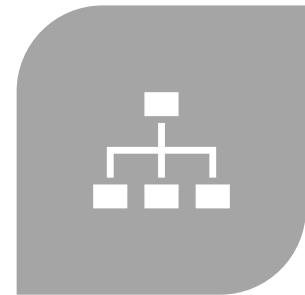


ARCHITECTURAL PATTERN

Introduzione



STRATIFICAZIONE DELLE
APPLICAZIONI AZIENDALI



STRUTTURAZIONE DELLA LOGICA DI
DOMINIO (AZIENDALE)



STRUTTURARE UN'INTERFACCIA
UTENTE WEB



COLLEGAMENTO DI MODULI IN
MEMORIA (IN PARTICOLARE
OGGETTI) A UN DATABASE
RELAZIONALE



GESTIONE DELLO STATO DELLA
SESSIONE IN AMBIENTI SENZA
STATO



PRINCIPI DI DISTRIBUZIONE

Progettare e sviluppare sistemi informatici è difficile.

Man mano che la **complessità del sistema aumenta, il compito di costruire il software diventa **esponenzialmente** più arduo.**

Come raggiungere allora l'obiettivo** ?**

Come in ogni professione, possiamo progredire solo imparando dai nostri errori e dai nostri successi.



«**Architettura**» è un termine che molti provano a definire (chi con più successo chi meno) ed eviteremo di aggiungerne altre.

Quello che possiamo affermare è che vi sono due **fattori critici**:

- **la scomposizione di un sistema dal suo livello più alto nelle sue parti**
- **decisioni da prendere che poi risultano difficili da cambiare in corso d'opera**



Col tempo si è presa coscienza del fatto che non esiste un solo modo per disegnare l'architettura di un sistema.

In molti casi può essere così complessa che, scendendo nel dettaglio, si debba scomporla in varie parti ognuna con una propria architettura.

Inoltre la **visione** di ciò che è architettonicamente significativo è quella che può cambiare nel corso della vita di un sistema.

Types of Solutions		
One Solution	No Solution	Many Solutions
Consistent Independent $X = 4, Y = 3$ $3X + Y = 17$ $4X - Y = 18$	Consistent Dependent $O = O, X = X$ $2X + 4Y = 8$ $X + 2Y = 4$	Inconsistent Independent $Z \neq S$ $3X + 2Y = 5$ $6X + 4Y = 8$

L'architettura è una cosa soggettiva, una comprensione condivisa del design di un sistema da parte degli sviluppatori esperti di un progetto.

Ciò influenza l'individuazione e la definizione dei componenti principali del sistema e del modo in cui interagiscono tra di loro e con gli altri sistemi.

Influenza anche le «**decisioni**», e, soprattutto quelle che si ritiene debbano essere prese «**subito**» in quanto percepite come difficili da cambiare.

Al di là di ciò, definire un'architettura si riduce all'individuazione delle «cose importanti», qualunque essa sia.



La complessità del
software

La complessità del software

Molti sono i fattori che possono incidere sul **successo** di un progetto o meno: la burocrazia, obiettivi poco chiari, mancanza di risorse solo per citarne alcuni.

Tra questi uno dei più rilevanti è l'**approccio alla progettazione**; ciò perché ha effetti diretti sulla complessità stessa del software.

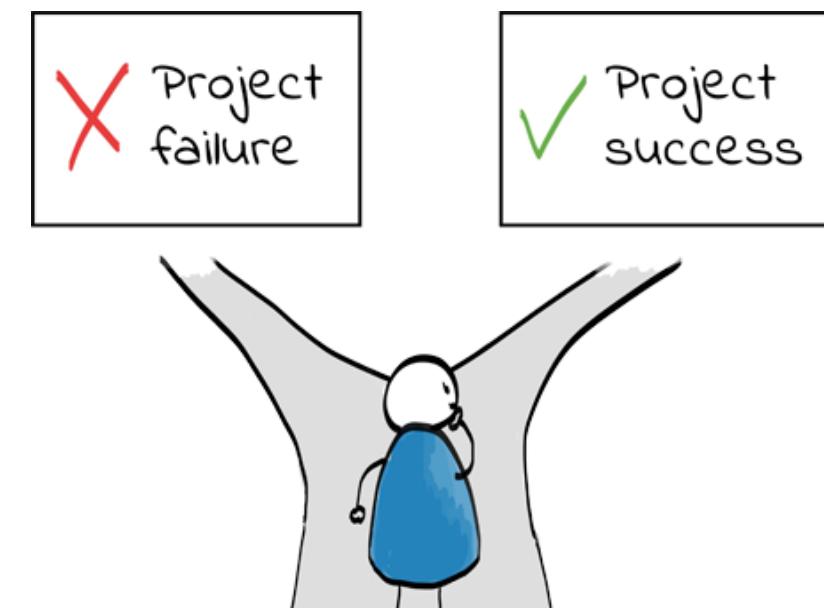
Quando la **complessità** sfugge di mano il software può perdere quelle caratteristiche che lo dovrebbero rendere facilmente modificabile ed estensibile.



Se si analizzano le statistiche sul numero di progetti consegnati in ritardo o con costi che hanno superato notevolmente il budget, più il numero di progetti falliti, si può restare stupiti per i risultati.

Il rapporto **CHAOS** 2015 dello Standish Group (<https://www.projectsmart.co.uk/white-papers/chaos-report.pdf>) suggerisce che dal 2011 al 2015, la percentuale di progetti IT di successo è rimasta invariata a un livello di solo il 22%.

Oltre il 19% dei progetti è fallito e il resto ha incontrato difficoltà.

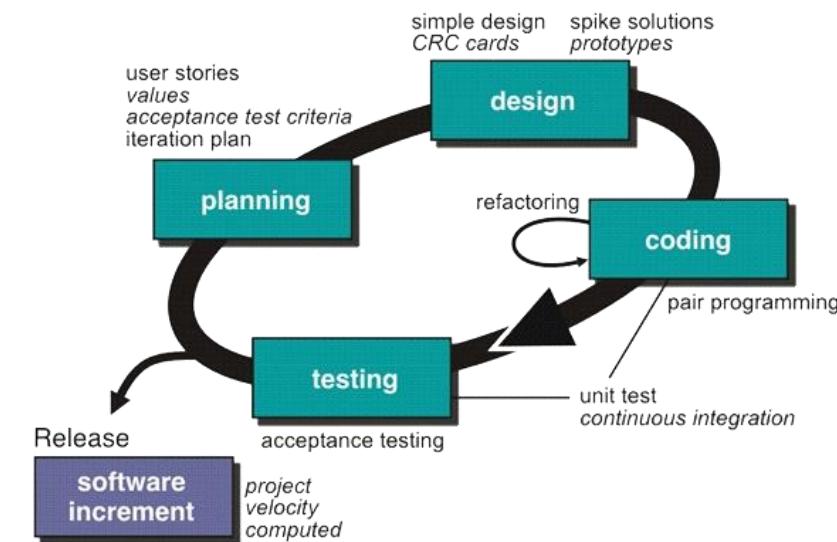
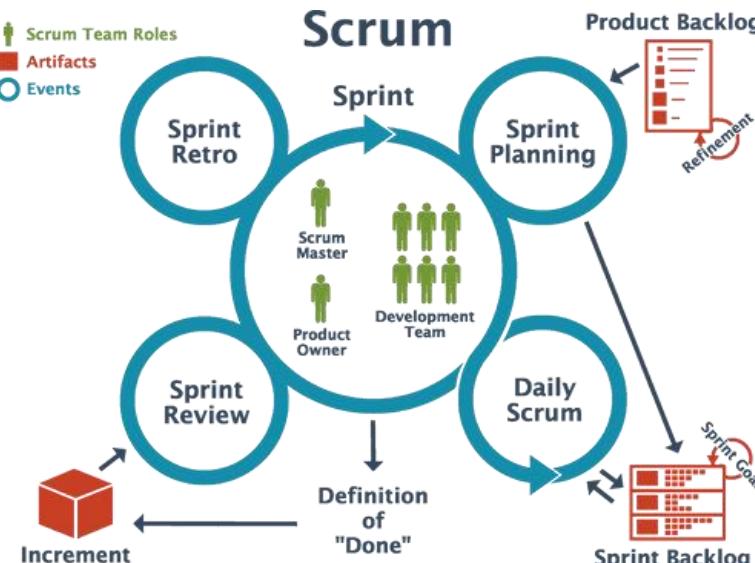


La comprensione dei problemi e le metodologie di sviluppo

Uno dei fattori critici che definiscono il successo di qualsiasi progetto IT è la **comprendere del problema che il sistema dovrebbe risolvere.**

L'esperienza ci insegna che molte volte i sistemi sviluppati o non risolvono i problemi a cui essi pretendono di rispondere o li risolvono in modo poco efficiente.

