Elicitazione dei requisiti e comprensione del dominio

Iniziamo con la parte di **Knowledge acquisition** → intuitivamente, attraverso questo termine, intendiamo “mettere un piede”, come ingegnere dei requisiti, nel nuovo **dominio** (qualunque esso sia: medico, ferroviario) in cui dobbiamo lavorare. Capiamo, allora, che il primo passo è di capire l’informazione di dominio in cui dobbiamo lavorare, ovverosia dobbiamo:

- raccogliere i requisiti;
- raccogliere le necessità degli utenti e formalizzarle e ordinarle.

> 💡 Dobbiamo, quindi, famigliarizzare con la terminologia e i nuovi concetti e ciò appunto prende il nome di Knowledge acquisition.

Per l’elicitazione dei requisiti e la comprensione del dominio, la prima attività è di **identificare gli stakeholders interessanti** per le attività che dobbiamo condurre, in quanto la cooperazione con gli stakeholders è essenziale per condurre correttamente l’identificazione dei requisiti, in quanto sono le persone che ci permettono di capire in maniera completa quali sono i servizi che il sistema dovrà fornire → capiamo, allora, che è fondamentale identificare bene le persone con cui dobbiamo parlare. Questo è un aspetto non banale, in quanto le persone che identifichiamo, devono essere **rappresentative** degli utenti finali e delle persone coinvolte nel sistema finale. “**Cosa intendiamo con rappresentativi?**” Nel senso, che tutti i punti di vista devono essere rappresentati e devono anche essere coinvolti nella giusta proporzione, in modo tale da raccogliere dati sensati ed effettivamente rappresentativi (della popolazione di persone che vogliamo identificare e caratterizzare) → capiamo immediatamente, che identificare inizialmente questo campione rappresentativo potrebbe non essere affatto semplice, in quanto procedendo con la raccolta dei requisiti, ci accorgiamo che esistono tipologie di utenti di cui non eravamo consapevoli inizialmente e di conseguenza, la definizione del campione rappresentativo potrebbe essere **dinamica**, ovverosia potrebbe

raffinarsi mano a mano che procediamo con la raccolta dei requisiti (quindi la definizione del campione rappresentativo può essere soggetta a revisione).

Sorge spontanea la domanda: "**Come selezioniamo gli stakeholders interessanti?**" Innanzitutto, dobbiamo analizzare quali sono i profili presenti nell'organizzazione e cercare di contattare le persone con ruoli apicali, perchè hanno una buona visibilità su un ampio spettro di attività all'interno dell'organizzazione. Successivamente, dobbiamo capire quali tra queste persone (con un ruolo apicale) hanno un ruolo attivo (nel senso che hanno un ruolo nel prendere le decisioni) all'interno dell'organizzazione e quali invece sono solamente degli esecutori. Dobbiamo poi identificare le persona esposte ai problemi aziendali, in quanto potranno descriverceli in maniera più completa e corretta.

Come abbiamo detto precedentemente, l'acquisizione di conoscenza parlando con le persone è un problema molto difficile, perchè ci possono essere molteplici **ostacoli** che ci impediscono di acquisire tutta la conoscenza di cui abbiamo bisogno per modellare effettivamente i requisiti. Alcuni di questi ostacoli sono:

- punti di vista conflittuali → una persona potrebbe dirci cose diverse e/o opposte da quello che ci dicono le altre persone;
- dato che dovremmo contattare le persone con ruoli apicali all'interno dell'organizzazine, è molto probabile che tali persone siano molto impegnate e di conseguenza difficilmente raggiungibili;
- le varie persone che contattiamo potrebbero avere un background culturale diverso e di conseguenza, ognuna potrebbe utilizzare un proprio linguaggio e una propria terminologia;
- potrebbe esserci della conoscenza tacita → una persona che ha lavorato per molto tempo all'interno di un'organizzazione, dà per scontato un certo concetto perchè per lui è ovvio e di conseguenza, non ha bisogno di essere descritto;
- l'acquisizione della conoscenza rilevante potrebbe nascondersi in mezzo a molti dettagli irrilevanti → quindi tali dettagli irrilevanti potrebbero rendere meno evidenti informazioni essenziali per l'identificazione dei requisiti;
- aspetti politici e/o organizzativi; resistenza al cambiamento; turnover del personale con conseguente perdita di conoscenza.

💡 Per parlare correttamente con gli stakeholder, oltre alle skills tecniche abbiamo bisogno anche di:

- skills comunicative (esempio: capacità di ascolto e di interpretazione);
- creare una relazione di fiducia con gli stakeholders;
- una volta raccolte le informazioni, vi deve essere una fase di ordinamento e formalizzazione delle informazioni, in modo tale che esse possono essere descritte all'interno di un documento. Una volta fatto ciò, possiamo tornare dagli stakeholders, al fine di verificare che le informazioni siano state raccolte correttamente e in maniera completa, in modo tale da evitare inconsistenze (e magari discutere con gli stakeholders, può far venire in mente a quest'ultimi nuove informazioni e completare l'informazione).

La prima modalità per acquisire informazioni di dominio è il **Background study** → attraverso questa modalità si raccoglie l'informazione di background (quindi informazione di letteratura e di documentazione) e la si legge. Quindi, dobbiamo **raccogliere**, **leggere** e **sintetizzare** documenti riguardanti:

- l'organizzazione → come ad esempio: organigrammi e business plan;
- il dominio della letteratura → come ad esempio: libri, articoli scientifici e regolamenti;
- il sistema as-is → come ad esempio: procedure, richieste di cambiamenti, regole di business e flussi di lavoro utilizzati all'interno dell'organizzazione.

In questo modo, arriviamo preparati alla fase di raccolta dei requisiti. Da notare, inoltre, che è importante svolgere questa fase prima di parlare con gli stakeholders, **in modo tale da mostrare una certa competenza e professionalità** → questa attività può risultare complicata, perchè la quantità di documentazione che potremmo trovare, potrebbe essere molto vasta e alcuna di questa potrebbe essere irrilevante (quindi dobbiamo anche fare attenzione a selezionare solamente la documentazione utile, in modo tale da non perdere tempo). Per risolvere questo problema, possiamo adottare la soluzione della **meta-knowledge**, la quale consiste essenzialmente nel concentrarsi solamente sulla documentazione realmente rilevante, mentre il resto della documentazione possiamo trascurarla. Sorge a questo punto la domanda: "**Come vengono**

raccolti i dati?" Prima di tutto dobbiamo raccogliere i dati di marketing (ovvero quelli finanziari), al fine di capire il business dell'azienda e di conseguenza, l'attività di Data Collection (ovvero di raccolta dei dati) potrebbe completare un background study → quindi, il background study ci permette di capire qual è la conoscenza necessaria per comprendere il nuovo il nuovo argomento, mentre il data collection ci permette di avere una conoscenza quantitativa.

> 💡 Il data collection può essere utile per elicitare i requisiti non-funzionali (esempio: vincoli di performance, vincoli quantitativi) e le due principali difficoltà legata al data collection sono:
>
> 1. ottenere dei dati affidabili può richiedere tempo;
> 2. i dati devono essere interpretati correttamente.

Una volta formata una certa conoscenza (utilizzando i background study e il data collection), possiamo iniziare a pensare di presentarci presso gli stakeholders e quindi iniziare una prima interazione con quest'ultimi. La prima modalità che possiamo adottare è quella dei **questionari** → un questionario è una lista di domande, alla quale qualcuno deve fornire delle risposte. Abbiamo, quindi, che presentiamo un elenco di domande agli stakeholder selezionati e quest'ultimi dovranno rispondere alle domande. In particolare, le domande possono essere di diverso tipo:

- domande a **risposta chiusa** → vi è quindi una lista di risposte e lo stakeholder deve selezionare la risposta più opportuna;
- domande di **ponderazione** → vi è un elenco di affermazioni da ponderare qualitativamente (alto, basso, medio) oppure quantitativamente (con percentuali), al fine di esprimere l'importanza percepita, la preferenza e il rischio.

I questionari sono una modalità molto efficace per:

- raccogliere molta informazione in poco tempo, perchè il tempo che la persona deve dedicare per rispondere a queste domande solitamente è abbastanza ridotto;
- raggiungere facilmente molte persone all'interno dell'organizzazione e/o del mercato di riferimento.

> 💡 Quindi, i questionari possono essere utilizzati per raccogliere informazioni in maniera massiva e per capire e quantificare alcuni aspetti, al fine di preparare un'interazione con gli stakeholders maggiormente focalizzata su argomenti specifici.

É importante realizzare correttamente i questionari, perchè altrimenti potrebbero raccogliere informazioni non del tutto attendibili. Per realizzare correttamente i questionari, dobbiamo:

- formulare correttamente le domande → in quanto una domanda formulata scorrettamente, potrebbe suggerire una risposta e quindi falsare i dati raccolti;
- scrivere le domande in maniera comprensibile e non ambigua.

Le **linee guida** per scrivere correttamente i questionari sono:

- selezionare un campione rappresentativo e statisticamente significativo di persone e fornire una motivazione valida per rispondere alle domande;
- verificare la copertura delle domande e delle possibili risposte;
- assicurarsi che le domande, le risposte e le formulazioni siano imparziali e non ambigue;
- aggiungere domande implicitamente ridondanti per rilevare risposte incoerenti;
- far controllare il questionario ad una terza parte.

Un'altra modalità per raccogliere informazioni sono le **Card sorts & repertory grids** → l'obiettivo di questa modalità è di **raffinare l'informazione**. Assumiamo, allora, che abbiamo già raccolto un po' di informazione con i background studies, i data collections e i questionari e a questo punto vogliamo raffinare un po' le informazioni raccolte. In particolare, attraverso le Card sorts utilizziamo l'informazione (che abbiamo raccolto) per catturare dei **concetti** → **ogni concetto che siamo riusciti ad identificare** (come ad esempio: le tipologie di utenti e le componenti del sistema) **viene scritto su una carta**. Successivamente, chiediamo agli utenti di raggruppare alcune azioni e/o alcune carte, che secondo loro sono in un qualche modo correlate e affini. Una volta che gli utenti hanno fatto i vari gruppi di carte, gli chiediamo il perchè hanno

formato questi gruppi (ovvero vogliamo sapere quali sono le correlazioni che hanno visto) → **questo permette di fare dei ragionamenti e di evidenziare delle informazioni, che magari erano rimaste implicite**.

> 💡 Le card sorts, quindi, sono una modalità molto efficace per fare emergere delle informazioni implicite. Inoltre, questa modalità può essere **iterabile**, ovverosia possiamo nuovamente chiedere agli utenti di fare dei gruppi di carte e spiegare il perchè hanno fatto tali gruppi.

Vediamo un esempio della metodologia card sorts: Vogliamo realizzare un sistema per pianificare le riunioni. Nella 1° iterazioni mettiamo insieme il concetto di '*meeting*' e il concetto di '*partecipante*'. Chiediamo all'utente la motivazione per cui questi due concetti sono finiti nello stesso gruppo e la motivazione è la seguente: '*i partecipanti devono essere invitati al meeting*'. Alla 2° iterazione mettiamo sempre insieme il concetto di '*meeting*' e il concetto di '*partecipante*'. In questa 2° iterazione, però, la motivazione è la seguente: '*i vincoli dei partecipanti devono essere noti per poter organizzare il meeting*'.

Una seconda modalità per raffinare le informazioni sono le **repertory grids** → una repertory grids è essenzialmente una matrice, che ha:

- sulle righe i **concetti**;
- sulle colonne gli **attributi** che possiamo utilizzare per i concetti.

Un esempio è: il concetto '*data*' potrebbe avere come attributi: 'lunedì' e 'martedì'.

Attraverso questa metodologia, coinvolgiamo gli stakeholders per aiutarci a classificare i concetti (che sono presenti nella griglia) in **classi** e **sotto-classi** → capiamo, allora , che gli stakeholders ci permettono di individuare le classi e le sotto-classi e di fatto, stiamo andando a creare una **gerarchia**. In questo modo, i gruppi più piccoli vengono sviluppati in gruppi più grandi e così via.

I **vantaggi** di questa metodologia sono:

- è semplice;
- è economica;
- è facile da utilizzare;

- ci permette di identificare delle informazioni, che prima non avevamo identificato.

Gli **svantaggi**, invece, sono:

- i risultati osservati potrebbero non essere soggettivi, perchè magari sono legati al singolo stakeholder con cui interagiamo;
- potrebbero esserci dettagli irrilevanti;
- potrebbero essere inaccurata.

Un'altra modalità per inferire informazioni sono gli **scenari e le storyboards** → l'obiettivo degli scenari e delle storyboard è di acquisire oppure raffinare le informazioni, che abbiamo già raccolto, utilizzando un **approccio narrativo** → in particolare, attraversi lo scenario andiamo a descrivere un'interazione (la quale è fatta di passi successivi) che gli stakeholders potrebbero avere con il sistema. Questo approccio narrativo per lo scenario, potrebbe essere utilizzato per:

- descrivere come il sistema as-is (ovvero il sistema prima dell'implementazione) funziona;
- oppure per descrivere come il sistema dovrebbe funzionare una volta che avremmo implementato il system-to-be.

La **storyboard**, invece, è una sequenza di **snapshots** successivi del nostro sistema (la storyboard, quindi, la potremmo paragonare ad un **fumetto**). Ogni snapshot può essere resa in modi diversi, ovverosia una snapshot può essere:

- una frase;
- un disegno;
- una slide;
- un'immagine.

La snapshot descrive essenzialmente:

- quali sono gli attori che partecipano all'iterazione;
- cosa succede all'interno dell'iterazione;
- perchè succede l'iterazione;
- potrebbero esserci anche delle condizioni WHAT IF, al fine di descrivere cosa succede se si verificano determinate condizioni e cosa succede se non si verificano.

La storyboard può procedere in due **modalità**:

1. **passiva** → allo stakeholder raccontiamo la storia e gli facciamo vedere la storyboard, al fine di verificare che l'informazione raccolta sia corretta → in questo modo, quindi, lo stakeholder valida e corregge le informazioni raccolte;
2. **attiva** → chiediamo allo stakeholder di scrivere e/o completare lo storyboard → in questo modo, raccogliamo le informazioni ed esploriamo (insieme allo stakeholder) i funzionamenti esistenti ed attesi del nostro sistema.

Lo **scenario**, invece, è una sequenza di interazioni tra:

- i vari componenti del sistema;
- oppure tra il sistema e i vari stakeholder, che hanno un ruolo all'interno dell'organizzazione.

Vediamo un esempio di scenario per lo scheduling dei meeting:

1. The **initiator** *asks* the **scheduler** for planning a meeting within some date range. The request includes a list of desired participants.
2. The **scheduler** checks that the initiator is entitled to do so and that the request is valid. It *confirms* to the **initiator** that the requested meeting is initiated.
3. The **scheduler** *asks* all **participants** in the submitted list to send their date and location constraints back within the prescribed date range.
4. When a **participant** *returns* her constraints, the **scheduler** validates them (e.g., with respect to the prescribed date range). **It** *confirms* to the **participant** that the constraints have been safely received.
5. Once all valid constraints are *received*, the **scheduler** determines a meeting date and location that fit them.
6. The **scheduler** *notifies* the scheduled meeting date and location to the **initiator** and to all invited **participants**

Gli scenari vengono molto utilizzati per spiegare com'è il sistema e/o per esplorare le possibili iterazioni con il sistema da implementare. Anche in questo caso, è opportuno interrogare gli stakeholders, al fine di capire il perchè vi sono le interazioni tra il sistema e gli stakeholders.

Gli scenari sono molto efficaci per essere trasformati successivamente in acceptance test, perchè descriviamo come il sistema potrebbe funzionare.

Ricordiamoci, che gli scenari descrivono sia quello che succede quando lo scenario va a buon fine, sia i comportamenti eccezionali che fanno fallire lo scenario e, quindi che devono essere gestiti. Inoltre, nello scenario l'utente è generico → in questo senso, gli scenari sono di diversa tipologia:

- scenari **positivi** → rappresentano il funzionamento atteso del sistema, ovvero quello che il sistema deve fare;
- scenari **negativi** → rappresentano un funzionamento che non dovrebbe mai avvenire, dato che è scorretto. Quindi, gli scenari negativi ci dicono cosa il sistema non dovrebbe fare e chiaramente sono difficili da raccogliere tutti;
- scenari **normali** → tutto procede come atteso;
- scenari **anormali** → vi sono delle situazioni eccezionali, che il sistema dovrebbe gestire → quindi il sistema non dovrebbe bloccarsi, bensì dovrebbe gestire la situazione nella maniera opportuna.

I **vantaggi** degli scenari sono:

- sono degli esempi concreti (e contro-esempi nel caso delle situazioni che non dovrebbero succedere) del funzionamento del nostro sistema;
- hanno una forma narrativa e di conseguenza, sono facilmente comprensibili per gli stakeholders, in quanto uno stakeholder riesce ad identificarsi nello scenario (dato che è un caso che potrebbe aver vissuto);
- sono sequenze di animazione oppure casi di test di accettazione.

Gli **svantaggi**, invece, sono:

- sono **parziali**, in quanto ci permettono di descrivere solamente alcune esecuzioni rispetto a tutte le possibili esecuzioni → quindi con gli stakeholders discutiamo solamente un sotto-insieme di esecuzioni. Capiamo, allora, che vi è un problema di copertura e di conseguenza, dobbiamo identificare gli scenari più importanti da discutere, al fine di coprire le casistiche più interessanti;
- dato che gli scenari possono essere moltissimi, abbiamo un'**esplosione combinatoria**;
- dal momento che gli scenari sono casi dinamici e rappresentano delle possibili esecuzioni del sistema, potrebbero avere all'interno dei dettagli necessari per istanziare un caso effettivo, ma tali dettagli non sono rilevanti

per capire il nostro problema e di conseguenza, potrebbero essere fuorvianti;

- nonostante gli scenari siano molto concreti, potrebbero comunque **nascondere alcuni requisiti impliciti**;

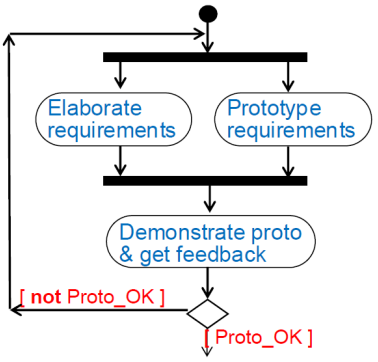
> 💡 Anche se non decidiamo di utilizzare gli scenari in maniera esplicita nel nostro processo di elicitazione dei requisiti, tipicamente prima o poi uno scenario “salta fuori”.

Un altro modo efficace (oltre agli scenari) per acquisire e comprendere i requisiti è quello dei **prototipi e dei mock-up** → l’obiettivo di quest’ultimi è di verificare i requisiti, raccogliendo un feedback diretto dagli stakeholders, mostrandogli un prototipo di quello che potrebbe essere il sistema (naturalmente scalato, ovvero con un numero limitato di servizi) una volta terminato → questo risulta molto utile per valutare se l’implementazione, che abbiamo pensato fino a questo momento, è allineato con quello che pensavano gli stakeholders. Inoltre, ci permette di focalizzarci su un (singolo) aspetto, che riteniamo importante da analizzare (perchè magari ci è poco chiaro e/o perchè è critico per il sistema). In particolare, il prototipo che mostriamo agli stakeholder potrebbe essere:

- un **prototipo funzionale** → prototipo effettivamente funzionante con cui lo stakeholder può interagire;
- un **prototipo di interfaccia utente** → si concentra sull'usabilità del sistema, mostrando moduli di input-output e pattern di dialogo.

Vi sono dei tool di tipo drag-and-drop, che ci permettono di realizzare dei mock-up o dei prototipi.

Questo può supportare un processo iterativo di raffinamento dei requisiti, in quanto possiamo avere il seguente schema:



Partiamo da un punto iniziale ed elaboriamo dei requisiti e li utilizziamo per l'implementazione di un prototipo. Dopodiché che i requisititi e il prototipo sono pronti, essi convergono e abbiamo una sessione con gli stakeholders per mostrargli il prototipo che abbiamo realizzato. A questo punto, quindi, gli stakeholders provano ad utilizzare il prototipo, in questo modo riusciamo ad ottenere dei feedback. Chiaramente, i feedback possono essere positivi (e quindi possiamo procedere con il processo), oppure possono negativi, i quali sono molto utili perchè ci permettono di far emergere degli aspetti a cui precedentemente non avevamo pensato → di conseguenza, il prototipo va raffinato e quindi vi saranno diverse iterazioni del processo.

I **vantaggi** dei prototipi e dei mock-up sono:

- forniscono un'idea molto concreta di quello che potrà essere l'applicativo una volta implementato e di conseguenza, permette allo stakeholder di ragionare concretamente sull'iterazione che avrà con il sistema → i prototipi, quindi, ci permettono di chiarificare i requisiti e di far emergere informazioni, che erano rimaste latenti fino a quel momento;
- potrebbero avere degli utilizzi collaterali rispetto a quello di fornire comprensione dei requisiti. Ad esempio, potrebbe essere utilizzato per la formazione degli utenti oppure potrebbero fornire uno stub per gli integration testing.

Gli **svantaggi**, invece, sono:

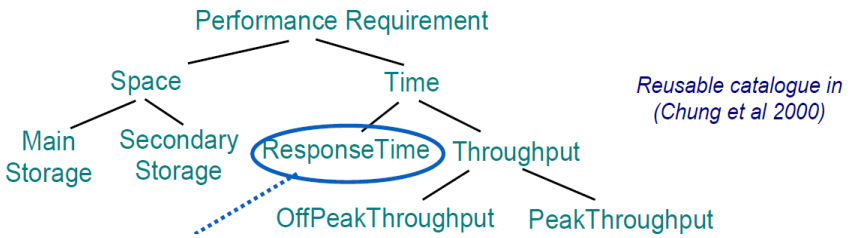
- è un sistema non completo, in quanto non copre tutti gli aspetti possibili, quindi alcune funzionalità sono mancanti e di conseguenza, non sono soggette a validazione degli stakeholders;
- non sono efficaci per validare i requisiti non-funzionali;
- potrebbero essere fuorvianti, perchè potrebbero far sorgere delle aspettative molto alte da parte degli stakeholders;

- se realizziamo un prototipo attraverso i tool (come ad esempio Figma), tipicamente l'applicaazione viene scritta con codice quick-and-dirty, il quale è funzionale solamente per il prototipo e difficilmente potrebbe essere riutilizzato nel sistema finale;
- potrebbero esserci delle inconsistenze tra quello che viene implementato nel prototipo e il requisito effettivamente raccolto.

Precedentemente abbiamo detto, che raccogliere le informazioni è molto prezioso, ma allo stesso tempo è anche molto dispendioso. Un modo per ridurre il costo della raccolta di informazione e per velocizzare questa fase è di utilizzare la **conoscenza**, la quale magari è stata raccolta durante progetti precedenti. "**Qual è il processo per riutilizzare l'informazione?**"

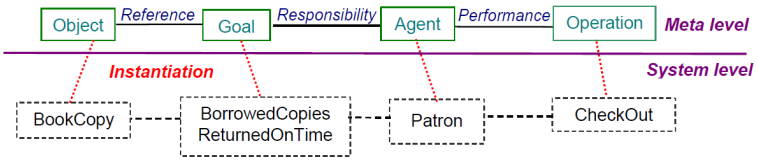
1. Prima di tutto dobbiamo recuperare l'informazione da DB contenti informazioni già raccolte in progetti condotti da me oppure condotti dall'organizzazione;
2. L'informazione può essere trasposta, quindi l'informazione può essere adattata al mio dominio → la trasposizione dell'informazione può essere condotta in tre modi:
   a. tramite istanziazione;
   b. tramite specializzazione;
   c. tramite riformulazione.

   Vediamo allora un esempio: Tipicamente l'uso di informazione passa per una tassonomia, ovvero passa per un'organizzazione gerarchica con una struttura ad albero.



Vediamo, allora, che questo concetto di Performance Requirement è sufficientemente astratto da essere riutilizzato in diversi progetti. In particolare, per utilizzare questo concetto, devo trasportare questa tassonomia nel contesto specifico all'interno del quale sto lavorando.

Un approccio alternativo (alla tassonomia) prende il nome di **Requirement Document Meta Model** → permette di raffinare le dipendenze e le relazioni tra i vari concetti, che possono essere presenti nel nostro dominio. In particolare, come informazione di riuso è possibile riutilizzare dei meta-modelli elaborati, dove per 'meta-modello' intendiamo un modello di modelli, ovverosia un modello che descrive come costruire altri modello. Questo permette di avere una visione più astratta e di poter riutilizzare la conoscenza in modo più efficace. Inoltre, all'interno dei meta-model ritroviamo dei concetti, che possono essere riutilizzati in vari progetti software, come ad esempio: il concetto di object, di goal, di agent e di operation.



Vediamo, che stiamo lavorando ad un meta-livello e che un goal fa riferimento a degli oggetti e che un goal è responsabilità di un particolare agente, il quale a sua volta attua una determinata operazione → **vediamo, allora, che abbiamo dei ruoli e abbiamo delle relazioni tra questi ruoli**.

A questo punto, specializziamo il meta-livello nel dominio all'interno del quale stiamo lavorando (e ad esempio, il dominio è quello di progettare un sistema software per la gestione di una biblioteca). Vediamo, allora, che il goal viene mappato nel goal di restituire le copie dei libri prima della scadenza del prestito. L'oggetto, invece, viene mappato nella copia del libro e l'agent diventa il patron e per riuscire a fare ciò, fa il check-out del libro.

Analogamente, vi sono dei domini astratti che possiamo concretizzare. Tipicamente vi sono:

- delle risorse;
- dei limiti di utilizzo (relativi alle risorse);
- un utente che intraprende un'operazione, al fine di ottenere una risorsa limitata.

Capiamo, allora, che anche in questo caso vi è una relazione astratta, la quale può essere **specializzata** con le risorse, che hanno importanza nel dominio che stiamo modellando. Attraverso questo approccio, quindi, abbiamo un riutilizzo della conoscenza specifica del dominio, che consiste

sostanzialmente nell'utilizzare l'informazione specifica del dominio astratto, avendo sempre come riferimento il dominio (che stiamo modellando), e utilizziamo tali informazioni per capire come si comporterà il sistema. Un esempio di frase da utilizzando nel documento dei requisiti relativo al dominio astratto è:

> '*Un utente non deve usare più di X risorse contemporaneamente*'

una volta che abbiamo concretizzato il nostro dominio astratto, **la frase può essere tradotta automaticamente in un requisito concreto nel nostro dominio**, nel seguente modo:

> '*Un patron non deve prendere in prestito più di X copie di libri contemporaneamente*'.

