analisi statica e dinamica* del codice al fine di trovare difetti nello stesso
  - *refactoring*

I metodi agili sono molto "elastici" e permettono la facile definizione di nuovi metodi facilmente adattabili al singolo progetto.

2.1 Scrum

Uno dei più famosi, tra i vari *metodi agili*, è **scrum** (figura 2.2).
In questo caso la parte di sviluppo e iterazione prende il nome di *sprint* ed

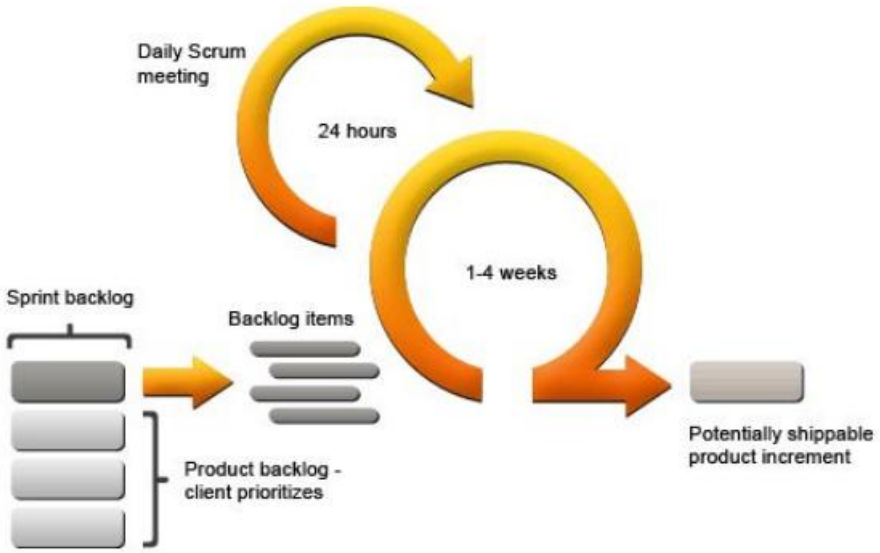


Figura 2.2: Rappresentazione grafica del processo scrum

ha una durata variabile tra una e quattro settimane, per avere un rilascio frequente e una veloce raccolta di feedback. I requisiti sono raccolti nel cosiddetto *product backlog*, con priorità basata sulla base delle indicazioni del committente. Ad ogni *sprint* si estrae dal *product backlog* lo *sprint backlog*, ovvero il requisito (o i requisiti) da implementare nello *sprint*. Lo *sprint backlog* viene analizzato nel dettaglio producendo i vari *backlog items*, ovvero le singole funzionalità che verranno implementate nello *sprint*. Si ottiene quindi di volta in volta un pezzo di prodotto finale, testato e documentato. Durante le settimane di *sprint* si effettua anche un meeting giornaliero utile per mantenere alti i livelli di comunicazione e visibilità dello sviluppo. Durante il meeting ogni sviluppatore risponde a tre domande:

1. Cosa è stato fatto dall'ultimo meeting?
2. Cosa farai fino al prossimo meeting?
3. Quali sono le difficoltà incontrate?

L'ultimo punto permette la cooperazione tra *team members*, consci di cosa ciascuno stia facendo.
Durante il processo *scrum* si hanno quindi tre ruoli:

1. il **product owner**, il committente che partecipa tramite feedback e definizione dei requisiti

2. il **team** che sviluppa
3. lo **scrum master** che controlla la correttezza di svolgimento del processo scrum

Essi collaborano nelle varie fasi:

- product owner e team collaborano nella definzione dei backlog e nella loro selezione ad inizio *sprint*
- durante lo *sprint* lavora solo il team
- nello studio del risultato collaborano tutti coloro che hanno un interesse diretto nel progetto (team, product owner e stakeholders)

Lo scrum master interagisce in ogni fase, fase che viene comunque guidata tramite meeting:

- **sprint planning meeting**, ad inizio *sprint*
- **daily scrum meeting**, il meeting giornaliero
- **sprint review meeting**, in uscita dallo *sprint* per lo studio dei risultati
- **sprint retrospective meeting**, in uscita dallo *sprint* per lo studio tra i membri del team di eventuali migliorie al processo e allo sviluppo del prodotto (anche dal punto di vista delle tecniche e delle tecnologie)

2.2 Extreme Programming

Un altro tipo di metodo agile è l'*extreme programming* (figura 2.3), ormai poco usato. I requisiti prendono i nomi di *stories*, delle narrazioni in cui l'attore (futuro utente del sistema) cerca di svolgere un compito. Vengono scelti quindi *stories* per la prossima iterazione, dove si hanno testing e revisione continua. Le release di ogni iterazione vengono catalogate per importanza (con anche la solita collezione di feedback). In *extreme programming* si hanno davvero molte pratiche:

- The Planning Game
- Short Releases
- Metaphor

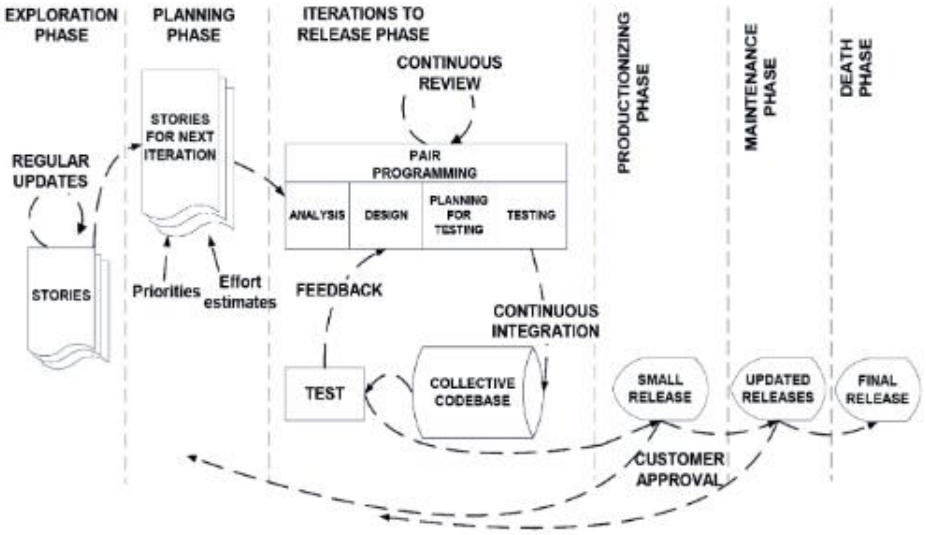


Figura 2.3: Rappresentazione grafica dell'extreme programming

- Simple Design
- Refactoring
- Test-First Design
- Pair Programming
- Collective Ownership (codice condiviso da tutti senza alcun owner, con tutti in grado di effettuare modifiche)
- Continuous Integration
- 40-Hour Week
- On-Site Customer
- Coding Standard (per avere il collective ownership avendo un solo standard di codifica comune a tutti)
- Open workspace

Capitolo 3

DevOps

Nel tempo si è passato dal *modello waterfall* (dove tutto era definito ad inizio progetto) al *modello agile*. Negli ultimi tempi si è sviluppato un altro metodo, chiamato **DevOps**, dove anche la parte di *operation* deve essere agile: il rilascio in produzione e il *deployment* devono essere agili quanto lo sviluppo. Amazon, Netflix, Facebook e molte altre società già adottano le pratiche *DevOps.*
Nel *DevOps* i team di sviluppo e operation sono indipendenti tra loro, diminuendo il costo di impegno necessario al team di sviluppo per la parte di deployment (che viene resa anche più sicura grazie alla diminuzione dell'intervento umano in favore di automazioni). Inoltre, avendo i due team dei tempi di lavoro diversi, si riesce a prevenire ritardi causati dalla non organicità delle operazioni.
DevOps promuove la collaborazione tra di due team al fine di ottenere una sorta di team unico che curi sia sviluppo che operation.
DevOps include quindi diversi processi che vengono automatizzati:

- Continuous Development
- Continuous Integration
- Continuous Testing
- Continuous Deployment (anche con tecnologie di virtualizzazione e con l'uso di *container* in *ambiente cloud*)
- Continuous Monitoring

Con DevOps il feedback arriva in primis dal software (i tool di *monitor*) inoltre il focus viene spostato sui processi automatici.
Nel DevOps si introducono nuovi ruoli:

- il **DevOps Evangelist**, simile allo *scrum master*, supervisiona l'intero processo di DevOps
- l'**automation expert**, dedicato a curare gli aspetti di automatismo
- un **Security Engineer**
- un **Software Developer**, nonché **Tester**
- un **Quality Assurance** che verifica la qualità del prodotto rispetto ai requisiti
- un **Code Release Manager** che si occupa sull'infrastruttura e sul deploy della release

Il DevOps (figura 3.1) si basa su sei principi base:

1. **Customer-Centric Action**, ovvero il committente è al centro dell'azione
2. **End-To-End Responsibility**, ovvero il team gestisce interamente il prodotto, avendone responsabilità totale
3. **Continuous Improvement**, ovvero cercare continuamente di migliorare senza sprechi il prodotto finale e i servizi
4. **Automate everything**, ovvero cercare di automatizzare l'intera infrastruttura di processo, dalle attività di testing e integrazione fino ad arrivare alla costruzione della release e del deployment
5. **Work as one team**, ovvero unificare tutti gli aspetti sotto un unico team o comunque con due team che collaborano fortemente come se fossero uno
6. **Monitor and test everything**, ovvero testare e monitorare costantemente il prodotto

Il quarto e il sesto punto sono i due punti tecnici principali.

Figura 3.1: Rappresentazione famosa del *lifecycle* di DevOps, con le due parti, con le relative parti fondamentali, di sviluppo e operation distinte dai colori ma unite in un singolo metodo unico

3.1 Build, Test e Release

Partiamo dalla parte "dev" di DevOps.
Bisogna pensare a questi step in ottica di automatismo vicina al DevOps.
Innanzitutto bisogna introdurre i sistemi di **version control**, in primis **Git**, un sistema di version control **distribuito**, dove ogni utente ha una copia della repository (con la storia dei cambiamenti), con la quale interagisce tramite *commit* e *update*. Esiste poi una repository lato server per permettere di condividere i vari cambiamenti tramite sincronizzazione.
Git permette anche lo sviluppo *multi-branch* per permettere di lavorare su più branch, pensando allo sviluppo separato dove ogni branch alla fine può essere unito (*merge*) con quello principale (solitamente chiamato *master*). Un altro branch standard è quello di sviluppo, detto *develop*. Un altro ancora è quello detto *hotfix*, per le modifiche più urgenti, che vive in uno stadio intermedio tra i due sopracitati, prima ancora del branch *develop*. Volendo ciascuna feature può essere creata in un branch dedicato. Si procede con le versioni "merge-ndo" quando necessario, "step by step" fino ad un branch *release* per poi andare dopo il testing su *master*. Sul branch *release* possono essere effettuati fix che poi verranno riportati anche in *develop*.
Lo sviluppo su multi-branch si collega alle operazioni di verifica che possono essere attivate automaticamente a seconda dell'evoluzione del codice su ogni branch. Avendo ogni branch una precisa semantica possiamo definire precise attività di verifica, corrispondenti a *pipelines* precise, solitamente innescate da un *push* di codice su un certo branch, in modo sia automatico che manuale. Le pipeline vengono attivate in fase di test di un componente, in fase di creazione di un sottosistema, di assembramento di un sistema intero o di

deployment in produzione. Si hanno quindi quattro fasi:

1. component phase
2. subsystem phase
3. system phase
4. production phase

Spesso le pipelines sono usate come *quality gates* per valutare se un push può essere accettato in un certo branch. Una pipeline può essere anche regolata temporalmente, in modo che avvenga solo ad un certo momento della giornata.

Component phase e Subsystem phase

Dove il fuoco è sulla più piccola unità testabile che viene aggiornata (una classe, un metodo etc...) che non può essere eseguita senza l'intero sistema. In tal caso si può fare:

- code review
- unit testing
- static code analysis

Un cambiamento può anche essere testato nell'ambito del sottosistema di cui fa parte, in tal caso si hanno anche check di prestazioni e sicurezza. Il servizio però potrebbe essere da testare in isolamento rispetto ad altri servizi, usando quindi dei *mocks* o degli *stubs*, ovvero creando degli alter ego dei servizi mancanti in modo che il servizio da testare possa funzionare.

System phase

In questo caso si testa l'intero sistema che viene "deployato" in ambiente di test. Si hanno:

- integration tests
- performance tests
- security tests

Tutti test che richiedono l'interezza del sistema e sono spesso molto dispendiosi e quindi bisogna regolare la frequenza di tali test in molti casi (sfruttando ad esempio la notte).

Production phase

Questa fase è legata alla necessità di creare gli artefatti che andranno direttamente "sul campo", ovvero il deployment in produzione. In tale fase potrebbe essere necessario creare container o macchine virtuali. Si hanno dei check molto veloci sugli artefatti finali (che non siano per esempio corretti), dando per assodato che la qualità del codice sia già stata testata. Si hanno quindi strategie anche di *deployment incrementale*, per cui esistono più versioni del software contemporaneamente con diversa accessibilità per gli utenti finali (accessibilità che viene man mano scalata). In tal caso si usano anche vari tool di monitor. Si hanno anche eventualmente tecniche di *zero downtime* (dove il software non è mai in uno stato *unstable*).

Fasi diverse corrispondono a branch diversi

3.2 Deploy, Operate e Monitor

Studiamo ora la parte "Ops" di DevOps.
Si studia l'evoluzione automatica del software da una versione all'altra in produzione. Avanzare di versione in modo *naive* e istantaneo è troppo rischioso (qualora la nuova versione fosse corrotta non si avrebbe controllo sull'update) e quindi spesso non attuabile (spesso anche per ragioni tecniche). Si ha quindi un insieme di tecniche che si basano in primis sull'*evoluzione incrementale*. Tali tecniche si distinguono in base alla dimensione su cui sono incrementati:

- **Incremental wrt users:** *Dark launching, Canary releases (and User Experimentation)*, ovvero legata agli utenti esposti alla nuova release
- **Incremental wrt requests:** *Gradual upgrades/rollout*, ovvero legata alle richieste per la nuova release
- **Incremental wrt components/replicas:** *Rolling upgrade*, incentrata sulle componenti che vengono aggiornate
- **Non-incremental with backups:** *Green/blue deployment, Rainbow deployment*, non incrementali ma che offrono comunque un backup di sicurezza

Tali schemi possono essere usati in un contesto DevOps.
Per studiare la prima tipologia (*Incremental wrt users*) abbiamo:

- **Dark launching** che arriva dal mondo di Facebook dal 2011. In tale schema l'update è esposto solo ad una parte della popolazione, per la quale viene effettuato il deployment per studiare gli effetti (tramite continuous monitoring) ed eventuali modifiche e migliorie al software, che infine verrà deployato per il resto della popolazione in modo comunque incrementale fino a che l'intera popolazione godrà della feature. Spesso usata per front-end
- **Canary releases**, che studia l'impatto di update relativi al back-end

Tali schemi spesso sono usati di pari passo per le varie sezioni del software, nonché possono essere usati in modo intercambiabile.
Collegato a questi schemi si ha l'approccio basato sull'**user experimentation**, che non è un reale schema di gestione dell'evoluzione del software ma è comunque correlato agli schemi sopra descritti. In questo approccio si studiano diverse varianti del sistema e il loro impatto esponendole agli utenti (perlomeno ad una sottoparte degli stessi in modo incrementale), cercando di capire per l'utente cosa sia meglio e come (si ispira alla *sperimentazione scientifico*). Si hanno quindi più release diverse, per parti di popolazione comparabili, tra le quali si sceglierà la migliore.
Per la seconda tipologia (*Incremental wrt requests*) si ha una divisione a seconda delle richieste fatte dagli utenti (esempio un momento d'uso diverso porta all'uso di componenti diverse), detto *gradual rollout*. Si ha quindi un *load balancer* che permette la coesistenza di due versioni, una nuova e una vecchia, dello stesso servizio. In modo graduale, partendo da pochissime, si passano le richieste alla versione nuova per poter studiare e testare la nuova versione (in caso di problemi il load balancer dirotterà tutte le richieste alla vecchia versione). Alla fine tutto il traffico sarà diretto verso la nuova versione, mentre la vecchia verrà dismessa.
Per la terza tipologia (*Incremental wrt components/replicas*), si ha lo schema del *rolling upgrade*, dove l'upgrade non riguarda un singolo upgrade ma tanti componenti di un sistema distribuito, verificando efficacia di ogni singolo update tramite il continuous monitoring prima di effettuare l'upgrade di un'altra componente. La stessa idea su applica anche a diverse versioni dello stesso prodotto, aggiornandone una prima e poi le altre progressivamente. Le nuove versioni delle componenti "upgradate" devono essere compatibili con quelle ancora prive di upgrade.
Per la quarta tipologia (*Non-incremental with backups*) si ha il **blue/green deployment**, dove vengono isolate due copie della stessa infrastruttura (anche hardware), dove una ospita la versione nuove l'altra la vecchia. Un router ridireziona le richieste degli utenti verso le due unità e quella che ospita la

nuova versione subirà le solite operazioni di test che, se superate, porteranno il router a direzionare verso quella unità, ignorando la vecchia. Se ci sono problemi si fa rollback alla vecchia unità che rimane come backup. Questo schema può essere generalizzato nel **rainbow deployment** dove il momento di coesistenza tra le due versioni (se non più versioni) viene prolungato al fine che vecchie richieste che richiedono una lunga elaborazione vengano elaborate dall'unità vecchia mentre le nuove dall'unità nuova.
In ogni caso le applicazioni devono essere costruite per supportare tutti questi schemi di deployment (a causa di stati delle applicazioni, backward compatibility etc...)

3.2.1 Deployable units

Il caso più tipico in merito alle unità dove fare deployment è il mondo del **cloud**, con **unità virtualizzate e virtual machine (VM)**, dove magari ogni servizio vive in una diversa VM. Si hanno diversi casi in merito a questo tipo di deployment:

- **cloud** basato su **VMs**, dove si ha un'infrastruttura gestita dal cloud provider che gestisce l'hardware e l'hypervisor. Ogni VM, che sono le nostre unità di deployment, ha un sistema operativo arbitrario che lavora con l'hardware mostrato dall'hypervisor. Ogni VM avrà una o più applicazioni e fare deployment porterà all'update di una o più VM. In alcuni casi si fa deployment di intere VM e in altri si modifica il software di una VM già in esecuzione. L'ambiente cloud solitamente è **multi-tenant** (ovvero su una piattaforma unica di un provider si hanno più VM di diverse organizzazioni). Una VM è grossa in quanto contiene un sistema operativo intero e la loro gestione può quindi essere difficoltosa

- **cloud** basato su **containers** che risolvono il problema della grandezza delle VM. In questo caso lo schema è il medesimo ma si ha un **container engine** al posto dell'hypervisor e ogni container non contiene l'intero sistema operativo ma solo il minimo necessario al funzionamento dell'applicazione (il sistema operativo viene condiviso dalla macchina sottostante, riducendo il volume dei singoli containers). In questo caso lo schema di update spesso consiste nel distruggere e ricreare i singoli containers. Anche qui si ha un contesto *multi-tenant*

- **bare metal**, dove i provider offrono direttamente risorse hardware, guadagnando prestazioni ma aumentano anche i costi eco-

nomici , che vengono comunque gestite dal cloud provider. Non si ha virtualizzazione ma accesso diretto alle risorse su cui fare deployment. Questa è una soluzione tipicamente **single-tenant** (sulla macchina gira il software di una sola organizzazione)

- **server dedicati**, un metodo ormai superato con difficoltà causate dall'uso di script, shell e connessione *ftp* completamente autogestiti dall'organizzazione e non da un provider

Il deployment "stile cloud" non è comunque l'unico possibile. Un esempio quotidiano è il deployment di app mobile sui vari store, dove il back-end probabilmente sarà gestito come sopra spiegato mentre l'app in sé viene rilasciata negli store e sarà l'utente finale a fare il deployment installando l'app sul proprio device. Le forme di monitoraggio e di feedback sono spesso diverse e provengono dagli utenti finali stessi.

3.2.2 Monitor

In ambiente cloud ci sono tante soluzioni per il **monitoring**, ad esempio lo **stack di ELK**, formato da:

- **Elasticsearch**
- **Logstash**
- **Kibana**

I dati, ad esempio log o metriche d'uso hardware, vengono raccolti e passano da **Logstash**, finendo in un database, per la memorizzazione di *time series* (serie temporali) di dati (questo in primis per le metriche d'uso che per i log), gestito da **Elasticsearch** e venendo visualizzati da una dashboard grafica, gestita da **Kibana**. Si ha quindi un ambiente di **continuous monitoring**.

3.2.3 DevOps tools

Ogni step del DevOps è gestito tramite moderne tecnologie e tools , con varie alternative per ogni fase (per questo servono figure esperte per ogni step). Vediamo qualche esempio (in ottica più spinta ad un progetto in Java):

- code: Git, Svn, Jira, Eclipse
- build: Apache Ant, Maven, Gradle
- test: JUnit

- release: Jenkins, Bamboo
- deploy: Pupper, Chef, Ansible, SaltStack
- monitor: New Relic, Sensu, Splunk, Nagios

Capitolo 4

Risk management

Partiamo da un semplice esempio:

Esempio 1. *Durante lo sviluppo di un progetto software l'unico dev, insoddisfatto del salario, che conosceva un modulo di importanza critica lascia la società, rallentando in modo serio lo sviluppo del progetto (nonché aumentandone i costi, nel tentativo di cambiare il modulo conosciuto solo dal dev) fino a farlo uscire troppo in ritardo rispetto, ad esempio, alle opportunità di marketing a cui puntava.*

Un esempio del genere non è così raro e il **risk management (*gestione dei rischi*)** si occupa di prevenire questo tipo di complicazioni e fallimenti.

Definizione 1. *Il **risk management** è la disciplina che si occupa di identificare, gestire e potenzialmente eliminate i rischi prima che questi diventino una "minaccia" per il successo del progetto (o anche per eventuali situazioni di revisione del progetto stesso).*

Dobbiamo però dare qualche definizione:

Definizione 2. *Definiamo **rischio** come la possibilità che ci sia un danno.*

Bisogna cercare di prevenire n evento che può portare ad un danno serio (e tanto più è serio il danno tanto è alto il rischio)

Definizione 3. *Definiamo **risk exposure**, che è una grandezza (calcolabile), per calcolare quanto un progetto sia esposto ad un rischio. Viene calcolato come:*

$$RE = P(UO) \cdot L(UO)$$

dove:

- $P(UO)$ *è la probabilità di un* unsatisfactory outcome*, ovvero la probabilità che effettivamente un danno (o comunque un risultato non soddisfacente) sia prodotto*
- $L(UO)$ *è l'entità del danno stesso, ovvero è la perdita per le parti interessate se il risultato non è soddisfacente*

Tanto più un rischio è probabile e tanto più il rischio crea un danno tanto cresce il ***risk exposure****.*

Definizione 4. *Definiamo* ***outcome unsatisfactory (risultato non soddisfacente)*** *come un risultato non positivo che riguarda diverse aree:*

- *l'area riguardante l'esperienza degli utenti, con un progetto che presenta le funzionalità sbagliate, una UI carente, problemi di prestazioni o di affidabilità etc.... In questo caso se i problemi sono gravi si hanno alti rischi, come l'utenza che smette di usare il prodotto, portando al fallimento del prodotto, o anche a conseguenze legali*
- *l'area riguardante i dev, con rischi che per esempio si ritrovano superamento del budget e prolungamenti delle deadlines*
- *l'area riguardante i manutentori, con rischi che per esempio si ritrovano nella qualità bassa di software e hardware*

Se abbiamo un rischio che produce un *outcome unsatisfactory* il primo elemento su cui soffermarsi è lo studio degli eventi che abilitano il rischio, detti **risk triggers**, per evitare che avvengano (comportando di conseguenza che il rischio diventi realtà comportando un *outcome unsatisfactory*, come nell'esempio 1). Sempre in base all'esempio 1 si potrebbe pensare di non sussumere un solo dev con una certa conoscenza o comunque di alzare la paga, per evitare di attivare i *risk triggers*.
Abbiamo quindi due principali **classi di rischio**:

1. **process-related risks**, rappresenti rischi con impatto negativo sul processo e sugli obbiettivi di sviluppo, come ritardi o superamento di costi
2. **product-related risks**, rappresenti rischi con impatto sul prodotto e su obbiettivi del sistema funzionali o meno, come fallimenti riguardanti la qualità del prodotto (sicurezza, prestazioni etc...) o la distribuzione dello stesso

Entrambe le classi possono portare al fallimento del progetto e quindi vanno gestite entrambe.
Bisogna quindi imparare a gestire i rischi. Si hanno principalmente due fasi:

1. una prima fase riguardante il **risk assessment (*valutazione del rischio*)**. In questa fase si hanno:
   - **risk identification**, ovvero l'identificazione dei rischi
   - **risk analysis**, ovvero l'analisi dei rischi identificati (tramite calcolo del *risk exposure* e studio dei triggers)
   - **risk prioritization**, ovvero la prioritizzazione dei rischi analizzati, per focalizzarsi sui più pericolosi per poi scalare ai meno pericolosi

   Alla fine si produce una lista ordinata sulla pericolosità dei rischi
2. una seconda fase riguardante il **risk control** e consiste, in primis, sul **risk management planning**, producendo piani di controllo di due tipi:
   (a) **piani di management**, per la gestione del rischio prima che si verifichino
   (b) **piani di contingency**, per il contenimento di rischi divenuti realtà qualora il *piano di management* fallisca, sapendo cosa fare a priori in caso di emergenza

   Si hanno quindi due sotto-fasi per i due tipi di piani:
   (a) **risk monitoring**
   (b) **risk resolution**

Queste due fasi vengono ciclicamente ripetute durante il ciclo di vita dello sviluppo di un software.

4.1 Risk identification

Si studia come identificare i rischi.
È un'operazione complessa legata alla competenza degli analisti. Un modo comune di farlo è usando delle **check-list**, liste che includono un insieme di rischi plausibili comuni a molti progetti. L'analista scorre tale lista cercando rischi che possono essere applicati al progetto in analisi.