Metodologie di sviluppo software come SCRUM e XP abbracciano l'interazione con gli utenti e la comprensione dei loro problemi.



Problem Space & Solution Space

Professionalmente costruiamo software per aiutare altre persone a svolgere il loro lavoro al meglio, più velocemente e in modo più efficiente.

Ciò significa che dobbiamo avere un problema che intendiamo risolvere.

La **psicologia cognitiva** definisce il problema come una restrizione tra lo stato attuale e lo stato desiderato.

Ogni problema reale richiede una **soluzione**, e se cerchiamo correttamente nello spazio del problema, possiamo delineare quali passi dobbiamo compiere per passare dallo stato iniziale allo stato desiderato.

Questo schema e tutti i dettagli sulla soluzione formano uno **spazio della soluzione**.



La storia classica degli spazi «**problema e soluzione**», che si staccano completamente l'uno dall'altro durante l'implementazione, è la **storia della scrittura nello spazio**.

«La storia racconta che negli anni '60, le nazioni che esploravano lo spazio si resero conto che le solite penne a sfera non avrebbero funzionato nello spazio a causa della mancanza di gravità».

«La **NASA** ha quindi speso un milione di dollari per sviluppare una penna che funzionasse nello spazio, e i sovietici hanno deciso di utilizzare la cara vecchia matita, che non costa quasi nulla.»

Questa storia è così avvincente che sta ancora circolando....

È così facile da credere, non solo perché siamo abituati a spese dispendiose da parte di enti finanziati dal governo, ma soprattutto perché abbiamo visto così tanti esempi di **inefficienza e interpretazione errata di problemi del mondo reale, aggiungendo un'enorme complessità non necessaria alle loro soluzioni proposte e risolvere problemi che non esistono.**

...ma questa storia è un mito.

La NASA ha anche provato a usare le matite, ma ha deciso di sbarazzarsene a causa della produzione di microdust, della rottura delle punte e della potenziale infiammabilità delle matite di legno.

Una società privata chiamata Fisher ha sviluppato quella che ora è conosciuta come una penna spaziale.

Successivamente, la NASA ha testato la penna e ha deciso di usarla.

L'azienda ha anche ricevuto un ordine dall'Unione Sovietica e le penne sono state vendute in tutto il mondo e il prezzo era lo stesso per tutti, \$2,39 per penna.

Analizziamo ora l'altra parte del rapporto **spazio del problema / spazio della soluzione**.

Sebbene il problema stesso sembrasse essere semplice, vincoli aggiuntivi, che potremmo anche chiamare **requisiti non funzionali** o, per essere più precisi, **requisiti operativi**, lo rendevano più complicato di quanto sembrasse a prima vista.

Saltare a una soluzione è molto semplice e poiché la maggior parte di noi ha un'**esperienza** piuttosto ricca di risoluzione dei problemi quotidiani, possiamo trovare soluzioni per molti problemi quasi immediatamente.

Tuttavia pensare alle soluzioni impedisce al nostro cervello di pensare al problema motivo per cui iniziamo ad approfondire la soluzione che per prima ci è venuta in mente, aggiungendo più livelli di dettaglio e rendendola la soluzione più ideale per un dato problema.

Trovare la soluzione al problema

C'è un altro aspetto da considerare quando si cerca una soluzione a un dato problema.

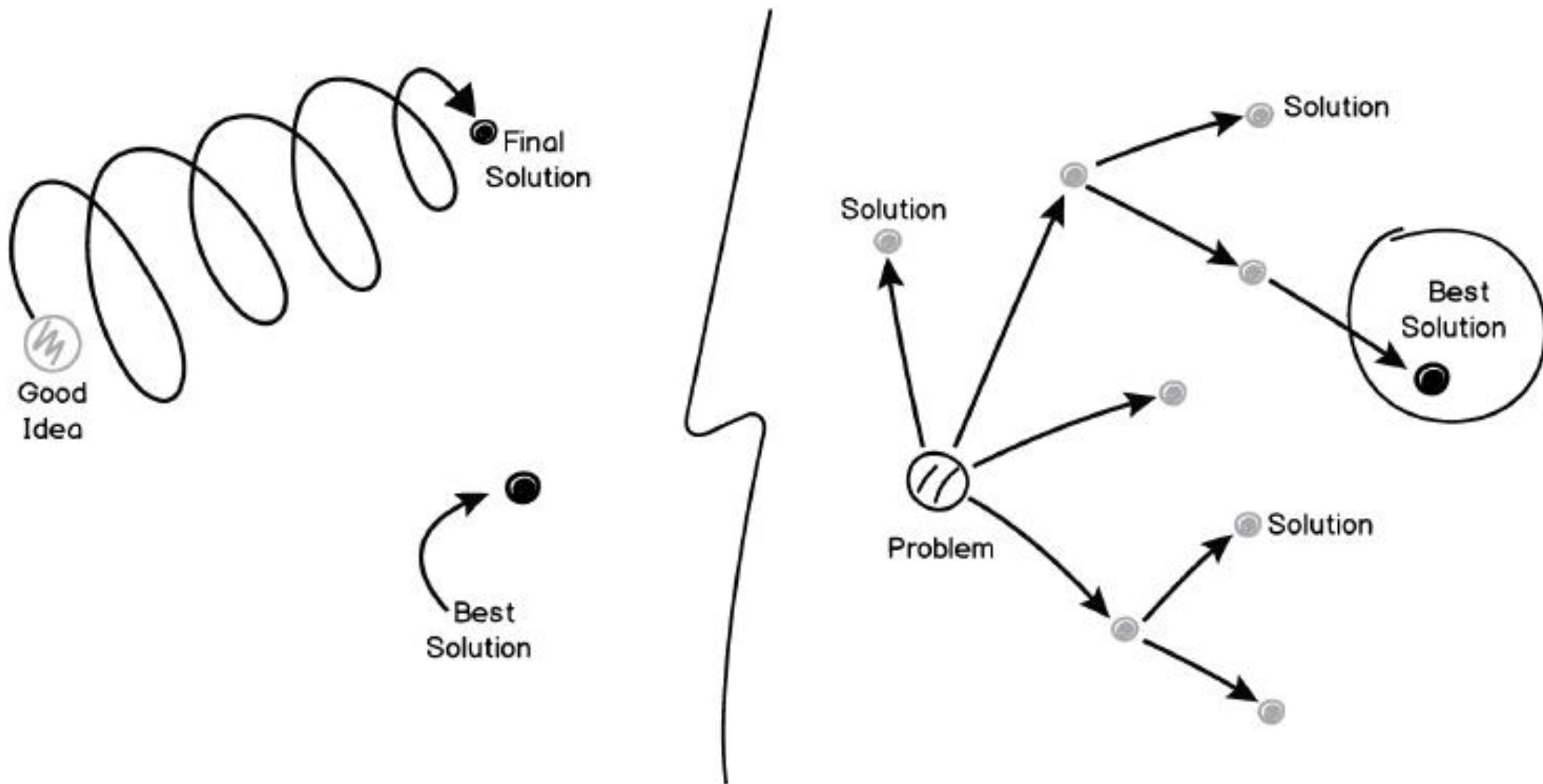
C'è il pericolo di concentrare tutta la propria attenzione su una particolare soluzione, che potrebbe non essere affatto la migliore, ma è stata la prima a venire in mente.