Questo è molto interessante, in quanto possiamo osservare che:

- lo stesso dominio astratto può avere molteplici specializzazioni concrete;
- lo stesso dominio concreto può avere molteplici specializzazioni dello stesso dominio astratto;
- di fatto posso riutilizzare molteplici strutture.

I **vantaggi** del riutilizzo della conoscenza sono:

- analisti esperti (rifacendosi all'esperienza pregressa) riutilizzano la conoscenza acquisita naturalmente;
- guida molto nell'elicitazione dei requisiti e ciò riduce l'effort nei nuovi progetti;
- si eredita la struttura e quindi, anche la qualità del dominio che stiamo utilizzando;
- è efficace nel produrre un documento dei requisiti, facendo in modo che non vengano tralasciati aspetti rilevanti, che sono già stati rilevati in progetti precedenti.

Gli **svantaggi**, invece, sono:

- l'efficacia del riutilizzo della conoscenza si ha solamente se siamo in grado di identificare un dominio astratto, che è abbastanza vicino al dominio concreto nel quale dobbiamo lavorare;
- definire i domini astratti è molto complicato;

- c’è un effort aggiuntivo per la validazione e l’integrazione;
- vi sono casi in cui il dominio astratto è allineato al nostro dominio concreto, ma vi è un leggero mismatch tra i due domini, il quale (ovvero il mismatch) deve essere identificato e successivamente allineato e questo chiaramente, comporta un certo effort.

3. I risultati ottenuti devono essere validati, adattati se necessario e integrati, in modo tale da essere utilizzabili.

Tecniche di elicitazione guidate dagli stakeholders

La tecnica principale (e la più utilizzata) è quella relativa alle **interviste** → attraverso tale tecnica, convochiamo gli stakeholders per intervistarli e conseguente per formulare delle domande, in modo tale da poter discutere con loro sulle possibili risposte. In questo modo, **cerchiamo di far emergere i requisiti direttamente dalle risposte, che gli stakeholders formulano alle nostre domande**. In particolare, le interviste si organizzano nei seguenti quattro punti:

1. selezionare gli stakeholders più rilevanti dell’attività → questo ci fa capire, che gli stakeholders da selezionare devono essere: esperti di dominio, manager, utenti finali e venditori;
2. una volta che abbiamo selezionato gli stakeholders, dobbiamo formulare le domande e registrare le risposte (la registrazione delle risposte può essere una lunga registrazione dell’audio di tutti gli stakeholders oppure dei semplici appunti);
3. una volta terminata l’intervista (ovvero una volta che gli stakeholders hanno risposto a tutte le domande) è importante scrivere un **report**, al fine di trascrivere e sintetizzare le informazioni ottenute durante l’intervista;
4. infine, il report che abbiamo realizzato deve essere consegnato agli stakeholders (che abbiamo intervistato), al fine di verificare e di conseguenza, **validare e raffinare** le informazioni che abbiamo raccolto.

💡 Una singola intervista può coinvolgere molteplici stakeholders (e non un singolo stakeholders), in modo tale da risparmiare tempo. Chiaramente, però, avremo un contatto meno diretto con gli stakeholders e alcuni potrebbero parlare di meno (e altri potrebbero invece parlare molto).

Esistono diverse tipologie di interviste:

- **intervista strutturata** → attraverso questa metodologia, arriviamo al momento dell'intervista già con una lista di domande, che vogliamo sottoporre al nostro intervistato. In particolare, le domande potrebbero essere:
  - domande aperte;
  - oppure domande chiuse → le domande chiuse sono più efficaci per ottenere una discussione più focalizzata.
- **intervista non-strutturata** → in questo caso, non arriviamo al momento dell'intervista con una lista di domande, bensì procediamo con una discussione più libera, parlando del sistema as-is, dei problemi e delle possibili soluzioni → inizialmente, potrebbe essere utile condurre un'intervista non-struttura, la quale è certamente più esplorativa e meno strutturata e successivamente procedere con un'intervista strutturata.

💡 Le interviste efficaci, infatti, sono ibride, ovverosia mescolano gli aspetti di entrambe le tipologie di interviste.

I **punti di forza** dell'intervista sono:

- ci permettono di raccogliere delle informazioni, che difficilmente potremmo raccogliere in modo diverso, in quanto attraverso le interviste parliamo direttamente con gli stakeholders;
- visto la dinamicità dell'intervista, potremmo accorgerci di un aspetto rilevante durante l'intervista e di conseguenza, possiamo focalizzare l'intervista su quell'aspetto (che magari non avevamo identificato precedentemente).

Le **difficoltà**, invece, sono:

- l’informazione che raccogliamo, potrebbe essere molto soggettiva, ovverosia potrebbe riguardare il singolo utente che stiamo intervistando e di conseguenza, potrebbe essere difficile da generalizzare;
- potremmo avere inconsistenza tra le informazioni raccolte da utenti diversi;
- l’efficacia dell’intevista dipende solo parzialmente dall’intervistatore, bensì dipende molto dalla disponibilità e dall’attutidine dell’intervistato a rispondere alle domande.

Per ovviare alle difficoltà (che abbiamo individuato) delle interviste, sono state individuate delle linee guida, che ci permettono di progettare e condurre delle interviste in maniera efficace. Alcune linee guida sono:

- Individuare il giusto campione di intervistati (con differenti responsabilità e mansioni) per una copertura completa dei problemi;
- Arrivare preparati, per concentrarsi sulla questione giusta al momento giusto;
- Centrare l'intervista sul lavoro e sulle preoccupazioni dell'intervistato;
- Mantenere il controllo dell'intervista;
- Far sentire l'intervistato a proprio agio.

Un approccio alternativo alle interviste sono le **osservazioni e gli studi etnografici** → si tratta di studi, che si focalizzano sul system as-is e consistono essenzialmente nell’osservare le task e le attività condotte dagli utenti del sistema as-is (ovvero come attualmente lavorano gli utenti). L’osservazione e gli studi etnografici, quindi, sono metodologie essenziali per comprendere i compiti all’interno dei sistemi esistenti, facilitando l’ossercazione diretta delle interazioni umane e delle pratiche lavorative. In particolare, le osservazioni possono essere di due tipi:

1. **osservazioni passive** → consiste nel non interferire con i soggetti, registrando le attività tramite note o video e analizzando i dati raccolti senza influenzare i comportamenti degli stakeholders;
2. **osservazioni attive** → l’osservatore partecipa direttamente alle attività, integrandosi nel gruppo per ottenere una comprensione interna delle dinamiche del compito.

Gli **studi etnografici**, invece, vengono effettuati su periodi prolungati e mirano a scoprire le caratteristiche emergenti di un gruppo sociale, andando ad evidenziare come i compiti vengano eseguiti e quali sono le norme culturali influenti.

> 💡 Questi metodi possono offrire spunti su come le persone lavorano veramente, rispetto a come dicono di lavorare o a come si crede lavorino.

I **vantaggi** dell'osservazione e degli studi etnografici sono:

- siamo in grado di rilevare anche delle **informazioni tacite**, che non verrebbero presentate da un intervistato, perchè magari la considera ovvia oppure perchè non gli viene in mente di raccontarla;
- identificano problemi nascosti e pratiche non documentate all'interno del system as-is;
- mettono in evidenza ed esplicitano aspetti, che sono specifici della cultura all'interno del quale stiamo lavorando;
- ci permettono di contestualizzare l'informazione raccolta attraverso alcuni scenari concreti.

Gli **svantaggi**, invece, sono:

- sono molto lenti e costosi, in quanto richiedono molto tempo sia da parte degli intervistati sia da parte degli ingegneri dei requisiti;
- potrebbero essere inaccurati, perchè le persone sono consapevoli di essere osservate e di conseguenza potrebbero non lavorare come al solito;
- complessità nella gestione e interpretazione di grandi volumi di dati;
- si focalizzano **solamente** sul system as-is.

Le sessioni di gruppo, sia strutturate che non, sono un'efficace metodologia per la raccolta di percezioni, giudizi e idee attraverso l'interazione tra partecipanti diversi (i quali hanno ruoli e profili diversi) → questo potrebbe arricchire notevolmente la qualità dei requisiti che riusciamo ad elicitare. Le sessioni possono variare da incontri focalizzati su obiettivi specifici a brainstorming aperti ed in particolare, abbiamo che:

- le sessioni di gruppo strutturate → si svolgono in workshop di gruppo, che possono durare alcuni giorni, con ruoli ben definiti per ciascun partecipante (leader, moderatore, manager, utente, sviluppatore, ecc.) e tipicamente, si focalizzano solamente sui requisiti di alto livello. Inoltre, utilizzano strumenti come supporti visivi e grafici per stimolare la discussione e documentare i risultati;
- le sessioni di gruppo non-strutturate (brainstorming) → in questi incontri, i partecipanti hanno ruoli meno definiti e il processo si divide due fasi:
  - generazione di idee senza censura, in modo tale che tutti possano proporre le proprie idee liberamente;
  - valutazione collettiva delle idee, secondo criteri prestabiliti come valore, costo e fattibilità, in modo tale da prioritizzare le idee.

> 💡 Le sessioni di gruppo possono portare alla luce aspetti nascosti del sistema attuale o futuro e favorire una più ampia esplorazione di problemi e soluzioni, stimolando l'invenzione e la collaborazione tra i partecipanti.

I **vantaggi** delle sessioni di gruppo sono:

- hanno una forma meno strutturata rispetto alle interviste e di conseguenza, permettono di raccogliere informazioni che magari l'intervistatore non aveva identificato sia del sistema as-is sia del sistema to-be → permettono, quindi, di raccogliere uno spettro molto più ampio di requisiti;
- permettono di attivare delle sinergie, siccome stiamo lavorando con un gruppo, possiamo già avviare una fase di risoluzione dei conflitti (nel senso, che se emergono punti di vista distinti, già in fase di lavoro di gruppo, si può cercare di mediare e capire quale potrebbe essere un compromesso accettabile).

Gli **svantaggi**, invece, sono:

- la composizione del gruppo è un aspetto critico, perchè se mettiamo insieme personalità totalmente incompatibili, rischiamo che il lavoro di gruppo sia inefficace;

- richiedono tempo e possono essere difficili da organizzare per persone molto impegnate;
- rischio di dominanza di alcune personalità, che possono influenzare l’intero gruppo;
- possibili discussioni prolisse senza risultati concreti ed una copertura superficiale di questioni tecniche.

Valutazione dei requisiti

Nel capitolo precedente, abbiamo parlato dell'elicitazione dei requisiti, la quale ci permette di ricavare informazioni dagli stakeholders, al fine di riuscire a descrivere i loro bisogni e i vincoli del sistema. Una volta che abbiamo terminato l'elicitazione, procediamo con la **valutazione dei requisiti, in modo tale** da valutare le possibili alternative, che dobbiamo prendere in considerazione, prima di arrivare ad una lista di requisiti condivisi dai nostri utenti finali → sorge spontanea la domanda: "**Perchè è importante valutare le alternative?**" É importante valutare le alternative, perchè la comprensione e la definizione dei requisiti è intrinsecamente un processo negoziale a vari livelli. Si tratta di un processo negoziale perchè:

- tipicamente stakeholder differenti hanno bisogni e necessità diverse (e potenzialmente anche contrastanti), quindi abbiamo la necessità di mediare tra gli stakeholders;
- come ingegneri del software abbiamo magari dei vincoli tecnologici , di tempo e/o di budget.

Capiamo, allora, che vi sono dei compromessi da prendere e per riuscire a prendere la decisione corretta, dobbiamo necessariamente capire quali sono le alternative disponibili, capire i relativi vantaggi e svantaggi (delle alternative) e discuterle, al fine di prendere una decisione ragionata e consapevole.

> 💡 Per essere precisi, non si tratta solamente di decidere un'alternativa piuttosto che un'altra solamente in base ai vincoli tecnologici, di budget e/o di tempo, bensì si tratta anche di una questione legata ai **rischi** → nel senso che, un'alternativa piuttosto che un'altra può comportare dei rischi diversi e quindi, è opportuno conoscere anche i rischi, al fine di prendere una decisione consapevole di tutte le conseguenze.

Gestione delle inconsistenze

Fino a questo momento, abbiamo parlato con i nostri stakeholders e abbiamo raccolto le loro necessità e tali necessità potrebbero comportare delle inconsistenze, dato che:

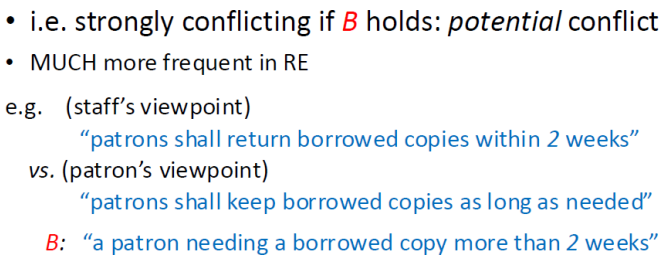
- differenti stakeholders hanno differenti bisogni e differenti punti di vista sul funzionamento del sistema software;
- vi sono delle **inconsistenze intrinseche** di ogni stakeholder (nel senso che vi sono dei requisiti, che per definizione sono inconsistenti tra di loro) → l'esempio tipico di ciò è la sicurezza contro l'usabilità: Un sistema è tanto più sicuro quanto più è chiuso all'esterno, ma al tempo stesso più è chiuso il sistema e meno quest'ultimo è utilizzabile.

Chiaramente, **è opportuno identificare il prima possibile tali inconsistenze, mentre la risoluzione delle inconsistenze è un aspetto delicato** → in particolare, la risoluzione delle inconsistenze possiamo decidere noi quando deve avvenire, perchè magari potremmo decidere di identificare solamente delle inconsistenze e non risolverle immediatamente (bensì vogliamo portare avanti le inconsistenze), perchè magari vogliamo prendere la decisione sulla risoluzione delle inconsistenze più avanti, visto che eliminare troppo presto delle alternative potrebbe precluderci delle alternative che vogliamo investigare successivamente. Dall'altra parte, però, non dobbiamo prendere queste decisioni troppo tardi, altrimenti i costi per la risoluzione delle inconsistenze potrebbero aumentare notevolmente. Durante la raccolta dei requisiti, possiamo avere le seguenti tipologie di inconsistenze:

- **terminology clash** → si ha quando lo stesso concetto, all'interno dei requisiti, viene catturato da termini diversi (per esempio: all'interno del sistema bibliotecario, alcune volte la persona che prende in prestito un libro prende il nome di 'borrower' mentre altre volte prende il nome di 'patron') → quindi, utilizzare i sinonimi, all'interno dei requisiti, per identificare lo stesso concetto è di per sè un problema. L'obiettivo, quindi, è di assegnare un unico termine per un concetto;
- **designation clash** → si ha nel momento in cui utilizziamo lo stesso termine per identificare concetti diversi. Questo è chiaramente un problema, perchè nel momento in cui utilizziamo un termine per identificare un certo concetto, allora il termine diventa una specie di keyword con una semantica ben precisa (per esempio: con il termine 'user' a volte viene utilizzato per identificare la persona che si reca in biblioteca, mentre altre volte identifica l'utente che utilizza il software della biblioteca);
- **structure clash** → si ha nel momento in cui lo stesso concetto viene strutturato in maniera diversa, a seconda del contesto in cui esso viene utilizzato (per esempio: in un contesto, la data di consegna del libro è un

tempo fisso, come ad esempio Venerdì alle 5 di pomeriggio, mentre in un altro contesto la data di consegna è un intervallo temporale, come per esempio Venerdì);

- **strong conflict** → si ha nel momento in cui nei requisiti abbiamo due frasi, dove una è l'opposto dell'altra;
- **weak conflict** (chiamati anche **divergenze**) → **strong conflict** → si ha nel momento in cui nei requisiti abbiamo due frasi che non sono esattamente l'una opposto dell'altra, bensì sono in conflitto nel momento in cui si verifica una certa condizione → quindi le due frasi non sono sempre in conflitto, bensì sono in conflitto solamente quando si verifica una certa condizione ed è per questo, che le divergenze sono difficilmente identificabili. Un esempio di divergenza è la seguente:



Sorge a questo punto spontanea la domanda: "**Quali strumenti possiamo utilizzare per identificare le inconsistenze?**" Possiamo utilizzare il seguente strumento:

- per prima cosa, dobbiamo definire un **glossario dei termini** → è molto complicato realizzare il glossario all'inizio, infatti tipicamente il glossario emerge man mano che lavoriamo con i requisiti e di conseguenza, ci accorgiamo delle keywords che stiamo utilizzando. Il glossario è molto utile nel momento in cui parliamo con stakeholders, che non hanno un background informatico, infatti il glossario rappresenta il primo step quando si realizza un progetto multi-disciplinare, che coinvolge quindi utenti con differenti background.

Il processo per gestire le inconsistenze è composto da quattro attività principali ed è il seguente:

1. **Identificare le frasi**, che abbiamo raccolto durante la fase di elicitazione, **che hanno una certa sovrapposizione** (ovvero che hanno un certo overlapping), come per esempio: tutte le frasi che riguardano il prestito dei libri oppure tutte le frasi che riguardano la chiusura delle porte del treno, e tutte queste frasi le mettiamo vicine;
2. Una volta che abbiamo identificato le frasi e le abbiamo messe vicine (quindi, in un certo senso le abbiamo raggruppate), le **analizziamo gruppo per gruppo e identifichiamo i potenziali conflitti e mettiamo in evidenza quest'ultimi**;
3. Una volta che abbiamo identificato tutti i potenziali conflitti, allora possiamo **avviare la fase di risoluzione dei conflitti**;
4. Infine, **valutiamo le possibili strategie di risoluzione dei conflitti e facciamo anche la valutazione delle alternative**. Selezioniamo la strategia migliore e poi iteriamo nuovamente il processo, fino a quando non riusciamo ad identificare altri conflitti.

Una volta che identifichiamo i potenziali conflitti, al fine di documentarli possiamo realizzare una **matrice di interazione**:

Statement	S1	S2	S3	S4	Total
S1	0	1000	1	1	**1002**
S2	1000	0	0	0	**1000**
S3	1	0	0	1	**2**
S4	1	0	1	0	**2**
Total	**1002**	**1000**	**2**	**2**	**2006**

vediamo che la 1° riga e la 1° colonna riguardano gli statements, i quali vengono numerati. I valori che inseriamo nelle celle della matrice sono:

- zero → se non vi è alcun overlap tra gli statements (infatti la diagonale della matrice è sempre a zero);
- 1000 → se non vi è alcun conflitto. Quindi i due statements hanno un overlap, ma non vi è alcun conflitto;
- 1 → se vi è sia overlap sia un conflitto tra i due statements.

Dopodiché facciamo la somma sia per riga sia per colonna e poi divido il totale ottenuto (sia di ogni riga sia di ogni colonna) per 1000 e mantengo il resto, in questo modo vedo quanti overlap e quanti conflitti ci sono per ogni statement.

Arrivati a questo punto, abbiamo identificato i conflitti e li abbiamo documentati. Adesso, quindi, dobbiamo identificare delle strategie di risoluzione dei conflitti → **idealmente**, per riuscire a fare ciò dovremmo:

- listare (quindi identificare) tutte le possibili alternative per risolvere ciascun conflitto → per riuscire a fare ciò, possiamo utilizzare le tecniche di elicitazione che abbiamo visto precedentemente (come ad esempio: interviste oppure sessioni di gruppo);
- confrontare tutte le alternative, che abbiamo individuato, e discuterle;
- verificare qual è la preferibile per ciascun conflitto.

Vediamo, allora, quali sono le possibili **strategie di risoluzione** dei conflitti:

- **evitare situazioni critiche** → per esempio: per fare in modo di avere sempre la disponibilità di copie dei libri molto richiesti, ci assicuriamo di avere delle copie in più che non possono essere date in prestito e quindi, sono sempre disponibili in biblioteca per essere consultate;
- **ripristinare delle condizioni** → per esempio: la copia del libro deve essere restituita entro 2 settimane, ma poi può essere presa di nuovo in prestito immediatamente, in modo tale che se gli utenti vogliono tenere la copia per più tempo possono farlo;
- **indebolire alcuni requisiti** → decidiamo qual è il requisito più importante, mentre quello meno importante possiamo indebolirlo, in modo tale da permettere al sistema di poter soddisfare il requisito indebolito. Per esempio: la copia deve essere restituita entro 2 settimane, salvo esplicita autorizzazione;
- **eliminare i requisiti che hanno un basso livello di priorità;**
- **specializzare uno oppure l'altro requisito** → per esempio: lo stato dei prestiti dei libri è un'informazione riservata e tale informazione possiamo specializzarla. In particolare, possiamo fare che lo stato dei libri è riservato, ma è noto solamente allo staff interno della biblioteca.

Queste tecniche di risoluzione ci guidano nella modifica dei requisiti, in modo tale da trovare un compromesso/soluzione compatibile che non abbia/abbiano conflitti.

Abbiamo realizzato una lista possibile di alternative e a questo punto, dobbiamo prendere una decisione, in modo da focalizzarci su una singola alternativa → per prendere una decisione dobbiamo innanzitutto definire dei **criteri oggettivi**:

- Uno dei possibili criteri è quello di valutare l'impatto, che le alternative hanno sui **requisiti non-funzionali**. Tipicamente, se i requisiti funzionali sono la risposta sì oppure no, i requisiti non-funzionali hanno un livello di granularità maggiore, il quale ci permette di capire quanto (in che misura) abbiamo raggiunto un requisito non-funzionale piuttosto che un altro (un esempio di requisito non-funzionale è il requisito sulle performance o sulla velocità di risposta);
- Altrimenti, le alternative possono essere valutate sull'impatto che hanno nel risolvere conflitti o nel generare altri conflitti o nel generare altri rischi.

> 💡 Una volta che abbiamo definito tali criteri oggettivi, è più semplice mettere in ordine le alternative, al fine di capire su quale alternativa focalizzarci.

Appena sopra, abbiamo parlato di rischi e quindi sorge spontanea la domanda: "**Che cos'è un rischio?**" Quando parliamo di requisiti, un rischio è tipicamente mappato sugli stakeholders. In questo caso, quindi, i rischi riguardano il fatto di non soddisfare gli stakeholders, ovverosia le necessità di quest'ultimi non vengono completamente soddisfatte (non diamo una risposta completa alle loro necessità) dal sistema che andiamo a realizzare → capiamo, allora, che **il rischio è rivolto al grado di soddisfazione che gli stakeholders avranno del prodotto che stiamo progettando**.

> 💡 Riassumendo, possiamo dire che il rischio è di non risolvere i problemi (degli stakeholders) che il software dovrebbe effettivamente risolvere oppure di risolverli in maniera non del tutto ottimale.

Il rischio è composto da:

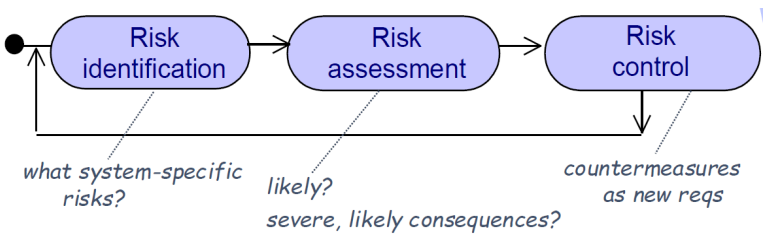
- la **probabilità** che il rischio si manifesti;

- le **conseguenze** che il rischio potrebbe avere nel caso in cui si manifestasse → notiamo, inoltre, che le conseguenze vengono catturate dalla **severità**, ovverosia il livello di insoddisfazione, che un particolare stakeholder potrebbe avere per un particolare obiettivo, che è stato identificato durante l'elicitazione dei requisiti.

Tipicamente, vi sono due tipologie di rischio:

1. **rischio legato al prodotto** → impatto negativo sugli obiettivi funzionali o non-funzionali del nostro sistema (esempio: problemi di sicurezza relativo all'utilizzo dei treni);
2. **rischio legato al processo** → impatto negativo sugli obiettivi di sviluppo (esempio: ritardo nelle consegne, sforamento dei costi).

Vediamo, a questo punto, il **risk management**:



Anche in questo caso, abbiamo un piccolo processo che ci guida nella gestione del rischio:

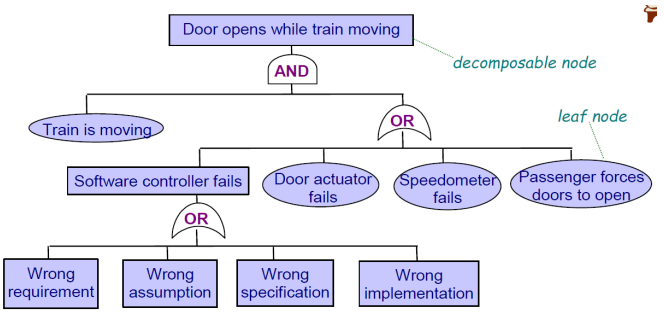
- Come prima cosa, il rischio va **identificato**, ovverosia dobbiamo capire quali sono i rischio che possiamo avere nel nostro sistema → l'identificazione dei rischi è molto complessa e per fare in modo da non dimenticarcene qualcuno, è buona norma utilizzare del **check-list**. Quindi, per ogni requisito dobbiamo verificare se è soggetto ad uno dei rischi della check-list.
  Capiamo, allora, che l'identificazione dei rischi è molto importante e per farlo in maniera corretta, possiamo procedere componente per componente, ovvero: Prendiamo tutti i componenti che compaiono all'interno del nostro sistema e ci chiamo quali rischi potrebbero essere associati a ciascun componente. Per ogni componente, quindi, ci poniamo le seguenti domande:
  - come potrebbe fallire?
  - qual è la motivazione del fallimento?

- quali potrebbero essere le possibili conseguenze del fallimento?

> 💡 Quando il componente è troppo complesso, è opportuno suddividerlo in sotto-componenti.

Per identificare i rischi, oltre alla check-list, viene utilizzato il **risk tree** → si tratta di una struttura ad albero, che **permette di collegare i fallimenti con le rispettive cause le rispettive conseguenze**. Tale albero si compone da:

- **failure node** → sono eventi oppure condizioni di fallimento indipendenti e ci permettono di ragionare. Tali nodi, a loro volta, vengono suddivisi in sotto-nodi che potrebbero essere messi in relazione in due modi:
  - **AND links** → tutte le condizioni sono necessarie per un fallimento;
  - **OR links** → basta una condizione per un fallimento.