Esempio 2. *Una lista di 10 punti comoda nell'ambito dello sviluppo include:*

1. *problemi con il personale*
2. *budget e deadlines irrealistici*
3. *sviluppo delle funzionalità sbagliate*
4. *sviluppo della UI sbagliata*
5. *gold-plating, ovvero aggiungere più funzionalità del necessario o usare tecnologie troppo avanzate*
6. *continue modifiche ai requisiti*
7. *problemi in componenti software fornite da esterni*
8. *problemi con task svolti da esterni*
9. *problemi con le prestazioni real-time*
10. *voler superare i limiti delle tecnologie attuali oltre le regole e le capacità della computer science*

Alcuni rischi, in astratto, possono verificarsi sempre ma va preso in considerazione solo per **motivi specifici identificabili** nel mio progetto.
Si hanno altri metodi per identificare i rischi:

- riunioni di confronto, *brainstorming* e *workshop*
- confronto con altre organizzazioni e con altri prodotti

4.2 Risk analysis

Per analizzare i rischi si sfrutta esperienza delle reali probabilità che un rischio diventi realtà. Anche in questo si hanno degli schemi su cui basarsi, come *modelli di stima dei costi*, *modelli delle prestazioni*, etc... basati su *simulazioni*, *prototipi*, *analogie con altri progetti* e *check-list* (con stime di probabilità e danno).
Le stime sono molto specifiche sul singolo progetto.
L'analisi dei rischi può anche comportare lo studio delle decisioni da prendere al fine di minimizzare il **risk exposure**, scegliendo o meno tra varie opzioni, scegliendo in modo "guidato" dai rischi. A tal fine si usano i *decision tree* con la radice che rappresenta il problema (un esempio si ha in figura

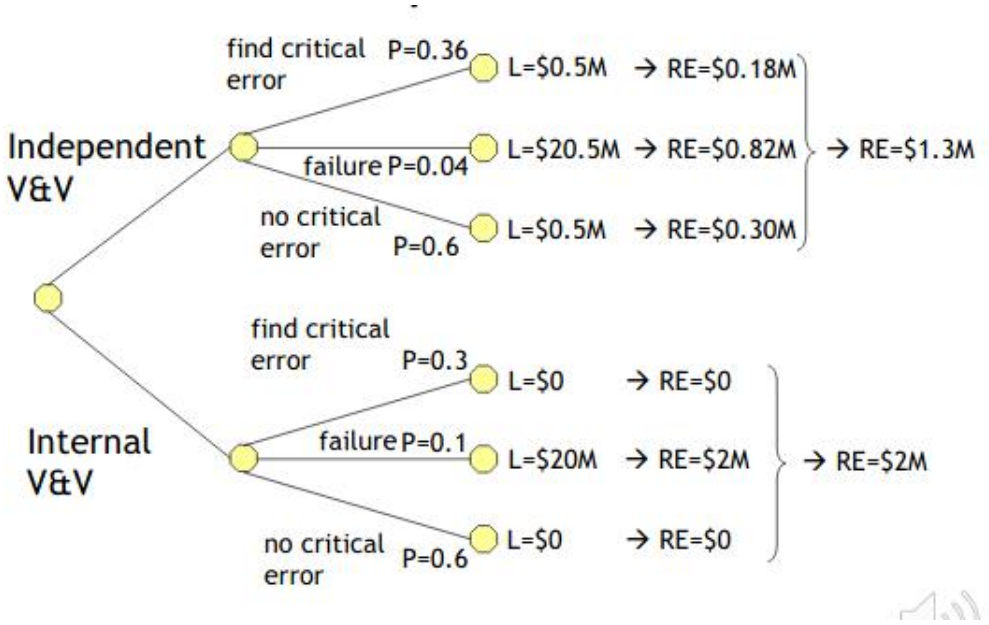


Figura 4.1: Esempio di decision tree

4.1). Si hanno di volta in volta i vari scenari, con le stime di probabilità di trovare un errore critico, di fallimento, di non trovare errori critici etc.... Tali probabilità verranno usate per il calcolo del *risk exposure* insieme ad un quantificatore di $L(UO)$ spesso pari all'effettivo costo che conseguirebbe al risultato ottenuto. Infine i vari *risk exposure* di ogni caso vengono sommati per ottenere il *risk exposure* finale. Si può fare un'**analisi di sensitività** cambiando le percentuali o i costi al fine di ottenere un certo risultato, in modo da capire come si dovrebbe comportare.
Ragionando sulle cause dei rischi usiamo il cosiddetto **risk tree** (esempio in figura 4.2). Questo albero ha come radice il rischio. Ogni nodo, detto **failure node**, è un evento che si può scomporre "via via" in altri eventi, fino alle foglie. La scomposizione è guidata da due tipi di **nodi link**:

1. **and-node**, dove i figli di tali nodi sono eventi legati dal un *and*
2. **or-node**, dove i figli di tali nodi sono eventi legati dal un *or*

Nodi And/Or vengono rappresentati tramite i simboli delle porte logiche.
Dato un *risk tree* cerco le combinazioni di eventi atomici che possono portare al rischio. Per farlo si esegue la **cut-set tree derivation**, ovvero, partendo dalla radice, si riporta in ogni nodo la combinazione di eventi che possono produrre il fallimento e si vanno a calcolare le varie combinazioni degli eventi

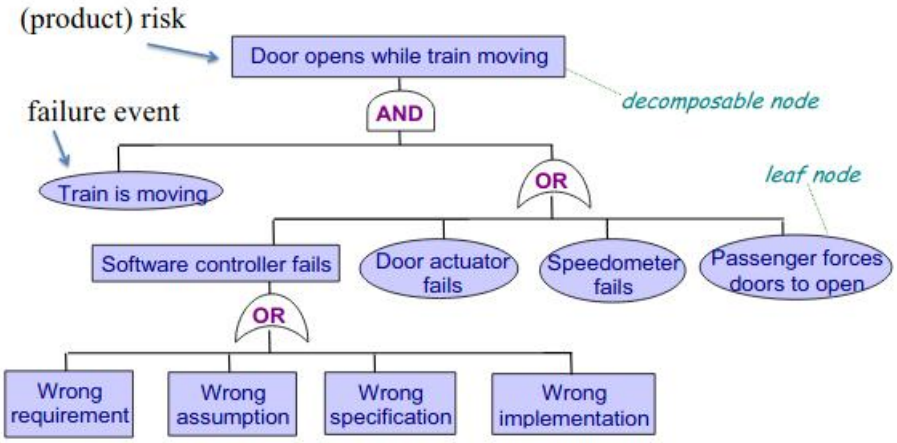


Figura 4.2: Esempio di risk tree

foglia. Praticamente si deriva un insieme di eventi non scomponibili sulle combinazioni dell'*and*.

4.3 Risk prioritization

Bisogna capire quali rischi sono più "rischiosi" degli altri. Per farlo si pongono i valori di $P(UO)$ e $L(UO)$ in un range, per esempio, da 1 a 10, ricalcolando il *risk exposure*. Una volta fatto si lavora in base al *risk-exposure* (che può anche essere un intervallo se le probabilità o i costi sono in un certo intervallo). Si procede plottando i dati su un piano con $P(UO)$ sull'asse delle y e $L(UO)$ su quello delle x e facendo lo *scatterplot* degli eventi (ponendoli quindi come punti in base a $P(UO)$ e $L(UO)$). Qualora i valori siano in un range si rappresentano con un segmento tra i due valori limite di *risk exposure*. Con delle curve posso identificare zone di rischio diverse in base al *risk exposure* per poter catalogare gli eventi. Solitamente alla fase di *risk control* passano una decina di eventi.

4.4 Risk control

Bisogna quindi capire come gestire i rischi.
Per ogni rischio bisogna definire e documentare un piano specifico indicante:

- cosa si sta gestendo
- come mitigare il rischio e quando farlo

- di chi è la responsabilità
- come approcciarsi al rischio
- il costo dell'approccio al rischio

Anche in questo caso ci vengono incontro liste e *check-list* con le tecniche di *risk management* più comuni in base al rischio specifico.
Ci sono comunque strategie generali:

- lavorare sulla probabilità che il rischio avvenga, sulla probabilità dei triggers. Bisogna capire come diminuire la probabilità
- lavorare, nel limite del possibile, sull'eliminazione stessa del rischio
- lavorare sulla riduzione della probabilità di avere conseguenze al danno, non viene quindi ridotto il rischio
- lavorare, nel limite del possibile, sull'eliminazione stessa del danno conseguente al rischio
- lavorare sul mitigare le conseguenze di un rischio, diminuendo l'entità del danno

Bisogna anche studiare le contromisure, da scegliere e attivare in base alla situazione. Si hanno due metodi quantitativi principali per ragionare quantitativamente sulle contromisure:

1. **risk-reduction leverage**, dove si calcola quanto una certa contromisura può ridurre un certo rischio, secondo la seguente formula:
$$RRL(r, cm) = \frac{RE(r) - RE\left(\frac{r}{cm}\right)}{cost(cm)}$$
dove r rappresenta il rischio, cm la contromisura e $\frac{r}{cm}$ la contromisura cm applicata al rischio r. Calcolo quindi la differenza di *risk exposure* avendo e non avendo la contromisura e la divido per il costo della contromisura.
La miglior contromisura è quella con il RRL maggiore, avendo minor costo e maggior efficacia dal punto di vista del *risk exposure*

2. **defect detection prevention**, più elaborato del primo è stato sviluppato dalla NASA. Questo metodo confronta le varie contromisure, confrontando anche gli obiettivi del progetto, in modo

quantitativo facendo un confronto indiretto, producendo matrici in cui si ragiona in modo indipendente sulle singole contromisure e sui singoli rischi ma confrontando anche in modo multiplo.

Si ha un ciclo a tre step:

(a) elaborare la matrice di impatto dei rischi, detta **risk impact matrix**. Questa matrice calcola l'impatto dei rischi sugli obiettivi del progetto. I valori della matrice, ovvero $impact(r, obj)$ (che ha per colonne i rischi r e righe gli obiettivi del progetto obj) variano da 0, nessun impatto, a 1, completa perdita di soddisfazione (e totale non raggiungimento dell'obiettivo indicato). Ogni rischio viene accompagnato dalla probabilità P che accada. Ogni obiettivo è accompagnato dal **peso** W che ha nel progetto (la somma di tutti i pesi è pari a 1). Si possono calcolare altri valori di sintesi. In primis la **criticità** di un rischio rispetto a tutti gli obiettivi indicati:

$$criticality(r) = P(r) \cdot \sum_{obj}(impact(r, obj) \cdot W(obj))$$

La criticità sale se sale l'impatto e se sale la probabilità del rischio.
Un altro dato è la **perdita di raggiungimento** di un obiettivo qualora tutti i rischi si verificassero:

$$loss(obj) = W(obj) \cdot \sum_{r}(impact(r, obj) \cdot P(r))$$

(b) elaborare contromisure efficaci per la matrice. In questa fase si usa il fattore di criticità del rischio. Viene prodotta una nuova matrice con colonne pari ai rischi (con probabilità e criticità) e righe pari alle contromisure. I valori saranno le riduzioni di rischio di una contromisura cm sul rischio r ($reduction(cm, r)$). La riduzione va da 0, nessuna riduzione, a 1, rischio eliminato. Si possono calcolare altri valori di sintesi. Possiamo calcolare la **combineReduction**, che ci dice quanto un rischio viene ridotto se tutte le contromisure sono attivate:

$$(combineReduction(r) = 1 - \prod_{cm}(1 - reduction(cm, r))$$

Un altro valore è l'**overallEffect**, ovvero l'effetto di ogni contromisura sull'insieme dei rischi considerato:

$$overalleffect(cm) = \sum_{r}(reduction(cm, r) \cdot criticality(r))$$

si avrà effetto maggior riducendo rischi molto critici

(c) determinare il bilanciamento migliore tra riduzione dei rischi e costo delle contromisure. Bisogna considerare anche il costo di ogni contromisure e quindi si fa il rapporto tra effetto di ciascuna contromisura e il suo costo e scegliendo il migliore

Analizzando il *contingency plan* viene attuato qualora il rischio si traduca in realtà.
I passa quindi al **risk monitoring/resolution**. Queste due parti sono tra loro integrate. I rischi vanno monitorati e occorrenza vanno risolti il prima possibile. Tutte queste attività sono costose e si lavora su un insieme limitato di rischi, una decina.

Capitolo 5

Capability Maturity Model Integration

Il **Capability Maturity Model Integration (*CMMI*)** che è un "programma" di formazione e valutazione per il miglioramento a livello di processo gestito dal *CMMI Institute*.
Bisogna prima introdurre il concetto di **maturità dei processi**. Questa nozione è abbastanza intuitiva. La probabilità di portare a termine un progetto dipende dalla *maturità del progetto* e la maturità dipende dal grado di controllo che si ha sulle azioni che si vanno a svolgere per realizzare il progetto. Si ha quindi che:

- il progetto è **immaturo** quando le azioni legate allo sviluppo non sono ben definite o ben controllate e quindi i dev hanno troppa libertà che rischia di degenerare in una sorta di anarchia nel controllo del processo di sviluppo, alzando la probabilità di fallimento
- il progetto è **maturo** quando le attività svolte sono ben definite, chiare a tutti i partecipanti e ben controllate. Si ha quindi un modo per osservare quanto si sta svolgendo e verificare che sia come pianificato, alzando le probabilità di successo e riducendo quelle di fallimento

Risulta quindi essenziale ragionare sulla *maturità del processo*.
La **maturità del processo** è definita tramite un insiemi di livelli di maturità con associate metriche per gestire i processi, questo è detto **Capability Maturity Model (*CMM*)**. In altri termini il modello CMM è una collezione dettagliata di *best practices* che aiutano le organizzazioni a migliorare e governare tutti gli aspetti relativi al processo di sviluppo, dalla gestione dei rischi al testing, dal design al project management etc. . ..

Un processo migliore porta ad un prodotto migliore.

La storia dei CMMs parte dagli anni novanta e porta alle versione di CMMI del 2018 che andremo ad analizzare.
Analizziamo quindi nel dettaglio il modello:

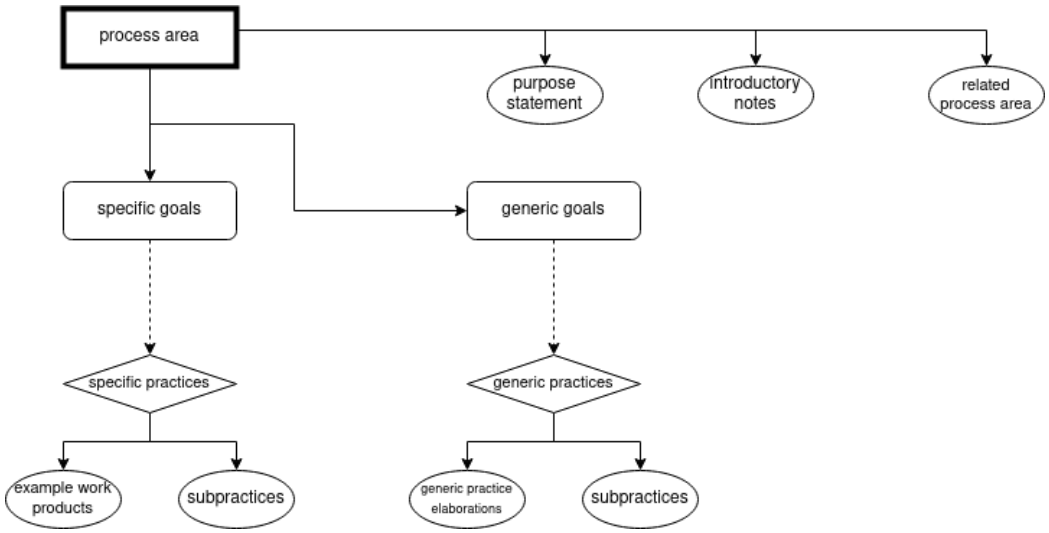


Figura 5.1: Diagramma dei componenti del CMMI, secondo l'organizzazione standard. Nei rettangoli (tranne *process area*) abbiamo i *required*, nei rombi gli *expected* e negli ovali gli *informative*

All'interno del diagramma (che rappresenta letteralmente un documento) notiamo:

- **process area**, che racchiude al suo interno una collezione di pratiche organizzate secondo obiettivi e riguarda una certa area del processo. Nel CMMI abbiamo 22 diverse *process area*, tra cui *configuration management*, *project planning*, *risk management* etc... Nel diagramma si ha lo studio di una generica *process area*
- ciascuna **process area** ha:
  - un **purpose statement**, che descrive lo scopo finale della *process area* stessa
  - un **introductory notes**, con nel note introduttive che descrivano i principali concetti della *process area*
  - un **related process area**, se utile, con la lista delle altre *process area* correlate a quella corrente

- le **process area** si dividono in due *tipologie di obiettivi*:

    1. **specific goals**, ovvero gli obiettivi specifici della singola *process area* in questione. Questi obiettivi caratterizzano la *process area*
    2. **generic goals**, ovvero gli obiettivi comuni a **tutte** le **process area**. Questi obiettivi rappresentano quanto la *process area* sia ben integrata e definita nel contesto del processo ma questi criteri sono generali

- all'interno di ogni **specific goals** (per questo la freccia tratteggiata) abbiamo una serie di **specific practices**, ovvero quelle azioni che se svolte permettono di raggiungere quell'obiettivo specifico e, a loro volta, tali pratiche sono organizzate in:

    – **example work product**, ovvero elenchi di esempi di prodotti che possono essere generati attraverso l'adempimento delle pratiche

    – **subpractices**, ovvero pratiche di "grana più fine"

- all'interno di ogni **generic goals** (per questo la freccia tratteggiata) abbiamo una serie di **generic practices**, comuni a tutti, con le pratiche che devono essere svolte per gestire positivamente una qualsiasi *process area* e, a loro volta, tali pratiche sono organizzate in:

    – **generic practices elaborations**, ovvero ulteriori informazioni di dettaglio per la singola pratica

    – **subpractices**, ovvero pratiche di "grana più fine"

    Tra i principali *generic goals (GG)* abbiamo:

    – *GG1*: raggiungere i *specific goals*, tramite l'esecuzione delle *specific practices*

    – *GG2*: "ufficializzare" un *managed process*, tramite training del personale, pianificazione del processo, controllo dei *work product* etc...

    – *GG3*: "ufficializzare" un *defined process*, tramite la definzione rigorosa del progetto e la raccolta di esperienze legate al processo

Per capire quanto un processo software è organizzato secondo questo standard bisogna "mappare" quali goals e quali pratiche si stanno perseguendo e seguendo e usare CMMI non solo come "ispirazione" ma come vero e proprio **standard** per definire le azioni da svolgere nonché per confrontare il nostro operato e studiarlo qualitativamente. Lo studio qualitativo mi permette di stabilire la **maturità** del progetti, secondo un certo livello di *compliance*, detto **CMMI level**. Tale qualità che può essere certificata da enti certificatori appositi (si ha anche una repository pubblica con i livelli di *compliance* di diversi progetti di organizzazioni famose).
Studiamo a fondo questi livelli di maturità e la loro codifica.
Si hanno due linee di sviluppo/miglioramento:

1. **capability levels (*CL*)**, che cattura e rappresenta quanto bene si sta gestendo una particolare *process area* (quindi ciascuna *process area* può raggiungere un diverso CL). Quindi per una singola *process area* mi dice quanto bene sto raggiungendo i *generic goals* (e di conseguenza anche i vari *specific goals*, in quanto quelli generici impongono il controllo di quelli specifici). A livello di grafico punta all'ovale con *specific goals*. Il CL ha valore da 0 a 3:

    - **level 0: *incomplete***, dove probabilmente non si stanno nemmeno svolgendo tutte le pratiche richieste per quella *process area* o sono state svolte solo parzialmente
    - **level 1: *performed***, dove si eseguono le pratiche e i vari *specific goals* sono soddisfatti
    - **level 2: *managed***, dove oltre alle pratiche si ha anche una gestione (controllo, allocazione di risorse, monitoring, review etc...) delle attività stesse, come indicato nello standard. Si ha una policy per l'esecuzione delle pratiche
    - **level 3: *defined***, dove l'intero processo è ben definito secondo lo standard, descritto rigorosamente e si ha un processo completamente su misura dell'organizzazione

    I CL di ciascuna *process area* possono essere rappresentate su un diagramma a barre, dove viene indicato il CL attuale e il **profile target**, ovvero il livello a cui quella *process area* deve arrivare (che non è per forza il terzo).

2. **maturity levels (*ML*)**, che cattura il livello raggiunto dall'intero processo di sviluppo ragionando su tutte le *process area* attivate.

Rappresenta quanto bene si sta lavorando sull'insieme intero della *process area*. A livello di grafico punta direttamente all'elemento specificante la *process area*. Il ML ha valore da 1 a 5:

- **level 1:** ***initial***, dove si ha un processo gestito in modo caotico
- **level 2:** ***managed***, dove si ha già un processo ben gestito secondo varie policy
- **level 3:** ***defined***, dove si ha un processo ben definito secondo lo standard aziendale
- **level 4:** ***quantitatively managed***, dove si stanno anche raccogliendo dati che misurano quanto bene sta funzionando il processo, facendo quindi una misura quantitativa dello stesso
- **level 5:** ***optimizing***, dove grazie alle informazioni raccolte nel livello precedente ottimizzo il processo, in un'idea di *continuous improvement* del progetto stesso

Gli ultimi due livelli sono davvero difficili da raggiungere.

Rapportiamo quindi CL e ML in base avari blocchi di *process area*, che scalano per "importanza":

- il processo è a ML=2 se tutte le *process area* del blocco (che si riferisce alle *process area* riferite al blocco *planned & executed*) sono svolte tutte a CL=2
- il processo è a ML=3 se aggiungo le *process area* del blocco *org. standard* e queste sono tutte a CL=3
- il processo è a ML=4 se aggiungo le due *process area* del blocco *quantitative PM* che devono essere svolte a CL=3
- il processo è a ML=5 se aggiungo le ultime due *process area*, ovvero quelle del blocco *continuous improvement*, che devono essere svolte a CL=3

Si possono confrontare CMMI e le pratiche agili.
Ciò che viene svolto ai livelli 2 e 3 (con qualche piccolo adattamento) di *maturity level* si fa ciò che viene fatto anche coi metodi agili. In merito ai livelli 4 e 5 di *maturity level* si hanno pratiche che non rientrano nell'ottica dei metodi agili. Quindi un'organizzazione può usare i metodi agili ed essere standardizzata rispetto CMMI raggiungendo un *maturity level* 2 o 3.
CMMI è quindi uno standard industriale con certificazioni ufficiali.

Capitolo 6

Requirements engineering

Quando si parla di **requirements engineering (*RE*, ingegnerizzazione dei requisiti)** di fatto ci si concentra sulla comprensione di come una soluzione software si deve comportare per risolvere un certo problema. In questo senso bisogna prima comprendere quale sia il **problema** da risolvere e in quale **contesto** tale problema si verifica, per poter arrivare ad una soluzione corretta ed efficacie a problemi "reali". Bisogna ben comprendere il problema, non si parla quindi della progettazione in se, ma del "cosa" deve fare il software.

Esempio 3. *Vediamo un esempio banale.*
Bisogna sviluppare il software per l'adaptive cruise control. In questo caso si ha:

- *il problema che consiste nel seguire correttamente la macchina che si ha davanti in autostrada*
- *il contesto che consiste in:*
  - *guida della macchina*
  - *intenzioni del guidatore*
  - *norme di sicurezza*
  - *etc...*

Esempio 4. *Vediamo ora l'analogia classica per spiegare il RE, il* problema mondo *e la* soluzione macchina*.*
Si hanno:

- *il **mondo**, con un problema derivante dal mondo reale, mondo stesso che produce tale problema che bisognerà risolvere con un calcolatore. Si hanno all'interno:*

  - componenti umane*, ovvero staff, operatori, organizzazioni etc...*
  - componenti fisiche*, ovvero device, software legacy, madre natura etc...*

- *la* ***macchina****, che bisogna sviluppare per risolvere il problema. Si ha necessità quindi di:*
  - software *da sviluppare o acquistare*
  - piattaforma hardware/software*, con eventuali device annessi*

- requirements engineering *che si occupa di:*
  - *definire gli effetti della* macchina *sul problema del* mondo
  - *definire assunzioni e proprietà principali del* mondo *stesso*

Macchina e mondo possono condividere alcuni componenti, con la macchina che modifica il mondo (nell'esempio precedente la macchina che disattiva il cruise control agisce sia a livello software che a livello del mondo esterno). Nei RE studiamo quindi il mondo*, senza definire come funziona internamente la macchina (nelle parti che non interagiscono direttamente col mondo, per esempio scelte di design etc…).*

Quando si parla di RE è bene distinguere due elementi:

1. ogni volta che prendiamo in consideriamo un problema esiste sempre un **system-as-is**, ovvero un sistema preesistente che già risolve il problema (anche se magari in modo non efficiente o qualitativamente sufficiente). Si ha quindi sempre un sistema da cui partire (esempio banale per il cruise control è dire che in realtà la soluzione al problema già esiste con la guida senza cruise control)

2. esiste sempre un **system-to-be**, ovvero il sistema che si andrà a realizzare. In altre parole è il sistema quando la *macchina/software* ci opera sopra

Studiare il *system-as-is* è essenziale per poter lavorare al *system-to-be*.

Definizione 5. *Il requirements engineering è, formalmente, un insieme di attività:*

- *per esplorare, valutare, documentare, consolidare, rivisitare e adattare gli obiettivi, le capacità, le qualità, i vincoli e le ipotesi su un* system-to-be
- *basate sul problema sorto dal system-as-is e sulle nuove opportunità tecnologiche*

L'output di queste attività è un ***documento di specifica dei requisiti*** *con tutto ciò che soddisfa il sistema. Se l'output non è un singolo documento si ha una collezione di singoli requisiti, che nel metodo agile sono storie/cards e in altri metodi un repository centrale con un db condiviso contenete i vari requisiti.*
In ogni caso si ha un insieme di requisiti su come si deve comportare il sistema che si realizza, descrivendo quanto descritto nei punti precedenti.

In un modello simil-cascata di sviluppo software il RE è una delle primissime attività, subito dopo quelle di definizione del sistema e di business plan. Per gli aspetti tecnici è probabilmente la prima attività svolta, occupandosi di *ottenere il giusto sistema da sviluppare (per fare la cosa giusto)*, prima di design, implementazione ed evoluzione software etc... che si occupano di *ottenere il software giusto, sviluppandolo nel modo corretto.*
Errare nel RE può portare ad un ottimo software che risolve i problemi sbagliati (o non tutti i problemi che dovrebbe risolvere). Sbagliare i RE è una causa di fallimento del progetto (anche in un contesto non agile).
Lavorare sui RE non è semplice per diversi motivi:

- si deve ragionare su tante versioni del sistema:
  - as-is
  - to-be
  - to-be-next, volendo essere lungimiranti per il comportamento del sistema, sapendo e prevedendo evoluzioni future
- si lavora in *ambienti ibridi*, tra umani, leggi, device, policy, leggi della fisica, etc..., quindi in un contesto eterogeneo che va compreso a fondo (ignorare degli aspetti può portare al fallimento)
- si hanno diversi aspetti funzionali, qualitativi e di sviluppo

- si hanno diversi livelli di astrazione, con obiettivi strategici a lungo termine di inserimento sul mercato e dettagli operazionali
- si hanno tanti stakeholders, quindi con diverse parti interessati di cui risolvere problemi e interessi (con potenziali conflitti tra i vari stakeholders)
- si hanno tante attività tecniche legate l'una con l'altra:
  - conflict management
  - risk management
  - evaluation of alternatives
  - prioritization
  - quality assurance
  - change anticipation

  per capire cosa fare, in che ordine, con che rischi, che livelli di qualità, etc...