Ciò accade in funzione delle esperienze precedenti, dell'attuale comprensione del problema e di altri fattori ancora.

L'approccio esplorativo per trovare e scegliere soluzioni richiede molto lavoro per provare alcune cose diverse, invece di concentrarsi sul miglioramento iterativo della buona idea originale.

La risposta che si trova durante questo tipo di esplorazione sarà probabilmente molto più precisa e preziosa.

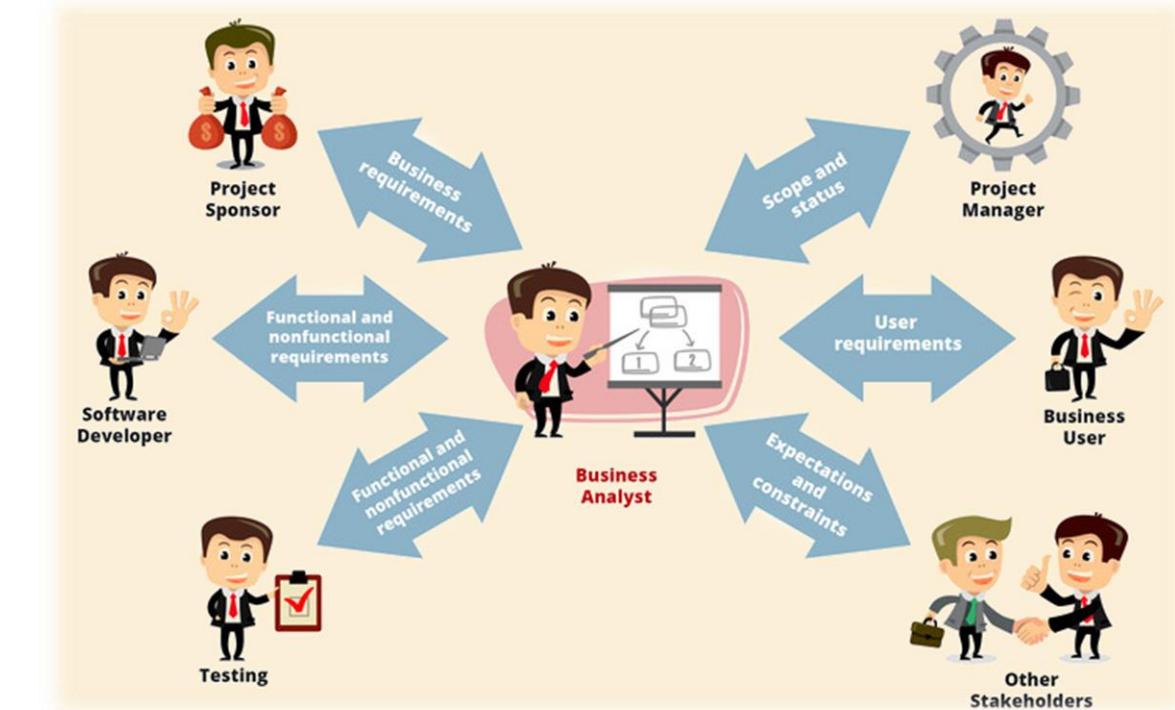
Trovare la soluzione al problema (Fowler)



Cosa è andato storto con i requisiti

Gli sviluppatori **raramente** hanno un contatto diretto con chi vuole risolvere un problema.

Di solito, figure preposte, come analisti dei requisiti, analisti aziendali o responsabili di prodotto, parlano con i **clienti** e generalizzano i risultati di queste conversazioni sotto forma di **requisiti funzionali**.



I requisiti possono essere codificati in documenti di grandi dimensioni (specifica dei requisiti) o, in ottica agile, come le storie degli utenti.

Cosa è andato storto con i requisiti

Analizziamo questa frase:

«Ogni giorno il sistema genera, per ogni hotel, un elenco di ospiti che dovrebbero effettuare il check in e il check out in quel giorno».

Essa descrive solo la soluzione.

Non possiamo sapere cosa sta facendo l'utente e quale problema risolverà il nostro sistema.

Potrebbero essere specificati requisiti aggiuntivi, perfezionando ulteriormente la soluzione, ma, solitamente, una descrizione esaustiva del problema non è mai inclusa nei requisiti funzionali.

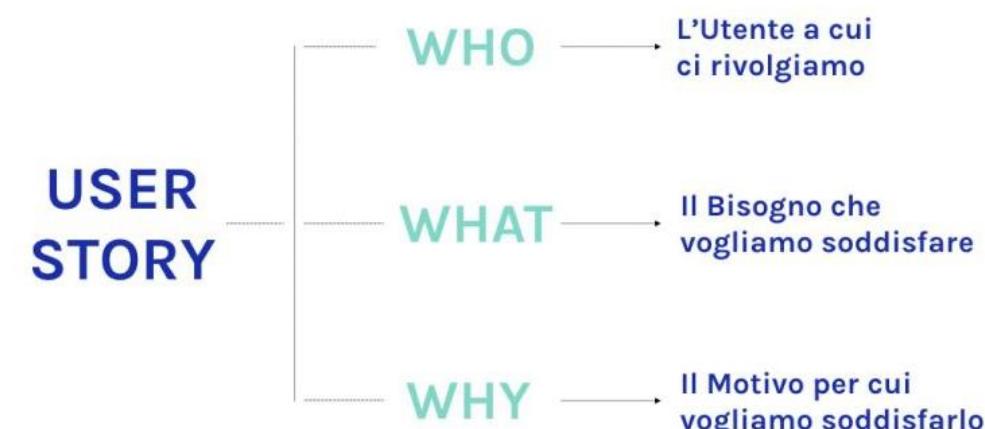
Cosa è andato storto con i requisiti

Al contrario, con le «**user stories**», abbiamo più informazioni su ciò che il nostro utente desidera. Esempio:

«In qualità di responsabile del magazzino, devo essere in grado di stampare un rapporto a livello di scorte o per poter ordinare gli articoli quando sono esauriti».

In questo caso, abbiamo un'idea di ciò che vuole il nostro utente.

Tuttavia, questa user story detta già ciò che gli sviluppatori devono fare ovvero «sta già descrivendo la soluzione».



Infatti, nonostante lo sforzo di analisi, restano fuori le seguenti domande:

«Il vero problema è che il cliente ha bisogno di un processo di approvvigionamento più efficiente in modo tale da non esaurire mai le scorte ?»

Oppure

«Il cliente ha bisogno di un sistema avanzato di previsione degli acquisti, in modo che possa migliorare la produttività senza accumulare scorte aggiuntive a magazzino ?».



Cosa è andato storto con i requisiti

Non dovremmo mai pensare che la raccolta dei requisiti siano una perdita di tempo.

Ci sono molti ottimi analisti che producono specifiche di requisiti di alta qualità.

Ma è fondamentale comprendere che questi requisiti «rappresentano quasi sempre la comprensione del problema reale dal punto di vista della persona che li ha scritti».

Ciò porta alla seguente considerazione «E' sbagliata l'idea secondo cui è necessario spendere sempre più tempo e denaro per scrivere requisiti di qualità superiore».

Le metodologie agili abbracciano una comunicazione più diretta tra sviluppatori e utenti finali.