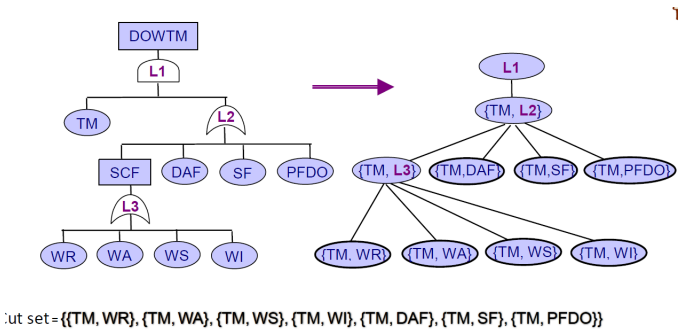
Vediamo la 1° condizione di fallimento: "Le porte si aprono mentre il treno è in movimento". Per la manifestazione di tale rischio, abbiamo bisogno di almeno 2 condizioni

Oltre alle check-list e ai risk trees, vi sono anche delle **linee guida** che ci permettono di identificare altri possibili rischi, leggendo ad esempio le key-words degli statements che stiamo scrivendo nel documento dei requisiti.

Partendo dal risk tree che abbiamo identificato, il passo successivo è di identificare i **cut set** → il cut set è la **minima combinazione di condizioni in AND delle foglie dell'albero, che causano il rischio che abbiamo nella root dell'albero**. L'algoritmo per identificare tutti i cut set consiste in:

- partire dalla root dell'albero;

- discendere nell'albero;
- se ho una condizione AND, allora dobbiamo prendere tutte le aggregazioni dell'AND;
- se ho una condizione OR, allora basta prendere solamente un'aggregazione e proseguiamo con quest'ultima fino ad arrivare ad un nodo foglia.



ut set = {{TM, WR}, {TM, WA}, {TM, WS}, {TM, WI}, {TM, DAF}, {TM, SF}, {TM, PFDO}}

Un altro modo per identificare i rischi è di utilizzare le tecniche di elicitazione (che abbiamo visto precedentemente), ovverosia possiamo utilizzare:

- gli **scenari** → prendendo in considerazione gli scenari che abbiamo a disposizione, possiamo formulare le WHAT IF QUESTIONS. Tali domande ci permettono di 'indebolire' la condizione e di conseguenza, capire se effettivamente essa rappresenta un rischio per il nostro requisito;
- il **riutilizzo della conoscenza** → dato che contesti simili hanno rischi simili;
- le **sessioni di gruppo** → ogni utente può portare il suo punto di vista e la propria esperienza, al fine di identificare i rischi che prima non erano stati identificati.

- Una volta che abbiamo identificato i rischi, dobbiamo capire:
  - qual è la **probabilità** e la **severità** del rischio;
  - qual è la **probabilità** delle conseguenze e di conseguenza, controllare i rischi ad alta priorità.

  Di fatto, quindi, stiamo facendo il **risk assessment** → inizialmente, il risk assessment viene condotto in maniera **qualitativa**, di conseguenza abbiamo che:

- la probabilità viene classificata come: poco probabile, probabile, molto probabile, etc.;
- la severità viene classificata come: catastrofica, severa, alta, moderata, etc.;

> 💡 Questo ci permette di riempire una **tabella di probabilità-conseguenze** (likelihood-consequence table) e di confrontare e prioritizzare i rischi in base al livello di severità.

Un esempio di likelihood-consequence table è la seguente:

	Risk likelihood		
Consequences	**Likely**	**Possible**	**Unlikely**
Loss of life	*Catastrophic*	*Catastrophic*	*Severe*
Serious injuries	*Catastrophic*	*Severe*	*High*
Train car damaged	*High*	*Moderate*	*Low*
#passengers decreased	*High*	*High*	*Low*
Bad airport reputation	*Moderate*	*Low*	*Low*

Likelihood level

Severity level

Il grande **vantaggio** di questa metodologia è che è molto facile da utilizzare, mentre lo **svantaggio** è che le conclusioni sono limitate, in quanto stiamo ragionando ad un livello di granularità abbastanza elevato e di conseguenza, vi è la possibilità che i rischi individuati siamo tra loro molto simili e quindi, tale metodologia fornisce un supporto limitato alla classificazione dei rischi.

Per cercare di dare delle risposte più quantitative, sono state proposte delle alternative quantitative (in modo tale da ragionare in termini numerici quando parliamo di rischi) → capiamo, allora, che non parliamo più di qualitative assessment, bensì parliamo di **quantitive assessment**, il quale prevede che:

- la probabilità venga stimata con dei numeri. Solitamente è difficile attribuire un numero alla probabilità e di conseguenza, si preferisce utilizzare degli intervalli numerici (esempio: 0-3; 3-5);
- la severità venga stimata attraverso una scala numerica, che va da 1 a 10.

Attraverso questi valori, possiamo calcolare l'**esposizione al rischio** e tale metodologia ha come **vantaggio** che utilizza un livello di dettaglio maggiore e di conseguenza, supporta una prioritizzazione più dettagliata ed esplicita dei nostri rischi. Lo **svantaggio**, invece, è che i valori sono ancora soggetti a decisioni soggettive.

- Infine, dobbiamo **controllare** i rischi → una volta che abbiamo identificato i rischi e gli abbiamo assegnato un valore, dobbiamo cercare di controllare i rischi, ovverosia dobbiamo ragionare sui rischi e cercare di ridurli → inizialmente cercheremo di ridurre i rischi a cui siamo maggiormente soggetti, ovverosia cercheremo di ridurre i rischi le cui conseguenze sono più pericolose per raggiungere gli obiettivi di progetto. Questo avviene in due fasi:

  1. **Esploriamo tutte le possibili contromisure** da adottare, al fine di limitare il rischio;
  2. **Prendiamo una decisione** su quali siano le contromisure preferibili.

  Visto che dobbiamo capire quali sono le possibili contromisure, possiamo utilizzare:

  - le tecniche di elicitazione dei requisiti (che sono: interviste e sessioni di gruppo), in modo tale da coinvolgere gli stakeholders;
  - le informazioni che abbiamo raccolto durante progetti simili → una volta che abbiamo individuato i rischi principali, ovverosia i rischi con le conseguenze peggiori (e di conseguenza i rischi a cui siamo maggiormente esposti) possiamo utilizzare varie tecniche per capire come procedere:
    - le **simulazioni** → esse ci permettono di capire se abbiamo delle performance non accettabili, perchè andiamo a simulare il sistema e di conseguenza, possiamo stimare le performance;
    - i **prototipi** → essi ci permettono di limitare i rischi legati alla limitata usabilità del sistema;
    - i **modelli di costo** → essi ci permettono di stimare correttamente il costo di sviluppo e quindi, ci permette di evitare i rischi di definire tempo di sviluppo (del software) irrealistici e/o dei costi irrealistici.
  - le **tattiche di riduzione del rischio** → le tattiche di riduzione sono molteplici:

    - possiamo ridurre la probabilità che un rischio si manifesti → questo tipicamente ci fa emergere un nuovo requisito e tale requisito, ci permette di controllare una certa quantità, in modo tale che non vada sotto una certa soglia;
    - possiamo evitare che il rischio si manifesti → questo tipicamente ci fa emergere un nuovo requisito e tale requisito, ci permette di controllare che l'occorrenza del rischio sia pari a zero;
    - possiamo ridurre la probabilità di conseguenze → questo tipicamente ci fa emergere un nuovo requisito e tale requisito, ci permette una diminuzione significativa della probabilità di conseguenze;
    - possiamo evitare le conseguenze del rischio → questo tipicamente ci fa emergere un nuovo requisito e tale requisito, ci permette di garantire che la conseguenza non si verifichi mai;
    - possiamo mitigare le conseguenze del rischio → questo tipicamente ci fa emergere un nuovo requisito e tale requisito, ci permette di ridurre la gravità delle conseguenze.

A questo punto, abbiamo individuato i rischi principali a cui siamo maggiormente esposti e abbiamo individuato una serie di possibili azioni, che ci permettono di dare risposta ai rischi e dobbiamo capire quale alternativa utilizzare per mitigare i rischi → capiamo, allora, che dobbiamo definire dei **criteri** per misurare (e quindi valutare) le alternative. In particolare, tali criteri sono:

- il loro **contributo ai requisiti non-funzionali** → i requisiti non-funzionali sono tipicamente quantitativi e di conseguenza, hanno un valore di soglia che bisogna massimizzare o minimizzare;
- il loro **contributo alla risoluzione dei rischi**;
- il loro **cost-effectiveness** → per capire se le alternative sono cost-effectiveness, dobbiamo calcolare il beneficio che ci portano rispetto al costo che dobbiamo pagare → questo si calcolare attraverso il **risk-reduction leverage e l'obiettivo è di scegliere l'alternativa, che massimizza tale valore.**

A questo punto, è importante capire che i rischi devono essere documentati, dato che abbiamo fatto un lavoro di valutazione delle alternative e delle contromisure e decisione sulle contromisure. Questo lavoro, quindi, deve

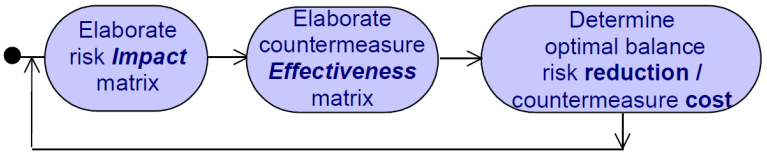
essere documentato perchè ci ha permesso di prendere una decisione e la motivazione che sottintende questa decisione deve essere documentata, in quanto se in futuro questa motivazione venisse a cadere, allora dobbiamo rivedere anche l'operazione di identificazione e decisione → l'idea, quindi, è di **tracciare le argomentazione che hanno supportato questa decisione**.
Capiamo, allora, che documentare un rischio significa specificare:

- le condizioni per cui il rischio si verifica;
- la probabilità che abbiamo stimato;
- le possibili cause e conseguenze;
- la probabilità stimata e la severity di ciascuna conseguenza;
- le contromisure identificate.

e queste informazioni devono poi essere mappate sul risk tree, in modo da strutturare tutta l'informazione che abbiamo raccolto.

DDP

Il **DDP** (**Defect Detection Prevention**) è una metodologia **quantitativa** per la gestione del rischio, che si articola in tre step:



1. Il 1° passo è di elaborare la **matrice degli impatti** (Impact matrix) → per riuscire a fare ciò, dobbiamo innanzitutto realizzare una tabella rischi-conseguenze (risk-consequence table) con degli esperti, in modo tale da:
   a. dare priorità ai rischi in base all'impatto critico su tutti gli obiettivi;
   b. evidenziare gli obiettivi più rischiosi.

   Realizzata la tabella, andiamo a definire:

   - per ogni obiettivo, definiamo l'impatto del rischio con un intervallo da zero a 1 → questo ci dice quanto il rischio può portare ad una perdita di soddisfazione per gli stakeholders;
   - l'impatto del rischio ci permette di definire la criticità di ogni rischio;

- l'impatto del rischio ci permette anche di stimare la perdita di soddisfazione per ogni obiettivo.

	Risks				
Objectives	Late returns (likelihood: 0.7)	Stolen copies (likelihood: 0.3)	Lost copies (likelihood: 0.1)	Long loan by staff (likelihood: 0.5)	**Loss obj.**
Regular availability of book copies (weight: 0.4)	0.30	0.60	0.60	0.20	**0.22**
Comprehensive library coverage (weight: 0.3)	0	0.20	0.20	0	**0.02**
Staff load reduced (weight: 0.1)	0.30	0.50	0.40	0.10	**0.04**
Operational costs decreased (weight: 0.2)	0.10	0.30	0.30	0.10	**0.05**
Risk criticality	**0.12**	**0.12**	**0.04**	**0.06**	

Vediamo che sulle colonne abbiamo i rischi, mentre sulle righe abbiamo gli obiettivi e all'interno della matrice, cerchiamo di indicare l'impatto di ogni rischio, la criticità e la perdita di soddisfazione.

2. il 2° passo è di calcolare un'altra matrice, chiamata **Effectiveness matrix**, che mette in relazione i rischi con le contromisure → in questo caso, dobbiamo stimare la riduzione del rischio, quando una certa contromisura viene adottata. Oltre a stimare la riduzione del rischio, calcoliamo anche:
   a. la riduzione combinata per ciascun rischio;
   b. gli effetti globali della contromisura per ogni rischio.
3. Infine, possiamo stimare il costo di ogni strategia e possiamo utilizzare i valori (che abbiamo calcolato) in maniera quantitativa, al fine di prendere le decisioni su quali contromisure adottare e quali invece no → stiamo, quindi, andando a combinare il costi-benefici per ogni contromisura.

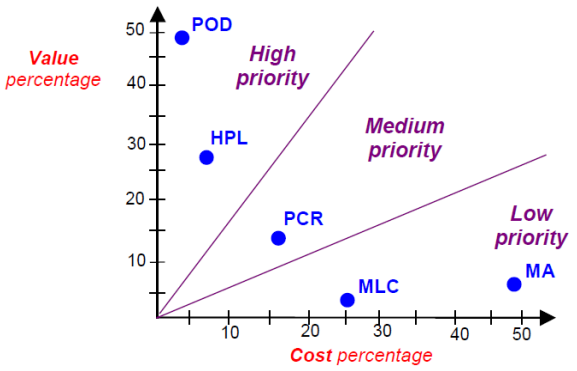
A questo punto, abbiamo molteplici alternative e possiamo adottare delle tecniche quantitative e qualitative per valutare le alternative e prendere delle decisioni. Vediamo, allora, alcune tecniche quantitative e qualitative:

- la prima tecnica qualitativa si basa sui requisiti non-funzionali e anche in questo caso abbiamo una tabella → l'obiettivo di questa tecnica è di determinare il contributo qualitativo di ciascuna opzione (dove con opzione intendiamo alternativa) rispetto ai requisiti non funzionali più importanti che abbiamo raccolto. Possiamo, quindi, decidere l'impatto che ogni opzione ha sui requisiti non-funzionali ed in particolare l'impatto può essere: positivo, molto positivo, negativo, molto negativo;
- la metodologia precedente ha anche un'equivalente quantitativo, la quale quindi ci permette di prendere delle decisioni informate in maniera

quantitativa.

Abbiamo visto, che spesso è necessario capire quali requisiti sono più importanti di altri, al fine di prendere una decisione sulle alternative. Vi sono diverse metodologie per **prioritizzare i requisiti**. Vediamone alcune:

- **prioritizzazione costi-benefici** → prevede che per ogni requisito, cerchiamo di stimare qual è il valore del requisito, ovverosia qual è il valore per i nostri stakeholders, che il requisito abbia una soluzione contro il costo di implementazione del requisito. Capiamo, allora, che per ogni requisito andiamo a stimare il valore ed il costo e realizziamo il seguente diagramma bidimensionale:



  Come possiamo vedere dall'immagine, andiamo a suddividere lo spazio cartesiano in tre settori:

  1. **alta priorità** → in cui i requisiti che hanno un alto valore ed un basso costo;
  2. **media priorità** → in cui i requisiti hanno un costo medio ed un valore medio.
  3. **bassa priorità** → in cui i requisiti hanno un alto costo ed un basso valore.

  > 💡 In questo modo, riusciamo a partizionare i requisiti in tre classi di equivalenza, in modo tale da prendere delle decisioni.

- **metodologia AHP** (**Analytic Hierarchy Process**) → metodologia più formale rispetto alla precedente, che sostanzialmente ci consente di ragionare sui

requisiti dando un criterio. Tipicamente, questa metodologia reitera due volte sui requisiti:

- la prima volta itera sul criterio del ‘valore’;
- la seconda volta itera sul criterio del ‘costo’.

Specification + Quality assurance

Abbiamo visto come raccogliere i requisiti dai nostri stakeholders (ad esempio, attraverso le interviste e i focus group) e abbiamo visto come possiamo esplorare le molteplici alternative che ci possono essere, quindi abbiamo visto come identificare le alternative e come prendere decisioni sulle alternative che abbiamo identificato, al fine di identificare delle soluzioni alternative e per mitigare i rischi. Arrivati a questo punto, facciamo uno step ulteriore e sostanzialmente **descriviamo come specificare e documentare i requisiti** → in particolare, con 'specificare' e 'documentare' intendiamo catturare in maniera scritta i requisiti (come li abbiamo lavorati fino ad ora) lungo tutte le direzioni, le quali (ovvero le direzioni) sono:

- gli obiettivi del sistema;
- i concetti;
- le domain properties;
- system requirements;
- responsabilità.

La specifica e la documentazione dei requisiti, quindi, sono passaggi fondamentali nel processo di sviluppo di sistemi, in quanto assicurano che tutte le parti interessate abbiano una comprensione chiara delle caratteristiche concordate del sistema.

> 💡 Uno degli obiettivi, capiamo immediatamente, è di arrivare a **scrivere il documento dei requisiti**.

La prima modalità per descrivere i requisiti è il **linguaggio naturale**, i quali **vantaggi** sono:

- enorme espressività;
- molto comunicativo;
- non è necessario alcun training per essere utilizzato.

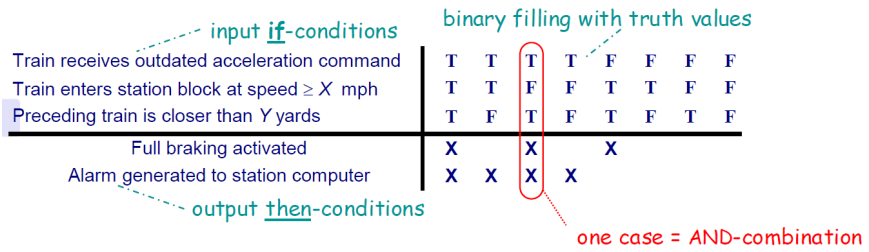
Gli **svantaggi**, invece, sono:

- la natura illimitata del linguaggio naturale, lo rende suscettibile a errori di specifica e difetti;
- attraverso il linguaggio naturale il testo è soggetto ad ambiguità, ovverosia la stessa frase può essere interpretata in diverse modi da persone diverse. Capiamo, allora, che attraverso il linguaggio naturale la semantica non è univoca e di conseguenza, la stessa frase può essere soggetta ad interpretazioni diverse.

Ci sono delle linee guida, che ci permettono di evitare i problemi, durante la scrittura dei requisiti, del linguaggio naturale. In particolare, alcune di queste linee guida sono:

- dobbiamo chiederci chi leggerà questo documento e quindi qual è il suo background, in modo tale da scrivere il documento in maniera opportuna;
- dobbiamo spiegare cosa stiamo per descrivere ancora prima di descriverlo effettivamente;
- dobbiamo prima scrivere le motivazioni per cui ci serve il requisito e poi lo andiamo a sintetizzare;
- dobbiamo definire i concetti prima di utilizzarli;
- ogni volta che scriviamo una frase, dobbiamo chiederci se essa è sufficientemente comprensibile e se effettivamente è rilevante per il progetto software;
- dobbiamo scrivere un solo requisito/proprietà di dominio per ogni frase;
- utilizzare ‘shall’ e ‘should’ in maniera corretta;
- è buona norma utilizzare degli esempi concreti per spiegare dei concetti astratti.

Oltre al diagramma, per descrivere i requisiti, è funzionale aggiungere anche una tabella, che prende il nome di ‘**decision table**’. Vediamo un esempio di decision table:



Come possiamo vedere dall'immagine, abbiamo:

- delle **condizioni in input** → che nel nostro esempio sono:
  - il treno riceve un comando di accelerazione non aggiornato;
  - il treno entra nel blocco stazione alla velocità ≥ X mph;
  - il treno precedente è più vicino di Y metri.
- delle **condizioni in output** → che nel nostro esempio sono:
  - frenata completa attivata;
  - allarme generato al computer di stazione.
- sul lato destro della tabella, abbiamo **tutte le possibili combinazioni dei nostri input** ed in particolare, con il simbolo X, indichiamo le azioni che vengono intraprese.

> 💡 Notiamo, che **attraverso la decision table rendiamo la frase, scritta in linguaggio naturale, non più ambigua**. Inoltre, si tratta di un metodo **completo** e il problema di ciò, è che la tabella diventa di dimensione 2^n, dove 'n' è il numero degli input →per alleviare questo problema, possiamo compattare l'input (o meglio dire, le colonne della tabella), nel caso in cui gli output sono uguali.
> Il vantaggio di avere questo approccio tabulare è che mi permette di scrivere (già in questa fase) i test di accettazione, ovverosia possiamo già scrivere i test, che il sistema (una volta implementato) dovrà soddisfare.

Un'altra regola per scrivere del linguaggio naturale in maniera comprensibile e funzionale, in modo tale da scrivere correttamente i requisiti, sono:

- adottare un criterio per trasformare i requisiti generici in requisiti misurabili (ovvero in **requisiti quantitativi**), in modo tale da facilitare la decisione se

un requisito è raggiunto o meno dal sistema, nel momento in cui quest'ultimo verrà implementato.

Oltre alle regole per scrivere correttamente i requisiti, è opportuno anche darsi delle **regole globali**, in modo tale da organizzare il documento dei requisiti in capitoli e conseguentemente, per raggruppare i concetti per similarità. Tipicamente, la scrittura del documento dei requisiti viene guidata da un **template**, il quale può essere generico oppure può essere specifico dell'organizzazione per cui stiamo lavorando. In particolare, il template IEEE Std-830 fornisce una struttura standardizzata per organizzare il documento dei requisiti, includendo specifiche per scopo, dominio e altri elementi critici. Questo standard aiuta le organizzazioni a mantenere una documentazione chiara e conforme alle migliori pratiche del settore.

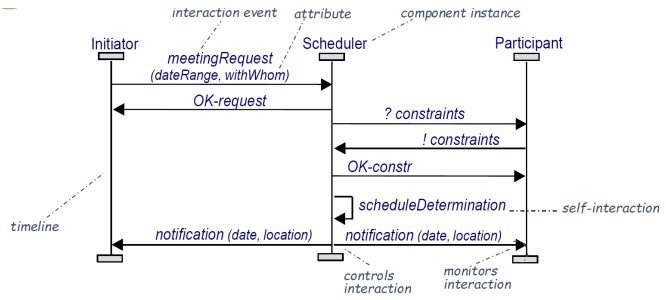
Una modalità alternativa per descrivere il sistema è di utilizzare **delle notazioni diagrammatiche.** Alcuni esempi di queste notazioni sono:

- **diagrammi entità-relazione** → essi dichiarano quali sono gli oggetti concettuali (all'interno dei requisiti) e quali sono le loro relazioni. Capiamo, allora, che:
  - l'**entità** è un'istanza di concetto e può avere identità distinte. Inoltre, può raccogliere le caratteristiche condivise in tutte le istanze delle molteplici entità;
  - le entità sono collegate attraverso **relazioni** N-arie (in particolare, le relazioni possono essere ternarie oppure binarie) e ogni relazioni specifica le molteplicità.

  Nei diagrammi entità-relazione dobbiamo citare anche due concetti fondamentali:

  - **specializzazione delle entità** → le entità possono essere specializzate in sotto-classi con caratteristiche specifiche (attributi, relazioni). Ad esempio, un ImportantParticipant eredita attributi e relazioni dalla superclass Participant, ma può avere attributi aggiuntivi come Preferences;
  - **annotazioni dei diagrammi** → essenziali per definire con precisione gli elementi ed evitare errori di specifica. Ad esempio, l'annotazione per Participant può specificare che una persona attesa alla riunione in un ruolo specifico.

- **diagrammi SADT** (Structured Analytics Design Technique) → non ci interessano per l'esame;
- **dataflow diagrams** → essi ci descrivono quali sono i flussi dei dati e sono più espressivi rispetto ai diagrammi SADT, in quanto specificano come i dati vengono generati e trasformati durante l'esecuzione del nostro sistema;
- per rappresentare le operazioni del sistema possiamo utilizzare gli **use case diagrams** → essi catturano tutte le operazioni, che possono essere intraprese all'interno del sistema con la relativa responsabilità;
- **event trace diagrams** → essi catturano lo scenario di utilizzo o di esecuzione del system-to-be. Intuitivamente, quindi, possiamo dire che l'event trace diagram rappresenta una serie di fotografie del nostro sistema. Vediamo un esempio:



Dall'esempio possiamo vedere che abbiamo tre attori (initiator, scheduler e partecipant) e ci focalizziamo sulle interazioni che hanno i vari attori (per esempio: l'initiator interagisce con lo scheduler mandando un meetingRequest e da notare che possiamo specificare gli attributi. A sua volta, lo scheduler risponde con una Ok-request). Possiamo, allora, osservare che ogni interazione ha una sorgente ed una destinazione e possiamo intendere, che il tempo scorre dall'alto verso il basso (così possiamo leggere le interazioni dall'alto verso il basso in ordine cronologico);

- per rappresentare il funzionamento del sistema, possiamo anche utilizzare gli **state machine diagrams** → essi rappresentano gli stati all'interno dei quali transisce il nostro sistema. Inoltre, ci permettono di rappresentare le transizioni da uno stato ad un altro.

> 💡 Notiamo, che con il termine ‘stato’ intendiamo un insieme di variabili, che hanno uno stesso valore consistente.

Anche in queste tipologie di diagrammi si parte da uno stato iniziale e si arriva ad uno stato finale. Riassumendo, quindi, possiamo dire che: I diagrammi delle macchine a stati catturano i comportamenti ammissibili dei componenti del sistema. Essi rappresentano la sequenza delle transizioni di stato per gli elementi controllati. Per comprendere questi diagrammi, abbiamo bisogno di conoscere i seguenti concetti:

- **Comportamento di un’istanza di componente** → sequenza di transizioni di stato per gli elementi che controlla;
- **Stato della Macchina a Stati** → insieme di situazioni in cui una variabile che caratterizza un elemento controllato ha sempre lo stesso valore. Ad esempio, lo stato “MeetingScheduled” ha sempre lo stesso valore per “Date” e “Location”;
- **Stati Iniziali e Finali** → stati in cui l’elemento appare o scompare. Gli stati possono avere una certa durata;
- **Transizione di Stato della Macchina a Stati** → causata da un evento associato. Se l’elemento `e nello stato di origine e l’evento si verifica, allora passa allo stato di destinazione. Gli eventi sono fenomeni istantanei.