6.1 Tipi di requisiti

Bisogna anche ragioni sui tipi di requisiti su cui si deve lavorare.
Una prima differenza si ha nel modo in cui sono scritti i requisiti. Si hanno quindi:

- **descriptive statements (*dichiarazioni descrittive*)**, che indicano dei requisiti non negoziabili, rappresentano dei comportamenti derivanti dalle leggi del mondo su cui lavora la macchina. Si ha quindi zero margine di modifica
- **prescriptive statements (*dichiarazioni prescrittive*)**, che indicano requisiti negoziabili

Esempio 5. *Vediamo un esempio.*
Per il primo modo si ha:

La stessa copia non può essere presa in prestito da due persone diverse contemporaneamente

per il secondo:

Un cliente abituale non può prendere in prestito più di tre libri contemporaneamente

Entrambi sono importanti e vanno considerati. Si possono avere requisiti non ovvi, o lo sono in un contesto specialistico e quindi spesso non ovvi a chi lavora sul software (che generalmente non è del settore specialistico), che magari nemmeno sa che esistono.
I requisiti possono inoltre differire per gli elementi che prendono in considerazione. Posso avere requisti:

- **system requirements**, che riguardano come si comporta l'ambiente, per capirne il funzionamento (che andrà ad influenzare il software)
- **software requirements** che studiano come si comporta il software nell'ambiente, studiando i cosiddetti *shared fenomena*

ricordando che:

$$sistema = ambiente + software$$

I requisti comunque non riguardano mai il comportamento interno del software (variabili, error code etc...).
Si hanno 3 dimensioni:

1. **dimensione del why**, dove si identificano, analizzano e rifiniscono i requisiti del **system-to-be** per: affrontare le carenze analizzate del **system-as-is**, in linea con gli obiettivi di business, sfruttando le opportunità tecnologiche.
   Si hanno le seguenti difficoltà:
   - acquisire conoscenza del dominio
   - valutare opzioni alternative (ad esempio modi alternativi per soddisfare lo stesso obiettivo)
   - abbinare problemi-opportunità e valutarli in termini di implicazioni, e rischi associati
   - gestire obiettivi contrastanti
2. **dimensione del what**, dove si identificano, analizzano e rifiniscono le funzionalità del **system-to-be** (servizi software e associate procedure manuali) per soddisfare gli obiettivi individuati in base a vincoli di qualità: sicurezza, prestazioni, etc...basati su ipotesi realistiche sull'ambiente.
   Si hanno le seguenti difficoltà:
   - identificare il giusto set di funzionalità

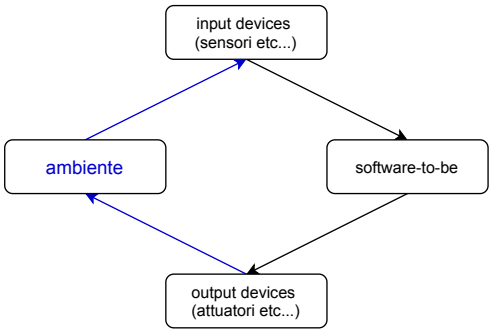


Figura 6.1: Raffigurazione del modello Parnas95

- specificarle precisamente per essere comprese da tutte le parti
- garantire la tracciabilità *backward* per gli obiettivi del sistema

3. **dimensione del who**, dove si assegnaresponsabilità per gli obiettivi, i servizi, i vincoli tra i componenti del **system-to-be** in base alle loro capacità e agli obiettivi del sistema, oltre i confini dell'ambiente software.
   La difficoltà è valutare opzioni alternative per decidere il giusto grado di automazione

I requisti che riguardano l'ambiente possono essere di due forme:

1. **domain properties**, ovvero proprietà riguardanti l'ambiente, immutabili

2. **assumptions**, ovvero le assunzioni fatte su come è fatto l'ambiente. Il software funzionerà solo negli ambienti che soddisfano quella certa assunzione (per esempio un numero massimo di chiamate API a cui il sistema è in grado di rispondere). Le assunzioni descrivono gli ambienti compatibili con il software. Se troppo restrittive possono portare al fallimento del progetto, che riguarderebbe pochissimi casi particolari

In ottica di un ragionamento di interazione tra sistema software e ambiente citiamo il **modello Parnas95**, della metà degli anni novanta.

In modo elementare il modello schematizza questa interazione software-ambiente in modo da esplicitare i device usati dal software-to-be per interagire con ambiente. Si ha uno schema composto da 4 elementi:

1. dei device di input, simil sensori, che percepiscono l'ambiente (ad esempio richieste che arrivano dal web)
2. dei device di output che permettono di interagire con l'ambiente (che possono ad esempio essere le risposte renderizzate nel browser)

Si hanno 4 tipi di variabili:

1. **input data (I)**, tra *input devices* e *software-to-be*
2. **output results (O)**, tra *software-to-be* e *output devices*
3. **controlled variables (C)**, tra *output devices* e *ambiente*
4. **monitored variables (M)**, tra *ambiente* e *input devices*

Si ha che:

$$\text{System requirements} \subseteq M \times C$$

$$\text{Software requirements} \subseteq I \times O$$

$$\text{Assumptions} \subseteq M \times C \cup M \times i \cup C \times O$$

Nel caso, ad esempio, di sistemi embedded, di domotica o simili tali device diventano parecchio complessi.

Definizione 6. *Definiamo* ***domain property*** *come un descriptive statement sui problemi legati ai fenomeni del mondo (a prescindere da qualsiasi* ***software-to-be****).*

Domain property $\subseteq M \times C$ *se leggi che non possono essere infrante*

Si ha che:

$$\text{Software requirements} = map(\text{System requirements, Software requirements, Domain property})$$

esempi su slide.

È utile quindi fare una distinzione per quanto concerne i requisiti del software. Abbiamo due famiglie principali:

1. **requisiti funzionali**, che indicano le funzionalità che un sistema, *system-to-be*, deve implementare, cosa deve essere in grado di fare. Non sono prevedibili prima di studiare il sistema

2. **requisiti non funzionali**, che indicano delle qualità o dei vincoli sulle funzionalità e quindi sui requisiti funzionali. Possono essere in termini di prestazioni, sicurezza, qualità etc... delle funzionalità. Questa famiglia di aspetti non funzionali è più o meno standard

Esempio 6. *Posso avere il requisito funzionale:*

Il cliente è in grado di prendere in prestito al massimo tre libri

e quello non funzionale:

Un cliente può prendere in prestito i libri solo dopo il riconoscimento

Si ha quindi una tassonomia dove si trovano organizzati aspetti non funzionali tali per cui, dopo aver individuato le funzionalità, possono essere usate per chiedersi se, per ciascun aspetto, esso è rilevante, individuando nuovi requisiti non funzionali da includere nell'insieme dei requisiti.
Per alcuni tipi di progetti requisiti funzionali e non sono difficilmente distinguibili, spesso mischiandosi (si pensi, ad esempio, alla sicurezza nello sviluppo dio un firewall).

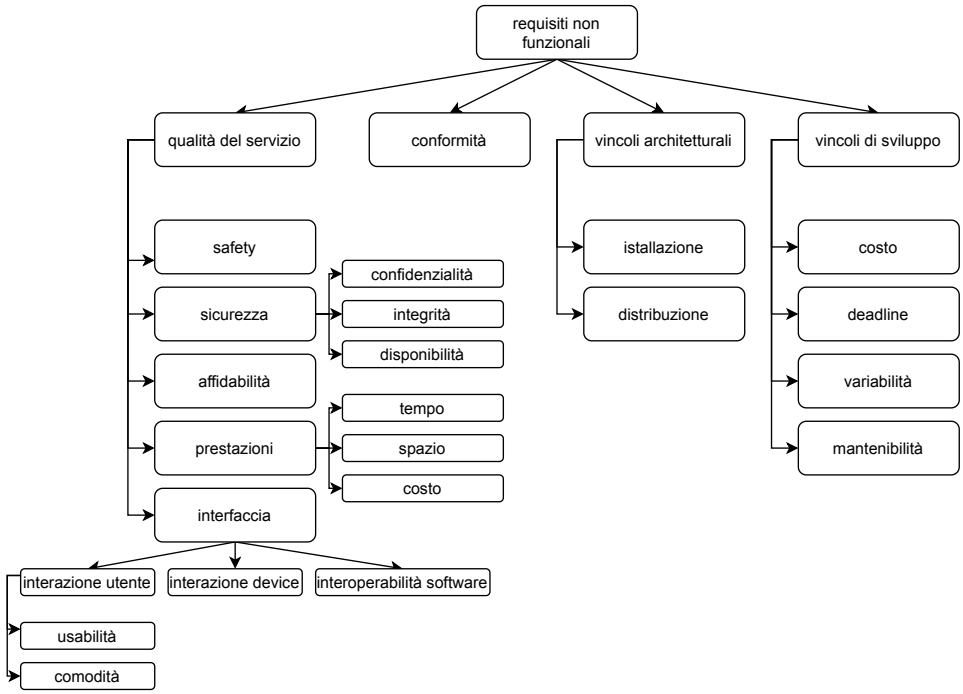


Figura 6.2: Tassonomia dei requisiti non funzionali

6.2 Qualità dei requisiti

Si hanno diversi aspetti di cui preoccuparsi nell'analisi dei requisiti. Si hanno tante qualità indispensabili:

- **completezza** di quanto descriviamo, descrivendo tutti i requisiti rilevanti del progetto, identificando tutti i comportamenti del sistema e documentarli in modo adeguato. È una qualità virtualmente irraggiungibile in modo assoluto, non è infatti verificabile. Inoltre i requisiti variano al proseguire del progetto, interagendo anche con gli stakeholders, rendendo questo uno degli aspetti più difficili da studiare
- **consistenza** dei requisti, senza conflitti. Nella mole di requisiti si possono purtroppo facilmente introdurre inconsistenze
- **non ambiguità** di ciò che si scrive. Tutto deve essere chiaro e non soggetto ad interpretazione
- **misurabilità**, quando rilevante, e quindi non basato su interpretazioni vaghe ma si misure concrete e specifiche

- **fattibilità**, ovvero non si devono avere requisiti basati su funzionalità irrealizzabili
- **comprensibilità**
- **buona struttura**
- **modificabilità**, con possibilità di avere il risultato mantenibile del tempo
- **tracciabilità**, individuando tutti gli artefatti ottenibili come conseguenza e tracciandoli. Si tracciano anche le dipendenze tra requisiti

Vediamo quindi gli errori che si fanno quando si va ad identificare i requisiti, per poter poi studiare tecniche per evitare, nel limite del possibile, tali errori:

- **omissioni**, non riuscendo ad identificare qualche requisito. Anche un riconoscimento tardivo è un problema in quanto comporta la modifica del documento, non sempre facile
- **contraddizioni**, avendo conflitti tra i requisiti (anche solo, banalmente, per i requisiti di un treno si potrebbe avere che non si possono aprire le porte tra due stazioni ma anche se si possono aprire le porte in caso d'emergenza)
- **inadeguatezza**, avendo requisiti che non sono adeguati per un determinato problema
- **ambiguità**, avendo requisiti interpretabili
- **non misurabilità** di certi requisiti (specialmente non funzionali), che comportano difficoltà di gestione, precludendo confronti etc. . .

Si hanno anche altri tipi di errore:

- essere **troppo specifici** nella definizione dei requisiti includendo anche comportamenti interni al software che non dovrebbero essere descritti in questa fase. Bisogna dire quali funzionalità bisogna realizzare, non come
- descrivere requisiti **non implementabili** considerando vincoli temporali o di costo
- descrivere requisti **complessi da leggere**, ad esempio con un uso eccessivo di acronimi

- avere **poca struttura** nella stesura dei requisiti, a livello visivo. Un documento tecnico deve essere facile da gestire e da usare
- se si produce un documento (e non nel caso dell'uso del db) bisogna evitare di fare riferimento a requisiti che non sono stati ancora descritti, ovvero evitando il **forward reference**
- evitare "**rimorsi**" di non aver definito nel momento giusto certi concetti che magari erano stati usati, senza definizione, precedentemente. Questi vanno definiti all'inizio del documento (nelle slide si dice tra parentesi)
- evitare la **poco modificabilità** del documento. È buona norma avere delle "costanti simboliche", definite all'inizio, a cui fare riferimento nel documento, in modo che un'eventuale modifica si rifletta su tutto il documento
- evitare l'**opacità/logica di fondo/rationale/motivazioni** dei requisti, in modo che sia chiaro il perché esso è stato incluso, portando a mettere in discussione requisiti in realtà sensati a cui si arriverebbe comunque dopo ulteriore analisi, dopo aver perso ulteriore tempo

6.3 Il processo RE

Vediamo le principali fasi del RE.
Si hanno 4 fasi a spirale:

1. **domain understanding & elicitation**
2. **evaluation & agreement**
3. **specification & documentation**
4. **validation & verification**

domain understanding & elicitation In questa fase si hanno due parti principali:

1. **domain understanding**
2. **requirements elicitation**

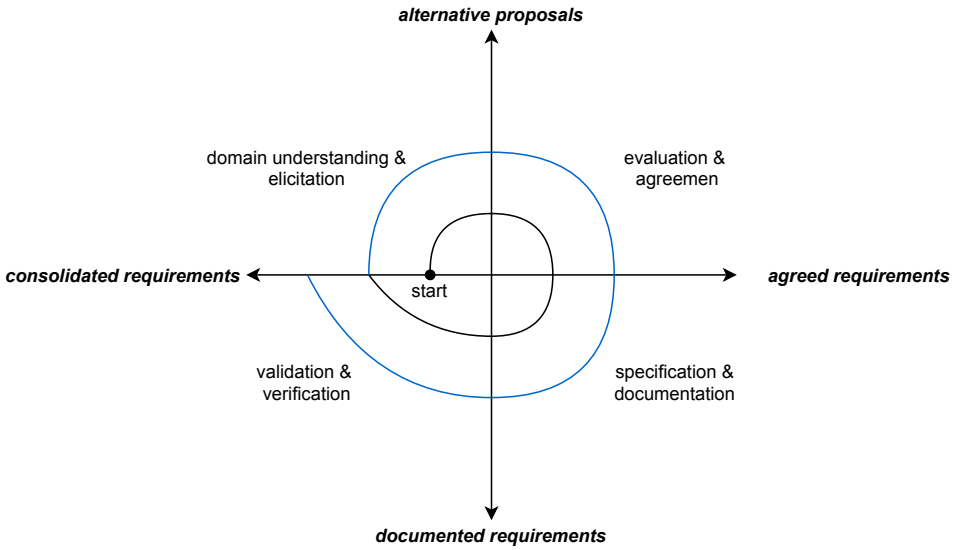


Figura 6.3: Rappresentazione a spirale delle quattro fasi del RE

Nella *domain understanding* si ha:

- lo studio del *system-as-is* in ottica di studio del dominio applicativo, studio del business organizzazione che vuole il prodotto. Si studiano anche forze e debolezze del *system-as-is*. Si studia il dominio applicativo, spesso sconosciuto e complesso, per ottenere il miglior *system-to-be* possibile
- si identificano gli stakeholders (e quindi tutte le parti interessate) del progetto per poter capire a fondo interessi e fini

Come **output** della *domain understanding* si hanno:

- le sezioni iniziali per la bozza di proposta preliminare
- il glossario dei termini

Passando alla *requirements elicitation* si uno studio più approfondito nel mondo:

- si ha un'ulteriore analisi dei problemi legati al *system-as-is*, alla ricerca di sintomi, cause e conseguenze
- vengono identificati, grazie all'aiuto degli stakeholders:
  - opportunità tecnologiche

  - condizioni del mercato
  - obiettivi di miglioramento
  - vincoli, organizzativi e tecnici, del *system-as-is*
  - alternative per raggiungere l'obiettivo e assegnare le responsabilità
  - scenari di ipotetica interazione software-ambiente
  - requisiti del software
  - assunzioni sull'ambiente

In **in output** alla *requirements elicitation* si hanno ulteriori sezioni per la bozza di proposta preliminare

evaluation & agreement Come risultato nella fase precedente si effettuando decisioni basate sull'interazione con i vari stakeholders (che spesso richiedono funzionalità conflittuali etc...), per poter valutare e decidere, avendo cambiamenti dei rischi in base a:

- identificazione e risoluzione di conflitti di interesse
- identificazione e risoluzione di rischi legati al sistema proposto
- comparazione e scelta tra le alternative proposte in merito a obiettivi e rischi
- prioritizzazione dei requisti, al fine di:
  - risolvere conflitti
  - definire vincoli di costi e tempi
  - supportare lo sviluppo incrementale

In **in output** a questa fase si hanno le sezioni finali per la bozza di proposta preliminare, dove si documentano gli obiettivi selezionati, i requisiti, le assunzioni e il *rationale*, la logica di fondo delle opzioni selezionate

specification & documentation In questa fase si raccoglie quanto detto nelle prime due fasi per produrre il documento, si hanno quindi:

- definizione precisa di tutte le funzionalità del sistema scelto, tra cui:

  - obiettivi, concetti, proprietà rilevanti del dominio d'interesse, requisti del sistema, requisti del software, assunzioni sull'ambiente e responsabilità
  - motivazioni e logica di fondo delle opzioni scelte
  - probabili evoluzioni del sistema ed eventuali variazioni
- organizzazione di quanto appena citato in una struttura coerente
- documentazione del tutto in un formato comprensibile a tutte le parti, mettendo in allegato:
  - costi
  - piano di lavoro
  - tempi di consegna del risultato

In **output** a questa fase si ha il vero e proprio **Requirements Document (*RD*)**

validation & verification In questa fase si studia l'RD, ovvero si studia la garanzia di qualità dell'RD (in modo equivalente a quanto può essere fatto durante la produzione di un software1, analizzando varie attività:

- **validazione**, ovvero vedendo se quanto contenuto nell'RD è adeguato con quanto si necessita
- **verifica**, controllando se ci sono omissioni o inconsistenze
- **correzione** di eventuali errori e difetti

In **output** a questa fase si ha un RD consolidato.

Ciclicità del processo Come è stato detto queste quattro fasi si ripetono in modo iterativo in modo "a spirale". Questo viene fatto in quanto si possono avere evoluzioni nel processo nonché correzioni di quanto già fatto che si propagano su tutto il documento. Le evoluzioni e le correzioni possono sopraggiungere durante:

- il RE stesso
- lo sviluppo del software
- dopo il deploy del software stesso

Dopo ogni ciclo si è molto più consci del sistema e si può passare al miglioramento del RE con più efficacia.
Questo sistema è molto compatibile con i vari **metodi agili**.

6.3.1 Domain understanding & elicitation

Approfondiamo questa fase analizzando i vari aspetti. Partiamo con la tecnica dell'**elicitation**, che consiste in tecniche atte allo scoprire requisiti che un progetto deve soddisfare.

Stakeholders

La prima parte è la selezione degli stakeholders del progetto. Questa fase è la prima che viene svolta per poter lavorare alla *elicitation*. In generale uno stakeholder è un'organizzazione o una persona che nutre un interesse rispetto al progetto. Conoscendo gli stakeholders avremo modo di prendere in considerazioni gli interessi degli stakeholders tramite diverse strategie. Idealmente si vuole realizzare qualcosa che soddisfi tutti gli interessi di tutti gli stakeholders.
I hanno diversi aspetti per la selezione degli stessi:

- posizione nell'organizzazione
- ruolo nel prendere decisioni sul *system-to-be*
- livello di esperienza del dominio applicativo
- esposizione al problema che il sistema deve risolvere
- influenza nell'accettazione del sistema
- obiettivi personali ed eventuali conflitti di interesse

Questa è un'attività insidiosa in quanto non si può dire se l'insieme degli stakeholders sia completo.
Si hanno anche stakeholders non sono dell'organizzazione ma anche associazioni, ulteriori organizzazioni, addetti alle normative di settore etc... producendo un insieme molto ampio e variegato, proporzionalmente alla grandezza del progetto. Si hanno anche stakeholders che non interagiscono direttamente col sistema.
Un modo per identificare gli stakeholders è tramite una serie di semplici domande, tramite una piccola attività di brainstorming (da fare anche insieme ad analisti):

- chi è influenzato positivamente e negativamente dal progetto?
- chi ha il potere di fargli avere successo (o farlo fallire)?
- chi prende le decisioni in materia di denaro?

- chi sono i fornitori?
- chi sono gli utenti finali?
- chi ha influenza, anche indiretta, sugli altri stakeholder?
- chi potrebbe risolvere potenziali problemi con il progetto?
- chi si occupa di assegnare o procurare risorse o strutture?
- chi ha competenze specialistiche cruciali per il progetto?

Si hanno vari check-list simili online.
Dimenticare uno stakeholders può portare ritardi o fallimenti del progetto, dovendo rivedere magari requisti (specifici di quello stakeholder) o dovendo rifare parti di sviluppo.
Si hanno varie difficoltà nell'acquisire informazioni dagli stakeholders, rendendo complesso il dialogo:

- fonti di conoscenza sul sistema distribuite sui vari stakeholders e tali fonti spesso sono contrastanti
- accesso difficile alle fonti
- ostacoli alla buona comunicazione, avendo background diversi sul dominio
- conoscenza non comunicata esplicitamente (magari anche solo perché date per scontate dagli esperti di un certo dominio o perché alcune informazioni sono sensibili o segretate) e bisogni nascosti
- fattori socio-politici
- condizioni instabili e mutabili, cambiano gli stakeholders, si hanno dinamiche aziendale mutevoli e cambi di ruoli. Anche per questo si hanno i metodi agili

Serve molta esperienza per gestire queste situazioni.
Servono quindi **buone capacità comunicative**, sapendo usare la giusta terminologia di dominio (ovvero quella dello stakeholders), arrivando dritti al punto e creando un rapporto di fiducia con gli stakeholders.
Si ha inoltre un piccola pratica, detta *knowledge reformulation*, ovvero quando si acquisiscono informazioni anche da fonti multiple è bene riformulare tale informazione allo stakeholder, per verificare una corretta comprensione.
È bene fare distinzione sulle tecniche di engagement con gli stakeholders, considerando due variabili, catalogandole in *low* e *high*:

1. potere decisionale
2. interesse nel progetto

Si ha quindi:

potere	interesse	strategia
high	high	fully engage
high	low	keep satisfied
low	high	keep satisfied
low	low	minimum effort

Dove nel dettaglio:

- **fully engage**: ovvero coinvolgimento regolare degli stakeholders che in questo caso sono della categoria principale. Devono essere estremamente soddisfatti
- **keep satisfied**: si hanno due casi:
  - se hanno alto potere decisionale si cerca di mantenerli informati e soddisfatti ma senza troppi dettagli
  - se hanno alto interesse, essendo spesso gli end-user, si cerca di consultarli spesso cercando di risolvere le problematiche indicate, coinvolgendoli regolarmente per ottenere dettagli e informazioni specifiche, essendo spesso i più informati sui dettagli
- **minimum effort**: mentendoli informati in modo generale e monitorandone eventuali cambi di ruolo, potere o interesse

Elicitation techniques

Passiamo quindi alle tecniche di elicitation. Si hanno due famiglie principali:

- **artefact-driven**, che fanno uso di artefatti per poter scoprire requisiti
- **stakeholders-driven**, che fanno invece uso degli stakeholders

Artefact-driven La prima tecnica è il **background study**, ovvero collezionare leggere e sintetizzare documenti su:

- le **organizzazioni stesse**, ovvero grafici, business plan, report finanziari, tempi delle riunioni etc...
- il **dominio applicativo**, ovvero libri, paper, report su sistemi simili etc...
- il **system-as-is**, ovvero workflow documentati, procedure, regole di business, report di errori, richieste di cambiamenti etc...