Questo è il motivo per cui i vecchi modelli di analisi e design vengono rivisti e si adeguano ai tempi e alle tecnologie moderne; in questo contesto si inserisce, ad esempio, il **DDD (Domain-Driven Design) dove diventa centrico:**

- La comprensione del «**problema**» da tutte le parti interessate (dagli utenti finali agli sviluppatori e tester)
- Trovare soluzioni insieme
- Eliminare ipotesi
- Costruire prototipi da valutare per gli utenti finali
- Etc.

Affrontare la complessità

Nel **software**, l'idea di **complessità** non è molto diversa dal mondo reale dato che la maggior parte del software è scritta per affrontarne i problemi.

Senza dubbio, la complessità dello spazio del problema si rifletterà nel software che cerca di risolverlo.

Rendersi conto del tipo di complessità con cui abbiamo a che fare durante la creazione del software diventa quindi preminente rispetto all'analisi e al design.



Tipo Di Complessità

La «**complessità essenziale**» deriva dal dominio, dal problema stesso e non può essere rimossa senza diminuire la portata del problema.

Al contrario, la «**complessità accidentale**» viene portata alla soluzione dalla soluzione stessa; potrebbe essere un framework, un database o qualche altra infrastruttura, con diversi tipi di ottimizzazione e integrazione.

Il livello di «**complessità accidentale**» dovrebbe diminuire notevolmente mano a mano che l'industria del software diventa più matura.

Linguaggi di programmazione di alto livello e strumenti efficienti danno ai programmatori più tempo per lavorare sui problemI aziendali.

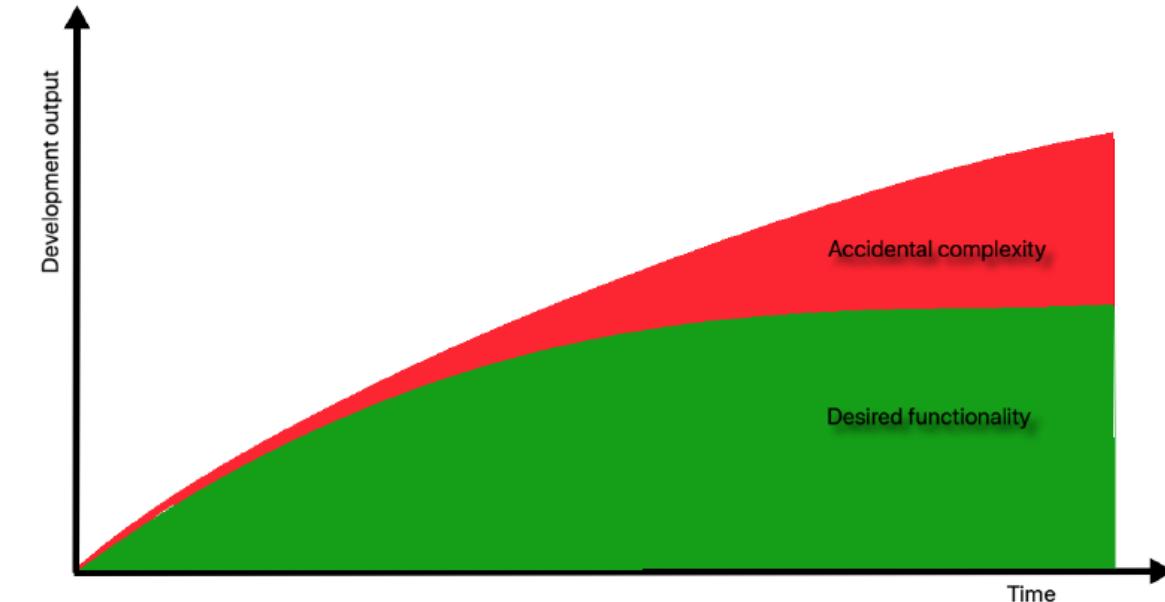
L'industria del software fatica ancora a combattere la «**complessità accidentale**».

Tipo Di Complessità

Gli sviluppatori amano principi come **DRY** (don't repeat yourself); cercano astrazioni che rendano il codice più elegante e conciso.

Ma questo sforzo potrebbe essere del tutto inutile; a volte cadono nella trappola di utilizzare qualche strumento o framework che promette di risolvere **facilmente** tutti i problemi del mondo.

Il grafico mostra che con la crescente «**complessità del sistema**», la «**complessità Essenziale**» viene ridotta e la «**complessità Accidentale**» prende il sopravvento.



Quando i sistemi diventano più importanti e quindi grandi, è necessario un enorme sforzo per farli funzionare nel loro insieme e per gestire i relativi modelli di dati (solitamente anche loro di grandi dimensioni).

Il codice cresce e aumenta e con esso cresce anche l'effort per mantenere il sistema in esecuzione:

- Introduciamo sistemi di cache
- ottimizziamo le query
- effettuiamo split o union di database

...e così via discorrendo

DDD aiuta a concentrarti sulla risoluzione di problemi di dominio complessi e si concentra sulla complessità essenziale.

Per un'azienda dovrebbe essere più importante ottenere prima qualcosa di utile e testabile piuttosto che ottenere un pezzo perfetto di software all'avanguardia che non coglie completamente il punto (ovvero che risolva il problema).

Per fare ciò, DDD offre diverse tecniche utili per la gestione della complessità suddividendo il sistema in parti più piccole e facendo in modo che queste parti si concentrino sulla risoluzione di una serie di problemi correlati.

La regola pratica quando si ha a che fare con la complessità è:

«Abbracciare l'essenziale ovvero dominare la complessità ed eliminare o ridurre la complessità accidentale».

L'obiettivo che uno sviluppatore dovrebbe porsi è NON CREARE troppa complessità accidentale.

Di fatto la complessità accidentale è causata da un'ingegnerizzazione eccessiva.

Classificazione della complessità

E' possibile misurare la complessità ?

Quando trattiamo «problemi», non sempre sappiamo se questi siano complessi o meno; in particolare dovremmo porci i seguenti quesiti:

- **Se sono complessi, quanto sono complessi ?**
- **Esiste uno strumento per misurare la complessità ?**
- **Se c'è, sarebbe utile misurare, o almeno categorizzare, la complessità del problema prima di iniziare a risolverlo ?**

Lo scopo di ciò è aiutare a regolare anche la complessità della soluzione, poiché, nella maggior parte dei casi, anche problemi complessi richiedono una soluzione complessa.

In **ingegneria del software** e in altre discipline informatiche, l'espressione «**dominio applicativo**» (o «**dominio dell'applicazione**»; in alcuni casi «**dominio del problema**») si riferisce al contesto in cui una applicazione software opera, soprattutto con riferimento alla natura e al significato delle informazioni che devono essere manipolate.

Nei più diffusi modelli e metodi di sviluppo del software, l'**analisi del dominio** (ovvero l'analisi volta a comprendere il contesto operativo in cui l'applicazione dovrà inserirsi) è una componente essenziale (e in genere preliminare) dell'analisi dei requisiti.

Cynefin (cu-NE-vin) è una parola gallese che in Inglese si traduce letteralmente come haunt o **habitat** (habitat anche in Italiano).

Il **Framework Cynefin** è stato creato da Dave Snowden di Cognitive Edge come uno strumento per aiutare il processo decisionale negli ambienti sociali complessi.

Snowden ha iniziato a lavorarci nel 1999, quando lavorava in **IBM**; Il lavoro è stato così prezioso che IBM ha istituito il Cynefin Center for Organizational Complexity, di cui Dave Snowden ne è stato fondatore e direttore.