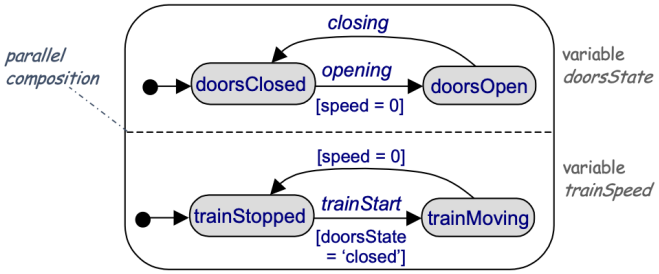
I **vantaggi** degli state machine diagrams sono essenzialmente due:

1. permettono di modellare chiaramente i comportamenti ammissibili dei componenti del sistema;
2. forniscono una rappresentazione dettagliata delle transizioni di stato.

Gli **svantaggi**, invece, sono:

1. possono diventare complessi con un numero elevato di stati e transizioni;
2. richiedono una comprensione dettagliata dei comportamenti dei componenti.

Per evitare l’esplosione combinatoria di stati è possibile utilizzare gli **statecharts**, i quali permettono di modellare comportamenti complessi in modo più chiaro e conciso.



Gli ingegneri del software mettono a disposizione la possibilità di verificare se vi è consistenza tra i diversi diagrammi o (al contrario) se vi sono delle inconsistenze. Capiamo, allora, che vi sono dei vincoli di consistenza sia sui singoli diagrammi sia sulle rappresentazioni che collegano i vari diagrammi.

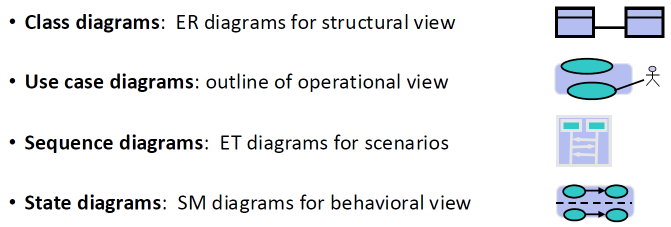
Avendo a disposizione una varietà di diagrammi per rappresentare i requisiti di sistema, possiamo integrare i diversi diagrammi dato che ciascuno fornisce una prospettiva unica del sistema. Una prima verifica, data dai sistemi automatici, può essere fatta per verificare la consistenza tra i diagrammi. Le inconsistenze che si possono verificare possono essere diverse, tra cui:

- tutti i componenti di un problem diagram devono essere presenti in un diagramma ER;
- tutti i fenomeno di un problem diagram devono essere presenti in un diagramma ER come un'entità, un attributo o una relazione.

UML fornisce una buona base di partenza per supportare la specifica dei requisiti, ed in particolare abbiamo che:

- il diagramma delle classi (class diagram) può essere utilizzato per rappresentare i diagrammi ER;
- gli use case diagrams possono essere utilizzati per rappresentare le operazioni;
- i sequence diagrams possono essere utilizzati per rappresentare gli event trace diagrams;
- gli state diagrams possono essere utilizzati per rappresentare le state machine.

Capiamo, allora, che utilizziamo dei linguaggi grafici ampliamente diffusi per rappresentare dei concetti specifici dell'ingegneria dei requisiti.



I **vantaggi** di questa notazione diagrammatica sono:

- Sono diversi dalla presentazione in linguaggio naturale, permettendo quindi di essere guidati;
- Hanno una semantica precisa, non ambigua;
- Mezzo di informazione più funzionale per il dialogo con gli stakeholder;
- Hanno una struttura grafica che permette di avere una panoramica di alto livello;
- Sono facili da capire e da comunicare;
- Permettono di fare un'analisi di primo livello e sono supportati da tool.

Gli **svantaggi**, invece, sono:

- Il linguaggio semantico può essere vago (con interpretazioni differenti);
- Spesso vengono formalizzati solo gli aspetti più evidenti e superficiali del sistema, lasciando non formalizzate le proprietà dettagliate degli elementi, il che può limitare la profondità dell'analisi;
- L'analisi automatizzata è limitata;
- Si modellano solo aspetti funzionali e strutturali.

Requirements Quality Assurance

Arrivati a questo punto, abbiamo documentato i requisiti e dobbiamo arrivare ad una versione consolidata dei requisiti, ovverosia dobbiamo validare e verificare i nostri requisiti, in modo tale da assicurarci che i requisiti:

- siano stati specificati in maniera corretta;
- siano consistenti;
- non contengano errori → è opportuno individuare gli errori in questa fase, perchè è economicamente vantaggioso risolverli. Quindi, trovare un errore successivamente potrebbe essere più costoso (e magari se il sistema è già progettato, un errore ci costringe a rivedere la progettazione del sistema e rifare delle implementazioni, che magari ci costringono ad accettare dei compromessi, i quali vanno a ridurre la qualità del sistema software).

La quality assurance tipicamente è articolata in due step:

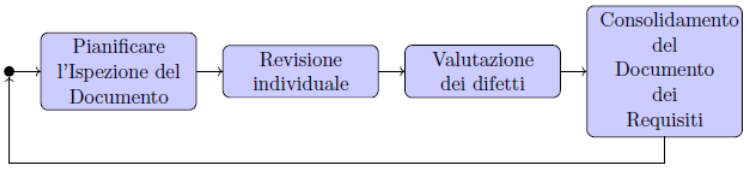
1. **identificare possibili errori o problemi nei requisiti** → questo primo step lo si effettua attraverso la validazione, la verifica e il controllo dei requisiti;
2. una volta identificati gli errori o i problemi, **dobbiamo documentarli** in modo tale da re-iterare il processo di quality assurance, per riuscire ad analizzare le cause che hanno portato a questi errori o problemi e risolverli (gli errori e i problemi).

> 💡 L'output di questa operazione, quindi, è la versione consolidata del documento dei requisiti. Oltre a tale documento possiamo considerare anche i test di accettazione consolidati, eventualmente un prototipo, un piano di sviluppo e potenzialmente un contratto di sviluppo software.

Il primo modo in cui possiamo verificare la qualità del documento dei requisiti è attraverso le **ispezioni e le revisioni** → questo consiste nell'identificare delle persone, le quali dovranno leggere il documento dei requisiti e lo valutano e lo correggono, in modo tale che sono loro a segnalare eventuali errori o problemi. Questa è una metodologia abbastanza efficace, in quanto si riescono ad individuare dei problemi rilevanti (soprattutto se questa metodologia viene fatta da persone diverse da quelle che hanno scritto i requisiti). Principalmente vengono utilizzate due tecniche:

1. Walkthrough → si tratta di una lettura del documento, con l'obiettivo di trovare errori;

2. Più strutturato → utilizzando degli ispettori esterni, dei meeting ben strutturati con gli ingegneri dei requisiti, report o altro. Se utilizziamo questo approccio strutturato, tipicamente viene articolato nelle seguenti fasi:



- Pianificare l'ispezione del documento → si tratta di definire chi sono gli ispettori,
  come si svolgerà l'ispezione, quali sono gli obiettivi e quali sono i criteri di accettazione;
- Revisione individuale → ogni ispettore legge il documento e cerca di individuare errori.
  Tale fase può seguire diverse modalità:
  - liberamente → ogni ispettore legge il documento e cerca di individuare errori;
  - guidata mediante checklist → ogni ispettore segue una lista di controllo basata sul
    dominio;
  - basato sul processo.
- Valutazione dei difetti → si tratta di valutare i difetti capendo come sono stati generati
  e come possono essere corretti;
- Consolidamento del documento dei requisiti → si tratta della fase in cui viene
  consolidato il documento dei requisiti.

  Tale processo può essere ripetuto più volte, fino a quando non si è soddisfatti della qualità del documento.

A questo punto, vediamo quali sono le **linee guida per la revisione**:

- Rapporto di Ispezione:
  - Deve essere informativo, accurato e costruttivo;

    - Non deve contenere opinioni personali o commenti che possano risultare offensivi;
    - Deve seguire una struttura standard che permetta l'aggiunta di commenti liberi e
      sia di facile lettura;
    - Può servire da guida per la revisione individuale.
- Gli ispettori:
    - Devono essere indipendenti dagli autori del documento sotto revisione;
    - Devono rappresentare tutti gli stakeholder e provenire da background diversi.
- Tempistica dell'Ispezione:
    - L'ispezione non deve avvenire n´e troppo presto n´e troppo tardi;
    - Incontri brevi e ripetuti risultano più efficaci.
- Focus Maggiore su Parti Critiche:
    - Maggiore è il numero di difetti in una parte, maggiore deve essere lo scrutinio su
      quella parte.

L'obiettivo principale delle liste di controllo per l'ispezione `e di dirigere la ricerca dei difetti verso specifiche problematiche, migliorando così l'efficienza e l'efficacia del processo di revisione. Le liste di controllo si suddividono in diverse categorie, ognuna con un focus particolare:

- **Liste basate sui difetti** → queste liste comprendono domande generiche che sono strutturate
  in base al tipo di difetto. L'intento è di coprire un ampio spettro di possibili anomalie in modo organizzato;
- **Liste specifiche per qualità** → tali liste affinano le domande delle liste basate sui
  difetti per concentrarsi su specifiche categorie di requisiti non funzionali (NFR), come
  la sicurezza, la performance e l'usabilit`a. Queste liste possono anche essere basate su parole guida e sono frequentemente orientate a identificare omissioni, migliorando così la completezza del software o del prodotto analizzato;

- **Liste specifiche per dominio** → queste liste rappresentano un'ulteriore specializzazione
  e si concentrano sui concetti e le operazioni specifici di un dominio. Sono particolarmente
  utili per offrire una guida dettagliata nella ricerca di difetti, assicurando che le peculiarità del dominio siano adeguatamente considerate;
- **Liste basate sul linguaggio** → queste liste adattano le liste basate sui difetti ai costrutti di linguaggi di specifica particolari, supportando processi di automazione. La ricchezza dei linguaggi di specifica consente implementazioni di controlli più sofisticati e mirati.

I **vantaggi** delle checklist sono essenzialmente due:

- **maggiore efficacia rispetto all'ispezione del codice** → in quanto vi è l'utilizzo di un modello basato su processi, che combina checklist basate sui difetti, specifiche per qualità e specifiche per linguaggio, risultando estremamente efficace;
- **ampia applicabilità** → sono adatte per ogni tipo di difetto e formato di specifica, rendendole strumenti versatili.

Gli **svantaggi**, invece, sono:

- **onere e costo del processo di ispezione** → la dimensione del materiale di ispezione e il tempo/costo necessari per gli ispettori esterni e le riunioni di revisione possono essere significativi;
- **nessuna garanzia di rilevare tutti i difetti importanti** → nonostante l'efficacia, le checklist non possono garantire il rilevamento di tutti i difetti critici.

Un'altra forma di quality assurance è rappresentata dalla **query** → rappresenta una metodologia alternativa per validare i nostri requisiti. In particolare, quando utilizziamo dei diagrammi, vi sono dei tool (chiamati CASE tool) che prendono i diagrammi ed estraggono l'informazione disponibile e mettono tale informazione all'interno di un database. Quindi, questi tool sono in grado di creare uno schema logico partendo dai diagrammi e popolare le tabelle del database → quindi, le verifiche della consistenza dei dati possono essere rappresentate come query al database (grazie all'utilizzo di linguaggi specifici per le query), che permettono in maniera automatizzata di implementare le verifiche dei requisiti come query del database.

Un'altra forma di verifica dei requisiti è quella che passa attraverso le **animazioni** → abbiamo visto, che vi sono diverse forme di specifica dei requisiti e una di queste (forme) sono gli scenari. Gli scenari, però, possono essere visualizzati attraverso un sequence diagram oppure attraverso un'animazione → l'animazione è molto funzionale, perchè permette di validare i requisiti sia attraverso i revisori sia attraverso gli stakeholders. Questo processo può essere attuato attraverso due principali approcci:

1. **Visualizzazione di Scenari di Interazione** → questo approccio consiste nel mostrare scenari di interazione concreti in azione, utilizzando strumenti di attuazione sui diagrammi di Evento Tempo (ET). Tuttavia, questo metodo presenta sfide legate alla copertura completa degli scenari, che può risultare incompleta;
2. **Utilizzo di Strumenti di Animazione delle Specifiche** → l'animazione delle specifiche si svolge in quattro fasi principali:
    a. generazione del modello esecutivo → si estrae o si genera un modello eseguibile direttamente dalle specifiche tecniche;
    b. simulazione del comportamento → il sistema viene simulato basandosi su questo modello, dove eventi stimolo vengono inviati per imitare il comportamento dell'ambiente circostante e si osserva la risposta del modello;
    c. visualizzazione della simulazione → la simulazione viene visualizzata mentre è in corso, permettendo una valutazione immediata e dinamica del comportamento del sistema;
    d. feedback degli utenti → i feedback degli utenti vengono raccolti e analizzati per valutare l'efficacia della simulazione e del modello proposto.

I **vantaggi** delle animazioni sono:

- **verifica dell'adeguazione** → metodo efficace per controllare l'adeguatezza dei requisiti rispetto alle necessità reali e all'ambiente attuale;
- **coinvolgimento degli stakeholder** → il principio "WYSIWYC" (What You See Is What You Check) permette agli stakeholder di visualizzare e verificare i requisiti direttamente;
- **estensione a controesempi** → possibilità di animare controesempi generati da altri strumenti come i verificatori di modelli, migliorando così la

robustezza del sistema;

- **riutilizzo degli scenari** → gli scenari di animazione possono essere conservati per replay futuri e usati come dati di test di accettazione.

Gli **svantaggi**, invece, sono essenzialmente due:

- **problema della copertura** → la progettazione degli scenari deve essere attentamente gestita per assicurare che non vengano tralasciati problemi significativi. Quindi, non vi è garanzia di identificare tutti i difetti importanti;
- è necessaria una **specifica formale dettagliata** per eseguire un'animazione efficace dei requisiti, richiedendo un investimento considerevole in termini di tempo e risorse.

Requirement evolution

Nelle lezioni precedenti abbiamo visto: come avviene l'elicitazione dei requisiti, come possiamo valutare le possibili alternative, in modo da arrivare ad una soluzione condivisa ed infine, abbiamo visto come dobbiamo documentare i requisiti, in modo tale da valutare e verificare i requisiti, in modo tale da poter certificare la qualità del lavoro svolto → intuitivamente, potremmo pensare che abbiamo concluso il ciclo del processo, ma in realtà quest'ultimo non è così lineare, ovverosia potenzialmente potremmo ciclare più volte i nostri quadranti (ovvero le fasi del processo), in modo tale da avviare cicli successivi di elicitazione, valutazione, specifica e validazione dei nostri requisiti → questa necessità di ciclare più volte i quadranti, deriva dal fatto che i requisiti possono evolvere nel tempo e anche perchè la conoscenza sui nostri requisiti potrebbe evolvere.

Come abbiamo detto precedentemente, i requisiti possono evolvere perchè l'azienda nel quale si lavora può cambiare e possono anche cambiare le necessità dei nostri utenti e degli stakeholders. In generale, anche se i requisiti non dovessero cambiare (cosa molto difficile) comunque potrebbe cambiare la conoscenza e il grado di familiarità che noi (come ingegneri dei requisiti) potremmo avere dei requisiti → per questi motivi, quindi, **è opportuno dotarsi di una metodologia strutturata per gestire e documentare l'evoluzione dei requisiti e valutarne gli impatti** (in modo tale da non subire l'evoluzione e il cambiamento). Fino ad ora abbiamo parlato di 'cambiamento', ma sorge spontanea la domanda: "**Cosa può riguardare il cambiamento?**" Il cambiamento può riguardare:

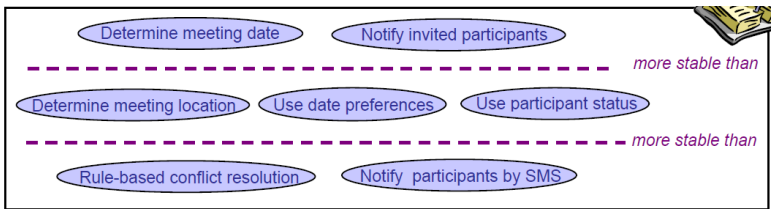
- le **funzionalità** → possono essere aggiunte delle nuove funzionalità e questo deve essere opportunamente gestito;
- **revisioni** e **varianti** → quando parliamo di 'revisione' abbiamo evoluzione del nostro requisiti (quindi vengono aggiunti e/o eliminati dei 'pezzi' ai nostri requisiti) e conseguentemente, una maturazione della conoscenza. Le varianti, invece, riguardano appunto le diverse varianti del prodotto per il quale andiamo a specificare i requisiti → questo significa, che le molteplici varianti sono vere contemporaneamente (basti pensare alle diverse linee produttive all'interno di un'organizzazione).

Vi possono essere molteplici motivi per accogliere un cambiamento all'interno dei nostri requisiti. Alcuni motivi sono i seguenti:

Cause	Change type	Version type	Change time
errors & flaws	corrective	revision	RE, design, implem, post-deployment
better understanding	corrective extension	revision	RE, post-deployment
new functionality	extension	revision, variant	post-deployment
improved feature	ameliorative	revision	post-deployment
new users/usage	adaptative	variant	RE, design, post-deployment

Come abbiamo detto precedentemente, dobbiamo arrivare ad identificare un cambiamento e prepararci, in modo tale da non subire il cambiamento, bensì dobbiamo organizzare i requisiti in maniera tale, che siano facilmente modificabili (in questo modo, i cambiamenti non abbiano un costo troppo elevato e il loro impatto sia basso). Per riuscire ad arrivare preparati al cambiamento dobbiamo:

1. identificare i requisiti che potrebbero cambiare con maggiore probabilità (tipicamente ciò lo si capisce già in fase di elicitazione dei requisiti) → una delle modalità principali per **identificare la probabilità di cambiamento dei requisiti è di raggruppare i requisiti (in maniera omogenea) per probabilità di cambiamento**;



   L'obiettivo di ciò (ovvero di avere dei livelli di stabilità dei requisiti) è di avere un confronto tra i requisiti più stabili rispetto a quelli meno stabili, ovvero vogliamo avere una definizione qualitativa della stabilità → per identificare la stabilità di un requisito in questa fase, possiamo affidarci a delle **regole euristiche**.

2. esplicitare i vincoli tra i singoli requisiti che abbiamo compreso, ovverosia **dobbiamo essere in grado di esplicitare la tracciabilità dei requisiti**. Un requisito è tracciabile quando siamo in grado di capire:
   a. il motivo per cui il requisito è stato scritto;

b. da dove proviene il requisito (è stato detto da uno stakeholder oppure è una conoscenza di dominio?);

c. chi utilizzerà la funzionalità.

> 💡 In questo modo, se successivamente dovremmo discutere del requisito, possiamo analizzare se le motivazioni che hanno portato all'emissione del requisito sono ancora valide oppure no.

Oltre ad avere un link all'indietro del requisito (ovvero oltre a capire il motivo, da dove proviene e chi lo utilizzerà) è importante anche avere un link in avanti del requisito, in quanto il requisito potrebbe essere la base di un altro requisito (quindi il requisito potrebbe essere la motivazione di un altro requisito) → di conseguenza, se il requisito cambia allora dovremmo vedere anche tutti gli altri requisiti che discendono da esso. Riassumendo, quindi, possiamo dire che: Un requisito è tracciabile se è possibile capire:

- da **dove** proveniva;
- **perchè** è stato inserito;
- per **cosa** verrà usato;
- **come** verrà soddisfatto.

Per implementare la tracciabilità ogni singolo elemento del documento dei requisiti deve essere documentato con un identificativo alfanumerico.
L'obiettivo della tracciabilità è quello di essere in grado di capire l'impatto di un cambiamento (da notare, che Il cambiamento potrebbe avere un impatto anche su diagrammi legati al documento dei requisiti, perciò è necessario che anche questi siano modificati). Il cambiamento ha due dimensioni:

- orizzontale → ha impatto sullo stesso dominio di applicazione;
- verticale → ha impatto su un dominio di applicazione diverso (componenti architetturali, codice).

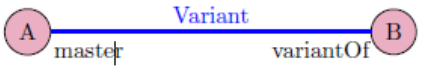
La tracciabilità può essere:

- all'**indietro** → per risalire alle motivazioni;
- in **avanti** → per capire l'impatto di un cambiamento.

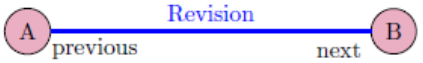
Oltre a ciò, esiste anche una **tassonomia** dei link di tracciabilità:

I link di tracciabilità possono essere di diversi tipi:

- **Link inter-versione**: tra versioni differenti:
  - Variante: ho due varianti del software differente (*premium, standard, ...*).

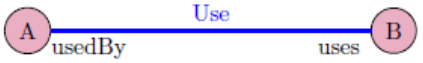


  - Revisione: si considera la dimensione temporale, dove ho una versione precedente e una successiva.

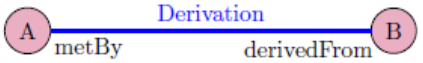


  È possibile aggiungere ulteriori annotazioni per facilitare la tracciabilità.

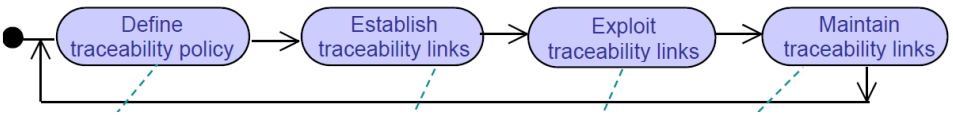
- **Link intra-versione**: tra elementi della stessa versione:
  - **Link di uso**: ho un link di uso quando il requisito A usa il requisito B.



  - **Link di Derivazione**: un requisito B è derivato da un requisito A.



A questo punto, dobbiamo dotarci di un **sistema di gestione della tracciabilità** → prima di iniziare è opportuno fare una riflessione su quanto vogliamo spingerci nella gestione della tracciabilità, perchè (come possiamo intuire) la tracciabilità ha un costo, visto che i link devono essere disegnati, capiti e mantenuti.



Naturalmente, questa scelta di quanto spingerci con la tracciabilità dipende dal progetto in sè e dalle sue dimensioni e anche dalla dimensione del team. Capiamo, allora, che:

- Prima di affrontare la tracciabilità è opportuno definire in anticipo quali sono le policy della tracciabilità, ovverosia dobbiamo definire in anticipo cosa vogliamo tracciare (questo naturalmente dipende sempre dall'organizzazione in cui si sta lavorando, dal tipo di progetto e dal team) → definire la policy significa definire qual è la granularità dei link che vogliamo propagare;

- Una volta definite le policy, il passo successivo è di disegnare i link di tracciabilità (in modo tale da poterli iniziare ad utilizzare), ovverosia andiamo a capire qual è il tipo dei link e quali sono i dati che vogliamo associare ad ogni link;
- Una volta che abbiamo terminato il 2° step, otteniamo il grado della tracciabilità e di conseguenza, possiamo passare al terzo step, che consiste nell'utilizzare effettivamente i link di tracciabilità. Le finalità dei link di tracciabilità sono molteplici:
    - sono un supporto alla documentazione dei requisiti perchè permettono di identificare le dipendenze quando discutiamo o ridefiniamo un requisito. Verificando l'impatto di qualche modifica;
    - posso ripercorrere all'indietro i link per verificare perchè un requisito è stato inserito;
    - analisi di copertura.
- Infine, abbiamo la manutenzione dei link, ovverosia quando abbiamo dei cambiamenti dobbiamo aggiornare anche i link di tracciabilità. Quindi, esso devono rimanere corretti e accurati durante tutto il ciclo di vita del progetto.

Metodologie di tracciabilità

- **Cross-referencing** → si basa sull'uso di un identificativo univoco per ogni elemento del documento, che permette di collegare i vari elementi, riferendosi a questi identificativi. Tali link supporteranno l'opzione di browsing e di ricerca. I **vantaggi** del cross-referencing sono:
    - facilità di implementazione;
    - facilità di utilizzo;
    - disponibili nella maggior parte degli strumenti di gestione dei documenti.

    Gli **svantaggi**, invece, sono:
    - i link sono tutti dello stesso tipo, non portando informazione semantica (semplice link ipertestuale);
    - non si traccia la motivazione, perciò aggiungiamo maggiore overhead.
- **Matrici di tracciabilità** → le matrici possono essere rappresentate per rappresentare grafi, adatte quindi per la rappresentazione di link di

tracciabilità. I **vantaggi** sono:

- facilità di navigazione backward e forward;
- semplice forma di analisi di copertura.

Gli **svantaggi**, invece, sono:

- aumenta la complessità con il crescere del numero di requisiti. La complessità è quadratica rispetto al numero di requisiti;
- abbiamo solamente informazioni binarie, quindi non possono rappresentare informazioni più complesse.

- **Liste di tracciabilità** → invece di rappresentare i link come matrici, possiamo rappresentarli come liste. Per ogni item della lista indichiamo quali sono gli archi uscenti di quell'item. I **vantaggi** sono:
  - più compatta rispetto alle matrici;
  - la navigazione in avanti è più semplice.

  Gli **svantaggi**,, invece, sono:
  - la navigazione all'indietro è più complessa;
  - la struttura dei dati è innaturale.
- **Feature diagram** → i feature diagram ci permettono di ragionare in maniera più naturale, rispetto alle strutture dati precedenti. Abbiamo una rappresentazione gerarchica delle feature. In particolare, abbiamo che:
  - Rappresentiamo con un arco l'aggregazione delle feature, dove la feature di più basso livello contribuisce alla feature di livello superiore;
  - Inseriamo un pallino pieno per rappresentare l'obbligatoriet`a di una feature;
  - Inseriamo un pallino vuoto per rappresentare la opzionalità di una feature;
  - Inseriamo un collegamento tra due archi per rappresentare la relazione di OR non-esclusivo;
  - Inseriamo un collegamento tra due archi e lo riempiamo, per rappresentare la relazione di OR esclusivo.

  I **vantaggi** sono:

    - rappresentazione più compatta e un possibile grande numero di varianti;
    - Rappresentiamo più versioni, permettendoci di ragionare su tutta la product line.
- **Database di tracciabilità** → in generale con i link è opportuno avere un database che permetta di gestire i link in modo efficiente. Il database permette di gestire in modo efficiente i link, permettendo di fare query complesse. I **vantaggi** sono:
    - molto scalabile rispetto al numero di requisiti;
    - generico, quindi è possibile estenderlo con diversi tipi di link;
    - ci sono tanti tools che permettono di gestire i database declinati per la tracciabilità.