Queste collezioni di dati ci permettono di informarci in modo autonomo sul mondo in cui si andrà a lavorare, senza coinvolgere lo stakeholders, in quanto costoso, dispendioso e limitato in termini di tempo. Gli stakeholders vanno interpellati non per informazioni reperibili autonomamente ma per estrarre conoscenza non pubblica e non documentata. Ci si presenta allo stakeholder già con una base di conoscenza e conoscendo già la terminologia corretta del dominio.
Si ha quindi l'attività di **data collection** dove si estraggono informazioni dai dati di marketing, statistici etc... Sono dati utili per studiare il target. Si possono fare attività di *survey*. Si collezionano dati anche già documentati.
L'attività di background study ha ovviamente dei limiti di scalabilità, non potendo leggere troppe cose, sia per tempo che per costo. Si ha quindi la *meta-knowledge* per selezionare le parti dei documenti più rilevanti. Queste attività sono essenziali all'avvio di un progetto.
Un'altra tecnica è quella dei **questionari**, ovvero una lista di questioni da presentare agli stakeholders con una lista di possibili risposte. Si possono così interrogare molti stakeholders contemporaneamente, senza una particolare selezione. Si hanno quindi domande spesso a risposta multipla (per capire se fare nel modo A o B, ad esempio) e raramente aperte, che spesso vengono ignorate o mal risposte, ed eventualmente con risposte "pesate", per indicare il livello di gradimento (si preferiscono poche alternative, ad esempio usando la **scala di Liker**, di 4 opzioni, due positive, una alta e una bassa, e due negative, una alta e una bassa, quindi senza opzione intermedia, spesso scelta ma poco utile). Le domande devono essere chiare e che portino a risposte affidabili. È bene fare pochi questionari efficaci, con risposte poco affette da rumore. Si deve avere una numerosità di domande adeguata. Questi questionari aiutano a preparare colloqui/interviste mirati più efficaci (il questionario non è un'alternativa all'intervista). Spesso si usa prima un piccolo campione per i questionari per poi mandarlo a tutti. Si devono evitare ambiguità e inserimenti di bias, ne positivo ne negativo, nelle domande, senza avere domande

che possano condizionare la risposta. La raggiungibilità dei soggetti scelti è essenziale, nonché la scelta degli stessi. È buona pratica avere domande di *cross-check* (magari due domande che chiedono la cosa opposta) per controllare chi sta rispondendo non dia risposte inconsistenti e incoerenti, sintomo del fatto che uno non risponda a caso.
Un altro artefatto usato sono le **storyboards**, ovvero narrazioni, tramite esempi, di uso del sistema, sia del *system-as-is* (per le problematiche, per capire com'è) che del *system-to-be* (che capire cosa si vuole ottenere). Sono quindi storie fatte tramite sequenze di snapshots (slide, figure, sketch etc...). Tali storie si creano in due modi:

1. **modo attivo**, dove lo stakeholder contribuisce alla costruzione della narrazione
2. **modo passivo**, dove si narra la storia costruita allo stakeholder

Ovviamente vengono fatte solo per i workflow chiare.
Legati alle storyboards si hanno gli **scenari** che descrivono, attraverso una sequenza di interazioni rappresentate con testi o diagrammi, l'utilizzo del sistema, sia *as-is* che *to-be*. L'uso di esempi rende semplice la comunicazione e l'interazione con gli altri. Si hanno 4 tipi di scenario:

1. **positivo**, ovvero come il sistema si dovrebbe comportare
2. **negativo**, ovvero cosa il sistema non dovrebbe fare
3. **normale**, ovvero tutto procede come dovrebbe
4. **anormale**, ovvero cosa succede in casi eccezionali

Gli ultimi due sono sotto-categorie degli **scenari positivi**.
Gli scenari sono un metodo naturale di interazione con lo stakeholders e possono guidare la stesura di **testi di accettazione**. Hanno comunque dei limiti, essendo solo esempi parziali e sono poco adatti alla combinazione tra essi. Glie esempi possono anche deviare la comprensione del sistema, che magari può comportarsi in molti altri modi, avendo anche dettagli irrilevanti. Gli scenari non si prestano all'interazione con molti stakeholder, a causa della variabilità di questi in ottica di conoscenza e posizione.
Esempi di scenari sulle slide.
Un'altra tecnica è la costruzione di **prototipi** e **mock-up**. In questo caso l'obiettivo è quello di controllare l'adeguatezza di un requisito, che viene mostrato in modo visuale nella sua ipotetica formula finale. Si hanno quindi piccoli esempi del software in azione, chiarendo e verificando che sia quello di cui l'utente ha bisogno. Ovviamente un mock-up non ha la logica applicative ma risposte costruite a priori. A seconda del tipo di mock-up il focus è su:

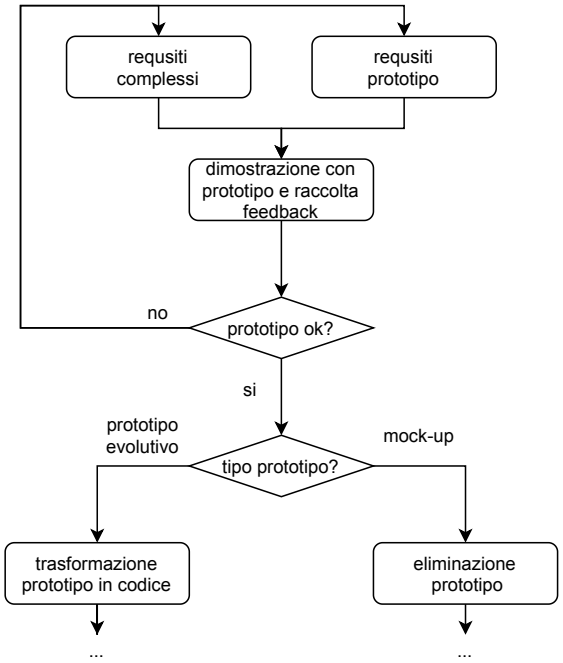


Figura 6.4: Rappresentazione del processo tramite prototipi

- **funzionalità**, se sono state comprese correttamente e in modo esaustivo
- **UI e UX**, avendo focus più orientati all'usabilità

Si parla di **mock-up** se dopo l'uso viene buttato e di **prototipo** se, in caso di approvazione, viene usato come base del software o comunque riutilizzato in qualche modo, fornendo eventualmente una base evolutiva del software finale.
Come pro si ha una sensazione di concretezza per lo stakeholder, anche visuale, permettendo anche già una sorta di training sul software che si produce. Purtroppo spesso i m mock-up sono costosi, inoltre la loro natura simulata può produrre aspettative troppo alte, magari anche solo in termini di prestazioni (il mock-up è super performante mentre il prodotto finale molto meno, avendo tutta la logica applicativa sotto). Inoltre in un'ottica di riuso un mock-up è spesso inutilizzabile a causa del **quick-and-dirty code**. Si ha anche il **knowledge reuse**, per velocizzare l'elicitation, riusando conoscenze pregresse da sistemi simili a quello sotto studio.
Si hanno 3 fasi:

1. trarre conoscenze e informazioni rilevanti da altri sistemi

2. trasporle nel sistema in studio
3. convalidare il risultato, adattarlo se necessario e integrarlo con la conoscenza del sistema già acquisita

Tali conoscenze possono essere dipendenti o indipendenti dal dominio.
Si hanno i seguenti pro:

- analisti esperti riutilizzano naturalmente dall'esperienza passata
- riduzione degli sforzi di eliciation
- ereditarietà della struttura e qualità delle specifiche del dominio astratto
- efficace per completare i requisiti con aspetti trascurati

e i seguenti contro:

- efficace solo se il dominio astratto è sufficientemente simile e accurato
- definire domini astratti per una riusabilità significativa è difficile
- si hanno forti sforzi di convalida e integrazione
- le corrispondenze vicine possono richiedere adattamenti complicati

Approfondimento su slide.
Un'altra tecnica **artefact-driven** è il **card sort**, che consiste nel chiedere agli stakeholder di suddividere un set di carte dove:

- ogni carta cattura un concetto in modo testuale o grafico
- carte raggruppate in sottoinsiemi in base ai criteri degli stakeholder

L'obiettivo è acquisire ulteriori informazioni sui concetti già evocati. Per ogni sottoinsieme, chiedere la proprietà condivisa implicita utilizzata per il raggruppamento per poi ripetere con le stesse carte per nuovi raggruppamenti / proprietà.

Stakeholders-driven Passiamo quindi alle tecniche in cui interagisce direttamente lo stakeholder, senza la mediazione di artefatti.
La prima tecnica è l'**intervista** che consiste in:

- selezione mirata dello stakeholder, in base alle informazioni necessarie
- fare l'intervista registrando le risposte
- scrivere il *transcript* dell'intervista e produrre subito il report
- sottomettere all'intervistato il report per validazione

Si può avere un'intervista anche con più stakeholder (perdendo però in parte la costruzione del rapporto personale con lo stakeholder). L'intervista è una tecnica costosa e le interviste possono essere poche, bisogna quindi procedere in modo attento.
Si hanno due tipi:

1. **interviste strutturate**, dove si parte con un insieme di domande già scelto per un certo obiettivo. Si da poco spazio ad una discussione aperta
2. **interviste non strutturate**, dove si da spazio alla discussione aperta e libera sul *system-as-is*, sulle problematiche e sulle soluzioni.

Spesso si hanno interviste miste, nella prima parte strutturate e poi con domande e argomenti liberi. Gli argomenti vanno calibrati così come il numero di domande, per evitare perdite di tempo e di attenzione da parte dello stakeholder.
Vediamo quindi qualche linea guida per la preparazione delle interviste.

- bisogna arrivare preparati, tramite il background study
- per costruire un rapporto con lo stakeholder le domande devono essere su misura dello stesso, in base al suo lavoro e ruolo
- mettere in centro all'intervista necessità e problematiche dell'intervistato, chiarendo di essere interessati al suo punto di vista
- evitare che la discussioni dilaghi su argomenti inutili
- fare in modo che l'intervistato si senta a suo agio, magari iniziando con qualche chiacchiera informale e domande semplici per poi spostarsi su domande difficili

- dimostrarsi affidabile
- chiedere sempre "perché" cercando il rationale di ciò che si chiede
- evitare domande bias che influenzino la risposta
- evitare domande ovvie che facciano pensare all'intervistato che stia perdendo tempo
- evitare domande a cui sicuramente l'intervistato non sa rispondere

Nel transcript bisogna includere reazioni personali (anche se queste parti possono essere tolte dalla copia di validazione).
Si hanno altri modi per interagire con lo stakeholder.
Talvolta si ha bisogno anche di altre modalità oltre la sola intervista.
Spesso è più facile infatti far osservare che descrivere a parole e quindi si hanno **studi osservazionali ed etnografici**, ovvero studi che si basano sull'osservazione degli stakeholders stessi all'azione, nell'ottica del *system-as-is*, osservando come svolgono vari task per cogliere problemi e funzionalità.
Si hanno due modalità di osservazione:

1. **osservazione passiva**, dove non si produce interferenza sulle azioni dello stakeholder, guardando da fuori, registrando record e producendo un transcript. Si hanno due particolari **osservazioni passive**:
   (a) **protocol analysis**, dove si studia qualcuno che svolge un certo protocollo, un certo task (la slide dice spiegando contemporaneamente)
   (b) **ethnographic studies**, studio che si svolge in un lungo periodo di tempo, dove si prova a scoprire le proprietà emergenti del gruppo sociale coinvolto
2. **osservazione attiva**, dove si svolge in prima persona i task, diventando eventualmente team member, capendo attivamente come deve essere svolto un task

Gli studi osservazionali aiutano a scoprire requisti che restano altrimenti taciti e a scoprire problemi nascosti, che non si noterebbero nell'intervista. Aiuta anche a contestualizzare le informazioni. È comunque un'operazione lenta e costosa, dovendo recarsi di persona ad osservare ed annotare. Può essere potenzialmente inaccurato in quanto una persona conscia di essere osservata potrebbe comportarsi in modo diverso dallo standard. L'osservazione si focalizza comunque solo sul *system-as-is* e non aiuta molto nel *system-to-be*.

Si hanno altri modi di interazione con gli stakeholders tra cui le **group sessions**, ovvero una famiglia di tecniche utili per la risoluzione di conflitti. L'elicitation prende la forma di un "workshop" di uno o più giorni in cui si ha una discussione tra vari partecipanti scelti in modo oculato a seconda dall'obiettivo, innescando una discussione utile a comprendere il *system-to-be*. Mettendo insieme le parti con visioni conflittuali si possono risolvere conflitti. Su hanno due tipologie di *group sessions*:

1. **strutturate** dove ogni partecipante ha un ruolo chiaramente definito: leader, moderatore, manager, utente, sviluppatore, etc... Ognuno contribuisce in funzione del ruolo, permettendo di ragionare su requisiti di più alto livello, comuni a più figure, e su conflitti ad alto livello

2. **non strutturate**, dette anche **sessioni brainstorming**, dove i partecipanti non hanno un ruolo definito. La sessione è formata da due fasi:

    (a) **generazione delle idee**, dove tutti espongono le proprie idee in merito al problema/conflitto che ha generato la sessione

    (b) **valutazione delle idee**, dove tutte le idee vengono analizzate una per una e valutate per valore, costo, fattibilità etc... in modo da arrivare una visione condivisa sugli approcci da usare secondo una certa prioritizzazione

Si ha il pro di poter esplorare varie opzioni e a portare a risoluzioni sfruttando anche l'inventiva dei partecipanti. La composizione del gruppo è comunque critica, molti potrebbero non avere tempo, alcuni potrebbero prendere posizioni dominanti introducendo bias, condizionando la discussione e le decisioni. Si rischia inoltre che la discussione diverga verso inutilità senza convergere sugli aspetti per cui è stata convocata la sessione.
Ogni attività, sia artefact-driven che stakeholder-driven viene svolta solo se si hanno chiari gli obiettivi che si vogliono ottenere con tale attività.

6.3.2 Evaluation & agreement

Requirements Evaluation

In questa fase si studia:

- *inconsistency management*, in ottica di:
  - tipi di inconsistenza
  - manipolazione delle stesse
  - gestione dei conflitti in modo sistematico
- la valutazione di alternative per prendere decisioni
- la prioritizzazione dei requisti, *requirements prioritization*

Consistency management

Definizione 7. *Definiamo* ***inconsistenza*** *come la violazione della regola di coerenza tra gli elementi.*

Si hanno due tipi:

1. **inter-viepoint**, quando ogni stakeholder ha il suo focus e il suo punto di vista
2. **intra-viewpoint**, quando si hanno conflitti tra i requisti di qualità (conflitti per esempio tra sicurezza e accessibilità)

A livello di tempo le inconsistenze vanno identificate e risolte:

- *non troppo presto*, per avere prima un'elicitation più approfondita
- *non troppo tardi*, per permettere lo sviluppo del software

Dal punto di vista dei tipi di inconsistenza abbiamo:

- **terminology clash**, avendo lo stesso concetto denominato diversamente in statement diversi
- **designation clash**, avendo lo stesso nome per concetti diversi in statement diversi
- **structure clash**, avendo lo stesso stesso concetto strutturato in modo diverso in statement diversi

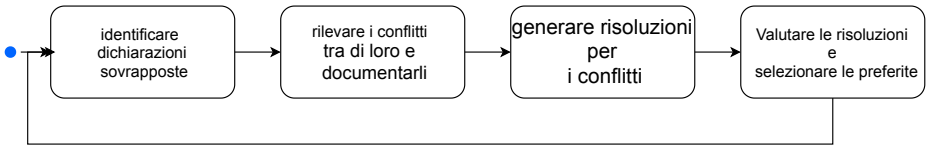
Distinguiamo inoltre:

- **conflitto forte**, avendo statement non soddisfacibili contemporaneamente (ad esempio $S \wedge \neg S$)
- **conflitto debole (*divergenza*)**, avendo statement non soddisfacibili contemporaneamente in certe condizioni e con certi vincoli

Per gestire i clash (scontri) in termini di terminologia, design e/o struttura si usa il glossario costruito nella fase di elicitation (dove vengono anche usati eventualmente acronimi).
Gestire le inconsistenze è difficoltoso a causa:

- obiettivi personali in conflitto da parte degli stakeholder (cosa che andrebbe gestita alla base propagandone i risultati al livello dei requirements)
- legati ad ambiti non funzionali, come ad esempio prestazioni vs sicurezza, confidenzialità vs consapevolezza, etc... (cosa che andrebbe gestita tramite uno studio dei tradeoff, i compromessi, migliori)

Come detto la gestione dei conflitti avviene in modo schematico, tramite il seguente schema:



Nel dettaglio:

- per la prima fase con sovrapposizione/overlap indichiamo il riferimento a termini o fenomeni comuni
- per la seconda il riconoscimento può essere fatto informalmente, tramite euristiche sulle categorie dei requisiti in conflitto, o formalmente. Il riconoscimento deve essere documentato per una successiva risoluzione e analisi dell'impatto. Si usano strumenti come l'**interaction matrix** che presenta su righe e colonne gli statement e negli incroci indica:
  - 1 per il conflitto
  - 0 per nessun overlap

  - 1000 per overlap senza conflitto

Avendo, per ogni statement S_i, indicando con S_i la riga/colonna corrispondente (sono uguali):

$$conflict(S_i) = \left(\sum_{s \in S_i} s\right) \mod 1000$$

$$nonConflictingOverlaps(S_i) = \left\lfloor \sum_{s \in S_i} 1000 \right\rfloor$$

statement	S_1	S_2	S_3	S_4	total
S_1	0	1000	1	1	1002
S_2	1000	0	0	0	1002
S_3	1	0	1	1	1002
S_4	1	0	1	0	1002
total	1002	1000	2	2	1002

Esempio 7.

$$conflict(S_1) = 2$$
$$nonConflictingOverlaps(S_2) = 1$$

- per la terza fase è meglio:
  - esplorare prima più risoluzioni, generate tramite tecniche di elicitation e usando tattiche di risoluzione dei conflitti:
    - evitare condizioni a contorno
    - ripristinare statement in conflitto
    - indebolire gli statement in conflitto
    - non considerare statement a bassa priorità
    - approfondire source e target del conflitto
  - confrontare, selezionare e concordare il preferito poi

  Si trasformano quindi statement in conflitto (e parti coinvolte) in nuovi requisiti

- per la quarta fase si usano vari criteri per la scelta:
  - ontributo a requisiti non funzionali critici
  - contributo alla risoluzione di altri conflitti e rischi
  - applicazione dei principi di *risk analysis*

Requirements prioritization Ai vari requisti va imposta una prioritizzazione, per vari scopi:

- risoluzione dei conflitti
- limitazioni delle risorse (budget, personale, programmi)
- sviluppo incrementale
- ripianificazione a causa di problemi imprevisti

e si hanno vari principi per una buona riuscita della stessa, che può essere svolta:

1. tramite livelli ordinati di uguale priorità, in un piccolo numero
2. tramite livelli relativi ("maggiore di" etc...)
3. tramite requisiti comparabili: stessa granularità, stesso livello di astrazione
4. tramite requisiti non mutuamente dipendenti (uno può essere mantenuto, un altro eliminato)
5. tramite un'accordo con i vari partecipanti e stakeholder

Per i primi tre posso usare una tecnica sistematica basata su diversi step:

- stimare il contributo relativo di ogni richiesta al **valore** del progetto
- stimare il contributo relativo di ciascuna richiesta al **costo** del progetto
- tracciare il **diagramma valore-costo**

Si usa quindi la tecnica AHP dalla **Decision Theory** dove si cerca di capire in che proporzione ogni requisito R_i contribuisce al criterio $crit$, che sarà prima il **valore**, $crit = value$, e poi il **costo**, $crit = cost$.
Si hanno quindi due step:

1. si usa la **comparison matrix** stimando i contributi di ogni requisito sul criterio da studiare.
   Avendo N requisiti:

$$R_{ij} = \frac{1}{R_{ij}} \text{ se } 1 \leq i, j \leq N$$

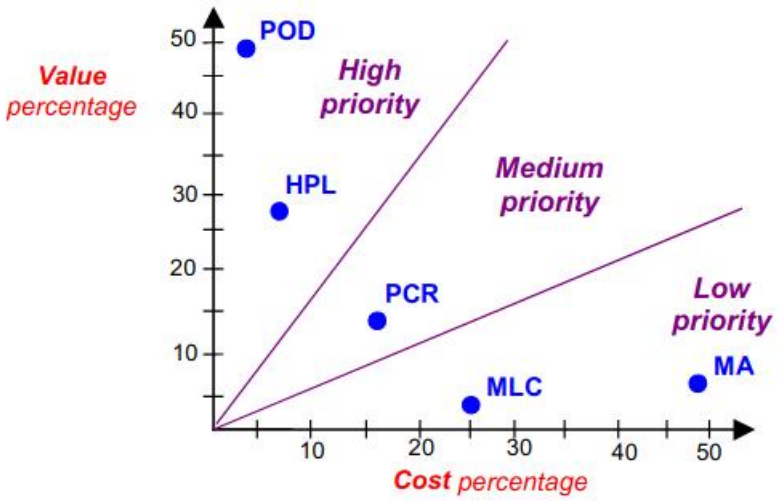


Figura 6.5: Grafico costo-valore con 5 requisti d'esempio: POD: Produce Optimal meeting Dates, HPL: Handle Preferred Locations, PCR: Parameterize Conflict Resolution, MLC: Support Multi-Lingual Communication e MA: Provide Meeting Assistant

Esempio 8. *(i conti potrebbero essere errati)*

crit=value	R_1	R_2	R_3	R_4	R_5
R_1	1	3	5	9	7
R_2	$\frac{1}{3}$	1	3	7	7
R_3	$\frac{1}{5}$	$\frac{1}{3}$	1	5	3
R_4	$\frac{1}{9}$	$\frac{1}{7}$	$\frac{1}{5}$	1	$\frac{1}{3}$
R_5	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{1}{3}$	3	1

2. si determina quanto questo si distribuisce tra tutti i requisti, tramite gli **autovalori** della matrice.
   Inoltre le colonne vengono normalizzate tramite:

$$R_{ij} = \frac{R_{ij}}{\sum_i R_{ij}}$$

e aggiungendo la media sulle righe per valutare il contributo del requisito sul criterio:

Esempio 9. *(i conti potrebbero essere errati)*

	R_1	R_2	R_3	R_4	R_5	contributo
R_1	0.56	0.65	0.52	0.36	0.38	0.49
R_2	0.19	0.22	0.31	0.28	0.38	0.28
R_3	0.11	0.07	0.10	0.20	0.16	0.13
R_4	0.06	0.03	0.02	0.04	0.02	0.03
R_5	0.08	0.03	0.03	0.12	0.05	0.07

AHP permette di assicurare stime e rapporti consistenti.

6.3.3 Specification & documentation

Parliamo quindi della fase di **specification & documentation**.
In questa fase si documentano in modo preciso i requisiti scoperti. Si descrivono in modo preciso le feature e i concetti rilevanti per il progetto. Nel dettaglio si definiscono in modo preciso:

- obiettivi, concetti, proprietà di dominio rilevanti, requisiti di sistema/software, ipotesi, responsabilità
- il rationale delle opzioni scelte
- un'indicazione sulle varianti e sulle evoluzioni previste

Il documento deve avere una struttura coerente. Tale documento è detto **requirements Document (*RD*)**. Qualora i requisti siano stati messi online si procede alla costruzione di un db che tiene traccia dei requisti. In ogni caso il contenuto deve essere accessibile e comprensibile da tutte le parti interessate, aggiungendo spesso in allegato anche costi, workplan e piani di delivery. Bisogna capire come effettivamente documentare i requisiti.
La prima opzione è l'utilizzo del linguaggio naturale in modo svincolato, in prosa. Si ha una forte espressività ma, in assenza di regole sulla scrittura si rischia di produrre una sorta di romanzo, producendo qualcosa di poco gestibile. Il linguaggio naturale inoltre può nascondere ambiguità (basti pensare anche solo a sequenze di *or* e *and* in assenza di uno schema logico fisso). Usando il linguaggio naturale privo di vincoli si hanno quindi rischi ma si può usare un *linguaggio naturale strutturato*. Si introducono quindi due tipi di regole:

1. **local rules**, che riguardano la scrittura del singolo requisito
2. **global rules**, che riguardano le regole sulla scrittura dell'intero documento e sull'insieme dei requisti

Abbiamo però, in primis, alcune regole stilistiche generali (le stesse che si dovrebbero applicare anche per una tesi):

- scrivere pensando a chi deve leggere, che deve essere ben identificato
- spiegare cosa stiamo per scrivere prima di specificarlo nel dettaglio
- prima motivare le scelte, indicare le scelte e poi fare un sommario delle stesse
- assicurarsi che ogni concetto usato nei requisiti sia stato prima ben definito
- chiedersi se quanto scritto è comprensibile e rilevante (per capirlo basta cancellare quanto scritto e vedere se si capisce ancora)
- per ogni frase/elemento del documento indicare uno e un solo requisito o assunzione o proprietà di dominio, evitando di mischiare troppe cose
- scrivere frasi brevi
- distinguere ciò che è obbligatorio da ciò che è desiderabile
- evitare acronimi non necessari ed evitare l'abuso del gergo informatico
- usare esempi esplicativi
- usare diagrammi o illustrazioni quando utile

In termini di local rules si hanno dei template per scrivere i requisiti, usando anche un solo standard per tutti i requisti. Un esempio di template potrebbe contenere:

- identificatore del requisito (con uno schema di naming significativo, magari con uno schema gerarchico con combinazioni di caratteri tramite separatore per specificare al meglio il requisito)
- categoria del requisito (funzionale, assunzione etc...)

- specifica del requisito (usando le regole stilistiche)
- criterio di fit (una sorta di test di accettazione, usato quindi per quantità/concetti misurabili, essendo critico quindi per requisiti non funzionali)
- fonti di elicitazione
- rationale
- interazioni con altri requisti (contribuzioni, dipendenze e conflitti)
- livello di priorità
- livelli di stabilità (per indicare le chance di cambiamenti)

Per le global rules si ha una forma standard data da **IEEE std-830**, il più diffuso. Si hanno varie macrocategorie a loro volta suddivise:

1. introduzione (dove si specifica anche quale parte del documento interessi ai vari stakeholder)
    (a) motivazioni del documento
    (b) scopo del prodotto
    (c) definizioni, acronimi, sigle e abbreviazioni
    (d) reference, fonti di elicitazione
    (e) overview dell'organizzazione del documento
2. descrizione generale
    (a) prospettive del prodotto
    (b) funzionalità principali
    (c) caratterizzazione degli utenti
    (d) vincoli generali sull'ambiente (che non possono esse modificati ma con i quali si deve integrare il prodotto)
    (e) assunzioni sul prodotto e dipendenze del prodotto
    (f) ripartizione dei requisti
3. requisiti specifici (spesso per i dev), usando magari il template visto prima. Si hanno quindi varie categorie di requisiti che articolano le varie sezioni del capitolo:

   (a) requisiti funzionali (che a sia volta un'organizzazione interna in base a vari fattori, come aree funzionali, componenti etc...)
   (b) Requisiti per l'interfaccia esterna
   (c) requisiti per le prestazioni
   (d) vincoli di progettazione
   (e) attributi di qualità del software
   (f) altri requisiti

4. appendice
5. indice

Una variante p il **template VOLERE** dove si hanno sezioni esplicite per proprietà del dominio, costi, rischi, piano di lavoro di sviluppo etc...
Si hanno anche opzioni aggiuntive oltre al linguaggio naturale. In primis si ha l'uso di diagrammi, con quindi una **notazione semi-formale**, in quanto si ha una sintassi formale essendo i vari diagrammi formalmente definiti ma interpretazione degli stessi può comunque ssere ambigua.
I diagrammi sono utili per complementare quanto scritto in linguaggio naturale, che risulta spesso troppo complesso e articolato. Un diagramma può semplificare quanto scritto e rappresentare il tutto in modo compatto. L'uso di diagrammi permette di comunicare più semplicemente e viene usato per aspetti specifici del sistema. Si ha comunque la stessa ambiguità di interpretazione che si aveva nei linguaggi naturali. vediamo alcuni diagrammi.
Un primo esempio è il **problem diagram** che descrive i requisiti a livello di sistema. Si mostrano le componenti del sistema dove ogni componente è indicato tramite un rettangolo con doppia linea a sinistra. Il sistema comunica con diversi elementi dell'ambiente, indicati da rettangoli. Si può specificare testualmente un requisito e riferirlo agli elementi dell'ambiente citati nel requisito, che è posto in un ellisse tratteggiato. Posso avere riferimenti e vincoli sugli elementi a partire dal requisito. Tutte le interazioni sono decorate con gli eventi che sono rilevanti per l'interazione con il produttore dell'evento indicato tramite le maiuscole e il nome dell'evento, a loro volta separati da "!".
Con questo tipo di diagramma si cattura il contesto del requisito, legandolo componenti dell'ambiente e componenti del sistema, tramite specifici eventi. Si identificano a colpo d'occhio gli elementi rilevanti per un certo requisito. Esempio di problem diagram nella figura 6.6. Questa rappresentazione può prendere la forma di veri e propri pattern detti **problem pattern** per semplificare la rappresentazione. Si ottengono i frame diagrams, che hanno

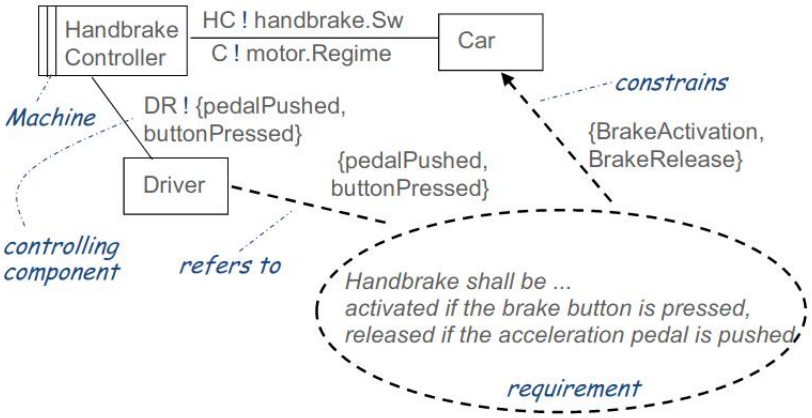


Figura 6.6: Esempio di problem diagram

la stessa rappresentazione dei problem diagram con la differenza che si ha qualche informazione in più:

- tipo di componente (indicato con un quadratino in basso a destra nel rettangolo della componente) specificato da una lettera:
  - C: causal, causa-effetto
  - B: biddable, non predicibile
  - X: lexical, lessicale, specifica artefatti
- tipo di evento (posto) dopo il "!" sopra descritto) specificato da una lettera:
  - C: causal, diretta conseguenza di altri eventi
  - E: event, che non sono diretta conseguenza ma che sono prodotti in modo spontaneo
  - X: lexical, dati che vengono utilizzati per produrre qualcosa

Tali pattern vengono istanziati nei vari casi specifici (esempio completo nelle slide). Si ha un catalogo di tali pattern caricato sulla pagina del corso.
Un altro diagramma tipico è quello ER per specificare entità che entrano in gioco in certi aspetti dei requisiti, specificandole in modo schematico. Si usano anche diagrammi di dominio, di classe etc..., non descrivendo più comportamenti ma strutture, domini etc...
Tipi di diagramma invece da approfondire sono i **SADT diagrams** che si

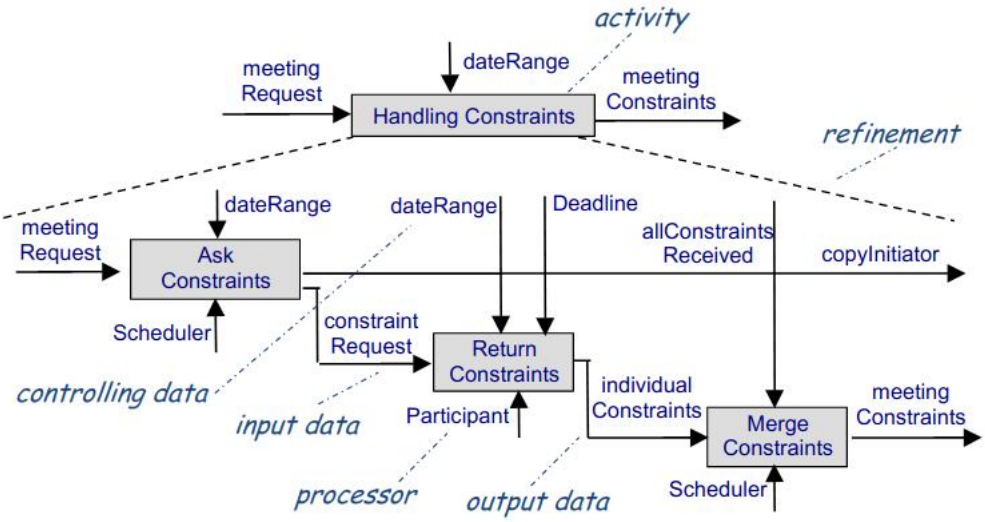


Figura 6.7: Esempio di actigram con sotto un raffinamento della parte superiore

dividono in due tipi e permettono di specificare il comportamento di alcune attività scomponendolo e aggiungendo varie informazioni.