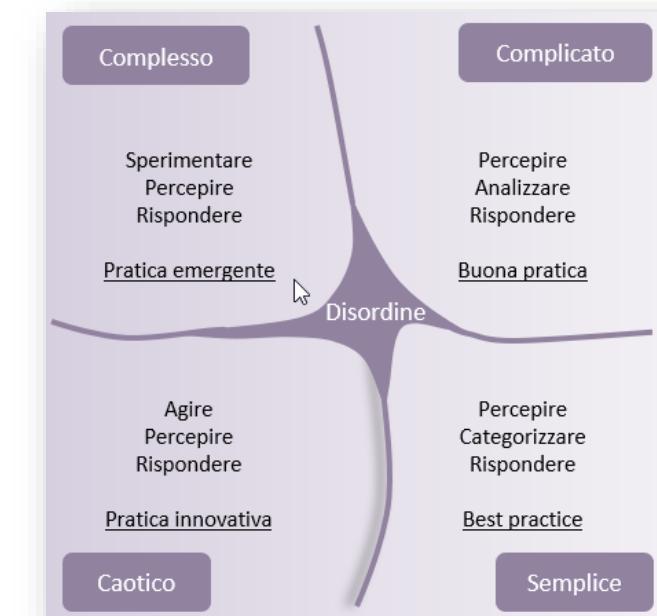
Classificazione Cynefin

Cynefin divide tutti i problemi in cinque «**categorie**» o «**domini di complessità**».

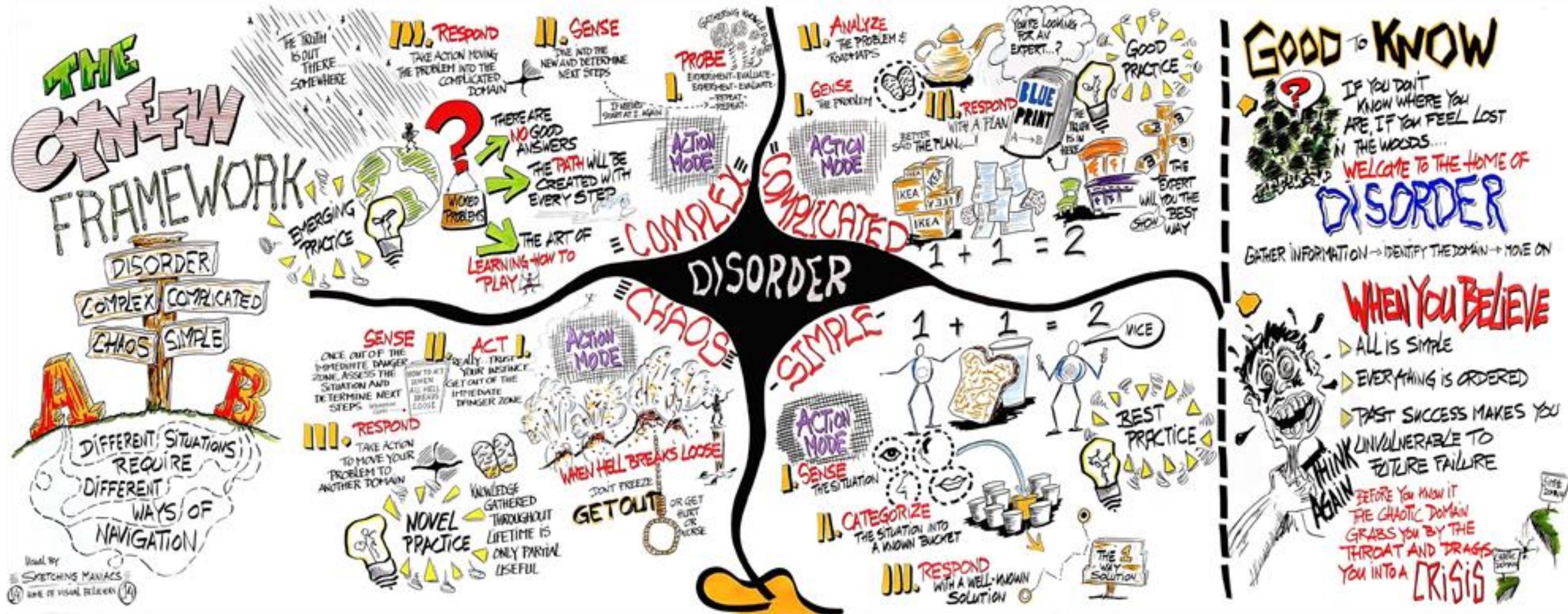
Fornisce un «**senso di collocazione**» per ogni dato problema attraverso la descrizione delle proprietà dei problemi «generici» che rientrano in ciascun dominio.

Dopo che il problema è stato **classificato** in uno dei **domini**, Cynefin offre anche alcuni approcci pratici per affrontarlo.

L'obiettivo è **identificare** a quale dominio appartenga un problema e comprendere come debba essere affrontato (e eventualmente risolto).



I Cinque Regni della Complessità



In esso categorizziamo i problemi che possono essere descritti come «noti».

Per la risoluzione di tali problemi risultano disponibili le migliori pratiche e un insieme stabilito di regole.

Nel dominio semplice vi sono relazioni causa ed effetto che sono prevedibili e ripetibili (se fai X, aspettati Y).

La sequenza di azioni per questo dominio è «**percepire-categorizzare-rispondere**».

- Stabilire fatti (percepire)
- Identificare processi e regole (categorizzare)
- Eseguire i processi (risposta).

In questo dominio bisogna evitare la tendenza delle persone a **classificare erroneamente i problemi come semplici**; ovvero:

- **Semplificazione eccessiva (Oversemplification)**: ciò è correlato ad alcuni dei pregiudizi cognitivi descritti nella sezione seguente.
- **Pensiero intrappolato (Entrained thinking)**: quando le persone usano ciecamente le abilità e le esperienze che hanno ottenuto in passato e quindi diventano cieche a nuovi modi di pensare.
- **Compiacimento (Complacency)**: quando le cose vanno bene, le persone tendono a rilassarsi e sopravvalutare la loro capacità di reagire al mondo che cambia. Il pericolo di questo caso è che quando un problema è classificato come semplice, può **rapidamente degenerare nel dominio caotico** a causa della mancata valutazione adeguata dei rischi. **Si noti la scorciatoia dal dominio semplice a quello caotico nella parte inferiore del diagramma, che spesso non viene rilevata da coloro che studiano il framework.**

Il Dominio «Complicated»

In questo dominio **categorizziamo** i problemi che richiedono **esperienza e abilità** per trovare la relazione tra causa ed effetto, poiché **non esiste un'unica risposta come soluzione**.

Vi sono ancora le relazioni **causa-effetto** ma non sono evidenti e richiedono competenze per analizzarle tant'è che i problemi sono definiti «incognite note».

È necessario eseguire un'analisi adeguata per identificare quali «**best practices**» applicare; quando viene eseguita un'analisi approfondita, il «**rischio di fallimento**» della implementazione è basso.

In questo caso, ha senso **applicare modelli DDD sia per la progettazione strategica e tattica, sia per l'implementazione** e ha senso assegnare a persone qualificate il compito di **progettare in anticipo** e poi eseguire l'implementazione.

La sequenza di azioni in questo dominio è «percepire-analizzare-rispondere».

Rispetto al dominio «semplice» l'azione «**analizzare**» sostituisce «**categorizzare**» perché in questo dominio non esiste una chiara categorizzazione dei fatti.

La «categorizzazione» può essere eseguita anche qui, ma è necessario effettuare più scelte e analizzare anche le **conseguenze.**

È qui che è necessaria l'esperienza pregressa.

I problemi di ingegneria risiedono tipicamente in questa categoria, dove un problema chiaramente compreso richiede una soluzione tecnica sofisticata.

Il Dominio «Complex»

In questo dominio categorizziamo i problemi che **nessuno ha mai risolto prima** ed è impossibile fare una stima (anche solo approssimativa).