  Gli **svantaggi**, invece, sono:
    - la customizzazione manuale può richiedere effort;
    - l'informazione di tracciabilità potrebbe non essere strutturata, ma solamente tipizzata.

Sorge a questo punto spontanea la domanda: "**Come faccio a generare i link di tracciabilità?**" Le prime modalità sono basate su tecniche di information retrieval, utilizzando delle keyword. Si cercando delle similarità tra il testo con altre parti del documento. I **vantaggi** dei generatori di link sono essenzialmente due:

- automatizzare la scoperta dei link e la loro manutenzione;
- applicabile in maniera semplice quando il documento dei requisiti è scritto in linguaggio naturale.

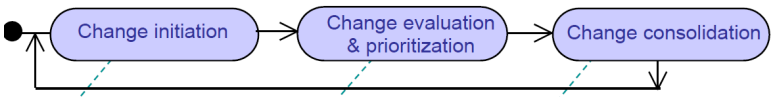
Gli **svantaggi**, invece, sono:

- puramente lessicali;
- potrebbero avere falsi positivi e falsi negativi.

Dobbiamo anche capire, qual è il compromesso ottimale tra l'overhead introdotto dalla generazione dei link con il beneficio che si potrebbe avere nel proprio progetto software. Il problema principale che possiamo riscontrare è quello della **delayed gratification**, ovverosia noi paghiamo

adesso un costo per un beneficio che vedremo in futuro (perchè avremo dei requisiti più facilmente gestibili e corretti).

Arrivati a questo punto, abbiamo documentato i link di tracciabilità e adesso vediamo come servircene per controllare il cambiamento. Durante il change control adottiamo i seguenti step:



Diciamo immediatamente, che quando viene proposto un cambiamento, quest'ultimo dovrebbe essere motivato per essere successivamente discusso. Una volta recepito una richiesta di cambiamento, questa (ovvero la richiesta di cambiamento) deve essere valutata nel secondo step, nel quale le richieste vengono discusse e prioritizzate. Le richieste che vengono approvate devono far attivare un nuovo ciclo dell'aspirale dei requisiti (ovverosia i requisiti devono essere: consolidati, valutati e verificati), in modo tale da poter entrare nel documento dei requisiti. Analizziamo, a questo punto, in maniera più approfondita i singoli step:

- **change initiation** → il processo parte con l'identificazione del cambiamento, il quale può accumulare richieste di cambiamento sia da membri del team sia da esterni. In particolare, le richieste di cambiamento vengono accumulate in una lista (di cambiamenti), che viene successivamente discussa → per discutere la lista dei cambiamenti è opportuno utilizzare un board e vanno documentate: le motivazioni, i vantaggi e gli svantaggi dei cambiamenti e il loro impatto.

  Capiamo, allora, che qualcuno propone un cambiamento dei requisiti e tale cambiamento può essere:

  - correttivo → per correggere un errore;
  - adottivo → per adattare il sistema a nuove condizioni;
  - migliorativo → per migliorare il sistema.

  Da notare, che deve essere menzionato il livello di urgenza del cambiamento, il livello di impatto sul sistema e anche la classe di stakeholders che propone questo cambiamento;

- **change evaluation & prioritization** → spesso ci si dota di un team indipendente per valutare il cambiamento, che porta una rappresentanza di

tutti gli stakeholder. Viene valutato il costo in termini di costo e beneficio e il risultato sarà il decidere se accettare o meno il cambiamento → capiamo, allora, che il risultato di questo 1° step (ovvero il change initiation) è l'approvazione o meno di un cambiamento.;

- **change consolidation** → i cambiamenti approvati arrivano a quest'ultima fase, quindi abbiamo che i cambiamenti vengono adottati nel documento dei requisiti e quindi, viene emanata una versione successiva (quindi un'evoluzione) del documento dei requisiti. Questo consiste nel seguire i link, in modo tale da propagare le modifiche in avanti per gli items, che dipendono dal requisito che è stato cambiato.

Goal Orientation

Nei capitoli precedenti, abbiamo analizzato i vari passaggi del ciclo di vita dei requisiti, quindi: come raccogliere i requisiti, come valutarli e documentarli, come valutare alternative ed identificare i relativi rischi e conseguentemente come gestirli, in modo tale da arrivare a collezionare l'informazione necessaria, la quale rappresenta il dato che dobbiamo gestire attraverso l'ingegneria dei requisiti. Adesso parleremo di una metodologia (la più condivisa) per la documentazione dei requisiti, che prende il nome di **Goal Orientation** → ci poniamo immediatamente una prima domanda fondamentale: "**Come mai la metodologia è orientata ai goal**?" Tale metodologia è orientata ai goal, perchè il goal rappresenta un obiettivo che si vuole raggiungere. Quindi, il goal nell'ingegneria dei requisiti è un concetto intrinseco quando si parla di requisiti. Di fatto, abbiamo sempre avuto a che fare con gli obiettivi, e la metodologia goal-oriented definisce il concetto di goal come astrazione chiave che ci guida nell'ingegneria dei requisiti.

Abbiamo parlato di goal, ma effettivamente che "**cos'è un goal?**" Un goal è una frase (o meglio dire uno **statement**) **prescrittiva** di un desiderio e/o di un obiettivo, che il sistema dovrebbe raggiungere e soddisfare attraverso la cooperazione degli agenti. In maniera più formale, quindi, **un goal è uno statement prescrittivo che esprime un desiderio, un obiettivo che il sistema deve raggiungere attraverso la cooperazione dei suoi agenti** → si tratta di uno statement prescrittivo, perchè può essere espresso attraverso "shall", "should", "must" e così via. Un esempio di goal è: *Le porte del treno devono essere chiuse quando il treno è in movimento*.

I goal, inoltre, sono formulati in termini di fenomeni problematici del mondo che osserviamo e si applicano al sistema, dove per sistema intendiamo sia il system-as-is (quindi il sistema attuale) sia il system-to-be (quindi il sistema che dobbiamo realizzare). Abbiamo utilizzato anche il termine '**agente**', il quale è un componente attivo del sistema as-is e to-be ed è responsabile del raggiungimento di un goal → capiamo, allora, che gli agenti possono essere umani, software o hardware. L'agente quindi è un ruolo che prende decisioni, in modo tale che il goal assegnato venga raggiunto. Per riuscire a fare ciò, l'agente dovrà monitorare e controllare delle grandezze. Come abbiamo detto precedentemente, per raggiungere un goal è tipicamente necessaria la cooperazione di molteplici agenti (ad esempio, per raggiungere il goal di

garantire un trasporto sicuro all'interno del sistema ferroviario, gli agenti che cooperano sono: il sistema di controllo del treno, il sistema di tracciamento, i passeggeri, l'autista del treno) e di conseguenza, possiamo capire che l'agente più che essere una persona o un componente, il concetto di agente cattura un **ruolo** che la persona o il componente (hw o sw) riveste, perchè è strettamente legato al concetto di responsabilità → la responsabilità di un agente è di restringere tutti i possibili funzionamenti del sistema, in modo tale che gli egenti soddisfino i goal (esempio: l'attuatore delle porte del treno deve aprire e chiudere le porte solamente in determinati momenti, o meglio dire scenari, in modo tale che si manifestano solamente gli scenari che permettono di verificare il goa) → per riuscire a fare ciò, ovverosia per avere un impatto ed una conseguenza sul sistema, è necessario che l'agente abbia dei sensori e degli attuatori. Esistono tre diverse tipologie di agenti:

- **agenti software** → essi si trovano sia nel sistema che modelliamo, sia nei software già esistenti che comunicano con il software che dobbiamo realizzare;
- **dispositivi** → quindi sono principalmente sensori ed attuatori;
- **utenti** → hanno una responsabilità all'interno del sistema software.

A questo punto, dobbiamo fare una distinzione tra goal e proprietà di dominio → le proprietà di dominio non sono prescrittive, bensì sono statement descrittivi che riguardano l'ambiente, in quanto non vengono espresse con "shall", "should" e "must", bensì per le proprietà di dominio utilizziamo tipicamente "is", "are" e così via. Un esempio: "*Se le porte del treno sono aperte, non sono chiuse.*" → capiamo, allora, che sulle proprietà di dominio non possiamo lavorarci, perchè esse sono vere e basta e di conseguenza:

- non possono essere negoziate, non possono essere indebolite e non possono essere prioritizzate;

è comunque importante modellare e documentarle, perchè ci permettono di collegare i concetti del sistema.

I goal possono essere espressi a livelli di granularità diversi. In particolare, possono essere:

- **goal ad alto livello** → sono statement strategici, che hanno quindi una granularità alta, come ad esempio "Migliorare la qualità del servizio del 50%";

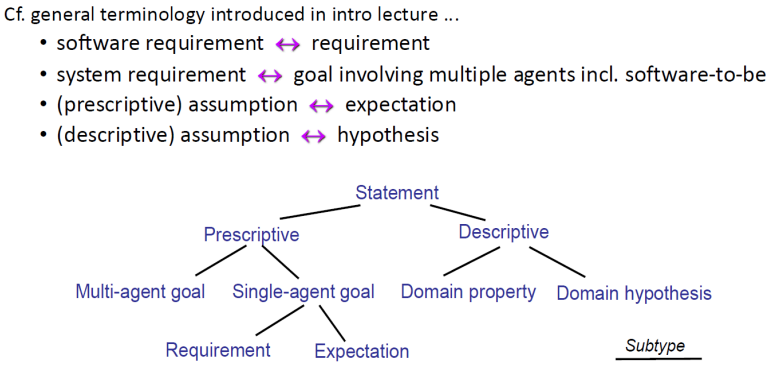
- **goal a basso livello** → sono statement tecnici, che hanno quindi una granularità bassa, come ad esempio “Il sistema invia un reminder per il rinnovo del prestito”.

> 💡 Chiaramente, **vi è una relazione tra i goal ad alto livello e i goal basso livello**, nel senso che: per raggiungere un goal al alto livello di granularità (quindi molto astratto) abbiamo bisogno di molteplici goal di basso livello, i quali insieme concorrono al goal di alto livello.

Abbiamo detto, che i goal possono essere assegnati agli agenti, nel senso che alcuni agenti hanno la responsabilità di raggiungere alcuni goal (esempio: l’operatore di treno ha la responsabilità di fare qualcosa, in modo tale che il treno sia in sicurezza) → capiamo, allora, che **più è astratto il goal e più agenti dovranno collaborare per raggiungere il goal; più il goal è concreto e meno agenti dovranno collaborare**. A questo punto, facciamo un’altra distinzione:

- **requirement**→ nella metodologia goal oriented, il requisito (ovvero il requirement) rappresenta una keyword e di conseguenza, ha un significato ben preciso che è: **Il requirement è un goal, che deve essere assegnato ad un unico agente nel software to-be** (come per esempio: il comando di accelerazione viene mandato ogni 3 secondi → l’agente responsabile è il computer della stazione) → di conseguenza, è responsabilità dell’agente raggiungere il requirement;
- **expectation**→ anch’essa nella metodologia goal oriented rappresenta una keyword, il cui significato è: **L’expectation è un goal, che deve essere soddisfatto da un unico agente nel environment**.

Riassumiamo, a questo punto, i concetti che abbiamo visto fino a questo momento, con il seguente schema:



Tutti gli statement possono essere prescrittivi oppure descrittivi. Gli statement prescrittivi, a loro volta, possono essere assegnati a più agenti oppure ad un singolo agente.

Tipologie di goal

Possiamo avere due tipologie di goal:

- **behavioral goals** → si tratta di goal prescrittivi, che esprimono il comportamento del sistema (quindi sono dei goal di tipo 'sì' oppure 'no'). Tali goal, a loro, volta si suddividono in due tipologie:
  - achieve goal → ovvero goal che riguardano il raggiungimento di una determinata condizione;
  - maintain/avoid goal → ovvero la condizione deve essere sempre vera/falsa.
- **soft goals** → esprimono una preferenza tra i funzionamenti distinti.

Analizziamo in maniera più approfondita queste due tipologie di goal:

- i behavioral goals sono statement prescrittivi e rappresentano dei vincoli, che rappresentano un sottoinsieme dei funzionamenti che il nostro sistema deve accettare a cui è possibile attribuire una risposta univoca, 'sì' oppure 'no' → tipicamente, i behavioral goal servono per restringere tutti i possibili funzionamenti del sistema a solamente quelli ammessi, ovverosia i behavioral goal vincolano quali sono gli stati in cui il sistema deve transire, per fare in modo che il goal venga raggiunto. Esistono diverse sotto-tipologie di behavioral goal:
  - achieve goal → rappresentano una condizione target da raggiungere e hanno la seguente forma:

Achieve [TargetCondition]:

- **[if** CurrentCondition **then]** **sooner-or-later** TargetCondition

Achieve [BookRequestSatisfied]:
if a book is requested **then sooner-or-later**
a copy of the book is borrowed by the requesting patron

- maintain goal → rappresentano che una condizione deve essere sempre vera e hanno la seguente forma:

Maintain [GoodCondition]:

- **[if** CurrentCondition **then]** **always** GoodCondition
- always (**if** CurrentCondition **then** GoodCondition)

Maintain *[DoorsClosedWhileMoving]:*
always (**if** a train is moving **then** its doors are closed)

- avoid goal → rappresentano che una condizione deve essere sempre falsa, ovvero evitano sempre una condizione indesiderata e hanno la seguente forma:

Avoid [BadCondition]: dual of *Maintain* ...

- **[if** CurrentCondition **then]** **never** BadCondition

Avoid *[BorrowerLoansDisclosed]:*
never patron loans disclosed to other patrons

- i soft goal catturano il concetto di preferenza tra le molteplici alternative. Quindi, i soft goal non possono essere soddisfatti in maniera booleana, in quanto riguardano maggiormente le preferenze (infatti i soft goal possono essere più o meno soddisfatti). Inoltre, contengono tipicamente delle grandezze da massimizzare o minimizzare oppure da incrementare o ridurre, come ad esempio *"La condizione di stress degli operatori deve essere minimizzata"*.

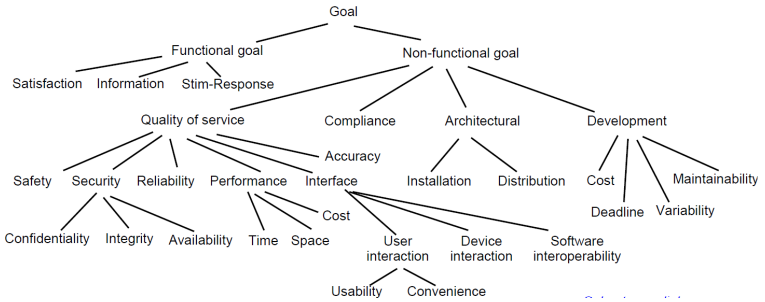
Categorie di goal

I goal possono essere classificati in diverse categorie, ma tale divisione non è sempre netta, in quanto un goal può appartenere a più categorie. Le categorie di goal sono:

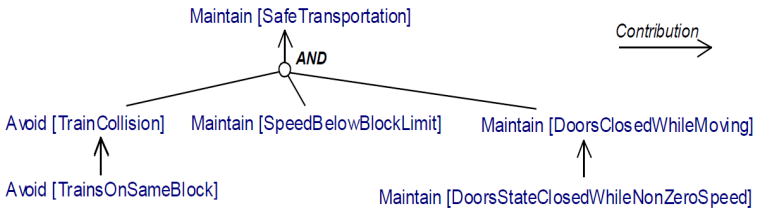
- **funzionali** → sono goal che esprimono le funzionalità e i servizi che il sistema dovrà offrire e vengono utilizzati per costruire un modello operativo del sistema;
- **qualitativi** → sono goal che esprimono le qualità che il sistema dovrà avere, come ad esempio: la sicurezza, l'usabilità, l'efficienza e le performance;

- **di sviluppo** → sono goal che esprimono le attività che dovranno essere svolte per lo sviluppo del sistema, come ad esempio: la documentazione e la formazione del personale.

I goal possono essere classificati secondo una tassonomia:



Come avevamo intuitivamente discusso precedentemente, vi sono dei goal più di alto livello (ovvero più astratti e molto generici) e goal di più basso livello (ovvero più concreti e specifici). Come avevamo già detto, i goal di basso livello sono collegati a quelli di alto livello, in quanto i goal di basso livello concorrono a raggiungere i goal di alto livello e tale relazione deve essere documentata. In questo senso, allora possiamo avere uno schema del genere:



vediamo che i goal sono rappresentati dai nodi del grafo, mentre le frecce dicono quali goal contribuiscono agli altri goal. Ad esempio, abbiamo il goal di mantenere il trasporto sicuro (il quale è un goal molto astratto) e per raggiungere tale goal, abbiamo dei goal più concreti (come ad esempio: evitare le collisioni tra i treni, mantenere le porte chiuse mentre il treno è in movimento) → capiamo, allora, che il ruolo dei goal è quello di supportare un meccanismo di raffinamento e astrazione dei goal, raggiungendo goal di più alto livello attraverso la cooperazione di goal in AND e OR di più basso livello.

I goal hanno un ruolo fondamentale nell'ingegneria dei requisiti, perchè ci permettono di ragionare in termini di soddisfacibilità (dei goal e dei requisiti utente) e possiamo rappresentare formalmente il concetto di goal:

$$\{\texttt{REQ}, \texttt{Exp}, \texttt{Dom}\} \models \texttt{G}$$

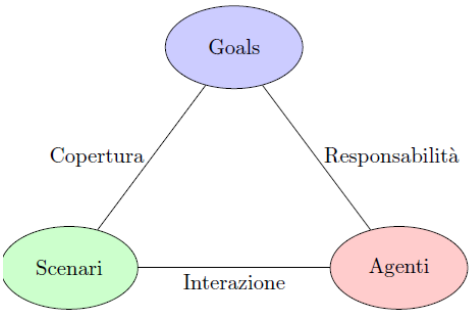
Che possiamo leggere come "*Considerate le proprietà di dominio, i requisiti/sottogoal che assicurano che il goal G sia soddisfatto sotto l'ipotesi delle proprietà di dominio*".

Ragionare in termini di goal, ci permette di ragionare secondo due dimensioni:

- **completezza** → assicurare che tutti i goal siano raggiunti;
- **rilevanza** → se un qualche requisito è utilizzato per raggiungere un goal, allora tale requisito è rilevante (quindi non abbiamo requisiti inutili).

Ragionando in termini di evoluzione è importante decidere, quali sono i requisiti più stabili e quelli meno stabili, in modo da osservarli con maggiore attenzione. Tipicamente i goal di alto livello sono più stabili, mentre i goal di basso livelli sono più instabili, perchè potremmo avere alternative e perchè sono più concreti e conseguentemente legati all'implementazione del sistema, anche data la presenza di OR decomposizione.

La metodologia goal oriented non è puramente top-down, nel senso che tipicamente si utilizzano entrambe le strategie, top-down e bottom-up. La metodologia bottom-up viene chiamata **goal-abstraction**, dove si cerca di estrarre le motivazioni per cui un determinato goal è stato definito. Tipicamente, però, si ragiona in termini di **scenario-oriented** e **agent-oriented** ed in questo senso, possiamo riassumere i concetti attraverso il seguente schema:



gli scenari devono coprire tutti i goal e abbiamo delle interazioni tra agenti e scenari e i goal sono responsabilità degli agenti.

Goal diagram

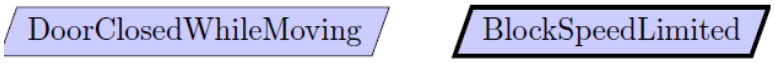
Riprendiamo il concetto di ‘goal orientation’ visto nella lezione precedente: Goal orientation significa andare ad identificare gli step prescrittivi, i quali dicono qual è l’intento del sistema che deve essere soddisfatto. Inoltre, gli step prescrittivi vengono formulati in termini di fenomeni del mondo e sono a molteplici livelli di granularità. Come abbiamo detto precedentemente, i goal possono essere:

- negoziati;
- indeboliti;
- prioritizzati.

> 💡 Quindi, i goal sono soggetti ad una valutazione, da parte degli ingegneri dei requisiti, indifferentemente dalle proprietà di dominio del sistema (le quali non possono essere modificate).
>
> Inoltre, abbiamo anche detto, che **il livello di granularità più fine dei goal, quest’ultimi possono essere assegnati ad un singolo agente** → da questo possiamo capire, che quando abbiamo un goal che è responsabilità di un solo agente, allora possiamo fermare il raffinamento.

Rappresentazione dei goal

All’interno del goal diagram i goal vengono rappresentati con un **parallelogramma (con i lati rivolti verso destra)**, il quale può o meno avere il bordo in grassetto.



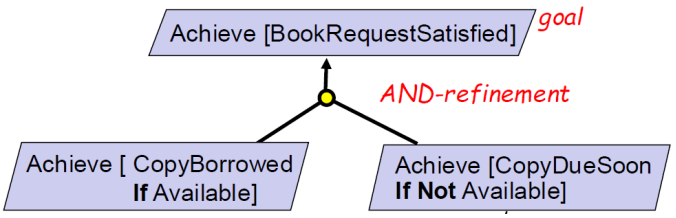
Ad ogni goal, inoltre, possiamo avere delle annotazioni per specificare ulteriori attributi del goal (come ad esempio: la definizione, la specifica formale, la categoria e la priorità) e possono avere delle precise definizioni a seconda del progetto software. Notiamo, inoltre, che all’inizio del progetto è opportuno

definire la sintassi delle notazioni, in modo tale da usarle in maniera consistente durante tutto il progetto.

Raffinamento dei goal

Un goal di alto livello può essere raffinato in molteplici sotto-goal più concreti, che ci spiegano come il goal deve essere composto → quindi, il raffinamento ci permette di suddividere un goal in sotto-goal per poterlo realizzare ed in particolare, il raffinamento può essere di due tipi:

1. **AND-refinement** → nell'AND-refinement il goal viene suddiviso in sotto goal, che devono essere soddisfatti **tutti** per poter soddisfare il goal principale. Vediamo un esempio:



Vediamo, dall'esempio, che abbiamo un goal di alto livello (ovvero Achieve[BookRequest]) e di conseguenza, possiamo raffinare il goal, in modo tale da capire come raggiungere effettivamente tale goal. Come possiamo vedere dall'immagine, se la copia è disponibile allora la possiamo dare in prestito, mentre se la copia non è disponibile deve essere riportata al più presto. Da notare, che **per soddisfare il goal di alto livello, i due sotto-goal devono essere veri/soddisfatti entrambi**.

💡 Il raffinamento di tipo AND deve essere **completo**, ovverosia il raggiungimento degli obiettivi in AND è una condizione sufficiente per il raggiungimento del goal di più alto livello.

In altri casi, invece, i goal non sono sufficienti a garantire che il goal di alto livello sia raggiunto e di conseguenza, dobbiamo aggiungere delle informazioni di dominio, le quali possono essere:

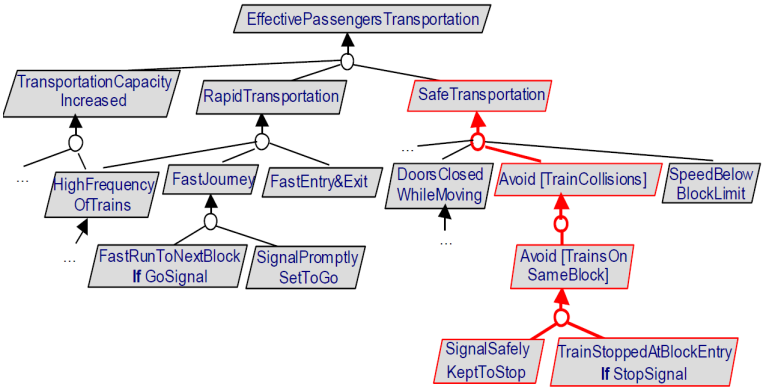
- invarianti di dominio (esse sono sempre vere) → un esempio di invariante di dominio è: "Le porte dei treni sono aperte o chiuse";
- oppure ipotesi di dominio (esse sono assunte per vere) → un esempio di ipotesi di dominio è: "i binari ferroviari sono in buone condizioni".

Queste informazioni, quindi, possono essere allegate al goal model, al fine di raggiungere il goal di alto livello.

Diciamo, inoltre, che i raffinamenti di tipo AND devono essere **consistenti** e **minimali**, ovverosia:

- **consistenti** → nel caso in cui avessimo *N* sotto-goal (G1, …., Gn) e le proprietà di dominio (Dom), dobbiamo fare in modo che i sotto-goal e le proprietà di dominio non si contraddicano a vicenda. Di conseguenza, i sotto-goal e le proprietà di dominio non devono essere contraddittorie tra di loro, bensì devono essere consistenti;
- **minimali** → il sotto-insieme di goal (che raffinano il goal di alto livello) deve essere minima, ovverosia se viene tolto anche solo un sotto-goal il goal principale non può essere raggiunto.

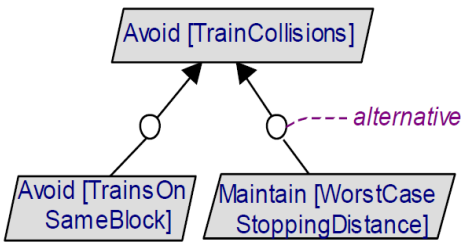
Da notare, che ripercorrendo il grafo dal basso verso l'alto, cerchiamo di capire come il goal viene soddisfatto e quali sono le modalità che possiamo adottare per raggiungere il goal. Questo concetto lo si può vedere meglio, attraverso il seguente esempio:



Quindi, investigando il grafo attraverso un approccio top-down osserviamo la dimensione del '*come*', ovverosia ci chiediamo come possiamo raggiungere un goal al quale stiamo ragionando. Invece, investigando il grafo attraverso un approccio bottom-up, ci fornisce la descrizione della motivazione per cui un goal è stato proposto e scritto.