1. actigram, di tipo activity-driven, si concentra sulle attività e mostra le dipendenze tra attività in termini di dati. Le attività sono indicate dentro rettangoli e si hanno una serie di frecce che a seconda della direzione variano il significato:

   - $\longrightarrow$ per l'input di dati
   - $\longleftarrow$ per l'output di dati
   - $\downarrow$ per il data/event controlling, ovvero dati o eventi che controllano il comportamento dell'attività, sono magari aspetti di configurazione etc... che influenzano l'attività
   - $\uparrow$ per l'unità che processerà l'attività (opzionale)

   Vedere figura 6.7.

2. datagram, di tipo data-driven, si concentra sui dati e mostra le dipendenze tra dati in termini di attività. I dati sono indicate dentro rettangoli e si hanno una serie di frecce che a seconda della direzione variano il significato:

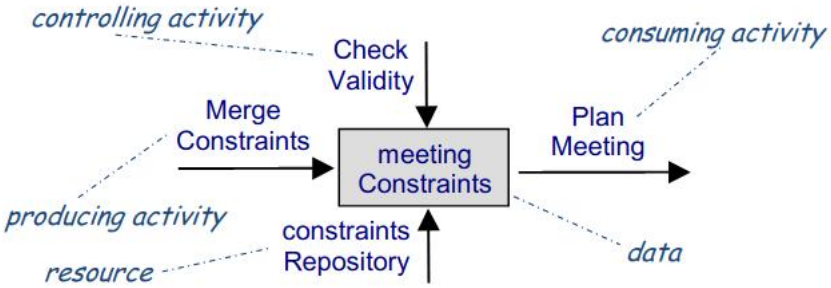


Figura 6.8: Esempio di datagram

- $\longrightarrow$ per l'input di attività che producono il dato
- $\longleftarrow$ per l'output di attività verso le quali serve il dato
- $\downarrow$ per le attività di validazione del dato
- $\uparrow$ per le risorse necessarie a memorizzare e gestire il dato

Vedere figura 6.8.

La coerenza tra di due diagrammi è fondamentale avendo una rapporto di **dualità** tra essi quindi i dati e le attività presenti in uno devono apparire anche nell'altro.

Questo strumenti si prestano a documentare workflow molto semplici. In ogni caso:

- gni attività deve avere un input e un output
- tutti i dati devono avere un produttore e un consumatore
- i dati I/O di un'attività devono apparire come dati I/O delle sottoattività
- ogni attività in un datagramm deve essere definita in un actigramma

Un altro diagramma classico è il **case use diagram** per visualizzare i requisiti identificati, che prendono forma di casi d'uso con gli attori che partecipano allo svolgimento dei requisiti. Si hanno le varie relazioni di inclusione etc... Un altro strumento utile nella definizione di workflow è l'**eventi trace diagrams**, ovvero i diagrammi di sequenza. Se ne hanno vari tipi con sintassi più o meno ricche ma in generale si hanno N elementi che partecipano all'esecuzione che viene mostrata visualmente tramite richieste e risposte, sincrone

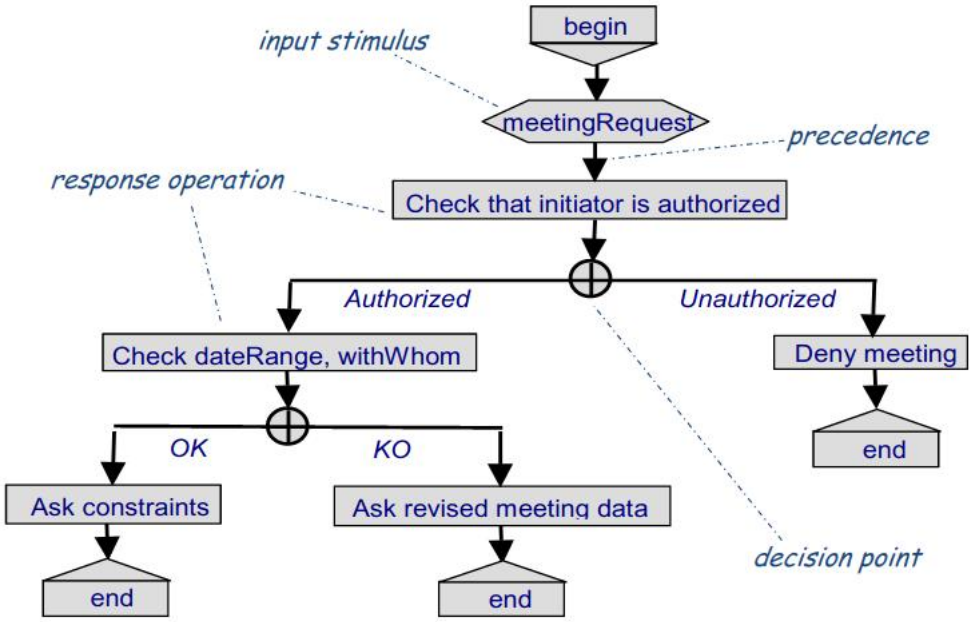


Figura 6.9: Esempio di R-net diagram

o meno, che vengono indicate cronologicamente dall'alto al basso. Questa visualizzazione compatta aiuta nel momento in cui il linguaggio naturale diventa troppo complesso e verboso per spiegare una certa sequenza di azioni.
Un altro diagramma usato è il **state machine diagram** per mostrare in quali stati un particolare elemento si trova e quali transizioni/eventi modificano i suoi stati. Questi diagrammi sono utili in quanto in modo compatto rappresenta il ciclo di vita di una certa componente, facilitando la creazione del sistema.
Come altro diagramma abbiamo il **R-net diagram** che permette di mostrare come reagisce un sistema in base ad un certo stimolo. È quindi un albero che parte con uno stimolo, dopo il begin, posto in un esagono e si sviluppo in base alle varie alternative in corrispondenza di punti di decisione (indicati come pallini). Nei rettangoli si hanno le azioni che sono svolte in conseguenza allo stimolo. Un esempio si ha in figura 6.9. I diagrammi sono tra loro complementari ma hanno anche delle intersezioni tra loro e anche con il testo e tutto deve comunque restare coerente. Si rischia di introdurre inconsistenze. Si hanno alcune regole di consistenza per i diagrammi che vengono usate da diversi strumenti per verificare la consistenza tra i vari diagrammi.
A questo punto è doveroso citare uno degli standard per la modellazione di diagrammi: **Unified Modeling Language (*UML*)**. Al suo interno si hanno, con regole di coerenza tra essi:

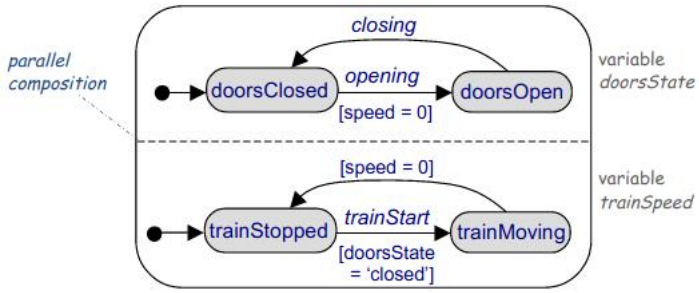
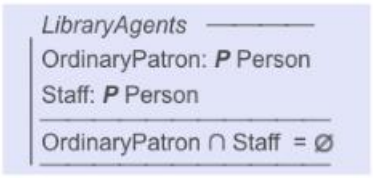


Figura 6.10: Esempio di R-net diagram

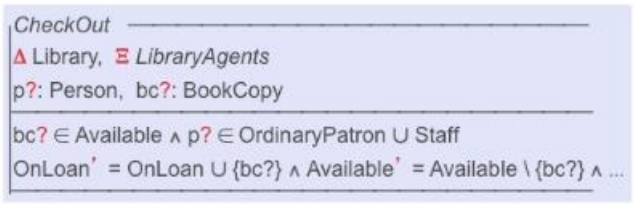
- class diagrams
- use case diagrams
- sequence diagrams
- state diagrams

In conclusione i diagrammi hanno il pro di essere in grado di dare una buona panoramica e struttura/comportamento, facile da trasmettere e comprendere, di aspetti importanti. Sono inoltre supportati da vari tool di analisi. Il contro è sempre la specifica ambigua semi-formale che può anche limitare l'analisi degli stessi. Inoltre si concentrano solo su aspetti funzionali e strutturali. Si hanno anche gli **statechart** (figura 6.10) per la descrizione di sistemi paralleli che usano la semantica interleaving ovvero per 2 transizioni che si attivano nello stesso stato, una viene presa dopo l'altra, in modo non deterministico. Si ha quindi in generale la necessità di una semantica formale per aspetti *mission-critical* e la cosa non viene garantita dai diagrammi ma si hanno linguaggi apposta (e diagrammi apposta) la cui creazione/gestione/documentazione è assai costosa per cui vengono usati solo in casi estremamente specifici. Approfondiamo quindi questi linguaggi e questa notazione.
Si parla sia di sintassi che di semantica. Tali definizioni formali permettono che il compilatore sia in grado di processare tali specifiche. Si ha quindi un livello di precisione alto e non si ha ambiguità, permettendo analisi di consistenza e coerenza delle specifiche (fatta direttamente dal calcolatore). Una sintassi/semantica formale è quella data dalla logica proposizionale e dalle **formule ben formate**. Si ha quindi un linguaggio basato sui connettori logici e sulle regole per avere formule ben formate. Ogni statement può quindi essere interpretato con il valore di verità con le regole definite dalla logica e dalle tabelle di verità (ovviamente può farlo il calcolatore). Si vede quanto

produrre tutto questo per un progetto enorme risulti costoso. Vengono quindi usate le classiche regole di inferenza. Si usa anche la **logica predicativa del primo ordine**.
Un altro strumento usato è la **logica temporale** con l'aggiunta dei connettivi temporali.
Si ha un approccio formale anche su specifiche state-based dove si specifica in modo formale cosa sia l'insieme degli stati che il sistema può attraversare e come delle operazioni modificano lo stato. Si usano linguaggi come Z basati sulla teoria degli insiemi. Ogni stato è caratterizzato da pre e post condizioni e si studiano le regole in termini di invarianti che lo stato deve soddisfare. Con Z si producono schemi come:



dove si specifica un certo elemento, gli attributi di stato con P che indica l'*insieme delle parti* che precede il tipo. Si ha poi un invariante per gli attributi che definisce l'elemento. Si ha quindi una caratterizzazione basata sulla teoria degli insiemi. Si hanno sia tipi primitivi che strutturati. Per definire invece le modifiche che portano a nuovi stati, ovvero per definire le operazioni si usa una simile sintassi:



Sopra si hanno tutti gli elementi coinvolti nell'operazione. La prima linea della parte sotto è per le precondizioni, quella sotto per le postcondizioni, entrambe con la notazione insiemistica. Gli elementi sono inoltre così decorati:

- `stateVar?:Type` indica una variabile di input
- `stateVar':Type` indica una variabile di output mutevoli (?)
- `stateVar!:Type` indica una variabile di output esterne e non mutevoli (?

- Δ `schema` indica che si ha una modifica delle variabili importate dallo schema
- Ξ `schema` indica che non si ha una modifica delle variabili importate dallo schema

Il cambiamento di stati ben definito diventa studiabile e simulabile, studiando lo spazio di comportamenti del sistema a questo livello di astrazione, individuando problemi che porterebbero alla correzione di requisiti. Si può avere anche l'inclusione di più schemi. Con questa specifica state-based si hanno vari pro:

- automazione semplice grazie a logica, matematica discreta, studio degli invarianti etc...
- ottimi meccanismi di strutturazione per comporre unità e strutture stati complessi
- permette l'analisi automatica: type checking, consistency checking, etc...

Di contro non permette di studiare altro se non aspetti funzionali e non ha un'historical referencing (anche se si hanno vari workaround per l'encoding di stati passati).
Si ha anche una **specifica algebrica** per formalizzare le leggi che compongono le operazioni, viste come funzioni matematiche senza esplicita nozione di stato. Si ha invece un system history avendo una "traccia" delle operazioni. Si hanno tipi di dato astratti, parametrizzazione, equazioni condizionali e funzioni parziali. Si hanno tre tipi di operazione (*capire bene questa cosa su slide*):

1. modifiers, per produrre qualsiasi istanza di concetto mediante composizione con altri modifier
2. generators, ovvero un sottoinsieme minimo di modifiers per generare qualsiasi istanza di concetto con un numero minimo di composizioni
3. obsevers, per ottenere informazioni pertinenti su qualsiasi istanza di concetto

Spesso si hanno funzioni ricorsive.
Vari esempi su slide.
Come vantaggi si hanno:

- analisi automatica efficiente
- specifiche eseguibili
- ricchi meccanismi di strutturazione (import, ereditarietà etc...)

Di contro si ha una limitata potenza espressiva (solo equazioni senza historical referencing) e una forte vicinanza alla programmazione (comportando difficoltà di validazione in caso, ad esempio, di ricorsioni).
Tendenzialmente i limiti della notazione formale sono sempre questi (oltre alla difficoltà di lettura/scrittura), anche se permette alta precisione, pochi difetti di specifica, analisi sofisticata (anche automatica), generazione di artefatti come test cases, codice etc.... Ricapitolando:

- il linguaggio naturale "puro" non è un'opzione
- il linguaggio naturale strutturato è quello più usato
- i diagrammi sono un'ottima aggiunta al linguaggio naturale
- la notazione formale è usata in rare circostanze

6.3.4 Validation & verification

Siamo quindi all'ultima delle quattro fasi, dove bisogna validare i requisiti.
Si hanno vari approcci per validare la qualità dei requisiti:

- analisi e revisione dei singoli requisiti. Questo è sempre applicabile ma va fatto manualmente impiegando molto tempo. In primis bisogna ricercare gli errori nel RD. Spesso questi sono gli errori più numerosi e pericolosi e possono avere varia natura:
  - omissione
  - contraddizione
  - inadeguatezza
  - ambiguità
  - incommensurabilità
  - rumore
  - eccesso di specificità
  - scarsa struttura
  - opacità

Bisogna quindi individuare quanti più problemi possibili nell'RD, validarli (vedendo se le informazioni utili sono necessarie), verificarli (vedendo se sono completi e consistenti) e confermare non ambiguità, misurabilità, fattibilità, buona struttura, etc... Bisogna quindi fare il report dei problemi, analizzarne la causa e correggerli.
Questa operazione viene fatta selezionando personale che ispeziona l'RD individualmente per poi confrontare le opinioni. Tali persone possono essere interne ai project members o recensori esterni (tramite poi incontri ben prepari report strutturati etc...). Questo viene spesso fatto anche per il codice sorgente e quindi si è mostrato che è empiricamente utile anche per le revisioni dell'RD.
La revisione può essere effettuata in vari modi:

- **free mode**, ovvero senza direttiva su cosa cercare e dove
- **checklist-based**, ovvero indicando una lista di cose da controllare
- **Process-based**, dove si hanno ruoli specifici, procedure dettagliate, tecniche di analisi etc..., rendendo questa la modalità più efficace

Il report deve essere fatto in modo informativo e accurato, senza opinioni o commenti offensivi (anche se può essere lasciato uno spazio per i commenti liberi). Ovviamente le revisione deve essere fatta possibilmente da personale diverso da quello che scrive il RD e tale personale deve rappresentare tutti gli stakeholder con le varie conoscenze pregresse.
A livelli di tempo il controllo deve essere effettuato ne troppo presto ne troppo tardi ma con incontri ripetuti per ottenere la massima efficacia.
Ci si deve inoltre concentrare sulle parti più critiche, che devono essere meglio analizzate.
Dal punto di vista delle checklist se ne hanno diverse:

- **Defect-driven**, con una lista dei difetti tipici, strutturate tramite un elenco di domande generiche
- **quality-specific**, con una lista più specializzata di quella defect-driven, specializzandosi su requisiti non funzionatali (sicurezza, prestazioni, usabilità etc...)

  - **domain-specific**, con una lista specializzata nei concetti e nelle operazioni di dominio per aiutare nella ricerca dei difetti
  - **language-based**, con una lista specializzata nei costrutti legati ai linguaggi (come la correttezza dei tipi, il rationale etc...). In questa parte si analizzano anche i diagrammi (presi singolarmente o nel complesso, che quindi può essere parzialmente automatizzata). Si analizzando anche linguaggi di specifica come Z.

Su slide esempi.
Si usano anche delle **checking decision tables** che hanno in input N condizioni (che saranno booleane nella tabella $\{T, F\}$) e una serie di eventi che si hanno in conseguenza allo stato delle condizioni, specificando se accadono o meno. Si ha che se il numero di combinazione dei vari casi specificati dalle condizioni in input è minore 2^N (se ho meno colonne del dovuto nella tabella di verità)allora mancano dei casi da analizzare mentre se è maggiore si hanno dei casi ridondanti.
Questo tipo di analisi è quindi applicabile potenzialmente alla ricerca di ogni difetto (anche se non si ha garanzia di riconoscerli tutti, per quanto critici, anche se con un mix delle tecniche spiegate si raggiungono ottimi risultati). I costi restano elevati sia in termini monetari (pagare i revisori) che di tempo

- interrogare il db delle specifiche. Questo, a livello superficiale, può essere fatto in modo automatico ma è appunto un check parziale. Questa tecnica è specifica per casi particolari, dove le specifiche sono memorizzate in un db (con quindi uno schema logico derivato dai diagrammi). Le query acquisiscono controlli per la coerenza strutturale intra o inter-diagrammi e tali query possono essere generate a partire dall'ER. Si parla anche di **press-button mode** quando si ha una lista di query prescritte per la violazione delle regole di coerenza standard

Esempio su slide

- usare le **specification animation**, ottime per i non esperti per identificare i problemi ma sono limitate alle specifiche "eseguibili",

consentendo quindi solo un check parziale. Si vuole verificare l'adeguatezza dei requisiti rispetto alle effettive necessità. Si hanno due approcci tipici:

1. mostrare una vera interazione cono lo scenario. In questo caso gli strumenti di "promulgazione" possono essere utilizzati sul diagramma NET (???) ma ovviamente si ha il problema del range di copertura dei problemi dato dallo scenario
2. usare tool per l'animazione delle specifiche. Si procedere generando un modello eseguibile e si procede alla simulazione (provvedendo a stimoli e vedendo il risultato) per poi raccogliere il feedback dell'utente. Si può avere:
   - formato testuale, con comandi di input e poi esecuzione
   - formato a diagrammi, con un input che comporta l'evoluzione di parti del diagramma (in stile token per le reti di Petri)
   - scena vera e propria con pannelli per l'input e scene animate nell'ambiente scelto

In ogni caso si ha un approccio **model-based**, si parla infatti di **model-driven development** (sviluppando fin dall'inizio modellando il comportamento del software), e si hanno vari tool per i vari linguaggi di specifica (SCR, LTSA etc...).
Tra i pro si hanno:

- è modo migliore per verificare l'adeguatezza rispetto ai bisogni reali, all'ambiente reale
- permette la facile interazione con gli stakeholder secondo la filosofia **WYSIWYC (What You See Is What You Check)**
- estendibile per animare controesempi generati da altri strumenti (come ad esempio tramite i model checkers)
- le animazioni possono essere riusate per altri scenari

Si hanno anche contro:

- si richiede un'attenta progettazione degli scenari che possono presentare problemi e non si ha garanzia di non trascurare problemi critici

  - necessita specifiche formali

- il check formale. Questo permette di individuare un ottimo range di problemi ma richiede la notazione formale, costosa, ed esperti. Si possono avere check sintattici per il linguaggio (di specifica come Z) usato (syntax checking, type checking, static semantics checking, le variabili utilizzate devono essere dichiarate e inizializzate, devono essere utilizzate nello scope corretto etc. . ., circularity checking, con il check delle postcondizioni etc. . .). Si può avere anche un check di completezza e consistenza grazie al linguaggio e la verifica di proprietà:

  - algoritmiche tramite **model checking**, per controllare se un modello di comportamento soddisfa un requisito, una proprietà di dominio o un presupposto, cercando violazione delle varie proprietà e generando eventuali controesempi o comunque specifiche della violazione (fattori utili nel debugging). Si possono controllare:

    * **raggiungibilità**, tramite un grafo di raggiungibilità
    * **proprietà di safety**, producendo controesempi
    * **proprietà di liveness**, che comporta che una condizione potrebbe non essere mai raggiunta in caso di risposta negativa del check

    I pro del model checking sono:

    * check completamente automatici
    * ricerca esaustiva che porta a non tralasciare alcun difetto
    * l'uso di controesempi può rivelare errori sottili, difficili da individuare altrimenti
    * è facile da abbinare agli animators per visualizzare le tracce
    * è sempre più usato per progetti mission-critical

    Ma si hanno anche dei contro:

    * si rischia di avere un'esplosione combinatoria degli stati da analizzare comportando

l'impossibilità di essere eseguito per sistemi molto grossi (anche se questo problema può essere limitato tramite astrazione, limitando però accuratezza e completezza)

* i controesempi possono essere complessi da capire e mostrano solo i sintomi dei problemi, non le cause

– deduttive tramite **theorem proving**. Il theorem proving viene usato per generare nuove specifiche tramite le regole di inferenza della logica. Si studiano le conseguenze logiche delle specifiche evidenziando eventuali inconsistenze in un processo generalmente iterativo per:

  * fornire, accettare o rifiutare i lemmi
  * suggerire strategie di prova

  Su slide esempio con Z.
  Si hanno diversi pro:

  * solidità e completezza del sistema formale utilizzato avendo che ogni conclusione è corretta e che ogni conclusione corretta è derivabile
  * mostra specifiche incoerenti, conseguenze inadeguate
  * può essere usato per sistemi molto grandi, gestendo una gran mole di dati tramite l'induzione

  ma si hanno ovviamente dei contro:

  * è di uso difficile e richiede personale esperto
  * non produce controesempi

Si hanno, per i check di consistenza quando servizi o comportamenti sono specificati come relazioni I/O capendo:

– se la relazione e una funzione (per isolare eventuali nondeterminismi)

– se la funzione è totale e quindi in grado di specificare ogni input

Esempio con Z su slide.

6.3.5 Requirements changes

I cambiamenti dei requisiti sono assai frequenti e quindi vanno gestiti. Si hanno infatti varie iterazioni delle 4 fasi sopra descritte, formando la spirale. In primis si hanno cambiamenti organizzativi, nuove normative, nuove opportunità, tecnologie alternative, priorità e vincoli in evoluzione oltre ad una migliore comprensione delle caratteristiche, dei punti di forza, dei limiti e dei difetti del sistema. Un altro aspetto riguarda il problema di gestione delle informazioni, in merito almantenimento della coerenza, propagazione delle modifiche, controllo delle versioni etc. . ..
La gestione dei cambiamenti di requisito consiste quindi nel processo di anticipazione, valutazione, accordo e propagazione dei cambiamenti nelle varie parti del RD.
Innanzitutto bisogna dentificare i cambiamenti probabili, valutare la probabilità e documentarli al fine di:

- anticipare una risposta adeguata quando si verificherà il cambiamento, studiando eventuali dipendenze tra i requisiti
- progettare l'architettura che rimanga stabile nonostante i cambiamenti

Inoltre, associando livelli di stabilità a gruppi di statement che definiscono le varie feature. Si punta ad avere un numero di livelli comunque ridotto e di cercare di permettere la comparabilità, che sia qualitativa e relativa.
Si usano regole euristiche per studiare i probabili cambiamenti, concentrandosi sui requisiti che è più facile cambino, limitando così i costi. Il **livello più alto di stabilità** viene definito per quelle feature presenti in ogni estensione/variante del sistema. Si ha che aspetti intenzionali e concettuali, così come aspetti funzionali e obiettivi chiave, sono più stabili di aspetti operazionali e non funzionali (per esempio la comparsa di una notifica è più stabile del modo in cui questa viene sviluppata. . . importante è che ci sia la notifica). La scelta tra varie alternative rende l'oggetto della scelta poco stabile anche perché tale scelta può essere basata su una conoscenza incompleta, presupposti volatili o generata da risoluzione di conflitti o contromisure a vari rischi. Quindi tali requisiti sono poco stabili.