Lavorare in questo dominio ha cause ed effetti che sono imprevedibili e che appaiono ovvi solo a posteriori.

Causa ed effetto **possono essere dedotti solo in retrospettiva** per cui non vi sono risposte giuste ma «incognite sconosciute» e non vi sono pratiche su cui fare affidamento e neanche l'esperienza pregressa può aiutare.

Questo è il dominio dell'INNOVAZIONE.

Il Dominio «Complex»

L'approccio è quello di «**sondare-percepire-rispondere**», cioè si dovrà sperimentare e testare per capire lo scopo dettagliato del lavoro.

Il corso dell'azione è **guidato da esperimenti e prototipi.**

Non ha molto senso creare un grande progetto in anticipo poiché non abbiamo idea di come funzionerà e di come il mondo reagirà a ciò che stiamo facendo.

Il lavoro qui deve essere svolto in piccole iterazioni con feedback continuo e intenso.

Tecniche avanzate di modellazione e implementazione sufficientemente snelle per rispondere rapidamente ai cambiamenti si adattano perfettamente a questo contesto.

Nota: Questo è il dominio di applicazione principale del modello DDD.

Il Dominio «Chaotic»

È qui che «brucia il fuoco infernale» e «la Terra gira più velocemente di quanto dovrebbe».

Nei sistemi caotici **non è possibile stabilire relazioni di causa ed effetto** e sono strettamente legati a problemi di innovazione.

Trovarsi in questo ambito significa chiaramente «**una notevole propensione al rischio**».

Le azioni del dominio sono «**agire-percepire-rispondere**».

Nota: Probabilmente non è l'ambito migliore per il DDD (sia in termini di tempo che di eventuale budget a disposizione).

C'è un altro modo in cui un progetto potrebbe diventare **caotico**.

In genere, i confini tra i domini possono essere oltrepassati (ad esempio quando, una volta eseguita l'analisi, il lavoro passa da complicato a semplice o da complicato a complesso).

Ma c'è un confine che è diverso dagli altri: quello tra semplice e caotico è spesso rappresentato come un dirupo.

Se la diagnosi nella fase del disordine è sbagliata e il lavoro viene considerato semplice quando non lo è, oppure il lavoro diventa più complicato e il team di gestione non risponde, il progetto potrebbe entrare nella «zona compiacente».

La governance del progetto si rivela totalmente inadeguata e precipita giù dal dirupo nel caos.

Il Dominio «Disorder»

Il **disordine** è il quinto e ultimo dominio, posto proprio nel mezzo degli altri domini (La terra di Mordor ☺)

La ragione di ciò è che quando ci si trova in questa fase, non è chiaro quale contesto di complessità si applichi alla situazione.

L'unico modo per uscire dal disordine è **provare a spezzare il problema in pezzi più piccoli** che possono essere poi classificati in quei quattro contesti di complessità e quindi affrontarli di conseguenza.

Processi decisionali e pregiudizi

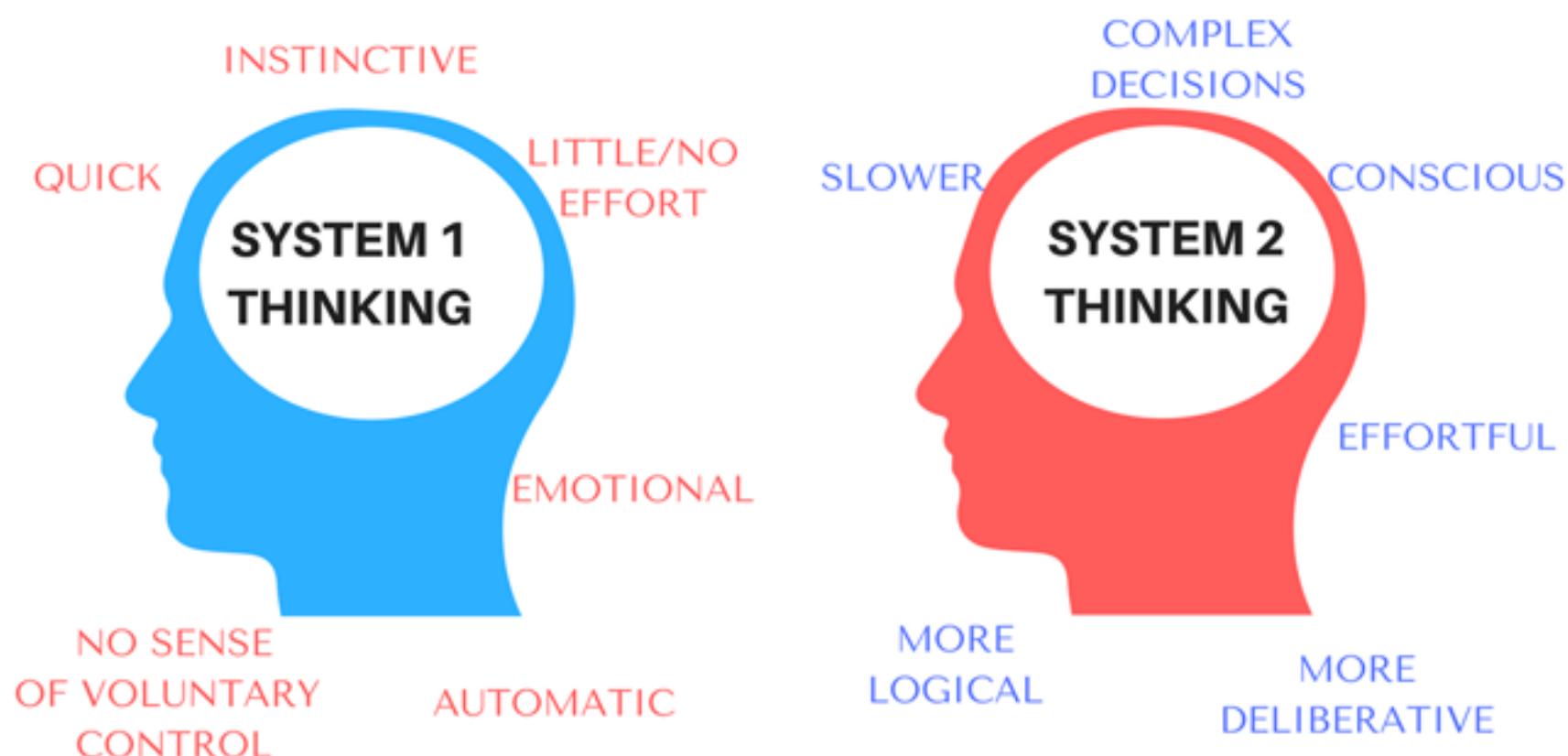
I «**processi mentali**» abbracciano la sfera dell'istinto, dell'abitudine nonché del pensiero, dell'apprendimento e del processo decisionale.

La «**teoria del doppio processo**» in psicologia suggerisce che questi tipi di attività cerebrale sono completamente diversi e divisi in due categorie:

- il «**processo implicito, automatico, inconscio**»: i processi inconsci si formano per molto tempo e sono anche molto difficili da cambiare perché cambiare un tale processo richiederebbe lo sviluppo di una nuova abitudine, e questo non è un compito facile.
- Il «**processo cosciente esplicito**»: i processi coscienti possono essere alterati attraverso il ragionamento logico e l'educazione.

Questi processi, o sistemi, coesistono felicemente in un cervello, ma sono piuttosto diversi nel modo in cui operano.

La teoria del doppio processo



SYSTEM 1

Intuition & instinct

95%

Unconscious
Fast
Associative
Automatic pilot



SYSTEM 2

Rational thinking

5%

Takes effort
Slow
Logical
Lazy
Indecisive

Cosa c'entra tutto questo con le **Architetture** ?