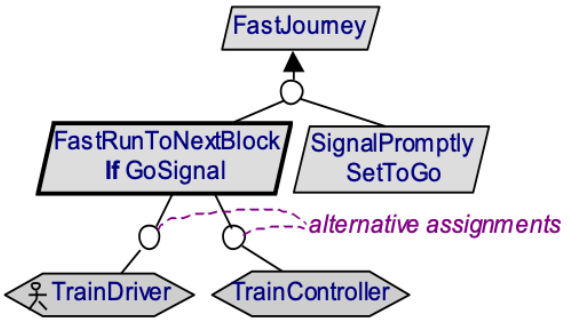
Fino ad ora, abbiamo discusso di quando vi è un goal che contribuisce positivamente ad un altro goal, ma ci potrebbero essere dei legami diversi tra i goal, ovverosia potrebbe esserci dei goal che sono tra loro **contrastanti**, nel senso che potrebbero esserci dei contributi negativi → la presenza di contraddizioni, potrebbe far emergere nuovi requisiti, rappresentando quindi delle temporanee lacune nella modellazione, non essendo quindi un pattern da evitare → è piuttosto importante documentare questi conflitti, per poterli risolvere in seguito, attraverso un **artefatto**.

2. **OR-refinement** → ci permette di catturare delle alternative, quando un goal di alto livello può essere raggiunto in diversi modi alternativi (quindi non tutti i modi sono necessari). Quindi, nella OR-refinement il goal viene suddiviso in sotto-goal che devono essere soddisfatti almeno uno per poter soddisfare il goal principale. Vediamo un esempio:



Dall'immagine possiamo vedere, che i due sotto-goal contribuiscono positivamente al goal di alto livello, ma contribuiscono in maniera distinta e di conseguenza, rappresentano delle alternative → quando avremo l'informazione completa (anche di tutti gli altri aspetti), allora potremmo procedere con la valutazione e la scelta dell'alternativa che decideremo di adottare.

Non abbiamo solamente la decomposizione OR tra goal e sub-goal, bensì potremmo avere anche un **OR-assignment**, il quale rappresenta delle alternative fra l'assegnamento di responsabilità agli agenti → nel caso in cui un goal possa essere raggiunto da più agenti diversi, in cui ognuno di essi può raggiungere il goal in modo indipendente, allora si può utilizzare un raffinamento di tipo OR:



💡 É opportuno in fase di modellazione segnalare queste alternative, che verranno discusse in futuro sulla base di costi, benefici e rischi.

Quindi, il nostro goal diagram contiene:

- decomposizioni AND;
- decomposizioni OR;
- il nodo radice (del diagramma) rappresenta il goal di più alto livello, che rappresenta il goal di sistema → essendo il goal di più alto livello sarò certamente il più astratto ed in particolare, tale goal può essere:
  - funzionale oppure *non-funzionale*;
  - *soft goal* oppure behavioral goal.

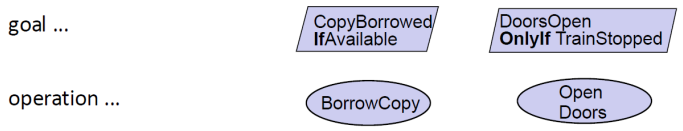
- i nodi foglia, invece, sono la granularità in cui possiamo decomporre i goal del nostro sistema ed in particolare, i nodi foglia saranno requisiti oppure expectations e li posso assegnare ad un solo agente (di sistema o di ambiente).

> 💡 Tipicamente, il goal diagram è un grafo orientato e aciclico, ma non è un albero (in quanto possiamo avere molteplici root goal).
>
> Possiamo aggiungere annotazioni e commenti, al fine di specificare ulteriori dettagli liberamente in modo da enfatizzare o cristallizzare il significato.

Euristiche per l'individualizzazione dei goal

- **Elicitazione dei goal preliminari** → la quale si suddivide in:
  - Analizzare gli obiettivi correnti del sistema as-is → andiamo a studiare il sistema as-is, per vedere quali sono i goal che il sistema deve raggiungere, visto che il sistema as-is quando verrà rimpiazzato dal sistema to-be molti goal rimarranno indifferenti. Quindi, dovremmo innanzitutto capire gli obiettivi e le policies del sistema as-is.
  - Cercare keyword all'interno del materiale elicitato (dove il materiale elicitato può essere: documenti disponibili e trascrizioni delle interviste fatte agli stakeholders) → tali keyword sono collegate alla nozione di goal/obiettivo che deve essere raggiunto → questo ci permette di raccogliere i goal e inoltre, ci permette di individuare la relazione di refinement tra i goal.
  - Cercare all'interno delle tassonomie di goal → sfogliare le tassonomie di obiettivi funzionali e non funzionali, alla ricerca di istanze specifiche del sistema.
- **Identificare i goal lungo i branch di raffinamento** → si suddivide in:
  - Farsi le domande "*Perché*?" (astrazione), quindi ispezioniamo il grafo in maniera bottom-up, e "*Come?*" (raffinamento), quindi ispezioniamo il grafo in maniera top-down.
  - Utilizzare gli scenari.

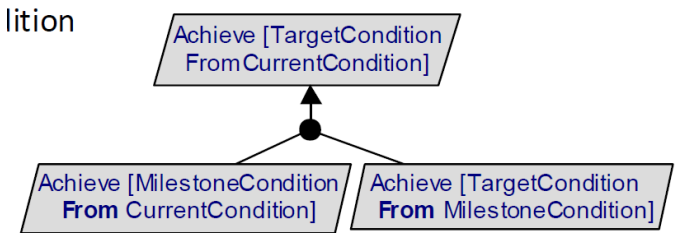
    - Dividere le responsabilità tra gli agenti → in questo modo, otteniamo dei sotto-goal, che coinvolgano un numero minore di agenti e muoversi verso requisiti e aspettative.
    - Identificare i goal da i pro e contro delle differenti alternative → da notare, che capire quali sono i vantaggi e gli svantaggi mi permette di individuare dei soft goal.
    - Dagli Achive goals posso identificare i corrispondenti Maintain goals (i quali solitamente non sono espressi, ovverosia sono impliciti).
    - Delimitare lo scope dei goal → dobbiamo capire quando andiamo a bloccare l'attività di raffinamento (quindi dobbiamo bloccarci quando abbiamo raggiunto i nodi foglia del nostro goal diagram) e allo stesso modo, dobbiamo fermarci quando abbiamo raggiunto il nodo radice nel diagramma → la regola generale, è che dobbiamo fermarci nel raffinamento, quando i goal foglia possono essere assegnati ad un singolo agente. Analogamente, raggiungiamo il nodo radice quando raggiungiamo il limite dello scope del nostro problema, ovverosia quando cominciamo a chiederci motivazioni che vanno fuori dal nostro dominio di sistema.
- **Non confondere i goal con le operazioni** → i goal non sono operazioni che vengono condotte dal sistema, bensì i goal sono obiettivi che vogliamo raggiugere. Le operazioni, invece, sono le funzionalità che il sistema mette a disposizione per raggiungere il goal. Vediamo un esempio:



  Capiamo, inoltre, che un goal può essere raggiunto attraverso molteplici operazioni e un'operazione può essere funzionale per raggiungere molteplici goal → inoltre, un goal restringe il funzionamento del sistema attraverso diversi stati, mentre le operazioni ci fanno transire da uno stato all'altro.
- Non confondere AND-refinement con OR-refinement.
- Evitare ambiguità

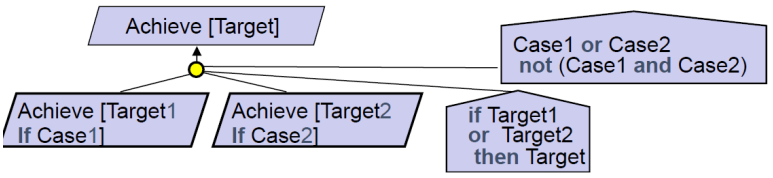
Infine, vediamo alcuni patter che sono interessanti da utilizzare per chiederci se vi siano o meno dei goal impliciti e in caso esplicitarli → i pattern, quindi, catturano delle tattiche di raffinamento e ci guidano nell'identificazione e modellazione dei goal. Vediamo, allora, i possibili pattern:

- **milestone pattern** → applicabile quando è possibile identificare gli stati intermedi tra la target condition e la current condition. La forma di tale pattern è la seguente:

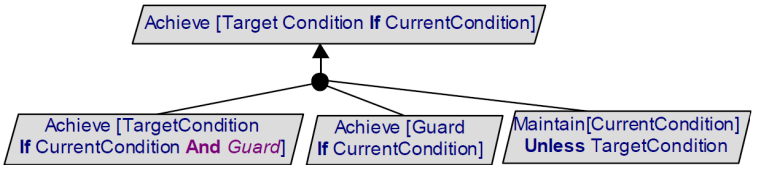


> 💡 Questo pattern ci permette di decomporre due fasi distinte, entrambi necessarie per raggiungere il goal di alto livello → capiamo, allora, che possiamo avere una o più milestone che ci permettono di raffinare il goal e ci permettono di spezzare il raggiungimento di un goal abbastanza astratto in varie fasi.

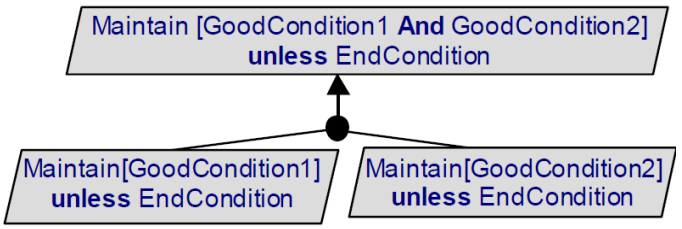
- **refinement by case** → applicabile quando lo spazio di soddisfazione del goal può essere partizionato in casi (disgiunti, che coprono tutte le possibilità). La forma del patter è la seguente:



- **guard introduction** → applicabile agli Achieve goal, dove è necessario stabilire una condizione di guardia per raggiungere l'obiettivo. La forma del patter è la seguente:



- **dividi-e-conquista** → applicabile ai Mantain goal in cui GoodCondition è una congiunzione. La struttura di questo patter è la seguente:

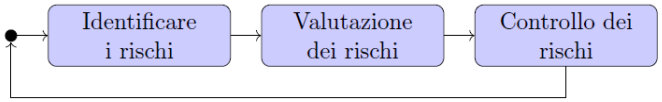


Risk analysis

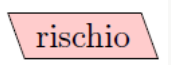
Il **rischio è un fattore di incertezza, che potrebbe risultare in una perdita di soddisfazione, da parte dei nostri stakeholders, per quanto riguarda gli obiettivi che ci siamo posti durante il progetto** → capiamo, allora, che il concetto di rischio è un concetto generico, che però è formulato in termini degli obiettivi (o meglio, in termini di raggiungimento degli obiettivi) → abbiamo già detto, che ogni rischio ha:

- una **probabilità**;
- delle **conseguenze** → a loro volta, le conseguenze possono avere:
  - una probabilità di manifestarsi quando si verifica/manifesta il rischio;
  - una severità.

Inoltre, già nella fase di modellazione dei requisiti abbiamo affrontato la risk analysis, identificando le fasi relative ai rischi:



💡 La mancata individuazione dei rischi è una delle principali cause di fallimento dei progetti software e i rischi possono essere rappresentati attraverso un parallelogramma, inclinato nella direzione opposta rispetto ai goal.



Goal ostruiti dai rischi

Tipicamente i goal sono identificati in maniera leggermente astratta e ottimistica rispetto a quello che il sistema dovrebbe fare, nel senso che **quando modelliamo i goal rischiamo di modellare solamente il funzionamento corretto del sistema, senza considerare i rischi che possono compromettere il raggiungimento di questi goal** (per esempio: alcuni goal potrebbero essere

violati facilmente da condizioni di contesto e/o da condizioni a cui non avevamo pensato). A questo punti, quindi, cerchiamo di capire:

- tutti i **problemi** che il nostro sistema potrebbe avere;
- tutti i **modi** in cui il nostro sistema potrebbe fallire (quindi non raggiungere alcuni degli obiettivi voluti dagli stakeholders).

> 💡 In generale, quindi, cerchiamo di capire tutti i problemi che potrebbero manifestarsi.

Definiamo, allora, il concetto di **ostacolo** → l'ostacolo è una condizione del sistema, che potrebbe far violare un'asserzione (tipicamente potrebbe far violare un goal). In particolare, l'ostacolo potrebbe:

- essere un ostacolo diretto, che prende il nome di '**obstruction**' → l'ostacolo messo insieme alle proprietà di dominio rappresenta una negazione diretta del goal;
- minare la **consistenza del dominio** → l'ostacolo insieme ad una proprietà di dominio risulta essere falso, nel senso che l'ostacolo nega qualcosa del dominio direttamente;
- essere **fattibile** → nel modo in cui sono definiti i goal e il sistema, l'ostacolo potrebbe effettivamente manifestarsi (quindi non c'è nulla che ne viene il manifestarsi).

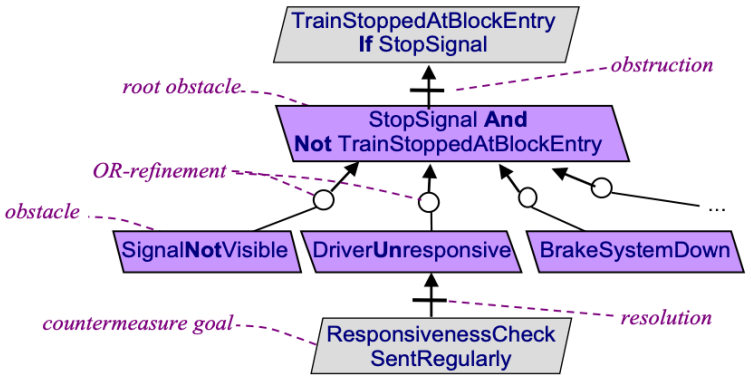
> 💡 Idealmente, l'insieme degli ostacoli dovrebbe essere **completo**, ovverosia se nessuno degli ostacoli si manifesta (o se tutti gli ostacoli sono negati) allora raggiungiamo il goal → quindi, l'insieme ideale di ostacoli di G dovrebbe essere la negazione di tutti i goal che permettono di raggiungere G → di fatto, allora, tutti gli ostacoli devono essere catturati.
> Questa attività è naturalmente molto complessa e costosa e di conseguenza, viene richiesta solamente quando abbiamo sistemi mission-critical.

Visto la difficolta nell'individuare tutti gli ostacoli, anche in questo caso possiamo dotarci di una **tassonomia**, la quale ricalca la tassonomia dei goal → tipicamente, possiamo ragionare in termini di ostacoli come negazione dei goal:

- **Hazard** obstacles obstruct Safety goals
- **Threat** obstacles obstruct Security goals
  - Disclosure, Corruption, DenialOfService, …
- **Inaccuracy** obstacles obstruct Accuracy goals
- **Misinformation** obstacles obstruct Information goals
  - NonInformation, WrongInformation, TooLateInformation, …
- **Dissatisfaction** obstacles obstruct Satisfaction goals
  - NonSatisfaction, PartialSatisfaction, TooLateSatisfaction, …
- **Unusability** obstacles obstruct Usability goals

Modellazione degli ostacoli

Vediamo attraverso la seguente immagine, la modellazione degli ostacoli:

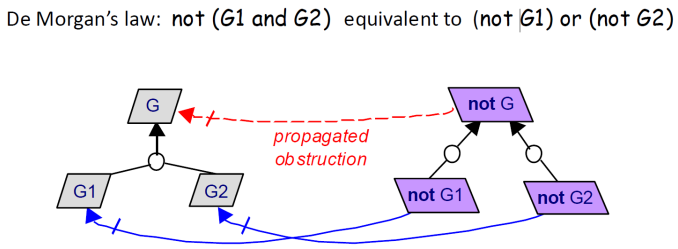


Possiamo vedere, che l'ostacolo è l'opposto/la negazione del goal.

La modellazione degli ostacoli è simile a quella dei goal e notiamo, che per collegare un ostacolo con il corrispondente goal viene utilizzato un **obstruction link (link di ostruzione), il quale vuole rappresentare l'opposto del refinement**. Notiamo, inoltre, che come i goal anche gli ostacoli possono avere un refinement ed in particolare, possiamo usare la AND-decomposition e la OR-decomposition per raffinare gli ostacoli. Infine, possiamo osservare che per mitigare un ostacolo, la soluzione standard è di definire un nuovo goal e di utilizzare un **link di risoluzione,** che collega il nuovo goal all'ostacolo (quindi il link parte dal goal e arriva all'ostacolo).

Gli ostacoli, allora, si propagano in modo bottom-up e top-down in maniera analoga ai goal, però gli ostacoli si articolano utilizzando la **legge di De Morgan**,

perchè vengono utilizzate le negazioni. In particolare, la legge di De Morgan prevede che:



Vediamo, allora, che si passa da un AND-decomposition ad un OR-decomposition.

Analisi degli ostacoli per migliorare la robustezza del sistema

Il nostro obiettivo è di identificare gli ostacoli, in modo tale da avere una modellazione dei goal più realistica, dato che attraverso gli ostacoli riusciamo a modellare i goal in maniera più realistica inserendo anche possibili problemi che potrebbero manifestarsi e quindi, non modelliamo solamente il caso migliore. Inoltre, ci permettono di arricchire i goal, perchè per ogni ostacolo dobbiamo definire dei goal, i quali cercano di mitigare e gestire l'ostacolo → quindi, il nostro obiettivo è di identificare:

- il maggior numero possibile di ostacoli;
- la probabilità di verifica di tali ostacoli;
- per ogni ostacolo, la severità delle conseguenze;
- ed infine dovremmo risolvere gli ostacoli → chiaramente potrebbe non essere fattibile dare una soluzione a tutti gli ostacoli (perchè magari sono troppi oppure perchè renderebbero il sistema eccessivamente complesso), però **sarebbe importante prioritizzarli e iniziare a gestire gli ostacoli con una probabilità ed una severità maggiore, che potrebbero far fallire dei goal importanti** → questo è conseguenza ad un'analisi degli ostacoli.

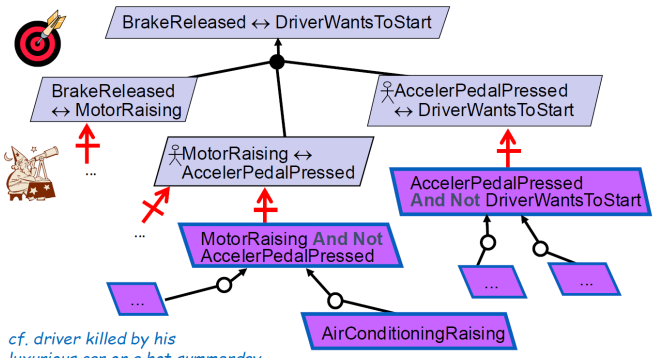
In realtà, l'analisi degli ostacoli (guidata dai goal) è un processo iterativo, perchè:

- una volta definiti gli ostacoli, dovremmo capire quali sono i nuovi goal che ci permettono di mitigare gli ostacoli;
- a questo punto, dovremmo iterare questi nuovi goal, in modo tale da capire se a loro volta hanno degli ostacoli → quindi, dobbiamo iterare fino a quando non raggiungiamo dei goal che non hanno ostacoli.

Sorge spontanea la domanda: "**Quali goal dobbiamo considerare per iniziare con la obstacle analysis?**" La linea guida è di iniziare con i goal più concreti e più dettagliati, in quanto sono quelli su cui è più facile ragionare sui modi in cui potrebbero fallire. La domanda successiva è: "**Quando terminiamo con il processo iterativo?**" Terminiamo quado abbiamo esaurito i goal più critici e quando le conseguenze potrebbero essere accettabili per il nostro sistema (perchè non vanno a violare un behavioral goal).

Analizziamo, a questo punto, le singoli fasi del processo di individuazione degli ostacoli:

1. **Identificare gli ostacoli** → per prima cosa si parte dai goal di alto livello, il primo step è identificare i root obstacle, ovvero gli ostacoli che impediscono il raggiungimento di un goal. Si procede poi con la AND-decomposition e la OR-decomposition per identificare tutte le possibili cause che possono impedire il raggiungimento di un goal, fin tanto che non si raggiungono ostacoli atomici. Per negare un goal top level utilizziamo la negazione con la legge di De Morgan. Vediamo un esempio:



2. Una volta che abbiamo identificato gli ostacoli, dobbiamo **valutare quest'ultimi** → per affrontare questa fase, solitamente, si coinvolgono degli esperti di dominio in cui si lavora, in modo da capire la probabilità di occorrenza e l'impatto di questi ostacoli. Per stimare la probabilità di un ostacolo, si calcola la probabilità di occorrenza di ogni sotto-ostacolo e si calcola la probabilità di occorrenza dell'ostacolo come il massimo tra i sotto-ostacoli nel caso di OR-decomposition (perchè abbiamo bisogno che si verifichi un solo sotto-ostacolo), mentre nel caso di AND-decomposition si calcola la probabilità di occorrenza minima tra i sotto-ostacoli (perchè abbiamo bisogno che si verifichino tutti i sotto-ostacoli) → tutto questo, quindi, ci permette di assegnare una probabilità agli ostacoli di alto livello.

3. **Risolvere gli ostacoli** → per risolvere un ostacolo si adottano delle contromisure identificando dei goal che permettono di limitare e/o superare l'ostacolo. Questi goal vengono collegati all'ostacolo tramite un link di risoluzione e per ogni ostacolo si può avere più di una soluzione, in questo caso quindi occorrerà considerare le alternative e selezionare la migliore basandosi su costi, benefici e rischi.

Per **esplorare le alternative** possiamo utilizzare le seguenti modalità:

- **Sostituzione del goal** → nel caso in cui nel goal top-level ci siano diverse alternative per raggiungere il goal, si possono esplorare queste alternative per capire quale sia la migliore anche in termini di rischi. Se una delle alternative contiene un rischio rispetto ad un'altra, allora si può considerare di scartare questa alternativa;
- **Sostituzione di agente** → in certe situazioni potrei trovarmi di fronte alla necessità di rivedere l'agente utilizzato per raggiungere il goal, in quanto potrebbe rimuovere la presenza di un ostacolo;
- **Indebolire il goal** → se il goal, così come è stato definito, è troppo difficile da raggiungere, si può considerare di indebolire il goal, in modo da renderlo più facile da gestire e da raggiungere.

  Maintain [TrafficControllerOnDutyOnSector]

  Tale goal viene ostruito da NoSectorControllerOnDuty, quindi si può considerare di indebolire il goal in modo da renderlo più facile da raggiungere.

  TrafficControllerOnDutyOnSector or WarningToNextSector
- **Prevenzione di ostacoli** → definiamo un nuovo goal, che parte dalla negazione dell'ostacolo (Avoid [obstacle]) in modo tale da cercare rimedio ad un ostacolo.

  AccelerationCommandCorrupted → Avoid [AccelerationCommandCorrupted]
- **Ripristino di goal** → rafforzando la condizione target come occorrenza di un ostacolo.

  if O then sooner-or-later TargetCondition
- **Riduzione di ostacoli** → definiamo tecnicamente un nuovo goal che cerca di rimediare un singolo ostacolo di basso livello.

- **Mitigazione di ostacoli** → introduciamo nuovi goal per mitigare le conseguenze di un ostacolo. Distinguiamo quindi due diverse strategie di mitigazione di ostacoli:
  - **mitigazione debole** → accettare l'ostacolo e ridefinire il goal in modo tale che una versione più debole del goal venga realizzata;
  - **mitigazione forte** → si raggiunge il goal di alto livello anche se ostruito.

Object model

L'object model rappresenta sostanzialmente una **vista strutturale del nostro sistema** (dove per sistema intendiamo sia il system-to-be sia il system as-is).

> 💡 Ribadiamo il concetto, che non stiamo modellando il software, bensì stiamo modellando il nostro sistema → Capiamo, allora, che non parliamo di oggetti nel senso di object oriented (ovvero non sono oggetti che rappresentano il codice), bensì sono elementi del nostro dominio che modelliamo, al fine di **rappresentare e modellare i concetti, le strutture e i collegamenti tra i vari oggetti, che rappresentano i requisiti del nostro sistema**.

Capiamo, allora, che le finalità di questo modello sono:

- documentare e modellare il system to-be oppure il systema as-is;
- descrivere le proprietà degli oggetti;
- fissare un vocabolario (ovvero servono per creare un glossario).

Dobbiamo fare una piccola precisazione sui termini, in quanto la rappresentazione avviene tramite i **Diagrammi delle Classi UML**, declinandolo verso l'uso di una terminologia differente → questo perchè nella terminologia goal oriented, quando parliamo di oggetti e classi, non ci riferiamo alla metodologia orientata agli oggetti, bensì modelliamo concetti del mondo reale → capiamo, allora, che ad esempio, l'oggetto '*treno*' oppure l'oggetto '*controllore*' non si riferiscono al software, ma al mondo dei requisiti che stiamo modellando e dato che, non modelliamo il software, le classi non hanno operazioni, ma solo attributi per modellare le proprietà degli oggetti del mondo che stiamo modellando. Riassumendo, quindi:

- gli **oggetti sono astrazioni**, ovverosia sono quelle cose che in object oriented chiamiamo classi. Quindi, per noi il '*treno*' è un oggetto;
- quando parliamo di **istanze**, facciamo allora riferimento alle istanze di un oggetto. Quindi, per esempio un'istanza è '*treno123*'.

A questo punto, diciamo innanzitutto cos'è un **oggetto concettuale** → un oggetto concettuale è un insieme di istanze, che godono delle seguenti proprietà:

- sono **identificabili in maniera distinta** (ovvero posso enumerare le istanze) e inoltre, ogni istanza è immutabile ed ha una propria identità;
- possono **essere enumerate in ogni stato del sistema**;
- **condividono caratteristiche comuni** (esempio: tutte le istanze sono caratterizzate da: un nome, una definizione, un tipo di dominio);
- possono **differire per lo stato in cui si trovano** → per *stato* intendiamo una **coppia chiave-valore** (quindi, ad ogni attributo X associamo un valore Y). Quindi, lo stato di un oggetto si rappresenta attraverso la lista di tuple Xi → Vi, dove:
  - Xi è l'attributo dell'oggetto;
  - Vi è il valore dell'istanza.

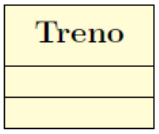
> 💡 La relazione tra gli oggetti e le istanze di oggetti è che un object instance è un'istanza di un oggetto, rappresentato come **instanceOf(o, Ob)**. A differenza dell'object oriented, nel corso dell'evoluzione del sistema, un oggetto può diventare un'istanza di un altro oggetto, quindi un'istanza può essere membro di più oggetti ed ottenere più tipi (in questo caso, si parla di **tipizzazione multipla**).
> Una cosa obbligatoria è di utilizzare le **notazioni**, ovverosia per ogni oggetto concettuale siamo obbligati a definire (all'interno della loro notazione) le condizioni necessarie e sufficienti affinché sia istanza di un particolare oggetto.