Un altro aspetto essenziale nello studio dell'evoluzione del progetto è legato alla gestione della tracciabilità (detto anche **Traceability Management (*TM*)**), che non è uno studio semplice nel caso dei requisiti ma torna comodo nel momento dei loro cambiamenti per gestire il cambiamento dell'intero progetto. Si ha che un certo aspetto è tracciabile sse si sa perfettamente:

- da dove viene
- perché esiste
- per cosa sarà usato
- come sarà usato

Possiamo quindi dire che il TM si occupa di identificare, documentare, recuperare la logica e l'impatto degli elementi contenuti nel RD.
Si possono avere anche tracciamenti in merito o obiettivi specifici per determinati requisiti, valutando l'impatto di eventuali modifiche e studiando come propagare facilmente i cambiamenti per mantenere la coerenza ad altri elementi dell'RD o, "scendendo di livello", ad oggetti software.
Un aspetto centrale nel TM è rappresentato dai collegamenti di tracciabilità tra gli elementi del RD. Tali collegamenti vanno scoperti, memorizzati ed eventualmente recuperati. Tali collegamenti sono bidirezionali e, ipotizzando uno schema verticale tra le varie fasi, si hanno:

- accessibilità dal source al target, detta **forward traceability** (che nello schema verticale sono le frecce verso il basso). Il source motiva l'esistenza del target
- accessibilità dal target al source, detta **backward traceability** (che nello schema verticale sono le frecce verso l'alto)

All'interno di una stessa fase posso avere anche collegamenti (che in questo caso non vengono direzionati a differenza dei due appena espressi e sono orizzontali). Dipendenze tra cambiamenti di fase sono sempre rappresentatati da una linea verticale.
la backward traceability viene usata per stabilire perché un certo elemento sia lì e da dove viene (anche ricorsivamente) mentre la forward per stabilire dove un elemento verrà considerato e con quali implicazioni (anche ricorsivamente). Ricorsivamente perché posso andare indietro/avanti di più step. Bisogna localizzare e valutare l'impatto dei cambiamenti lungo i collegamenti orizzontali/verticali.
Elencando le varie tecniche TM, per tracciare i vari collegamenti, si hanno:

- cross referencing
- matrici di tracciabilità
- feature diagrams (nel dettaglio quelli con supporto a più variant link type)

- db di tracciabilità
- db di modelli di tracciabilità
- tracciabilità Specification-based

È comunque un discorso molto complesso, costoso e studiato.
Approfondiamo il **cross referencing**.
Con questa tecnica si seleziona ogni elemento da tracciare e gli si assegna un nome unico. Si definisce poi lo schema di indice/tag per collegarli lessicalmente e si configura un motore di ricerca su tale schema. Si recuperano quindi gli elementi seguendo le catene di riferimenti incrociati, le *cross-reference chains*. Un esempio banale in un documento potrebbe essere specificare che sezione dello stesso documento andare a guardare per chiarire un certo concetto.
Come pro si hanno:

- la leggerezza e la disponibilità
- il supporto di ogni livello di granularità (assegnando identificatori a ciò che vogliamo)

Come contro:

- un solo tipo di collegamento, quello lessicale, privo di semantica
- informazioni di tracciabilità nascoste
- il costo del mantenimento dello schema di indicizzazione (anche se si ha un basso costo iniziale)
- controllo e analisi limitate

In merito alla **matrice di tracciabilità** abbiamo che essa è la matrice di adiacenza di un grafo a singola relazione di tracciabilità, detto **dependency graph**. In posizione i, j troviamo 1 se si ha un arco orientato tra l'item T_i e l'item T_j, 0 altrimenti. Possiamo quindi dire che sulla riga i-sima troviamo gli elementi che dipendono da T_i e sulla colonna i-sima gli elementi che sono dipendenza di T_i.
Come pro si hanno:

- navigazione backward e forward
- semplice analisi dello schema (ad esempio è semplice identificare un ciclo di dipendenza, avendo un ciclo nel grafo)

Come contro:

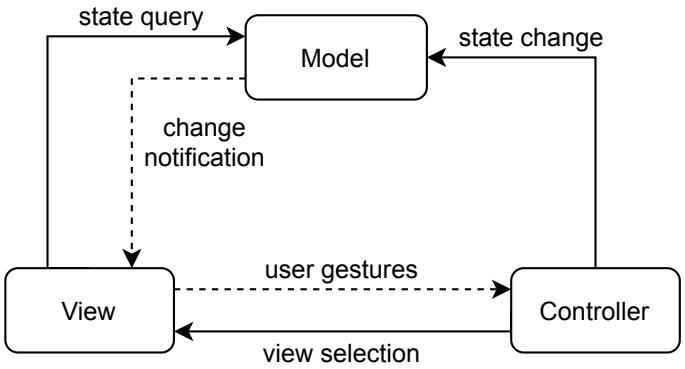
- grafi di grandi dimensioni sono soggetti ad errori oltre ad essere difficilmente gestibili
- supportano un solo tipo di relazione
- dovrei creare più matrici per collegamenti tra oggetti con semantiche diverse

Quest'ultimo contro può essere risolto coi **feature diagrams** che sono rappresentazione grafica dei punti in comune e delle varianti del sistema, permettendo la rappresentrazione compatta di un gran numero di varianti. Avere più varianti permette di distribuire il sistema con un ampio numero di feature (una versione free per una certa piattaforma, una non free per un'altra etc...). Nel diagramma le feature sono rettangoli e possono essere obbligatorie (con un pallino nero) o opzionali (con un pallino bianco). Si hanno anche potenziali *or esclusivi*, *or non-esclusivi* e *and* per separare in diversi modi feature in sotto-feature. Le varie combinazioni espresse dalle feature del digramma, seguendo la logica di opzionali o meno e delle varie suddivisioni esprimono le varianti del progetto. **Esempio di diagramma su slide**.

Capitolo 7

Model-View-Controller

Il **Model-View-controller (*MVC*)** è un pattern architetturale molto diffuso nello sviluppo di sistemi software, in particolare nell'ambito della programmazione orientata agli oggetti e in applicazioni web, in grado di separare la logica di presentazione dei dati dalla logica di business. Partiamo analizzando il pattern architetturale:



Dove con le frecce piene si hanno le invocazioni di metodi e con quelle tratteggiate gli eventi. Analizzando meglio le tre componenti si ha che:

1. **Model** incapsula lo stato dell'applicazione, risponde alle query di stato, espone le funzionalità dell'applicazione e notifica la vista delle modifiche. In altri termini fornisce i metodi per accedere ai dati utili dell'applicazione. È implementato dalle classi che realizzano la logica applicativa, possiamo dire che sia il "backend"

2. **View** renderizza il modello, richiede l'aggiornamento dai modelli, invia le operazioni dell'utente al Controller e gli permette di selezionare la vista. In altri termini visualizza i dati contenuti nel

model e si occupa dell'interazione con utenti e agenti. Presenta lo stato del model all'utente, magari con una GUI

3. **Controller** definisce il comportamento dell'applicazione, mappa le azioni dell'utente ad update del Model, seleziona le viste per le risposte. In altri termini riceve i comandi dell'utente (in genere attraverso il View) e li attua modificando lo stato degli altri due componenti. Si ha un Controller per funzionalità. È un mediatore tra View e Controller

La View raccoglie gli input dell'utente e li inoltra al Controller, che li mappa in operazioni sul Model, che viene modificato. A questo punto il Controller seleziona la nuova View da mostrare all'utente, che a sua volta interagisce per avere i dati con il Model. Cambi del model sono notificati alla View per eventuali cambiamenti dei dati.

Esempio 10. *Vediamo un esempio pratico semplice per una pagina HTML:*

- ***Model**: i file HTML e CSS*
- ***View**: il browser*
- ***Controller**: il web server*

Esempio 11. *Vediamo un esempio pratico semplice per una pagina dinamica:*

- ***Model**: la logica applicativa (la business logic), per esempio implementata in Java*
- ***View**: il render dei dati, tramite magari pagine JSP*
- ***Controller**: la logica di presentazione, magari tramite Servlet*

In questa parte si ha il ripasso di alcuni concetti utili.
Ricordiamo che gli **HTML forms** permettono di passare informazioni, formate da `URI` più dati, dal browser all'applicazione web. Un form è formato da:

- `ACTION` per processare i dati
- `METHOD` il metodo in cui mandare le informazioni:
  - `GET`, dove si manda solo l'`URI`, coi dati contenuti nell'`URI` stesso

  - `POST`, dove i dati sono nel corpo della richiesta direttamente

Servlet è una componente web basata su Java fatta per generare contenuto dinamico. Una servlet ha il seguente ciclo di vita:

- non esiste
- caricamento tramite il costruttore
- inizializzazione tramite un **init**
- esecuzione della richiesta (il `.service()` limita il tempo di vita della `request`)
- distruzione

Si hanno metodi come `doPost` e `doGet`. Per ottenere informazioni da un form si ha `getParameter(param)`, con `param` che può specificare un elemento della form, come metodo della request.

Esempio 12. *Vediamo un esempio di servlet:*

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.h = p.*;
public class Hello extends H = pServlet {
  public void doGet(H = pServletRequestreq,
                    H = pServletResponse res)
          throws ServletExcepion, IOExcep3on {

    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    String name = req.getParameter("name");
    out.println("<HTML>");
    out.println("<HEAD><TITLE>Hello,    " +
                name + "</TITLE></HEAD>");
    out.println("<BODY>");
    out.println("Hello,    " + name);
    out.println("</BODY></HTML>");
  }
```

Un **cookies** è una piccola stringa nominata salvata sul client. I cookie possono essere usati per:

- per identificare l'utente con una sessione web di lunga durata
- memorizzare alcuni dati di convenienza sul client (nickname, id, numero di carta di credito, etc...)
- personalizzare il sito in base al profilo utente (avendo banner mirati)

Il browser restituisce una serie di cookise al sito da cui provengono e la web app le usa nel suo processo logico. Per creare un cookie si può fare:
```
Cookie c = new Cookie('name', 'value');
resp.addCookie(c);
```
e per ottenere un cookie:
```
Cookie cArray[] = req.getCookies();
```

Definizione 8. *Definiamo* ***sessione*** *come una serie di richieste associate tra loro*

A causa della natura del protocollo HTTP è quindi difficile mantenere una sessione e si usano diversi meccanismi di tracciamento:

- cookies
- riscrittura dell'URL
- form nascosti

In ogni caso è il server a svolgere il lavoro:

- ottenere la sessione:
  `essionsession = req.getSession(true);`
- salvare le informazioni delle sessione nella sessione corrente (ad esempio per un carrello in un e-commerce etc...), ad esempio:
  `session.setAttribute(name, valueObject);`
- recuperare informazioni dalla sessione, ad esempio:
  `session.getAttribute(name);`

Si hanno diversi luoghi dove una servlet memorizza i dati a seconda dello scopo:

- nella **request**, nel caso dell'esecuzione di `.service()`
- nella **sessione**, per multiple request dello stesso utente
- nel **servletContext**, per i dati riguardanti l'intero applicativo

le servlet ragionano per *container*, avendo una servlet per container che è condivisa tra più utenti e richieste. Bisogna quindi preoccuparsi del multithreading e fare attenzione quando si accede ai dati nella sessione e nel SessionContext.
Java Server Pages (*JSP*) è usato per includere elementi dinamici in una pagina web statica. Le pagine JSP vengono compilate in servlet Java prima di essere eseguite secondo il seguente processo:

1. si ha la request
2. si genera la servlet dalla pagina JSP
3. si salva la servlet per usi futuri
4. si esegue la servlet

Con `<%= %>` si visualizzano le variabili Java (sono le *JSP expression*) e con `<% %>` si include codice Java (sono le *JSP Scriplet*).

Esempio 13. *Vediamo due esempi di codice, prima con una expression:*

```
<HTML>
<BODY>
Hello, sailor, it is now <%=new java.util.Date().toString()%>
</BODY>
</HTML>
```

e con uno scriptlet:

```
<HTML>
<BODY>
<% java.util.Date theDate = new java.util.Date(); %>
<% int theHour = theDate.getHours(); %>
<% if(theHour<12)  %>
Good morning,
<%  else  %>
Good afternoon,<%  %>
sailor. It is now <%= theDate.toString() %>.
</BODY>
</HTML>
```

In JSP si hanno variabili predefiniti per `request, response, out, session,`etc... e dei tag di azioni standard come `<jsp:useBean>, <jsp:getProperty>, <jsp:setProperty>`. Si hanno anche librerie per i vari tag, includibili con: `<%@taglib uri="URIToTagLibrary" prefix="tagPrefix"%>`.
Esempio grosso su slide.

Torniamo a parlare di MVC.
Durante la triennale (ad APS) si sono visti i design pattern della **Gang Of Four (*GOF*)** che erano i design pattern relativi al Model.
Dal punto di vista del **Controller design** si hanno vari design pattern:

- **Page Controller**, che si occupa di controllare i parametri delle richieste (tipo una `POST`) e richiamare la business logic sulla base della richiesta, di selezionare la view successiva da mostrare e di preparare i dati per la presentazione. Si ha il seguente flusso di controllo:

    1. il controllo si sposta dal server Web al Page Controller
    2. il Page Controller estrae i parametri dalla richiesta
    3. il Page Controller utilizza alcuni oggetti di business
    4. il Page Controller decide la view successiva
    5. il Page Controller prepara i dati da mostrare nella view successiva
    6. il Page Controller porta il controllo (e i dati) alla view

    Dal punto di vista implementativo si possono usare soluzioni con poca logica di controllo (ASP; JSP, PHP etc...) o con una forte logica di controllo (CGI scritp, servlet etc...). Con questo pattern si implementa un controller per ciascuna "pagina", avendo, per la scalabilità, la crescita finita del controller proporzionale al numero di "pagine" necessarie, producendo un numero comunque finito e gestibile di file di codice piccoli.

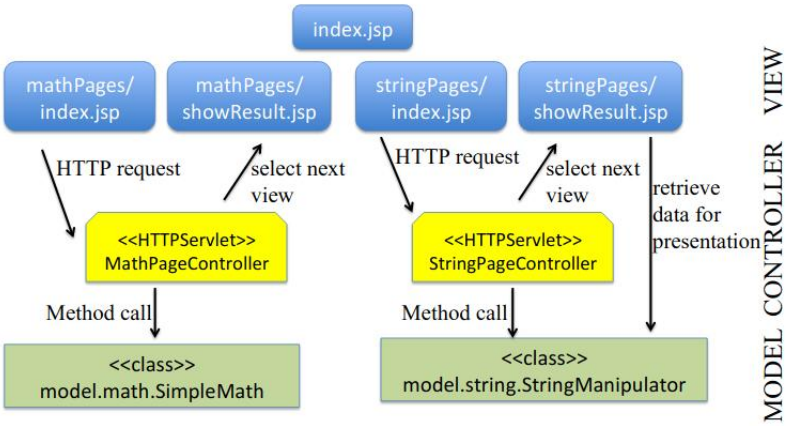
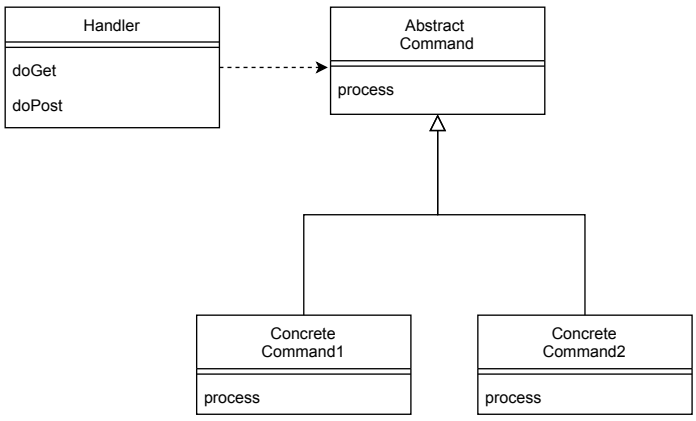


Figura 7.1: Esempio di page controller, con una semplice implementazione con due controller (uno per la matematica e uno per le stringhe), implementati come servlet. Si nota anche il collegamento diretto tra la view e il model

- **Front Controller** che definisce un singolo componente che gestisce tutte le richieste, eseguendo le operazioni comuni a tutte le richieste (come autenticazione, questioni di sicurezza, tracciamento etc...) e poi esegue la specifica operazione che è stata richiesta, passando il parametro come parametro. Ogni comando è rappresentato da una classe, con un metodo che esegue il comando (ad esempio `process`), interagendo con la logica applicativa per svolgere quanto deve. Vediamo quindi l'UML associato:



anche nel caso di un front controller si potrebbero avere più controller (ma solitamente se ne ha uno solo).

Cresce la gerarchia di comandi e non l'handler, avendo quindi controllo sulla scalabilità. Nel dettaglio l'*handler*:

- riceve la richiesta dal server
- esegue operazioni generali/comuni a tutti i comandi
- decide l'operazione che deve essere eseguita e alloca l'istanza del comando
- delega l'esecuzione al *concrete command*

mentre il *concrete command*:

- estrae i parametri dalla richiesta
- invoca metodi implementati nella business logic
- determina la vista successiva
- dà il controllo al View

Il front controller è più complesso del page controller. Inoltre evita la duplicaizone del codice tra i vari Controller (avendo mediamente un solo front controller e avendo un entry point unico per tutte le richeiste), permette una semplice configurazione del server avendo una sola servlet, permette di gestire dinamicamente dinamicamente nuovi comandi e facilita l'estensione del Controller.

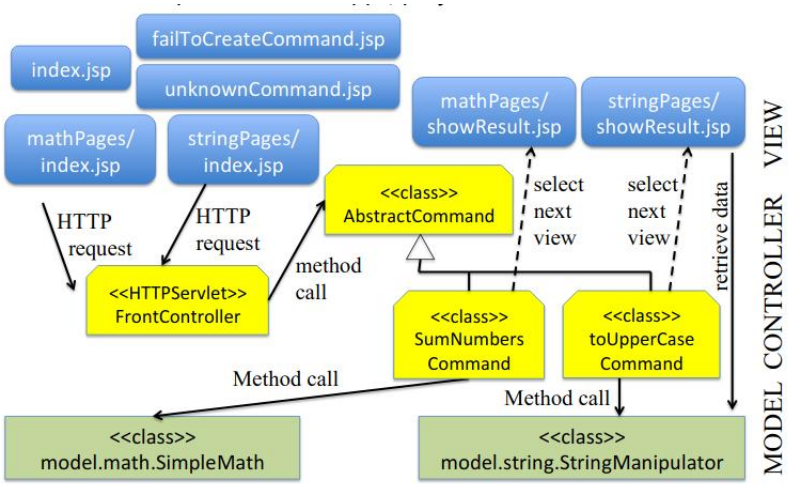
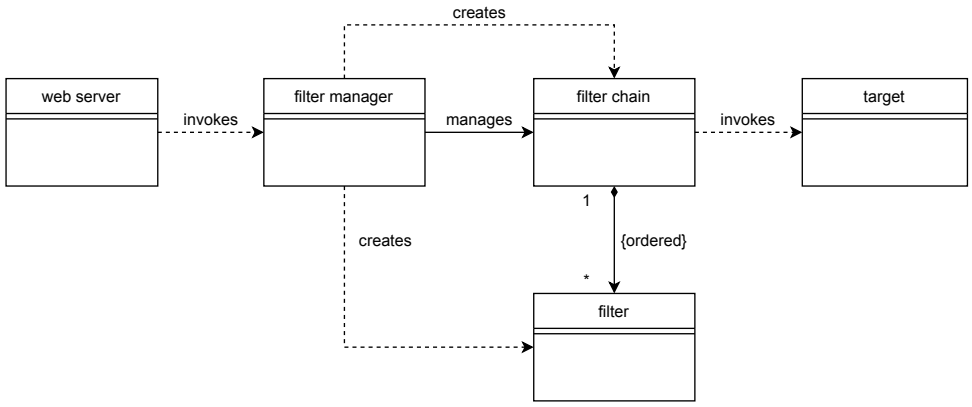


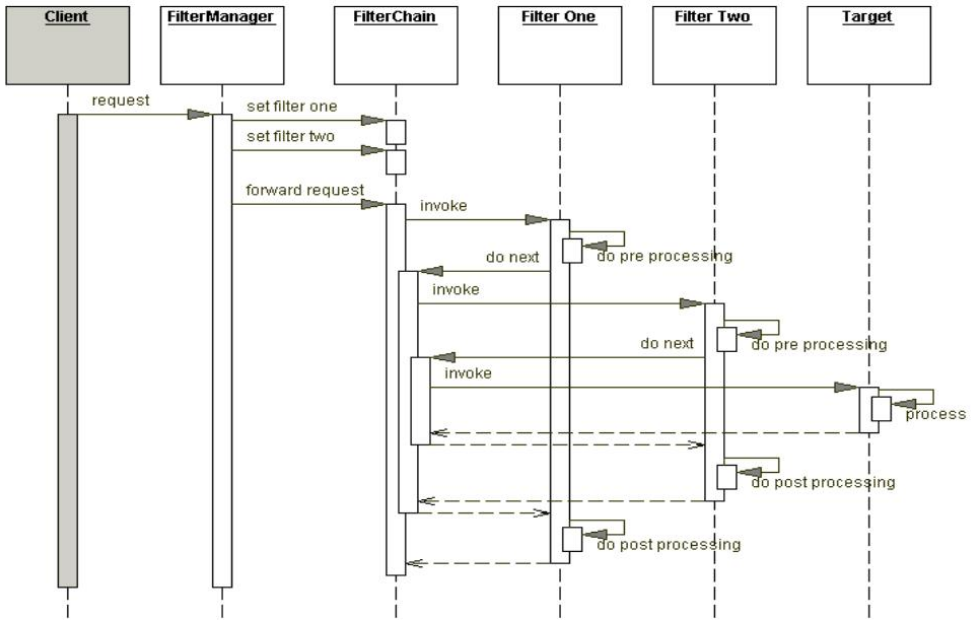
Figura 7.2: Esempio di front controller, con una semplice implementazione

Si hanno framework dove la gerarchia dei comandi è gestita diversamente

- **Intercepting Filter** che è utile per gestire richieste e risposte prima che vengano servite. Viene spesso usato insieme al front controller per "decorare" richieste e risposte con funzioni aggiuntive, come autenticazione, logging, conversione dati etc...
  Si ha quindi il seguente UML:

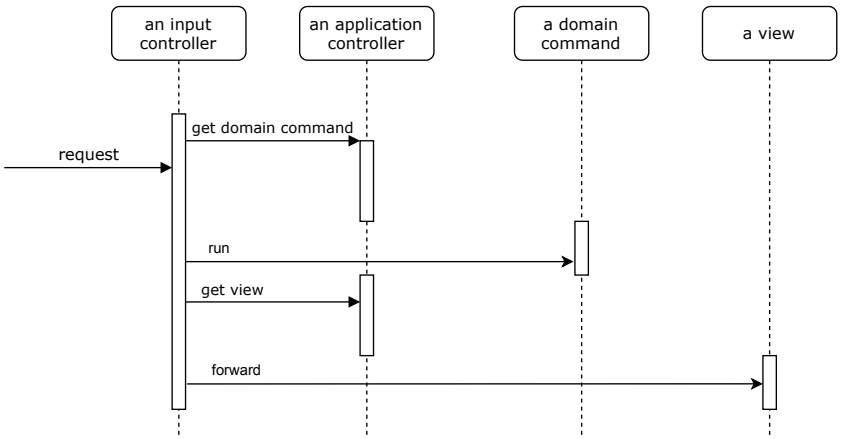


e il seguente comportamento:



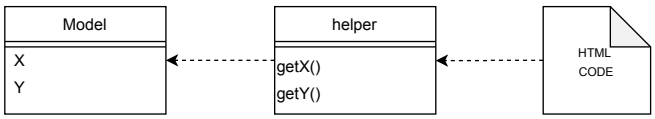
Dal punto di vista implementativo il web server fornisce *Filter Manager* e *Filter chain*, dovendo quindi implementare e dichiarare solo il filter (o più di uno)

- **Application Controller** usato in quanto all'aumentare della complessità del flusso, la gestione del flusso potrebbe essere concentrata in una classe. Solitamente viene usato dal page controller o dal front conteoller, secondo il seguente comportamento:

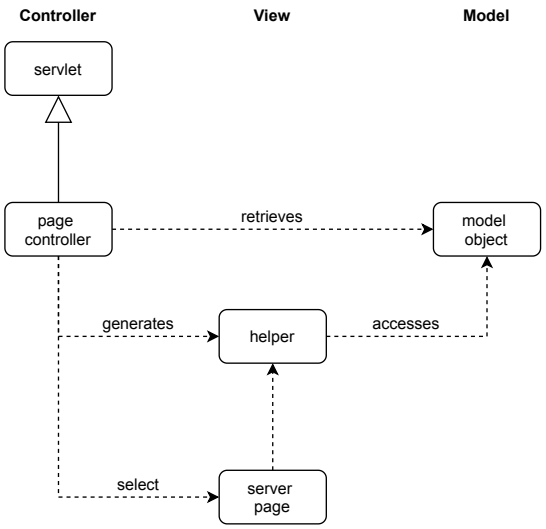


Passiamo quindi al **design view**, dove si hanno vari design pattern:

- **Template View** che genera codice HTML dalle pagine template che includono chiamate al modello, avendo:

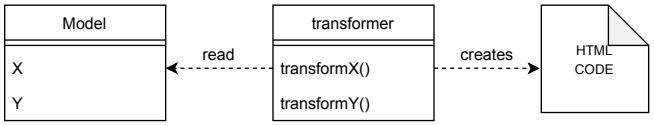


  Dove nel codice HTML si hanno vari marker e l'*helper* genera i dati del dominio e separa la vista dalla logica di implementazione. L'helper è creato dal Controller ad è accedibile da una vista. Si ha quindi un modello del tipo:



Un esempio di implementazione è JSP, dove appunto si recuperano elementi della view dai model. Si usano anche i metodi *helper* del controller per recuperare i dati dal Model, permettendo di preparare dei dati ad hoc per la parte di presentazione, tramite *getter* specifici.

- **Transformation View** che trasforma le entità di dominio in HTML:



Dove per il Model tipicamente ogni classe implementa un metodo `.toXML()` mentre per il *transformer* si ha ripicamente l'implementazione tramite **eXtensible Style Language for Transformations (*XSLT*)**. Si ha quindi che ci si concentra sull'entità che deve essere trasformata, piuttosto che sulla pagina di output. Si ha una produzione interamente dinamica della pagina da parte del transformer. Si hanno specifiche regole di trasformazioni per passare da XML e XSTL ad HTML, con regole a seconda dei tag. Purtroppo è difficile includere la logica di implementazione nella view anche se il testing è facile (magari controllando il flusso dopo la trasformazione XSLT). È facile da applicare a dati in formato XML anche se **What You See Is What You Get (WYSIWYG)** sarebbe più intuitivo e facile da implementare. Al contrario

di template view non vediamo cosa stiamo facendo fino a che non visualizziamo. Template view è usato spesso per costruire solo parti di pagine.

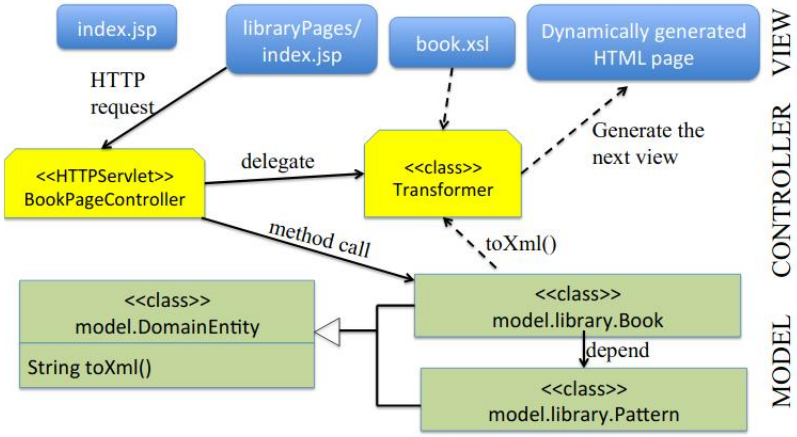
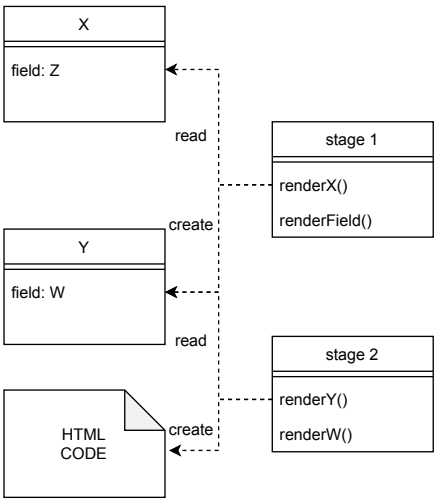


Figura 7.3: Esempio di page controller con template view, con una semplice implementazione relativa all'esempio di codice XML e XSLT che si trova sulle slide

- **Two-Steps View** dove si ha la generazione della pagina HTML in due passaggi:

    1. si genera un pagina logica
    2. si renderizza la pagina

  Si ha un modello del tipo:



Si ha che il "Look & Feel" è facile da cambiare perché il rendering non è ottenuto tramite interazione con la strutturazione delle informazioni che devono essere visualizzate nella pagina.
Dal punto di vista implementativo quindi si ha:

- una sequenza di due trasformazione XSLT, la prima per la costruzione della pagina logica e la seconda per il suo render
- una vista template con tag custom, la pagina HTML è quindi la pagina logica e i tag custom il render

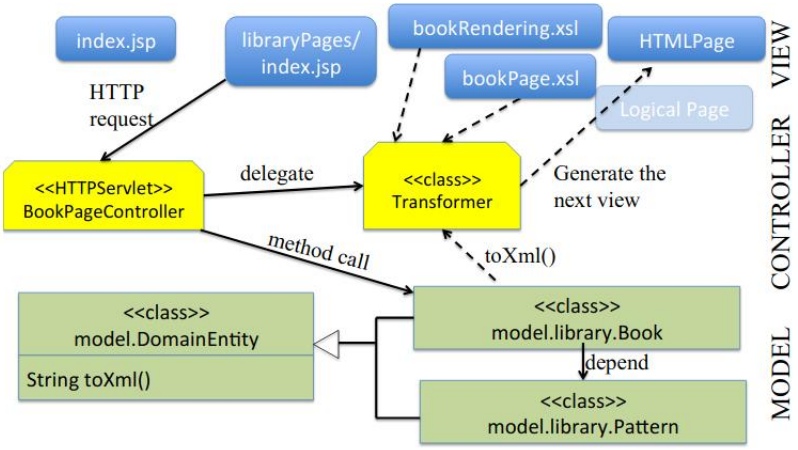


Figura 7.4: Esempio di page controller con two steps, con una semplice implementazione

Si ha *SPRING* che è un framework con l'implementazione integrata di più design pattern:

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

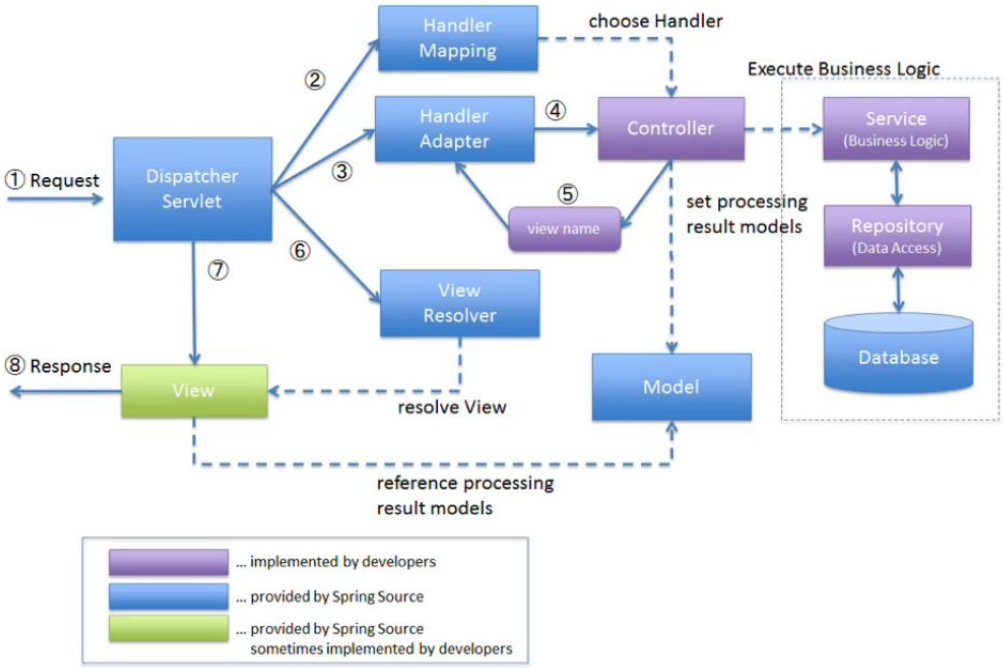


Figura 7.5: Esempio di sequenza di processo con Spring. Si hanno terminologie anche nuove. Sotto il nome di handler si ha il front controller che cattura le richieste. Il controller poi implementa la gestione della richiesta. La ricerca viene intercettata dal dispatcher servlet e viene inoltrata all'handler mapping che seleziona un controller per servire la richiesta che è stata ricevuta. Si passa poi all'handler adapter che invoca la business logic del controller selezionato (che viene da noi implementato). Poi il controller interagisce coi vari servizi del model. Il controller poi sceglie la view da visualizzare in base al risultato delle operazioni. Tale scelta arriva all'adapter che lo inoltra al view resolver che effettivamente prende la view scelta. View se necesario può accedere al modello.

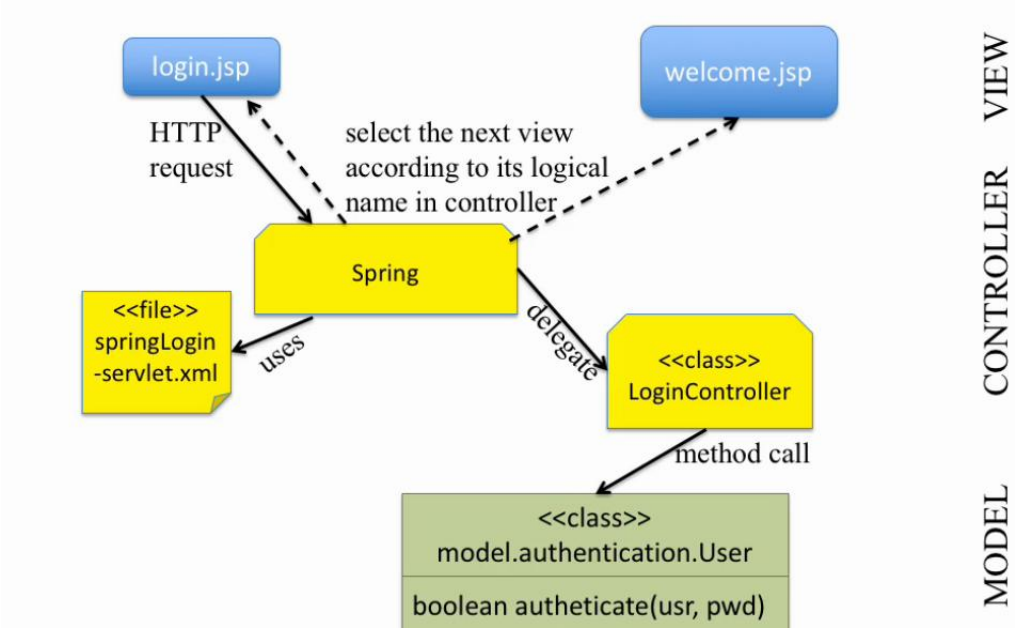


Figura 7.6: Esempio di implementazione base con Spring

Con questo framework si definisce il flusso e non la codifica in se. L'implementazione di un'operazione in genere richiede:

- implementazione della view
- implementazione dell'azione
- definizione del flusso

Si ha un file `web.xml` per permettere a SPRING di catturare le richieste, indipendentemente dall'applicazione.

7.1 Object Relational Mapping

Come sappiamo non si ha una relazione diretta tra OOP e modello relazionale e per questo si usa l'**Object Relational Mapping (*ORM*)** quando si lavora con la OOP ma con dati persistenti in un db (con le classi che vanno "mappate" nelle entità e viceversa). Si cercano implementazioni il più trasparenti possibili. Molti dei concetti si applicano ad altri mapping con modelli anche NoSQL. Si sfrutta quindi un *API gateway* tra il client e le risorse dati (il database in questo caso) che "incapsula" l'accesso a una risorsa esterna con una classe e traduce le richieste di accesso alla risorsa esterna in chiamate

all'API. Con questo approccio si nasconde l'accesso alle risorse (rendendone anche facile la modifica) e la complessità delle API.
Si hanno tipicamente 4 pattern per il **db gateway** (di cui approfondiremo solo gli ultimi due in quanto i primi due sono più primitivi):

- **Table Data Gateway**, che prevede un oggetto per tabella. Prevede classi stateless ed è comodo nel paradigma procedurale ma meno in quello OOP in quanto i metodi ritornano dati row e non oggetti
- **Row Data Gateway**, che prevede un oggetto per record, comportando che si possono avere più copie di oggetti persistenti in memoria, avendo una classe gateway che funge da alter-ego del db che viene chiamata dalla classe coi metodi *finder* (che però potrebbero essere implementati direttamente nella classe gateway).
- **Active Record**, che prevede un oggetto per record
- **Data Mapper**, che prevede un layer applicativo dedicato all'ORM

7.1.1 Active record

Questo pattern è una versione ottimizzata del *row data gateway.*
Considerate le classi che implementano i dati che devono essere persistenti si arricchiscono le classi con metodi per permettere l'interazione col db. Un **active record** include quindi:

- la business logic
- il mapping logico al db, avendo statici i *metodi finder* (invocabili avendo accesso alla classe) che comunque restituiscono risultati già nel dominio OOP (oggetti o collezioni di oggetti) e non statici i *metodi gateway* (per lavorare con le istanze)

Tra i metodi abbiamo quindi:

- **metodo load**, che crea un'istanza a partire dai risultati di una query SQL. Potrebbe essere necessario creare altri oggetti con accesso a più tabelle nel casi di reference
- **costruttore**, per creare nuove istanze che saranno poi rese persistenti invocando metodi appositi

- **metodi finder**, statici, che incapsulano la query SQL e ritornano una collezione di oggetti
- **metodi write**, tra cui si hanno:
  - **update**, per aggiornare un record a seconda dei valori degli attributi
  - **insert**, per aggiungere un record a seconda dei valori degli attributi
  - **delete**, per cancellare il record corrispondente all'oggetto in analisi

  serve sincronizzazione tra oggetto in memoria ed entità nel db. Per farlo si ha un attributo identificatore nella classe che compare anche nel record e che serve per trovare il record corrispondente ad un oggetto
- **metodi getter e setter**, che a seconda del caso devono essere subito sincronizzati con il db
- **metodo business**

Si nota quindi che è una semplice implementazione ORM senza separazione tra business logic e meccanismo di persistenza, tendendo a forzare una corrispondenza "uno a uno" tra oggetti della OOP e istanze nell'ER (anche se questo vale per molti pattern).

7.1.2 Data mapper

Con il pattern **data mapper** invece si ha un layer software indipendente dedicato alla persistenza e all'ORM, in modo che la logica di dominio non conosca la struttura del db e nemmeno le query SQL. Si usano quindi interfacce per consentire l'accesso ai mapper indipendentemente dalla loro implementazione. Framework moderni implementano questo tipo di pattern (che offre la parte di mapping senza doverli implementare ,a solo configurare).

7.1.3 Ulteriori pattern ORM

In generale si hanno dei pattern di comportamento per risolvere altri problemi ORM:

- **unit of work**

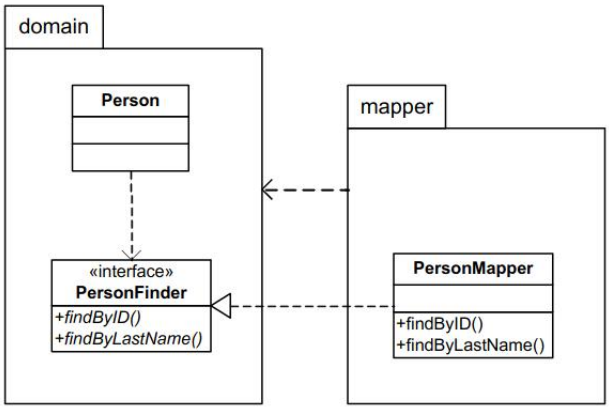


Figura 7.7: Esempio di UML di data mapper, dove si nota la separazione della classe dai metodi dedicati alla persistenza (nel mapper).

- **identity map**
- **lazy load**

e anche pattern strutturali:

- **value holder**
- **identify field**
- **embedded value**
- **LOB**

Unit of work

In questo caso si approfondiscono le operazioni ACID cercando di capire quando sincronizzare le informazioni *in memory* con il db. Questo aspetto è importante introducendo il concetto di **business transaction** per operazioni con semantica di tipo *all or nothing* dove o si accettano tutte le operazioni richieste o nessuna (si pensi ad esempio alla prenotazione di un biglietto). Le business transaction vanno quindi mappate nelle **system transaction** fatte dal sistema sul db.
Si hanno varie soluzioni (**immagini delle soluzioni su slide**):

- *aggiornare il db ogni volta che si ha una modifica in memory.* In pratica business transaction e system transaction coincidono ogni volta, dall'apertura alla chiusura oppure ogni volta che l'utente

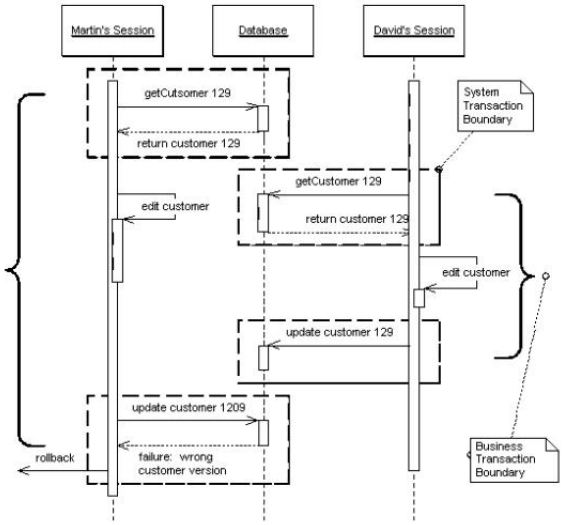
modifica qualcosa si effettua una system transaction. La prima soluzione è poco pratica a causa di transazioni molto lunghe (*long-lasting db transactions*) in quanto un perfetto isolamento implica gestire/servire una sola query alla volta mentre un isolamento imperfetto implica che si hanno transazioni che leggono dati non definitivi, comportando inconsistenze. L'utente può aprire la transazione e restarci minuti. Con il secondo caso si rischia di non poter implementare l'atomicità delle transazioni di business, che se in un certo momento fallisce non impedisce alle vecchie modifiche di essere già scritte, rompendo la *all or nothing*, avendo molte difficoltà nel fare gli *undo*

- *tracciare le operazioni (nella transazione di business) e attuare tutte le modifiche in una singola transazione di sistema.* Tutte le operazioni sul sistema si eseguono alla fine

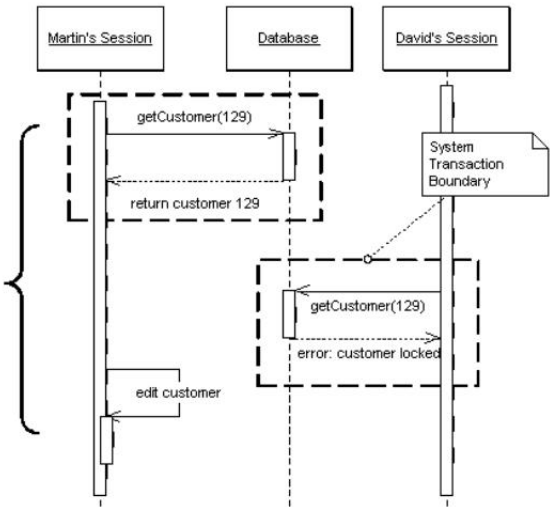
Esempio di codice Java su slide.
La unit of work quindi traccia ogni cambiamento (coi metodi *register*) e lo attuta in una singola system transaction finale (con il *commit*). Per fare questo incapsula ogni update/insert/delete, mantenendo l'integrità del db ed evitando deadlock.
A proposito di locking si ha che due business transaction possono essere attive nello stesso momento e bisogna capire come devono lavorare sul db. In questo caso si hanno due strategie:

- **optimistic locking**, usato quando si hanno poche probabilità di generare conflitti, avendo diversi utenti che lavorano spesso su dati differenti, e si bassa su un numero che identifica la versione degli oggetti registrati. Due business transaction possono quindi interferire accedendo e modificando i record ma si sfrutta il numero di versione per tenere traccia delle modifiche e se una transazione vede che il numero di versione di un record è incrementato e in tal caso si ricarica il record tramite un rollback.

Esempio di diagramma di sequenza:

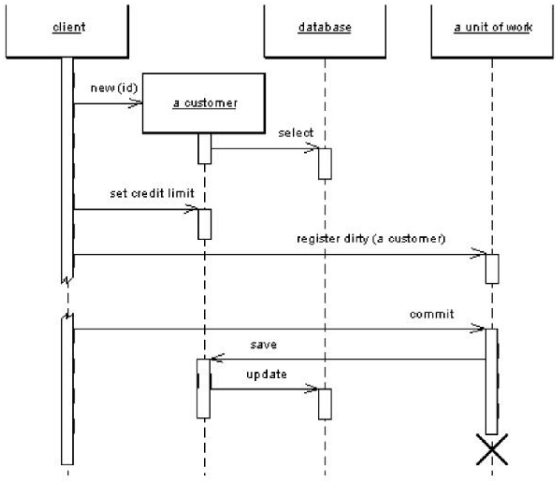


- **pessimistic locking**, usato quando hanno alte probabilità di generare conflitti. Gli oggetti sono bloccati non appena vengono usati riducendo la concorrenza. Non si permette quindi alle transazioni di lavorare concorrentemente riducendo però le prestazioni del sistema ma aumentando la sicurezza. Esempio di diagramma di sequenza:

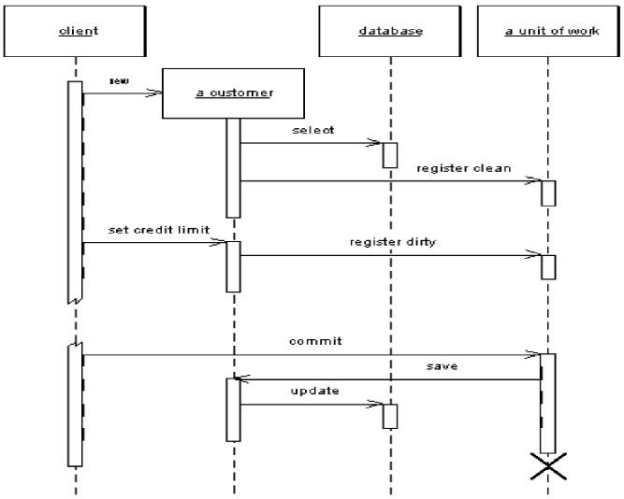


Poi, per avvisare la unit of work che un'operazione di persistenza è stata schedulata, si hanno due pattern:

- **caller registration**, dove chi modifica l'entità lo notifica direttamente alla unit of work. Questa soluzione è rischiosa in quanto il programmatore potrebbe dimenticarsi ma permette maggior flessibilità, permettendo di decidere dinamicamente se i cambiamenti devono "riflettersi" sullo storage persistente. Esempio di diagramma di sequenza (basato su active record):



- **object registration**, dove è l'entità modificata a notifica la unit of work, che però deve obbligatoriamente avere accesso globale. Ogni cambiamento viene comunque notificato. Esempio di diagramma di sequenza (basato su active record):

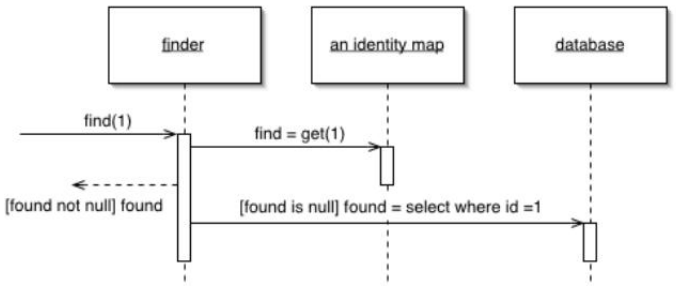


Solitamente si usa questa soluzione ma nella variante in cui le classi persistenti non includono il codice per la notifica ma tramite una strumentazione trasparente offerta dai framework

Esempio di codice Java su slide per entrambe le tecniche.

Identity map

In questo caso si studia il load di un'istanza direttamente a partire dal db. Se un certo record viene richiesto due volte si generano due oggetti uguali (esempio prima chiedo lo *user1* e poi tutti gli *user* (che ipotizziamo siano *user1* e *user2*), avendo alla fine due copie istanziate di *user1*). In pratica uno stesso oggetto può essere caricato più volte, o da azioni differenti dello stesso utente (rischiando inconsistenze, se una istanza viene modificata, oltre ad avere ridondanza di dati) o da utenti diversi (in questo spesso è intenzionale).
Identity map è quindi un oggetto con la responsabilità di identificare gli oggetti caricati in una sessione, funzionando come una sorta di cache, controllando l'eventuale presenza dell'oggetto "in cache" (anche se non si ha una vera cache) prima di caricarne uno nuovo (se già presente si ritorna un riferimento all'oggetto). Identity map quindi mantiene i riferimenti agli oggetti tramite un sistema chiave-valore. Vediamo il diagramma di sequenza:



Generalmente la chiave della mappa è quella primaria del db. Si hanno diverse identity map:

- una per applicazione, se chiavi di entità differenti sono disgiunte
- una per classe
- una per tabella, generalmente meglio se una per classe

La mappa è sempre accessibile.
Esempio di codice Java su slide.

Lazy load

A volte è necessario avere solo una parte di un oggetto per eseguire una certa operazione anche perché alcuni attributi (collection, bitmaps, grafi etc...) possono essere particolarmente pesanti. Si parla quindi di efficienza.
Si procede quindi con il pattern **lazy load** che crea oggetti inizializzando solo alcuni attributi (evitando spesso quelli più "pesanti") lasciando comunque possibilità si caricare in seguito ulteriori attributi non ancora inizializzati se ce ne fosse necessità, in modo *on-demand*, tramite i *finder*.
Su slide esempio di codice Java dedicato.

Value holder

Con questa tecnica si usa un oggetto, il **value holder**, come uno storage intermediario per un attributo. Anche in questo caso si ha un'inizializzazione "lazy" implementata da questo oggetto che caso per caso potrebbe essere istanziato con diverse strategie di load. Si separa la locgia di dominio da quella "lazy" di loa3ding.
Esempio di codice Java su slide.

Identify field

Normalmente si hanno:

- l'identità in memory rappresentata dalla reference dell'oggetto
- l'identità nel db rappresentata dalla chiave primaria

e si ha bisogno di accoppiare/sincronizzare le due identità per la consistenza dei dati.
La soluzione è usare l'**identity field** ovvero un "nuovo" attributo nell'oggetto il cui valore diventa la "chiave primaria" (solitamente chiamato `id`, di tipo `long`). Non si usano i normali attributi/identificatori della classe perché potrebbero modificare nel tempo di sviluppo del programma.
Si hanno infatti due tipi di chiave:

- una chiave derivata da attributi dell'oggetto (che sono usati anche dall'applicazione). Si ha comunque rischio di duplicazione, avere assenza di valore e avere cambiamenti sulla logica di dominio che impattano su quella di persistenza
- chiavi dedicate, sia semplici (un solo attributi) che composte (di più attributi), avendo una chiave per tabella e una per database.

Nella pratica spesso hanno tipo `long` che ben supporta il check di uguaglianza (`==`) e la generazione di nuove chiavi

Si hanno varie strategie per la generazione delle chiavi:

- direttamente dal db (con sequenze o id delle colonne)
- tramite *GUID*, ovvero Globally Unique IDentifier, tramite opportuni software e libreria
- dall'applicazione, principalmente in due modi:
  - *table scan*, usando una query SQL per determinare il valore della prossima chiave
  - *key table*, usando tabelle speciali con il nome delle altre tabelle e il valore successivo della chiave

I moderni framework ORM supportano i vari design pattern appena discussi e quelli mancanti, relativi a chiavi esterne, cascading, ereditarietà etc...
In ogni caso i pattern appena discussi sono tutti supportati dai moderni framework ORM.

Embedded value

Con questo pattern si hanno semplici oggetti, senza un chiaro concetto di identità, che possono essere inclusi in oggetti che fungono da alter-ego delle tabelle anziché creare tabelle dedicate.

LOB

Vediamo ora il pattern **Serialized Large Object (*LOB*)** che consiste in un grafo di oggetti memorizzati del db tramite serializzazione. Questo pattern è comodo finché il LOB può essere gestito come un embedded value ma si hanno problemi se vengono usati attributi serializzati nelle query o se il grafo ha dipendenze con oggetti non serializzati.
Si ha **Binary LOB (*BLOB*)**, semplice da implementare (ad esempio con *Java serialization*) e compatto. Purtroppo si ha che un BLOB non è printabile e si ha che il db deve supportare tipi binari. Crea anche problemi con il versioning delle classi.
Si ha anche **Character BLOB (CLOB)**, ad esempio tramite *XML*, che però richiede librerie per il parsing e non è compatto, anche se printabile e supportato generalmente dai db

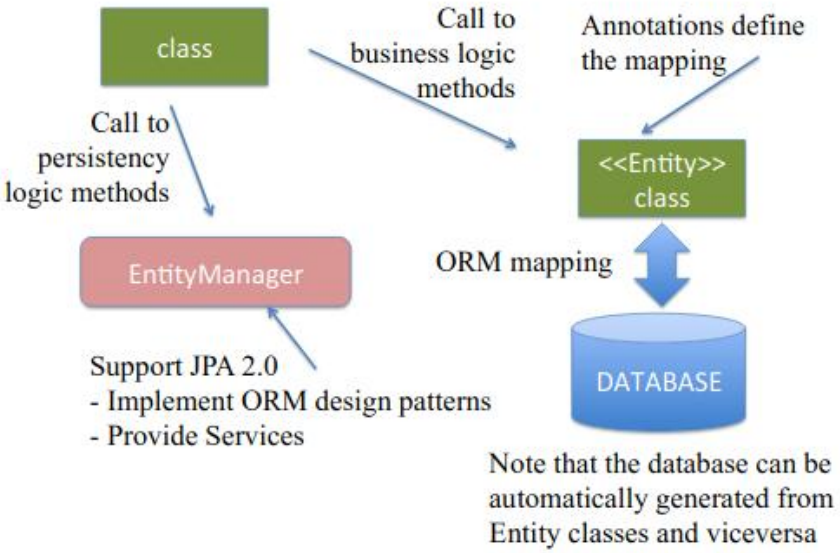


Figura 7.8: Schema generale di JPA. Si hanno varie classi con la business logic tra cui le entity class, che contengono i dati che si mappano sul db, definite tramite annotazioni. Si ha un servizio, detto *EntityManager* che implementa i vari pattern ORM. Volendo il database può essere generato automaticamente dalle entity class o viceversa.

7.1.4 JPA 2.0

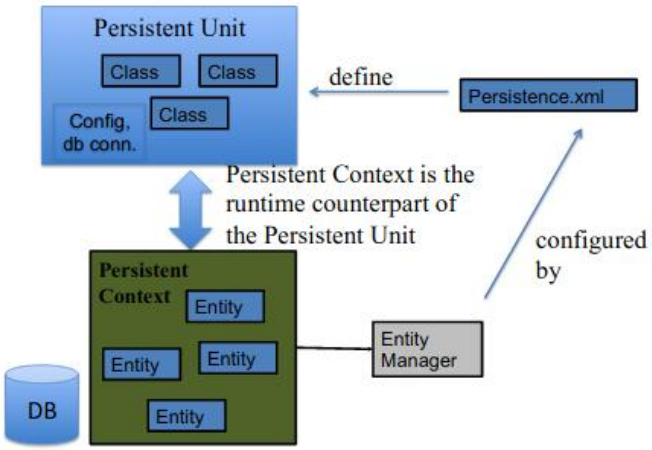
Vediamo quindi un framework Java che implementa ORM: **JPA**.
Le **Java Persistence API (*JPA*)** sono un framework per il linguaggio di programmazione Java che si occupa della gestione della persistenza dei dati di un DBMS relazionale nelle applicazioni.
Si ha quindi un grande set di pattern già implementati che usiamo in modo dichiarativo specificando il mapping tra enità di dominio e db. Si hanno varie implementazioni di JPA con ORM (tramite annotations e services, per comandare operazioni di persistenza, schema in figura 7.8), tra cui *Hibernate*.
Studiamo quindi nel dettaglio le varie componenti.

EntityManager

Chiamiamo **Plain Old Java Objects (*POJO*)** le entity class, con oggetti che vengono creati tramite il `new` e che sono oggetti regolari fino a che non interagiscono con l'**EntityManager** (sono normali classi con annotazioni).
Un'entità in memoria può essere in due stati:

- **managed/attached**, se l'entità è sotto gestione dell'EntityManager
- **unmanaged/detached**, se l'entità non è sotto gestione dell'EntityManager, che non sa che esiste quell'entità

Posso spostare esplicitamente un oggetto da uno stato all'altro in quanto può essere utile avere oggetti unmanaged, spesso in caso di distribuzione durante l'invio ad un altro nodo (diciamo che sono situazioni obbligate dalla pratica).