La risposta alla domanda ruota intorno «al come prendiamo le decisioni».

È scientificamente provato che tutti gli esseri umani hanno dei preconcetti (o, se vogliamo, pregiudizi).

A maggior ragione gli sviluppatori, abbiamo i nostri modi per risolvere i problemi

.

Analogamente, anche i clienti sono prevenuti e influenzati dal come già funziona la propria azienda (e in particolare dal come funzionano i loro sistemi software legacy).

Il modello di complessità Cynefin richiede di classificare la complessità con cui abbiamo a che fare nel nostro «**spazio del problema**» (e talvolta anche nello «**spazio della soluzione**»).

Ma per **assegnare la giusta categoria, dobbiamo prendere decisioni**, e qui spesso facciamo in modo che il nostro «**Sistema 1**» risponda e formuli ipotesi basate sui nostri pregiudizi ed esperienze del passato, piuttosto che coinvolgere il «**Sistema 2**» per iniziare a ragionare e pensare.

Pregiudizi che influenzano la progettazione

Supporto alla scelta: se si è scelto qualcosa, si tende ad essere positivi anche in presenza di difetti significativi evidenti (Es. quando scegliamo un framework).

Conferma: tendiamo ad ascoltare solo gli argomenti che sono a supporto della nostra scelta e ad ignorare quelli contro (strettamente legato al pregiudizio sul supporto della scelta)

Effetto carrozzone: quando in un Gruppo la maggioranza propende per una soluzione, ciò condiziona anche la minoranza che è in disaccordo, per cui alla fine anche questa finirà per supportarla.

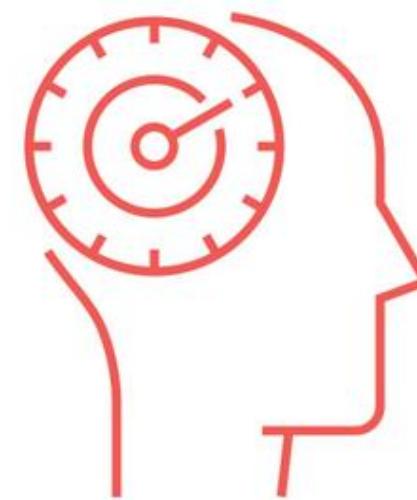
Eccessiva sicurezza: troppo spesso, le persone tendono ad essere troppo ottimiste riguardo alle proprie capacità. Questo pregiudizio potrebbe indurre a prendere rischi più significativi e decisioni sbagliate che non hanno basi oggettive ma poggiano esclusivamente su di un'opinione.

Euristica della disponibilità: le informazioni che abbiamo non è detto che siano sufficienti per conoscere appieno un problema particolare; ciò accade perché le persone tendono a basare le proprie decisioni solo sulle informazioni in mano, senza nemmeno cercare di ottenere maggiori dettagli.

La diretta conseguenza è che si tenderà a una **esemplificazione eccessiva** del problema del dominio e a una sottostima della complessità.

Questa euristica può anche **ingannarci** quando prendiamo decisioni tecnologiche e scegliamo qualcosa che ha sempre funzionato senza analizzare i requisiti operativi, che potrebbero essere molto più alti di quanto la nostra tecnologia possa gestire.

Dobbiamo ricordarci di attivare il «Sistema 2» per prendere decisioni migliori che non siano basate su emozioni e pregiudizi dettate dal «Sistema 1».



SYSTEM 1
Fast Thinking



SYSTEM 2
Slow Thinking

Essere **osessionati dalle soluzioni** invece di **comprendere il problema**, ignorare la complessità essenziale e conformarsi ai pregiudizi: tutti questi fattori ci influenzano quando sviluppiamo software.

Non appena avremo più esperienza e impareremo dai nostri errori e, preferibilmente, dagli errori degli altri, ci renderemo conto che la parte più cruciale della scrittura di software utile e di valore è la conoscenza dello spazio del problema per il quale stiamo costruendo una soluzione.



La Conoscenza
Del Dominio

La conoscenza del dominio

Nel contesto dell'IT, la conoscenza del dominio è la conoscenza dell'ambito in cui opereranno i sistemi software che si intende progettare e sviluppare.



La conoscenza del dominio

Non tutta la conoscenza è ugualmente utile quando si costruisce un sistema software e, inoltre, la **conoscenza di un dominio** potrebbe essere molto diversa da quella necessaria per un altro dominio.

Gli sviluppatori e gli analisti hanno solitamente a che fare con domini che non sono quelli in cui abitualmente operano e per i quali ottenere la conoscenza non è un compito facile.



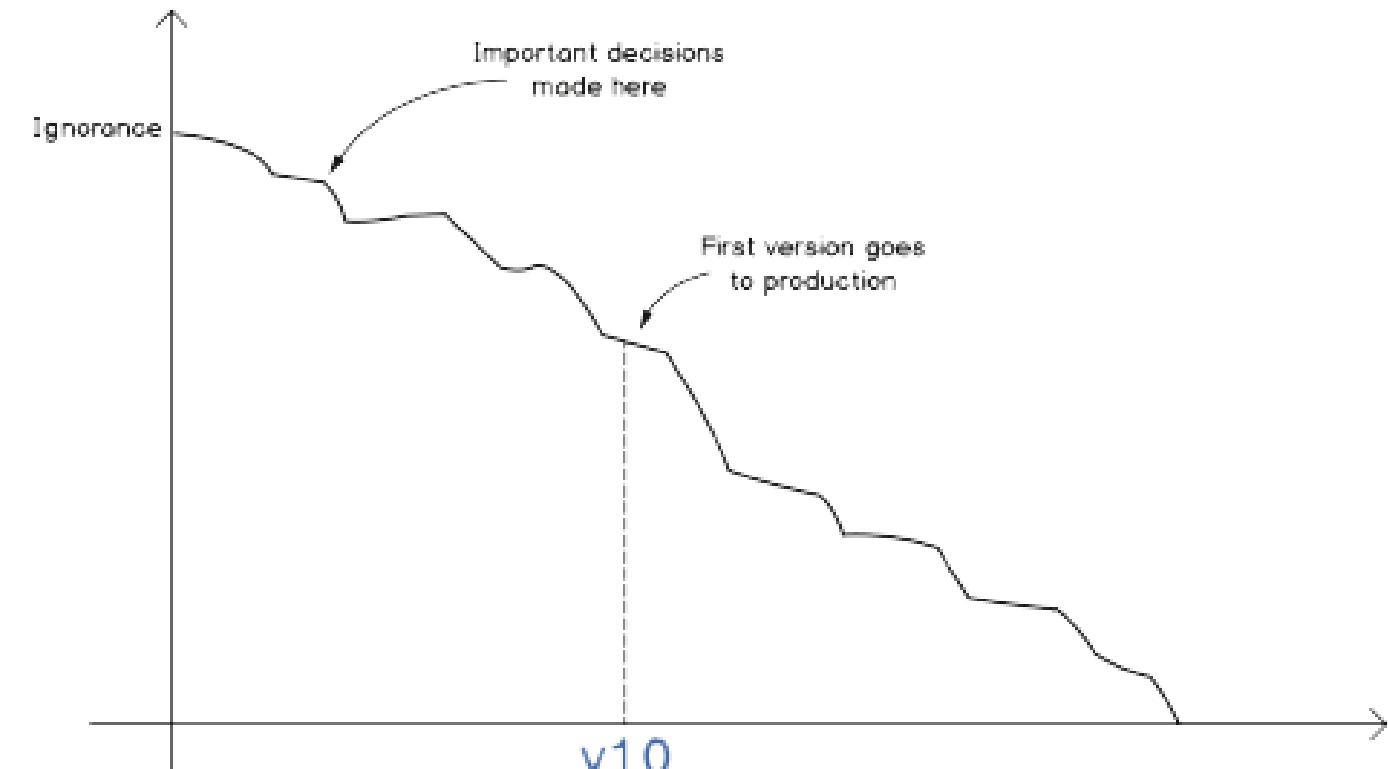
- Parlare con quante più persone possibili che abbiamo la conoscenza del dominio
- Essere buoni **osservatori**: andare sul campo, vedere gli attori lavorare e le attività svolte e poi effettuare le interviste/riunioni
- Usare **metodologie** per cogliere le azioni e rappresentarle graficamente (pittogrammi o altro); spiegare tale metodologia a chi partecipa alle riunioni.
- Usare nei workshop tecniche come quelle dei **post-it** da mettere su un muro
- Consentire la **scoperta** di attività, flussi di lavoro, processi aziendali etc.. E condividerli con tutti cercando di usare (e imparare) la terminologia corretta (ovvero quella usata nel dominio oggetto di studio)
- Etc. Etc.