Tipi degli oggetti concettuali

I tipi degli oggetti concettuali sono quattro:

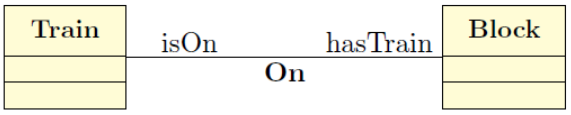
1. **Entità** → un'entità è un oggetto concettuale **autonomo** e **passivo**, ovverosia potrebbero esistere all'interno del sistema indipendentemente dalla presenza o meno di altri oggetti concettuali e non possono controllare il

funzionamento e lo stato degli altri oggetti (del sistema). Le entità sono rappresentate attraverso le classi UML:



Inoltre, le entità sono distintamente identificabili e possono essere enumerate. Infine, possono differire per quanto riguarda gli stati e le transizioni. Necessitano della annotazione **Def**, la quale specifica la definizione dell'oggetto;

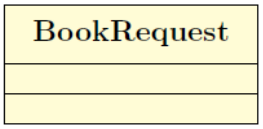
2. **Associazione** → un'associazione è un oggetto concettuale che collega altri oggetti, ed ogni oggetto ha un ruolo specifico → da notare, che le associazioni dipendono dagli oggetti concettuali che linkano. Tali oggetti si rappresentano attraverso le associazioni UML. Analogamente ad UML, l'associazione ha un nome e dei ruoli, quindi come in UML andiamo a specificare il ruolo di ogni entità, che gioca all'interno dell'associazione:



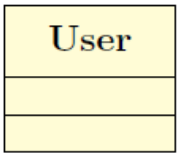
💡 Le associazioni possono **evolvere**, ovverosia un'associazione può nasce all'interno del nostro modello concettuale oppure potrebbe essere cancellata.
Le associazioni, inoltre, possono essere **binarie** (quando collegano due oggetti concettuali), ma possono essere anche **n-arie** (quindi collegare più oggetti concettuali) e chiaramente, le associazioni possono essere anche **riflessive**, ovvero uno stesso oggetto concettuale può partecipare a più associazioni con ruoli differenti (esempio: un treno può seguire un altro treno e può anche essere seguito da un altro treno).
Inoltre, nelle associazioni rappresentiamo anche la **molteplicità** (ovvero la cardinalità) → la molteplicità è un vincolo, che specifica il numero minimo e massimo di oggetti che possono essere coinvolti in un'associazione. La molteplicità può essere:

- 0..1 → 0 o 1 oggetti;
- 0..* → 0 o più oggetti;
- 1..* → 1 o più oggetti;
- 1..1 → 1 oggetto.

3. **Evento** → un evento è un oggetto concettuale che ha un senso istantaneo (ovvero non ha una durata). Gli oggetti che sono istanze di eventi soddisfano la proprietà *Occurs(Ev) e g*li eventi sono rappresentati tramite le classi UML:



4. **Agente** → un agente è un oggetto concettuale **attivo** e **autonomo**, ovverosia hanno un funzionamento autonomo e possono controllare o modificare lo stato di altri oggetti, causando transizioni di stato. Anche gli agenti sono rappresentati tramite le classi UML:

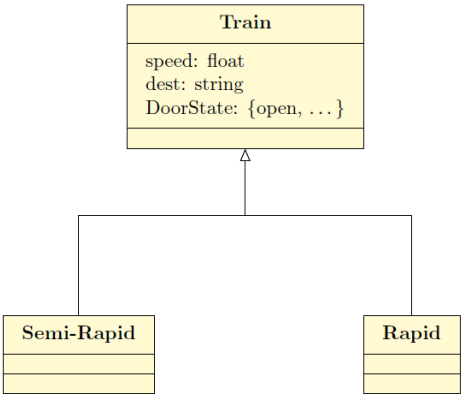


💡 Gli **attributi** sono proprietà degli oggetti che modelliamo, e sono rappresentati mediante gli attributi delle classi UML. Anche le associazioni, in quanto oggetti, possono avere attributi.

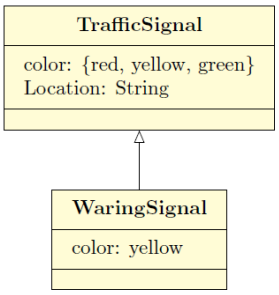
Le molteplicità (ovvero la cardinalità) sono collegate a proprietà di dominio (come per esempio: "*Un treno può essere presente al massimo su un binario alla volta*"), però potrebbero esserci alcune molteplicità che hanno una semantica prescrittiva (per esempio: *"Un blocco non può ospitare più di un treno in qualsiasi momento*") → queste informazioni le trovo all'interno del goal model, però la stessa cosa è vera anche viceversa, ovverosia quando ci chiediamo quale molteplicità dobbiamo specificare nella modellazione concettuale, di fatto implicitamente ci stiamo chiedendo se vi è un goal che vincola quel valore → questo per dire, che **la modellazione concettuale degli oggetti ci permette di ragionare in termini strutturali e di conseguenza è guidata dai goal, ma al tempo stesso fornisce anche dei feedback ai goal**.

A questo punto, parliamo di associazioni particolari, che sono la **specializzazione** e l'**ereditarietà**:

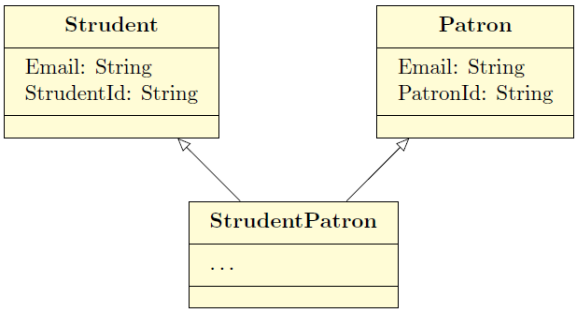
- **Specializzazione** → abbiamo la specializzazione quando utilizziamo il concetto di sotto-classe, ovvero abbiamo una classe concreta che eredità le proprietà della classe più astratta. Quindi, la specializzazione è un'associazione tra una classe generale e una classe specializzata, dove la classe specializzata eredita gli attributi della classe generale. La specializzazione è rappresentata tramite l'ereditarietà delle classi in UML:



Quando si parla di specializzazione, possiamo avere il concetto di **inibizione dell'ereditarietà** → in questo modo specializzare un oggetto, definendo un attributo in maniera più specifica rispetto all'attributo generale (si tratta di una ridefinizione):



Inoltre, possiamo avere l'**ereditarietà multipla** → ovvero un oggetto può essere specializzato in più oggetti, ereditando gli attributi di più classi:



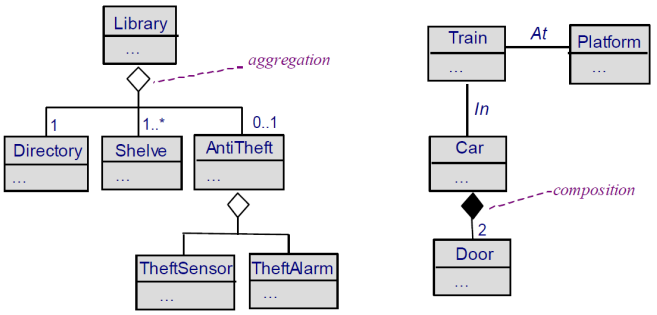
💡 Possiamo avere dei conflitti, in questo caso, avendo il campo Email in entrambe le classi, dobbiamo rinominare il campo in uno dei due oggetti.

I **vantaggi** della specializzazione sono essenzialmente due:

- avere un modello più semplice evitando la duplicazione di attributi;
- l'eliminazione di attributi è possibile su uniche entità evitando disallineamenti.

Ultimo tipo di associazione che possiamo avere è l'**aggregazione** → abbiamo un'aggregazione quando abbiamo due entità (una entità è il composto, mentre l'altra è il componente) e si applica a: entità, agenti, eventi e associazioni. Un esempio comune di aggregazione potrebbe essere una libreria e i libri che contiene. La libreria è l'aggregato e i libri sono le parti. I libri possono esistere al di fuori della libreria, suggerendo che la libreria è solo un contenitore per i libri, quindi non una necessità per l'esistenza dei libri.

Quindi, l'aggregazione è una composizione di diversi oggetti, dove ogni oggetto rappresenta una parte. In questa relazione, l'oggetto aggregato può essere considerato un contenitore dei suoi componenti, ma i componenti possono esistere indipendentemente dall'aggregato. Vediamo un esempio:



Dall'immagine sopra, possiamo capire la differenza tra aggregazione e composizione: **Mentre l'aggregazione permette che le parti esistano indipendentemente dall'aggregato, la composizione è una forma più forte di aggregazione dove le parti non hanno un'esistenza indipendente dall'aggregato** (nel nostro esempio: se un vagone viene eliminato dal sistema, allora vengono eliminate anche le porte, perchè non hanno più senso di esistere) → se un aggregato viene distrutto, tutte le sue parti vengono distrutte con esso. Questo legame è utile per rappresentare relazioni vita o morte tra componenti (come tra un corpo e gli organi interni).

Euristiche per la modellazione degli oggetti

Nel processo di modellazione degli oggetti, è fondamentale seguire determinate euristiche per assicurare una progettazione chiara e funzionale. Di seguito sono elencate le principali linee guida da considerare:

- **Revisione delle specifiche** → è cruciale rivedere tutte le specifiche degli obiettivi e delle proprietà di dominio contenute nel modello degli obiettivi, al fine di garantire che tutte le necessità siano adeguatamente rappresentate e comprese. Quindi, dobbiamo prendere tutti i concetti che soddisfano i criteri per essere oggetti concettuali e dobbiamo capire quali devono essere rappresentati. In particolare, gli oggetti concettuali che devono essere rappresentati, sono oggetti le cui istanze sono:
  - distintamente identificabili;
  - enumerabili in ogni stato;
  - condividono caratteristiche simili;
  - tra loro differiscono per lo stato in cui si trovano.

  Nei goal, oltre alle entità, riusciamo anche a individuare delle associazioni;
- **Assegnazione degli obiettivi** → per quanto riguarda le grandezze non direttamente controllabili e le grandezze non direttamente monitorabili, l'implementazione degli obiettivi nel software da realizzare, è essenziale introdurre rappresentazioni condivise degli oggetti dell'ambiente riferiti dagli obiettivi;
- **Definizione degli attributi** → un elemento è da considerare un attributo se è una funzione che restituisce un valore unico, le sue istanze non necessitano distinzione, non si prevede di aggiungervi attributi o associazioni, e non si intende specializzarne l'intervallo di valori;
- **Oggetti concettuali negli obiettivi** → gli oggetti concettuali definiti da uno stato unico (esempio: StartTrain) implicano eventi, mentre gli oggetti che attivano e controllano comportamenti di altre istanze (esempio: DoorsActuator) agiscono come agenti;
- **Attributi nelle associazioni** → un attributo deve essere collegato a un'associazione se caratterizza tutti gli oggetti partecipanti, sia esplicitamente che implicitamente;
- **Link strutturali e associativi** → considerare la creazione di un'associazione se esiste un link strutturale tra oggetti compositi e componenti che soddisfa specifiche condizioni operative o strutturali;

- **Specializzazioni e generalizzazioni** → è importante identificare specializzazioni e generalizzazioni attraverso l'analisi di attributi comuni, associazioni e invarianti di dominio, contribuendo alla creazione di un modello pi`u organizzato e scalabile;
- **Utilizzo di associazioni invece di puntatori** → evitare l'utilizzo di "puntatori" agli altri oggetti come attributi e preferire l'uso di associazioni binarie;
- **Nomi e definizioni chiare** → utilizzare nomi suggeriti e chiari per oggetti e attributi, evitando ambiguità e assicurando una definizione precisa e dettagliata. In questo senso essendo che non stiamo progettando il sistema software, non utilizziamo termini legati all'implementazione e utilizziamo termini di uso comune (quindi non inventati).

Agent responsibility

Avendo parlato ampiamente dei goal, parliamo degli **agenti**, ovvero chi ha la responsabilità che i goal vengano raggiunti. Non si parla solo di **responsabilità**, ma anche di **capacità**, ovverosia quali sono gli agenti che sono in grado di raggiungere i goal, chi ha interesse a raggiungerli e quelli a cui verrà assegnata la responsabilità del raggiungimento del goal → quindi, attraverso gli agent model:

- documentiamo la situazione di responsabilità che abbiamo deciso (nel senso di quali goal abbiamo assegnato ai vari agenti, ovvero a chi è stata assegnata la responsabilità di perseguire tali goal);
- possiamo fare la load analysis;
- possiamo analizzare lo scope del sistema;
- possiamo condurre la vulnerability analysis.

Riprendiamo, allora, il concetto di **agente** → gli agenti sono **oggetti attivi**, ovverosia controllano il funzionamento del sistema (quindi, sono in grado di impostare dei valori e di cambiare lo stato di alcuni oggetti concettuali). Inoltre, abbiamo visto che gli agenti sono responsabili del raggiungimento dei goal, ovverosia gli agenti sono in grado di vincolare/restringere il funzionamento del sistema, in modo da evitare i flussi di esecuzioni (possibili) che potrebbero andare a violare i goal → quindi, in un qualche modo, gli agenti impongono dei vincoli al sistema e guidano quest'ultimo, in modo tale che i goal vengano mantenuti e/o raggiunti.

L'assegnamento di responsabilità consiste nel prendere i nodi foglia, ovvero i goal più concreti, e assegnarli agli agenti. Notiamo, inoltre, che potrebbero esserci diverse categorie di agenti:

- agenti nel **software-to-be**;
- agenti dell'**ambiente** (quindi, gli agenti potrebbero essere: persone, dispositivi, software di terze parti).

Vediamo, a questo punto, le **caratteristiche** degli agenti:

- hanno una **definizione** → condizioni per cui un individuo è (al momento) un agente;
- hanno degli **attributi**;
- definiamo la **categoria** → quindi, definiamo se si tratta di un agente software oppure un agente dell'ambiente;
- definiamo le **capability** → ovvero definiamo cosa l'agente può fare. Principalmente, gli agenti possono fare 2 cose:
  - monitorare degli oggetti (quando hanno visibilità sull'oggetto e possono leggere lo stato di un oggetto concettuale);
  - controllare degli oggetti (quando possono assegnare i valori dello stato dell'oggetto concettuale oppure quando possono creare o eliminare un oggetto concettuale).
- definiamo le **responsabilità** → ovvero definiamo i goal a cui gli agenti sono responsabili;
- definiamo le **performance** → ovvero definiamo le operazioni che l'agente può intraprendere.

Iniziamo a parlare delle **capability** dell'agente → gli agenti possono **monitorare** o **controllare** delle grandezze del sistema, ovvero possono avere la capacità di osservare o modificare lo stato del sistema. Ad esempio, quando parliamo di monitoraggio, pensiamo ad un sensore di velocità, il quale avrà la capacità di monitorare la velocità del treno → quindi, è importante capire quali sono le grandezze che un agente può vedere o sapere, perchè ovviamente le responsabilità dovranno essere allineate. Il controllo, invece, avviene su quegli oggetti su cui l'agente può esercitare qualche tipo di controllo. Pensiamo ad esempio, al controllore del treno che imposta lo stato delle porte da aperto a chiuso. Oppure, il controllo si ha anche quando l'agente può controllare le relazioni dell'oggetto (magari l'agente può cancellare la relazione presente tra due entità). Le capability dell'agente vengono rappresentate nel seguente modo:

Come possiamo vedere dall'immagine, abbiamo che:

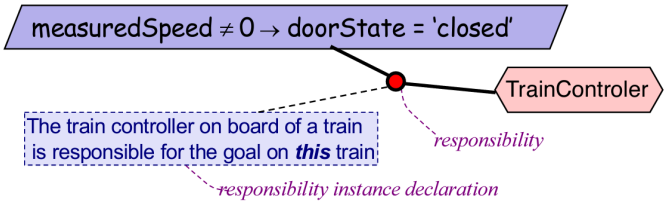
- in input all'agente abbiamo grandezze che l'agente monitora;

- in output all'agente abbiamo grandezze che l'agente controlla.

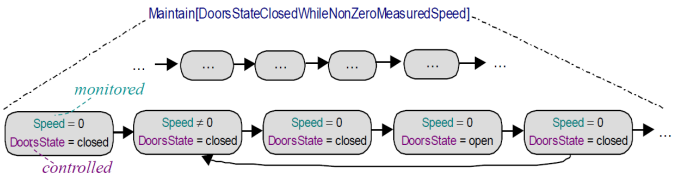
Di fatto, quello che gli agenti possono monitorare non sono solo dati, ma ogni tipologia di oggetto, ad esempio un agente potrebbe monitorare un campo di un oggetto, la presenza di associazioni tra oggetti, ecc.

> 💡 Diciamo, inoltre, che **una variabile può essere controllata al massimo da un agente, al fine di evitare potenziali conflitti.**

A questo punto, parliamo delle **responsabilità** → gli agenti sono responsabili per il mantenimento e/o per il raggiungimento di un determinato goal. La responsabilità di un agente viene rappresentata con un **responsibility link**, ovvero un collegamento tra l'agente e il goal che l'agente deve raggiungere e ciò viene rappresentato nel seguente modo:



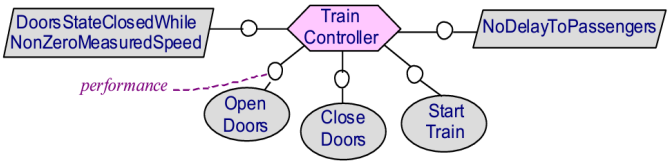
Quando assegniamo le responsabilità, dobbiamo assicurarci che l'agente sia in grado di realizzare il goal, ovverosia dobbiamo assicurarci che l'agente abbia tutte le capabilities, che gli permettono di raggiungere il goal e questo naturalmente, ci guida nell'assegnamento delle responsabilità → quindi, l'assegnamento delle responsabilità deve essere subordinato al fatto, che l'agente possieda le capacità di monitorare e controllare le grandezze necessarie per il raggiungimento del goal. Per farlo si definisce una **lista di transizioni di stato** che l'agente può monitorare e controllare, in modo che il goal sia soddisfatto. Capiamo, allora, che un goal potrebbe essere non realizzabile per l'agente quando delle variabili importanti non possono essere monitorate o controllate dall'agente stesso. Se vi sono variabili di stato che vengono valutate in stati futuri, quando il goal esprime delle condizioni istantanee, allora il goal non è realizzabile. Un'altra condizione si verifica quando il goal non è soddisfacibile sotto determinate condizioni, ovvero il goal non è realizzabile in alcune situazioni.



Riassumendo, quindi, le motivazioni per cui un goal non è realizzabile da parte di un agente sono:

- **mancanza di monitorabilità** → le variabili importanti per il raggiungimento del goal, non sono visibili all'agente;
- **mancanza di controllabilità** → l'agente non può impostare il valore delle variabili, che sono necessarie da controllare, per raggiungere il goal;
- potremmo avere un problema con gli achieve goal, in quanto potremmo rimandare all'infinito (visto che nella modellazione non abbiamo impostato una scadenza) il raggiungimento dell'achieve goal.

Sorge a questo punto spontanea la domanda: "**Come fanno gli agenti a raggiungere o mantenere i goal?**" Per raggiungere gli obiettivi, gli agenti devono prendere delle decisioni intraprendendo delle **operazioni**, le quali (ovvero le operazioni) vengono rappresentate tramite delle ellissi e sostanzialmente, rappresentano le 'cose' che l'agente può fare per raggiungere il goal. Fare un'operazione, quindi, vuol dire che l'agente deve impostare o leggere delle variabili negli oggetti che controlla o monitora e deve farlo nelle condizioni in cui il sistema opera. Vediamo un esempio:



Come possiamo vedere dall'immagine, il collegamento tra l'agente e l'operazione che l'agente deve compiere viene rappresentato con un **performance link**.

Quando ragioniamo in termini di agenti, oltre a ragionare in termini di capabilities, sarebbe importante anche modellare i desideri (detti **wishes**) dell'agente, che sono sostanzialmente le cose che l'agente vorrebbe fare → un agente umano potrebbe avere dei desideri, ovvero degli obiettivi personali che non sono necessariamente in linea con i goal del sistema. Un esempio di desiderio:

Un Patron vuole che il periodo di prestito sia lungo.

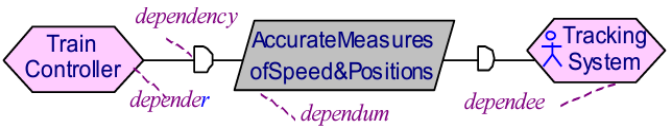
Ci forniscono un modo per guidare l'assegnamento delle responsabilità,, in quanto un agente umano potrebbe avere un desiderio che non è in linea con i goal del sistema. Pensare ai desideri degli agenti può quindi già guidare in fase di assegnamento delle responsabilità.

L'aspetto dei wishes (degli agenti) può essere ulteriormente approfondito nella **conoscenza e credenze/opinioni degli agenti** → è possibile modellare sia la conoscenza sia le credenze degli agenti rispetto una **memoria locale**, che l'agente possiede rispetto all'ambiente reale:

- Un fatto F è creduto dall'agente se F è nella memoria locale dell'agente;
- Un fatto F è conosciuto dall'agente se F è nella memoria locale dell'agente e F è vero, proprietà posseduta dall'ambiente o dal sistema.

> 💡 Tale considerazione è importante, perchè ci permette di fare delle analisi sulla **sicurezza**.

La prima applicazione degli agenti è la **agent dependencies** → come abbiamo già detto in precedenza, vi è una relazione di dipendenza tra i goal (ovvero alcuni goal sono necessari per raggiungere altri goal) e tali goal potrebbero essere assegnati ad agenti distinti. Questo ne consegue, che per raggiungere un particolare goal un agente deve aspettare che prima altri goal vengano raggiunti, che magari sono responsabilità di altri agenti → capiamo, allora, che alcuni agenti (per raggiungere i propri goal) dipendono da altri agenti.



Riassumendo, quindi, una dipendenza tra due agenti si verifica quando un agente è responsabile del raggiungimento di un goal, ma un altro agente, per aggiungere un determinato goal deve aspettare che il primo agente abbia raggiunto il suo goal.

💡 Tali dipendenze possono propagarsi a cascata e ci indicherebbero quali goal non verrebbero raggiunti, se un goal non venisse raggiunto. Chiaramente, in sistemi critici è opportuno non avere catene troppo lunghe e in caso la catena sia troppo lunga, dobbiamo pensare a dei modi (per esempio: raffinare degli obiettivi oppure degli assegnamenti di responsabilità alternativi) per spezzare la catena, in modo tale da ridurre le vulnerabilità.

Il diagramma delle agent dependencies ci permette di intavolare la **vulnerability analysis**, perchè se un goal non viene raggiunto, allora vi saranno automaticamente una serie di altri goal (intermedi) che non vengono raggiunti.

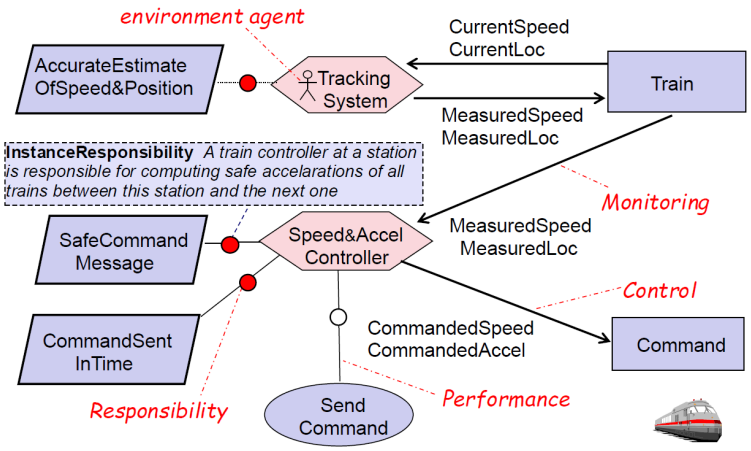
Rappresentazione della modellazione degli agenti

L'agent diagram è un diagramma che ha come prospettiva principale gli agenti e permette di rappresentare:

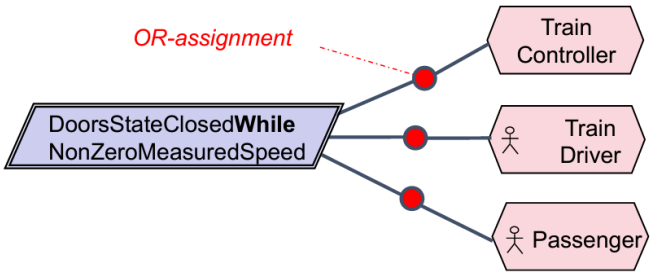
- le **responsabilità**;
- le **capacità**;
- le **operazioni** degli agenti.

Capiamo, allora che all'interno di un agent diagram:

- i nodi sono:
  - i goal che vogliamo raggiungere;
  - gli agenti;
  - gli oggetti concettuali;
- gli archi sono:
  - i responsibility assignment;
  - i performance link;
  - i control e monitoring link.



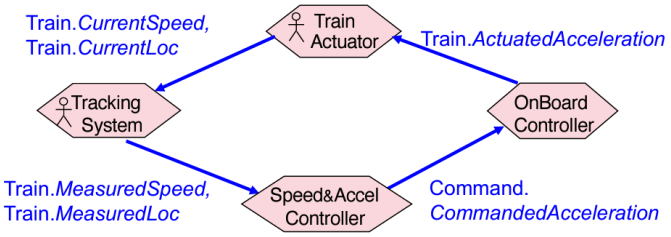
Come abbiamo detto, l'assegnamento delle responsabilità può essere alternativo, ovverosia possiamo, come nel caso dei goal, avere degli **OR-assignments**, ovverosia un goal può essere raggiunto da diversi agenti (a seconda della configurazione che vogliamo avere) e il goal è soddisfatto se almeno uno degli agenti ha raggiunto il goal. Ricordiamoci che **ogni goal deve essere assegnato ad un solo agente, per questo motivo non possiamo avere AND-assignments**. Vediamo un esempio:



Il goal può essere assegnato al train controller oppure al train driver oppure al passenger, a seconda della configurazione che vogliamo avere. Quindi, attraverso l'OR assignment identifichiamo 3 possibili alternative all'assegnamento delle responsabilità ad uno di questi 3 agenti.