Tramite tale interazione con l'EntityManager le entità sono memorizzate nell'*identity map* ed eventuali cambiamenti sono tracciati dalla *unit of work*, con notifiche tramite l'EntityManager (con *caller responsability*). D'altro canto create/remove sono responsabilità dello sviluppatore mentre le update sono rilevate automaticamente dall'EntityManager.
Si ha quindi uno schema del tipo:



Nello schema si ha in alto la parte di configurazione e in basso quella di runtime. Le classi sono divise in varie **persistent unit**, ciascuna che può lavorare con un db diverso.
Il file `persistence.xml` è contenuto nella cartella `META-INF` del progetto e gestisce tutte le varie configurazioni. Sono indicate nel file le classi della *persistent unit* ovvero tutte le classi di entità nel progetto java (con il tag `<class>`) o contenute in un jar (con il tag `<jar-file>`).
In merito al mapping tra classi e db si ha che esso è definito tramite annotazioni nelle classi e l'orm-mapping è definito nel `persistence.xml` tramite il tag `<mapping-file>`.
Esempio di `persistence.xml` e di codice java su slide.

Sulle slide si trova anche una spiegazione dei vari metodi in Java che possono tornare utili in quanto è stato ritenuto inutile metterli in questi appunti.
Nelle query si usano comunque i nomi degli oggetti.

Mapping Hierachies

Nella OOP posso avere gerarchie tra gli oggetti ed ereditarietà ma questi concetti non si hanno nei database e quindi serve un mapping dedicato per eseguire query che sfruttano le gerarchie. Il mapping deve essere esplicitamente specificata.
Si hanno tre strategie per il mapping (**su slide esempi delle varie annotazioni per Java**):

1. **single table**, che prevede una tabella per ogni classe in rapporto di gerarchia tra loro. Si ha quindi una colonna che deve rappresentare il tipo di oggetto (per distinguere le varie classi) di tipo stringa. È un metodo facile da implementare e veloce ma obbliga ad avere tabelle che possono contenere valori NULL (una classe derivata ha attributi in più di quella orinale e diversi da un'altra classe derivata dalla stessa classe originale etc...). Si ha comunque uno spreco di risorse

2. **one table per concrete class**, dove ogni classe (sia padre che figli) hanno una tavella associata con un campo per ogni attributo (per le figlie anche quelli ereditati). Tutti gli attributi di una classe sono quindi in una singola tabella dedicata. Si ha un maggior controllo dei vincoli e un mapping ancora semplice ma è difficile da gestire con l'EntityManager e il DBMS deve supportare la *SQL UNION*

3. **one table per subclass** dove si ha una classe per tabella ma senza ricopia degli attributi per le classi figlie (che sono caratterizzate solo dall'id e dagli attributi extra rispetto alla classe padre). Si ha quindi un mapping "uno a uno". Questa strategia supporta i vincoli tabellari e non richiede la *SQL UNION* ma si ha un istanza "splittata" su più tabelle ed è generalmente una soluzione lenta in esecuzione (dovendo eseguire i *join*)

Capitolo 8

Component-based systems

Parliamo ora di **sviluppo per componenti**.
Partiamo con alcuni design pattern per gestire le dipendenze per poi passare ad un modello per lo sviluppo per componenti, ovvero **EJB 3.0**, che è uno standard implementato in framework come **Glassfish** che verrà usato spesso negli esempi presenti nelle slide e nel materiale del corso.
Con sviluppo per componenti si parla di uno sviluppo che ci permette di creare unità, i componenti, di cui si può fare indipendentemente il deployment, senza quindi dipendenze statiche ma con dipendenze che possono essere soddisfatte a runtime nell'ambiente di esecuzione.

8.1 Gestione delle dipendenze

Parliamo quindi nel dettaglio dei pattern di gestione delle dipendenze.
Un aspetto chiave è il **decoupling**, ovvero rendere indipendenti le componenti da deployare anche se a runtime tali componenti devono comunque interagire. Non si parla quindi della rimozione della dipendenza funzionale ma si ha una soddisfazione dinamica (a runtime) delle dipendenze tramite linking di diverse componenti per differenti sistemi, senza che queste interferiscano con il deployment.
Su slide esempio di codice con dipendenza statica dove si crea un'istanza in una classe usando il costruttore di un'altra classe.
Si hanno quindi alcuni design pattern per gestire le dipendenze dinamicamente (**su slide anche esempi con diagrammi UML**):

- **inversion of control**, dove i campi che memorizzano riferimenti ad altri componenti vengono popolati automaticamente da un componente esterno anche se non viene chiesto di farlo. Questo pattern è anche chiamato **dependency injection**. Si introduce

un elemento esterno chiamato *assembler* che ha la responsabilità di individuare il componente che deve essere usato e "iniettare" la dipendenza di cui ha bisogno, questo senza che la classe che necessita della dipendenza faccia nulla. L'assembler, se pensiamo all'UML, ad esempio, è collegato all'interfaccia della classe in cui iniettare le dipendenze.
La dipendenza viene iniettata alla creazione dell'oggetto nella maggior parte dei casi concreti (nella maggioranza dei framework che implementano il pattern). Per fare l'iniezione si hanno diverse modalità:

  - **constructor injection**, dove si usa un parametro del costruttore
  - **setter or field injection**, dove si usa un metodo setter
  - **interface injection**, dove si usa un'interfaccia dedicata (ma questo metodo non è molto usato)

Esempi di codice Java dei tre casi su slide.
Solitamente comunque l'assembler è già messo a disposizione dai vari framework e può essere configurato eventualmente tramite file di configurazione (per la scelta dell'implementazione) e annotazioni nelle classi che devono subire la injection di dipendenze. In alcuni framework le classi vengono "strumentate" per non dover aggiungere esplicitamente i metodi delle injection (magari usando annotazioni specifiche per dire come valorizzare una variabile tramite dependency injection, tipo `@PersistenceContext`). In ogni caso i riferimenti non devono essere ambigui e devono essere ovvi nel contesto. Eventualmente si possono usare informazioni aggiuntive per disambiguare il tutto. L'accesso alla variabile viene fatto dall'assembler, in java, fine fatto tramite accesso diretto, con la **reflection**, o "strumentando" la classe aggiungendo metodi di setting. Il framework rende tutto questo "trasparente" nel codice

- **service locator**, dove la classe che ha bisogno di popolare un campo con un riferimento chiede esplicitamente al service locator il valore del riferimento. Questo pattern è anche chiamato **dependency lookup**. L'idea è quella di avere un "registro", il service locator appunto, che conosce qual è l'implementazione che deve essere usata di volta in volta. A questo punto la classe che deve

soddisfare la dipendenza fa un accesso al service locator chiedendo che venga ritornato un riferimento al componente che deve essere utilizzato. Si ha l'assembler che popola il service locator per specificare che componenti esistono e che interfaccia implementano queste componenti. Il service locator può essere più o meno complesso a seconda delle informazioni che devono essere aggiunte al suo interno e al tipo di query che devono essere usate per recuperare i vari riferimenti.
Nella sua accezione più semplice il service locator è un *singleton* che memorizza coppie chiave-valore dove il valore è il riferimento al componente che deve essere usato e la chiave può essere l'interfaccia che il componente implementa. In questo modo nei componenti possiamo chiedere quali altri implementano l'interfaccia con cui bisogna interagire per poi utilizzarli (???). Nelle classi, per interrogare il service locator, serve un riferimento al service locator stesso che può essere stabilito tramite dependency injection, si aggiunge quindi un'annotazione e l'assembler, in modo trasparente, imposta un reference al service locator che andremo ad usare direttamente nel codice. Alcuni framework permettono di assegnare nomi ai componenti che scriviamo e poi laddove in un altro componente serva un reference ad un componente già scritto posso usare annotazioni che specificano il nome cosicché il framework recuperi in modo trasparente il riferimento del componente, che nel frattempo viene registrato nel service locator, assegnando quindi il valore della variabile

Si raggiunge quindi un ottimo livello di astrazione.

8.2 EJB 3.0

Vediamo quindi un modello di sviluppo per componenti per Java: **EJB 3.0**, uno standard implementato in molti framework.
Tra i framework principali che implementano questo standard abbiamo:

- **Glassfish**
- **JBoss**

Si hanno varie tecnologie come EJB, ogni tecnologia con il suo modello, ciclo di vita dei componenti, servizi. EJB comunque include tanti servizi, fornendo un supporto completo, ma si hanno framework che supportano solo

sottoinsiemi degli stessi (tra questi troviamo anche Spring).
In generale all'interno di applicazioni EJB troviamo, lato server-side e component-model:

- implementazione della business logic
- distribuzione dei componenti (che quindi possono essree in esecuzione su macchine diverse), usando **Remote Method Invocation (*RMI*)** dietro le quinte (ricordando che RMI è una tecnologia che consente a processi Java distribuiti di comunicare attraverso una rete)
- facilità di integrazione
- presenza di numerosi servizi configurabili in fase di deploy in merito a sicurezza, transazioni, persistenza etc... Nel dettaglio per la persistenza si ha come standard JPA come meccanismo di ORM
- presenza di comunicazione tra componenti sia sincrona che asincorna, quest'ultima usando un **message-oriented-middleware** che implementa **Java Message Service (*JMS*)**, ovvero l'insieme di API, appartenente a Java EE, che consente ad applicazioni Java presenti in una rete di scambiarsi messaggi tra loro
- implementazione di interfacce di web service, con supporto al protocollo **SOAP** (per lo scambio di messaggi tra componenti software) e **Web Services Description Language (*WSDL*)**, ovvero un linguaggio formale in formato XML utilizzato per la creazione di "documenti" per la descrizione di web service. Si hanno anche **Remote Procedure Call (RPC)** (attivazione da parte di un programma di una procedura o subroutine attivata su un computer diverso da quello sul quale il programma viene eseguitoattivazione da parte di un programma di una procedura o subroutine attivata su un computer diverso da quello sul quale il programma viene eseguito) e chiamate *document-style* tramite **Java API for XML Web Services (*JAX-WS*)**, ovvero un insieme di interfacce (API) del linguaggio di programmazione Java dedicate allo sviluppo di servizi web, tramite annotazioni

Inoltre con le tecnologie di EJB si possono costruire diverse view.

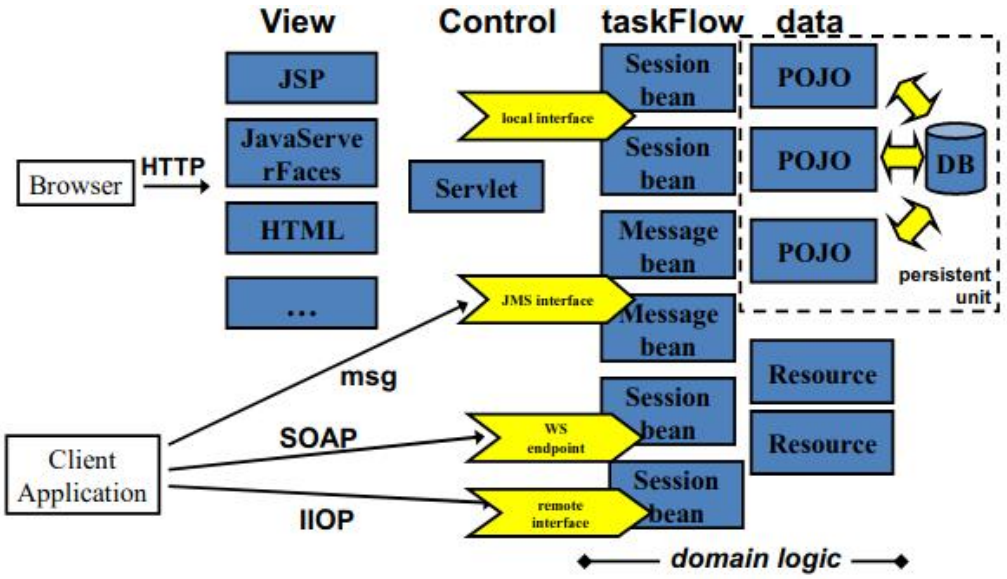


Figura 8.1: Schema d'esempio di un'applicazione **Java 2 Enterprise Edition (*J2EE*)**, con le entità persistenti dei POJO. I *bean* implementano la logica applicativa e sono divisi in due gruppi: *session bean* e *message bean* (i primi sincroni e i secondi asincroni). Si hanno vari protocolli per interagire con i bean e si hanno poi i vari componenti per le view.

EJB è conosciuto anche come la *tecnologia dei tre bean*, chiamando quindi con il termine bean i componenti di EJB. Si hanno, avendo già discusso di due di questi in figura 8.1:

1. **entity bean**, che altro non sono che i POJO nel dominio di EJB
2. **session bean**, per le operazioni sincrone della business logic.
   Un session bean è una classe che implementa un'interfaccia locale e/o una remota e/o un interfaccia endpoint. Implementa delle operazioni simil API e interagisce con gli entity bean e con altri session bean. Si hanno quindi varie annotazioni:
   - **local interface**, annotata con `@javax.ejb.Local` specifica metodi che possono essere invocati localmente
   - **remote interface**, annotata con `@javax.ejb.Remote` specifica metodi che possono essere invocati da remoto con RMI
   - **endpoint interface**, annotata con `@javax.jws.WebService` specifica metodi che possono essere invocati da remoto

usano SOAP con JAX-RPC, avendo WSDL generato automaticamente

I session bean possono essere *stateless* o *statefull* (non contenendo o contendo informazioni di stato) specificando:

- `@javax.ejb.Stateful`
- `@javax.ejb.Stateless`

e permettendo al framework di gestire diversamente il session bean a seconda del caso.
Esempio di implementazione in Java su slide.

3. **message bean**, per le operazioni asincrone della business logic. Questi sono classi che implementano un'interfaccia ben definita che viene eseguita quando si riceve un messaggio asincrono. Tali classi hanno l'annotazione `@javax.ejb.MessageDriven` e i messaggi JMS usano `javax.jms.MessageListener` nella loro interfaccia.
   Bisogna implementare un metodo `onMessage(Message message)` che ha appunto come parametro il messaggio ricevuto (messaggio solitamente di tipo chiave-valore).
   Esempio di implementazione in Java su slide.

Gli ultimi due implementano task serve-side e sono *transient* in quanto la loro esecuzione può modificare il db ma i bean in se non sono persistenti.
In un framework che implementa EJB dobbiamo immaginare le classi/bean come incapsulate in *container* che intercetta le richieste fatte al bean. Si possono, nella realtà implementativa,e avere componenti dette *stub*, indipendenti dal bean oppure "strumentando" il bean che va ad includere i vari *stub*, che vengono comunque prima eseguiti del bean (???). Avere questo ulteriore layer che protegge l'accesso al bean si possono aggiungere servizi di sicurezza, transazioni, instance pooling, gestione del ciclo di vita etc... usando un modo dichiarativo per specificare come deve comportarsi il software per quei vari aspetti, confutazione che comporta poi la natura degli *stub* (in pratica si richiede una certa logica, non la si programma).
I vari vendor che implementano EJB competono sulla parte dei *container*, oltre che sulla parte di prestazioni per i vari aspetti standard e sulla parte di servizi extra fuori standard. Attualmente i container sono auto-generati dai bean, dalle interfacce, dalle annotazioni, da file di configurazione etc... Per configurare i container si ha:

- un comportamento di default definito dal vendor

- annotazioni che sovrascrivono il comportamento di default che vengono definite dal programmatore
- dei *descrittori*, file di configurazione esterni spesso in XML, che descrivono il comportamento con priorità rispetto alle annotazioni. Sono definiti da un amministratore di sistema in quanto utili/necessari in fase di deployment

In alcuni casi accedere ad un container dal bean per risalire ad alcune informazioni di contesto dell'applicazione. Per farlo si sfruttando le dependency injection sfruttando l'annotazione `@Resource` per inserire in una variabile la `SessionContext`.
I bean hanno un ciclo di vita, creati quando utile e poi distrutti. Si hanno quindi vari metodi chiamati poi dal container tramite callback quando si hanno eventi alla creazione del bean o alla sua distruzione (esempio per liberare risorse). Si hanno quindi annotazione per i metodi, sempre del bean, come ad esempio `@PreDestroy`.
EJB supporta, per la gestione delle dipendenze:

- *setter/field dependency injection*
- *service locator*, tramite **Java Naming and Directory Service (*JNDI*)**

Su slide descrizione dei codici di esempio trovabili sulla pagina del corso.
Vediamo ora i servizi che un container può dare ad una applicazione EJB.
Si hanno due aspetti:

- **resource management** per la gestione efficiente delle risorse. Infatti applicazioni enterprise potrebbero avere migliaia di client causando la generazione di milioni di oggetti server-side, che devono essere gestiti in modo efficace
- **services** per i vari servizi configurabili ed usabili nell'applicazione

Lato *resource manager troviamo* l'**instance pooling**, un meccanismo chiave per la gestione delle risorse in quanto si riduce il numero di oggetti necessari a soddisfare una certa richiesta, sfruttando la presenza del container per il riuso dei vari oggetti, non rendendo necessario la ricreazione degli stessi (che nel dettaglio sono session bean). In generale infatti non serve avere un oggetto per ogni richiesta. In base all'annotazione che definisce se un bean è stateless o statefull si ha un impatto sulla gestione dell'istance pooling. Se un bean è stateless lo stesso bean, portato in memoria per effettuare delle

operazioni, può essere usato per servire altre richieste (finendo in un pool delle istanze con lo stato *ready*). Si ha quindi un pool di stateless session bean con i container (di tipo `EJBObject`) che intercettano le richieste e ottimizzano l'uso delle risorse tramite il pool.
Se invece si hanno bean statefull si ha che essi non possono essere usate per richieste di client differenti, cosa che porterebbe ad ambiguità nello stato. Quindi si ha un meccanismo di **activation/passivation**. Si ha il container che riceve le richieste quindi i bean possono anche non essere mantenuti in memoria se non vengono usati e quindi, se questi non sono utilizzati per lungo tempo (definito nella configurazione del framework) vengono "passivizzati", ovvero rimossi dalla memoria e salvati su disco (tramite `serialization` che scrive in binario un oggetto in memoria). Qualora servissero vengono nuovamente "attivizzati", rimessi in memoria ed usati. Anche in questo caso si ha quindi un'ottimizzazione anche se minore al caso dei bean stateless.
L'istance pooling viene poi accompagnato da vari service, nel dettaglio otto (alcuni già visti che verranno quindi meno approfonditi):

1. **concurrency**. Nel dettaglio in realtà non si ha concurrency non potendo implementare codice concorrente nei bean ma si può avere concorrenza su client multipli, ognuno che carica una propria copia dei dati, avendo quindi concorrenza tra i bean ma non nei bean. Per gestire i conflitti si hanno le strategia di locking ottimistiche o pessimistiche già viste, bloccando eventuali record per alcuni client etc... nel caso pessimistico o suando i numeri di versione in quello ottimistico. Anche lato messaggi non si ha concorrenza

2. **transaction management**, con le transazioni definite in modo dichiarativo specificando la semantica transazionale dei bean tramite annotazioni. Si ha che ua transazione inizia quando inizia l'esecuzione di un metodo associato a una transazione e termina quando termina l'esecuzione di un metodo associato a una transazione.
   Si separa la business logic dalla logica transazionale (anche se è possibile unirle con una gestione manuale delle transazioni).
   L'annotazione `@TransactionAttribute(<option>)` può essere utilizzata per specificare la semantica transazionale di un metodo nel bean. Bisogna studiare quando il chiamante apre una transazione. Si hanno varie opzioni per l'annotazione (**su slide immagine per ogni opzione**):

- `NotSupported`, a prescindere dal contesto transazionale del chiamante quando il bean viene eseguito non viene eseguito all'interno del contesto transazionale (e in caso di rollback non si fa rollback di ciò che è stato fatto nel bean)
- `Supports`, che indica che il bean segue il contesto transazionale del chiamante e quindi ciò che viene fatto dal bean fa parte della transazione (e può essere annullata con essa), se è stata aperta una transazione. Se non è stata aperta una transazione quello che viene fatto nel bean non fa parte di alcuna transazione
- `Required` (*default*), che indica che se il chiamante ha aperto una transazione le operazioni del bean faranno parte della transaction altrimenti, se non è stata aperta, se ne apre una apposta per l'operazione del bean che viene chiusa subito dopo
- `RequiresNew`, dove il bean lavora in una transaction indipendente da quella del chiamante, sia che ha aperto una transaction che no
- `Mandatory`, dove è responsabilità del chiamante aver creato una transazione (nella quale si ha anche l'esecuzione del bean) altrimenti si ha errore/eccezione
- `Never`, dove un'operazione può essere eseguita solo in un contesto non transazionale altrimenti si ha errore/eccezione (in pratica se il chiamante apre una transaction si ha errore)

Posso usare la stessa annotazione a livello di classe per adottare la cosa per ogni singolo metodo.
In merito alle annotazioni:

- per i session bean non si hanno regole
- per gli entity bean, interagendo con il db, si ha che bisogna vere sempre semantica transazionale, quindi tramite `Required, RequisresNew, Mandatory`
- per i messagedriven bean si ha che sono asincroni e quindi sono indipendenti dal contesto transazionale del chiamante. Si usano quindi `NotSupported, RequiresNew`

Su slide esempio grafico di sistema con transazioni.
Questo sistema è presente anche in altri framework Java e non solo in quelli che implementano EJB (ma anche non Java).
Parlando di persistenza e transazioni parliamo anche di **isolation** e **database locking**, ovvero quando si creano transazioni non necessariamente sono perfettamente "isolate" (anche se le vorremmo perfette). L'isolation può essere configurato per permettere un certo livello di interferenza con la creazione temporanea di oggetti da parte di transazioni non ancor concluse ma che possono essere letti da altre transaction. Si hanno vari livelli di interferenza per non avere isolation perfetta e i suoi costi di prestazioni (**immagini d'esempio per i tre modi su slide**):

- **dirty reads**, dove due transaction possono interferire avendo una transazione che può leggere dati modificati dall'altra transazione che però non si è ancora conclusa. In ogni caso le due transaction non si comportano come se l'altra non esistesse avendo interferenza tra scritture e letture
- **unrepeatable reads**, quando le stesse operazioni di lettura eseguite con una stessa transazione potrebbero restituire risultati diversi, in quanto dovuti a commit di un'altra transaction
- **phantom reads**, quando una transazione potrebbe leggere nuovi record generati (non si parla più di modifica ma di aggiunta) dopo l'avvio dell'altra transazione, che si è chiusa prima

A seconda dei casi alcune interferenze sono accettabili e altre no. Si hanno quindi vari livelli di isolation (dalla meno isolata e più performante alla più isolata ma meno performante):

- **read uncommitted**, con dirty reads, unrepeatable reads e phantom reads
- **read committed**, con unrepeatable reads e phantom reads
- **repeatable read**, con phantom rwad
- **serializable**, l'isolation perfetta

Con la massima isolation ho scarse prestazioni in quanto due transazioni che accedono agli stessi dati non possono essere eseguite

contemporaneamente e d'altro canto avere bassa isolation permette a transazioni non isolate di poter lavorare su dati spuri e non aggiornati, aumentando le prestazioni. Si ha il locking ottimistico usando l'attributo *version* che viene incrementato ogni volta che l'entità viene modificata e che viene controllato a commit time (e se viene rilevato un conglitto si effettua il rollback). Questa è una soluzione altamente efficace quando la probabilità di rollback è bassa, altrimenti inefficiente. **EJB** supporta il locking ottimistico tramite l'annotazione `@version`.

3. **persistency**, gestito da JPA
4. **distribution**, tramite protocolli (RMI, SOAP, JMS etc...) che fanno parte dello standard e che possono essere usati in modo trasparente per far comunicare i componenti tramite messaggi. È sia client-server (entity bean serializzati possono essere trasferiti al client) che server-side (avendo RMI utilizzato anche per la comunicazione tra i componenti EJB lato server)
5. **naming**, gestito da JNDI con dependency injection
6. **security**. Si hanno tre livelli basati su **Java Authentication and Authorization Service (*JAAS*)**:
    - **authentication**, per validare l'identità dell'utente tramite login forms, certificati etc...
    - **authorization**, per applicare criteri di sicurezza per verificare se un utente può eseguire l'operazione richiesta
    - **secure communications** per creare canali sicuri

    La sicurezza viene definita in modo dichiarativo e poi gestita dal framework
7. **asynchronous communication**, con JMS
8. **timer**, tramite un servizio di timing per comportamenti proattivi regolari in base ai setting