A questo punto, vediamo il **context diagram** → esso rappresenta solamente gli agenti con dei link e abbiamo un link tra un agente ed un altro, quando un agente controlla delle grandezze che sono monitorate dal secondo agente. Intuitivamente, il context diagram rappresenta un flusso dei dati, in quanto nel context diagram rappresentiamo gli agenti e abbiamo un link orientato tra due agenti quando l'agente 'source' (ovvero l'agente1) controlla una variabile controllata dall'agente 'target' (ovvero l'agente2).

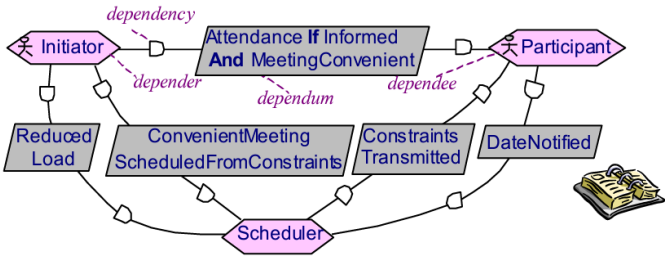
Riassumendo, quindi, il context diagram è una **vista** che rappresenta solamente la relazione tra variabili controllate e monitorate dagli agenti. Un esempio di context diagram è il seguente:



Abbiamo i vari agenti e per esempio, il controllore della velocità imposta il comando di accelerazione, il quale viene letto dall'OnBoard controller e in base al comando che legge, l'OnBoar controller imposta la velocità del treno., la quale a sua volta è letta dall'attuatoree e in base alla velocità letta, l'attuatore imposta la posizione corrente e la velocità corrente.

> 💡 Un agente potrà prendere decisioni in relazione alle informazioni in termine di variabili controllate e monitorate.

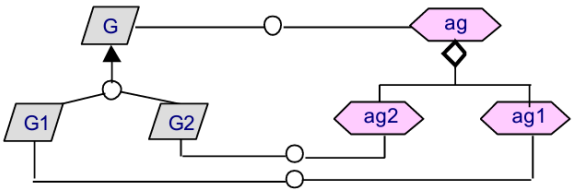
Vediamo, infine, un esempio completo di **dependency diagram** → esso è una vista, che mette in evidenza tutte le dipendenze tra gli agenti. Come possiamo vedere dall'immagine sotto, abbiamo vari goal e vari agenti. Abbiamo, ad esempio, che l'agente 'Partecipante' imposta delle dipendenze e questo, ci permette di capire come i fallimenti si propagano nei goal e quali altri agenti potrebbero far fallire (a catena) i goal.



Inoltre, attraverso il diagramma delle dipendenze possiamo identificare numerose catene di dipendenza, che ci permettono di capire come il sistema

potrebbe fallire e quali fallimenti a cascata potrebbero essere possibili.

Gli agenti possono essere raffinati in sotto-agenti più raffinati, ovvero possiamo avere una decomposizione gerarchica degli agenti. Questo permette di avere una visione più dettagliata degli agenti e delle loro responsabilità. Laddove un goal di alto livello è assegnato ad un agenti di alto livello, questo può essere raffinato in sotto-agenti che si occupano di raggiungere i goal di basso livello rispetto ad una mappatura 1 a 1.



Come possiamo vedere dall'immagine, il raffinamento degli agenti è rappresentata come una specie di aggregazione, perchè di fatto i due sotto-agenti potrebbero essere considerati come un singolo agente a cui viene assegnato il goal *G*. Vediamo, allora, che il raffinamento degli agenti potrebbe avere una struttura simile al raffinamento dei goal e di conseguenza, ad una AND-decomposition di un goal, potrebbe corrispondere un'aggregazione di due sotto-agenti, ognuno dei quali è distintamente responsabile di uno dei due sotto-goal.

Euristiche per la modellazione degli agenti

- **Assegnamento di responsabilità e identificazione degli agenti** → innanzitutto, per identificare gli agenti dobbiamo chiederci quali sono gli oggetti attivi all'interno del nostro sistema, ovverosia gli oggetti che in qualche modo intraprendono azioni per vincolare oppure per modificare lo stato di alcune entità del nostro sistema. Quindi, quando assegniamo le responsabilità dobbiamo chiederci chi ha possibilità di:
  - modificare lo stato del sistema;
  - leggere le variabili di sistema;
  - controllare le variabili di sistema.

  Questo, allora, ci guida nell'identificazione degli agenti e il relativo assegnamento di responsabilità.

- **Quando assegniamo una responsabilità ad un agente umano bisogna valutare anche i desideri dell'utente** (i goal di sicurezza informatica devono essere assegnati a persone che hanno a cuore la sicurezza del sistema, quindi in contrasto con il desiderio dell'utente finale) → quindi, identifichiamo gli agenti che, tra i loro desideri, hanno quello di raggiungere un determinato goal (naturalmente se hanno le capabilities opportune);
- Durante l'assegnamento delle responsabilità, è molto importante **non confondere agenti a livello di prodotto con gli stakeholder a livello di processo**. Gli agenti sono dei ruoli, mentre gli stakeholder sono delle persone che intervisto, di fatto la stessa persona potrebbe avere ruoli diversi ed essere modellato in agenti diversi;
- **Assegnare le responsabilità a componenti software**, piuttosto che agli utenti del sistema, in modo tale da raggiungere un livello di automazione maggiore;
- **Raffinare gli agenti in linea con i goal**, in modo da raggiungere una granularità tale da poter assegnare responsabilità in modo chiaro;
- Durante l'assegnamento delle responsabilità**, dobbiamo tenere in considerazione la vulnerabilità che stiano introducendo all'interno del sistema**, ovverosia se abbiamo catene di dipendenze molto lunghe, probabilmente l'assegnamento di responsabilità che abbiamo deciso non è ottimale.

"**Quali sono le euristiche che utilizziamo per disegnare i diagrammi?**" Il context diagram può essere direttamente ricavato dal nostro goal diagram e tipicamente, dobbiamo andare a vedere come abbiamo abbiamo scritto i goal. In questo senso, se abbiamo un goal come segue, lo dobbiamo trasformare nel seguente modo:

```
G:  CurrentCondition [monitoredVariables]
        ⇒ [sooner-or-later/always] TargetCondition [controlledVariables]
```

Quindi, implicitamente i goal possono far riferimento a variabili controllate e monitorate, quindi possiamo creare i link che monitora la variabile monitorata e controlla la variabile controllata, iterando agente per agente.

Modeling system operations

In questo capitolo vediamo l'**operation model**, in cui il punto di vista sono le **operazioni** → le operazioni sono il modo in cui i goal vengono raggiunti e di conseguenza, nella vista vediamo quali funzionalità e servizi vengono messi a disposizione e implementati dal sistema, al fine di raggiungere gli obiettivi di cui abbiamo discusso e naturalmente tali servizi, vengono messi a disposizioni in particolari condizioni → quindi, l'obiettivo è di capire e documentare quali sono le condizioni che attivano i vari servizi, perchè successivamente saranno funzionali alla progettazione dettagliata del sistema → in particolare, la rappresentazione avviene mediante l'ausilio dei **diagrammi UML,** declinati in maniera funzionale alla modellazione dei requisiti. Gli obiettivi di queste viste sono molteplici:

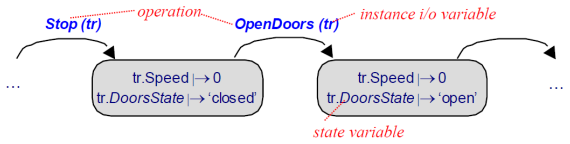
- cerchiamo di definire quali sono le specifiche del software, che successivamente sarà un input per il team di sviluppo del sistema;
- descriviamo l'ambiente, le task e le procedure.

Le utilità di queste viste sono molteplici:

- possiamo iniziare a scrivere qualche test di accettazione e di conseguenza, possiamo iniziare a capire quali dati dobbiamo utilizzare per scrivere i test di accettazione, che scriviamo prima (ovvero scriviamo prima i test di accettazione) di iniziare la progettazione del software;
- possiamo iniziare a definire i function points (i quali ci permettono di stimare l'effort necessario per implementare i vari componenti del sistema);
- permette di creare un link tra tutte le funzionalità del sistema e i goal che si vogliono raggiungere.

A questo punto, ci facciamo una prima domanda fondamentale: "**Che cos'è un'operazione?**" **Un'operazione è una mappa tra uno stato di input ed uno stato di output**. Di fatto, un'operazione è una funzione che cambia lo stato del nostro sistema ed in particolare, fa transire il sistema dallo stato di input per arrivare ad uno stato di output → quindi, vi è una relazione tra gli stati ed in particolare, uno stato è una lista di coppie (X,V) dove: X è una variabile e V è il corrispettivo valore della variabile → capiamo, allora, che **un'operazione permette di cambiare il valore di alcune variabili** (perchè un'operazione parte da uno stato iniziale, che è una coppia di variabile-valore, e va in uno stato

finale, che anch'esso è una coppia differente di variabile-valore) → quindi, le variabili di input sono le variabili sulle quali si applicano le operazioni, mentre le variabili di output sono le variabili su cui le operazioni hanno un effetto (ovvero ne cambiano il valore) e quindi le variabili hanno degli attributi..



💡 **L'obiettivo del nostro sistema software è di controllare la transizione degli stati e attivare la transizione solamente quando è opportuno e restringere l'evoluzione degli stati successivi solo a quelli che soddisfano i goal del nostro sistema**. Chiaramente, la transizione avrà degli effetti sulle variabili e sull'ambiente, che dobbiamo controllare → L'obiettivo, allora, è di operazionalizzare i goal del goal model, ovvero permettere in maniera operativa di raggiungere i goal.

Continuiamo, dicendo che un'operazione operazionalizza i goal del nostro goal model, ovverosia: abbiamo i nostri goal di alto livello, che si decompongono in goal sempre più concreti e di basso livello. Poi, abbiamo le operazioni che di fatto agiscono sul sistema, in modo tale da garantire che un goal venga soddisfatto → quindi, **vi è un link diretto tra un goal e un'operazione, in quanto un'operazione fa in modo che il goal venga raggiunto. Le operazioni, quindi, sono il modo in cui gli agenti possono modificare l'ambiente e il sistema, al fine di raggiungere i goal di cui sono responsabili.**

Tipicamente, le operazioni sono:

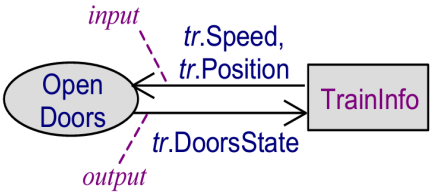
- **deterministiche** → ovvero non hanno un comportamento probabilistico (nel senso che hanno sempre un certo tipo di output);
- **atomiche** → non possono essere suddivise in operazioni più piccole;
- **possono essere applicate in modo concorrente** → possono essere eseguite in modo parallelo.

"**Come definiamo le operazioni?**" Le operazioni hanno quattro elementi fondamentali:

- un nome;

- una definizione;
- una categoria;
- una firma.

Vengono rappresentate mediate l'ellisse come nel Use Case Diagram e hanno un **input** (rappresentato mediante una freccia che esce dall'oggetto concettuale ed entra nell'operazione) ed un **output** (rappresentato mediante una freccia uscente dall'operazione ed entrante nell'oggetto concettuale). In alternativa, possiamo rappresentare le feature dell'operazione mediante una notazione testuale. Vediamo un esempio di operazione:



Come abbiamo detto precedentemente, un'operazione è caratterizzata da un cambiamento del sistema e tipicamente, un'operazione può avvenire solamente in determinate condizioni, in quanto necessita di alcune pre-condizioni per attivarsi. Capiamo, allora, che un'operazione ha delle pre-condizioni e delle post-condizioni ed in particolare:

- le pre-condizioni vengono denominate con la keyword '**DomPre**', che sono le pre-condizioni di dominio per l'attivazione dell'operazione. Le pre-condizioni di dominio sono le caratteristiche del dominio (all'interno del quale opera il sistema) che sono necessarie per attivare l'operazione (quindi le pre-condizioni devono essere vere per attivare l'operazione);
- le post-condizioni vengono denominate con la keyword '**DomPost**', che sono le post-condizioni di dominio, che caratterizzano l'avvenuta operazione. Quindi, sono le condizioni che devono essere verificate affinché l'operazione sia stata eseguita correttamente.

Sia le DomPre sia le DomPost sono assolute e descrittive e non prescrittive.

Notiamo, inoltre, che vi è un legame tra l'operazione e l'agente → tipicamente, un'agente attua un'operazione per raggiungere il goal per il quale è responsabile (quindi, intuitivamente possiamo dire che vi è una relazione a 3: agente - operazione - goal) e naturalmente, abbiamo due regole di consistenza:

1. Ogni variabile di input e di output, nella firma di un'operazione (ovvero le variabili che un'operazione legge e scrive), devono obbligatoriamente essere tra le variabili monitorate e controllate dell'agente che è responsabile del goal → di fatto, un agente deve poter leggere e scrivere le variabili che sono in input e in output all'operazione;
2. Ogni operazione è eseguita da un solo agente, che sarà quello responsabile del goal corrispondente e tale regola prende il nome di **Unique Controller**.

Operazionalizzazione dei goal

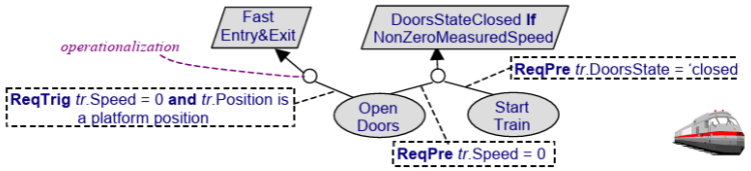
Precedentemente abbiamo visto le DomPre e le DomPost. A questo punto, vediamo le:

- **ReqPre** → sono le condizioni **necessarie** per soddisfare un goal, ovverosia sono le condizioni dell'input di una certa operazione, che sono necessarie per soddisfare un certo goal. Questo vuol dire, che **quando una DomPre è vera, allora un'operazione viene applicata solo se la ReqPre è vera (quindi, le ReqPre sono condizioni necessarie per applicare un'operazione)** → Con le ReqPre vi è quindi marginalità, ovverosia possiamo o meno attivare una certa operazione
- **ReqTrig** → sono condizioni **sufficienti**, ovverosia: **se abbiamo che le DomPre sono vere, allora un'operazione deve attivarsi non appena la ReqTrig è vera** (si tratta, quindi, di una condizione per triggerare l'operazione) → in questo caso, non vi è quindi marginalità, ovverosia l'operazione deve iniziare, quindi la ReqTrig è un evento che fa scattare/iniziare l'operazione immediatamente;
- **ReqPost** → condizione sull'output, ovverosia una volta terminata l'operazione, questa diventa vera. Quindi, sono le condizioni che devono essere verificate affinché l'operazione abbia soddisfatto il goal.

💡 Sia le ReqPre sia le ReqPost sono prescrittive (e non descrittive). Inoltre, ognuna delle tre devono far riferimento ad un goal distinto. Per esempio, se la stessa operazione operazionalizza più goal, allora dobbiamo distinguere quali sono le ReqPre per il primo goal e quali per il secondo goal e così via.

Vediamo un esempio:

- Operazione: OpenDoors
  - Def: Operazione che controlla l'apertura di tutte le porte di un treno.
  - Input: tr: TrainInfo / {Speed, Position}
  - Output: tr: TrainInfo / {DoorState}
  - DomPre: Le porte del treno tr sono chiuse.
  - DomPost: Le porte del treno tr sono aperte.
  - ReqPre: La velocità del treno tr è zero.
  - ReqPre: Il treno tr è in stazione.
  - ReqTrig: Il treno tr si è e appena fermato in stazione.



Dall'immagine, possiamo vedere che abbiamo l'operazione 'Open doors', la quale operazionalizza 'Fast entry and exit' ed operazionalizza la 'Doors state closed'. La condizione, che lega l'operazione di Open doors al goal di Fast entry and exit è la ReqTrig 'Speed = 0 and Position is a platform'. Tra Open doors e il goal, invece, vi è la ReqPre 'Speed = 0'.

La **operazionalizzazione** è il legame fra un goal e un'operazione, in quanto l'operazione di fatto operazionalizza il goal, ovverosia attua dei mutamenti nel nostro sistema, in modo tale che il goal venga raggiunto. In generale, abbiamo che un goal può essere operazionalizzato da molteplici operazioni (quindi ci possono essere diverse operazioni per un singolo goal), perchè magari è

necessario svolgere diverse “cose” per raggiungere il goal e di conseguenza, un’operazione operazionalizza, in generale, più goal → quindi se abbiamo le nostre pre-condizioni e le post-condizioni sui goal, queste sono implicitamente congiunte con le pre e post condizioni sul dominio. Se una precondizione di dominio è vera, deve essere applicata appena la condizione di trigger diventa vera. Invece, se una precondizione di dominio può essere applicata, quando tutte le ReqPre sono vere.

Abbiamo poi un **vincolo di consistenza**:

> 💡 Se una precondizione di dominio è vera e almeno una delle condizioni di trigger è vera, allora tutte le preconditioni di operalizzazione dei goal **devono** essere vere, altrimenti abbiamo un problema di consistenza.

Agent commitments

Come abbiamo detto in precedenza, un agente ha come responsabilità di garantire uno o più goal e ha come capacità, quella di attivare una o più operazioni, al fine di garantire i goal. Per ogni goal sotto la responsabilità di un agente e per ogni operazione che operazionalizza un goal, abbiamo che l’agente deve garantire che l’operazione sia attivata:

- Quando le operazioni **DomPre** sono verificate;
- Non appena una condizione **ReqTrig** sia verificata;
- Solo se le operazioni **ReqPre** sono verificate;
- In Modo tale che le operazioni **DomPost** e **ReqPost** siano verificate.

Anche in questo caso, dobbiamo rispettare **due regole di consistenza**:

1. se l’agente, per raggiungere il goal G, deve operazionalizzare tutte le operazioni relative al goal G;
2. se tali operazioni operazionalizzano altri goal, allora l’agente dovrebbe essere responsabile anche per gli altri goal.

Di fronte agli agenti, ci troviamo di fronte a un non determinismo, poiché potremmo trovarci di fronte a due categorie di agenti:

1. agente **eager** → agente estremamente pro-attivo, il quale fa partire un'operazione non appena tutte le ReqPre sono vere;
2. agente **lazy** → agente completamente opposto all'agente eager, in quanto l'agente lazy è estremamente pigro, dato che anche se le operazioni sono attivate, non le fa partire fino a quando non è necessariamente obbligato, ovverosia aspetta fino a quando non vi è una ReqTrig.

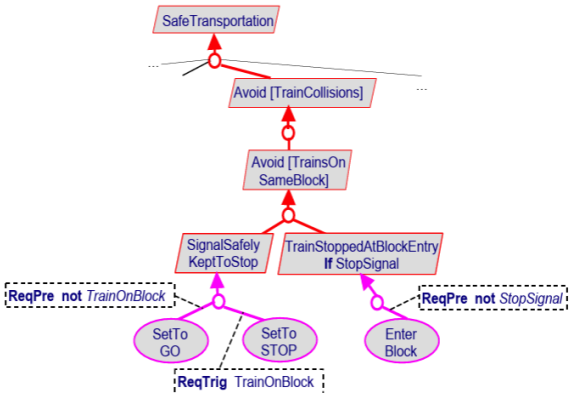
Chiaramente, tra queste due tipologie di agenti (quindi tra l'agente eager e l'agente lazy) vi sono molteplici tipologie di agenti e questo ci fa capire, che ci troviamo di fronte a un certo grado di libertà per gli agenti, che possono decidere quando far partire le operazioni e una sorta di concorrenza tra le gli agenti stessi.

Operalizzazione dei goal e soddisfacimento

Consideriamo l'operalizzazione di un goal in concomitanza con la soddisfacibilità dei goal, l'operation model e il goal model possono argomentare sugli obiettivi di alto livello, che vengono raggiunti. Di fatto, possono essere percorsi:

- **top-down** → per capire **perchè** un goal è stato raggiunto;
- **bottom-up**→ per capire **come** un goal può essere raggiunto.

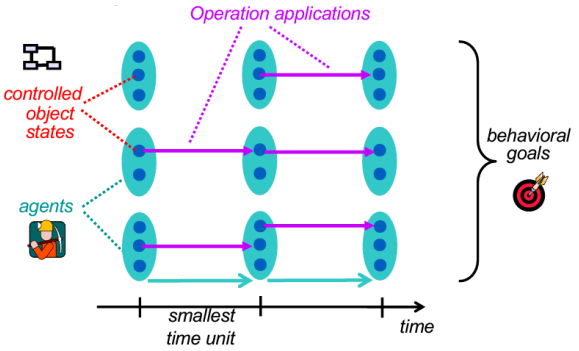
Vediamo un esempio:



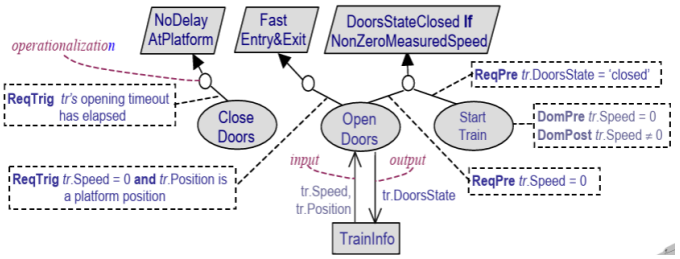
Partiamo da un goal di alto livello, ovvero "SafeTransportation" e viene articolato in vari sotto-goal. Per raggiungere il goal dei trasporti sicuri, quindi, dobbiamo evitare collisioni tra i treni. A sua volta, per riuscire a fare ciò, dobbiamo evitare che i treni siano sugli stessi binari e così via. Alla fine, quindi, abbiamo diverse operazioni, come ad esempio: impostare lo stato del treno a GO oppure a STOP e l'operazione di entrare nel prossimo blocco ed occuparlo.

💡 Dall'immagine, quindi, possiamo capire che indirettamente abbiamo una relazione tra le precondizioni per far partire le operazioni e raggiungere il goal "SignalSafelyKeptToStop" e tali precondizioni risalgono la gerarchia del modello, fino a raggiungere il goal di alto livello → quindi, le precondizioni dovrebbero essere collegate per raggiungere il goal di alto livello.

Rappresentazione semantica delle operazioni



Come possiamo vedere dall'immagine, abbiamo uno stato del nostro sistema con degli oggetti controllati e questo sistema transisce da uno stato all'altro. La modifica degli oggetto avviene attraverso gli agenti, i quali permettono di fatto al sistema di transire da uno stato all'altro. Le operazioni si applicano nel corso del tempo e possiamo notare, che la successione degli stati avviene in parallelo (ovvero gli oggetti evolvono in parallelo, ma vi sono dei pezzi diversi che vengono modificati da agenti), in modo tale che i goal del sistema vengono raggiunti → in qualche modo l'agente, allora, modifica gli stati al fine di restringere tutti i possibili behavior del sistema, a solo quelli che rendono il sistema corretto, ovvero solo a quelli che permettono di soddisfare i goal. Un esempio di operationalization diagram è il seguente:



Possiamo vedere che abbiamo i nostri goal, i quali sono operazionalizzati da alcune operazioni (quindi anche in questo diagramma abbiamo gli operationalization link, che però in questo caso legano un'operazione ad un goal e sono annotati con i vari requirements). Inoltre, le operazioni possono essere collegate agli oggetti concettuali, che usano come input e output. All'interno operationalization diagram, quindi, abbiamo:

- i **goal**;
- le **operazioni**;
- gli **oggetti concettuali**, per evidenziare gli input e gli output.

L'operationalization diagram, inoltre, può essere rappresentato come un UML use case diagram. Quindi, le operazioni possono essere rappresentate mediante un **use case diagram**, rappresentando le operazioni e gli agenti. In particolare, nel use case diagram, possiamo avere delle interazioni con:

- l'agente che controlla le operazioni in input;
- l'agente che monitora operazioni in output;
- ci sono dei link tra le operazioni di extend e include:
  - **Include** → l'operazione include un'altra operazione;
  - **Extend** → l'operazione estende un'altra operazione.

Dobbiamo però ricordarci, che le operazioni sono: atomiche, non decomponibili in sotto-operazioni e di conseguenza, questa tipologia di link non sarebbe da utilizzare. Capiamo, allora, che **la decomposizione si applica ai goal e non alle operazioni**.

Derivare operazioni dai goal in maniera fluida

La prima modalità consiste nell'utilizzare i **fluents**, dove per fluents intendiamo delle **condizioni che identificano l'inizio e la fine di un'operazione**.
Consideriamo il goal "*Le porte del treno sono chiuse se il treno è in movimento*", il quale viene catturato da due condizioni:

1. la condizione sul movimento;
2. la condizione sulle porte chiuse.

Entrambe le condizioni sono considerate fluents, perchè transiscono da uno stato vero ad uno stato falso numerose volte durante l'esecuzione del nostro

sistema → quindi, per ogni fluent abbiamo un'operazione che lo porta a vero quando era falso (chiamata initiating operation) e un'operazione che lo porta a falso quando era vero (chiamata terminating operation). Quindi, l'identificazione del fluent all'interno dei goal, ci permette prima di tutto di identificare le operazioni necessarie per raggiungere il goal principale (nel nostro caso, che le porte del treno devono essere chiuse se il treno è in movimento) e in secondo luogo, ci permettono di capire runtime come tali operazioni si susseguiranno in termini temporali.

La seconda metodologia è di identificare le operazioni partendo dagli scenari di interazione. Quindi, per ogni evento di interazione in uno scenario (dove lo scenario è un'applicazione operativa) da parte dell'agente sorgente con output monitorato dall'agente di destinazione e per ogni interazione, dobbiamo decidere quali sono le operazioni che operazionalizzano i nostri goal.

La terza modalità è il rafforzamento delle precondizioni e postcondizioni con le condizioni richieste dai goal. Per riuscire a fare ciò, dobbiamo:

- **Identificare i permessi richiesti** → se l'effetto DomPost di un'operazione può violare un obiettivo ***G*** sotto condizione ***C;***
- Allo stesso tempo dobbiamo **identificare le obbligazioni richieste** → se l'effetto DomPost di un'operazione è prescritto da un obiettivo ***G*** da mantenere ogni volta che la condizione ***C*** diventa true;
- Infine, dobbiamo **identificare gli effetti aggiuntivi richiesti** → se il DomPost di un'operazione non è sufficiente per garantire la condizione target ***T*** di un obiettivo ***G***.