Riduzione dell'ignoranza

L'obiettivo è quello di vedere lo sviluppo del software come «acquisizione di conoscenza e riduzione dell'ignoranza».

Aumentare la conoscenza del dominio e diminuire l'ignoranza sono due chiavi per creare software che offra valore.

L'ignoranza viene suddivisa in cinque livelli.



I livelli dell'ignoranza – Livello 0

Il livello di ignoranza **zero**, che gli autori chiamano «**mancanza di ignoranza**», è il più basso.

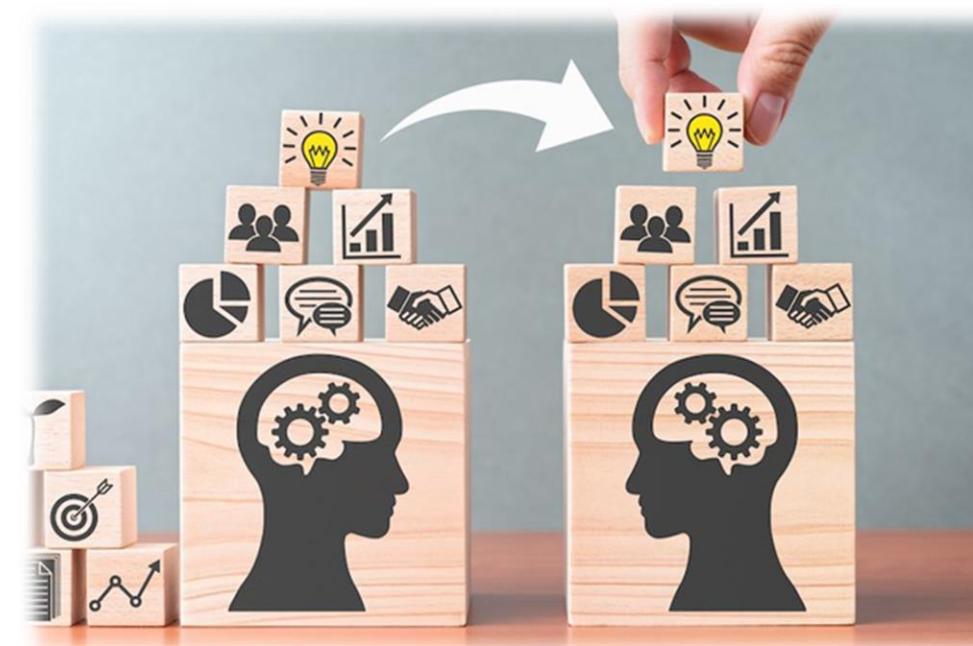
A questo livello si ha la maggior parte della conoscenza, si sa cosa fare e come farlo.



Il primo livello è la «mancanza di conoscenza».

È quando non si conosce un qualcosa, ce ne si rende conto e si accetta come fatto.

A questo livello l'obiettivo è ottenere più conoscenza e ridurre l'ignoranza al livello zero con la consapevolezza che si posseggono i canali per raggiungerlo.



I livelli dell'ignoranza – Livello 2

Il **secondo** livello chiamato anche «**mancanza di consapevolezza**», è quando «**non sai di non sapere**».

Più comunemente, ciò si verifica quando si ottiene una specifica che descrive una soluzione senza definire quale problema questa soluzione sta cercando di risolvere.

Questo livello può essere osservato anche quando le persone fingono di avere competenze che non possiedono e allo stesso tempo le ignorano.

Queste persone potrebbero essere prive di conoscenze sia commerciali che tecniche.

Molte decisioni sbagliate vengono prese a questo livello di ignoranza.

"Tutti coloro che sono incapaci di imparare si sono messi ad insegnare"
Oscar Wilde



Il **terzo** livello è la «**mancanza di processo**»; a questo livello, non si sa nemmeno come scoprire la propria mancanza di consapevolezza e letteralmente, non si ha un modo per comprendere che «non sai di non sapere».

È difficile fare qualsiasi cosa a questo livello poiché apparentemente non c'è modo di accedere agli utenti finali, nemmeno per chiedere se si capisce o meno il loro problema, al fine di scendere al livello due.

In sostanza, con la mancanza di processo, è quasi impossibile scoprire se il problema che stai cercando di risolvere esiste. La costruzione di un sistema potrebbe essere l'unica scelta in questo caso, poiché sarà l'unico modo per ottenere **feedback**.



Il quarto e ultimo livello di ignoranza è la «meta-ignoranza».

È quando non si conoscono i cinque gradi dell'ignoranza.

L'unico modo per diminuire l'ignoranza è «aumentare la comprensione» del dominio.

Un alto livello di ignoranza, consci o subconsci, porta ad una mancanza di conoscenza e ad un'errata interpretazione del problema, e quindi **aumenta la possibilità di costruire la soluzione sbagliata.**



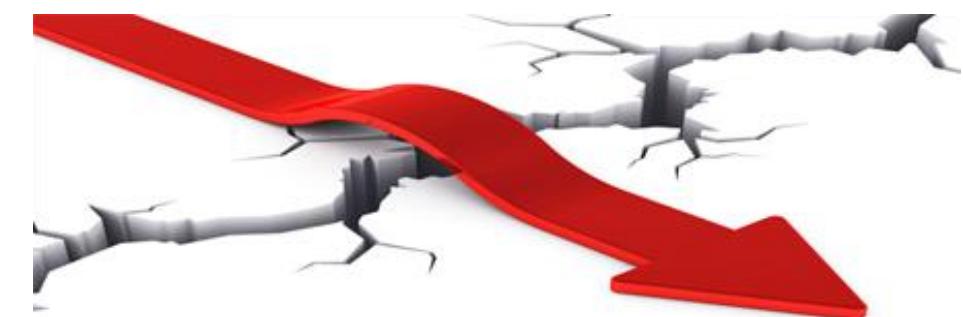
L'esperienza ci dice che il solo partire con un progetto ci porta ad essere sul secondo livello di ignoranza; ciò comporta fin da subito di essere consapevoli che:

- Durante il Progetto accadranno eventi imprevedibili
- Essendo imprevedibili significa che non abbiamo conoscenza
- Ciò si tradurrà in un impatto negativo sul progetto



- **Cercare la conoscenza sin da subito.**
- **Poiché non tutta la conoscenza è ugualmente importante, dobbiamo cercare di identificare quelle aree sensibili in cui l'ignoranza possa creare i maggiori ostacoli.**
- **Aumentando i livelli di conoscenza in queste aree, consentiamo il progresso.**
- **Allo stesso tempo, dobbiamo tenere d'occhio le nuove aree problematiche e risolverle**

Questo processo è continuo e iterativo.



Enterprise Application

Ma cosa si intende con «Enterprise Application» ?

Non possiamo dare una definizione precisa, ma solo qualche indicazione sul suo significato

Esistono diversi tipi di software ognuno dei quali presenta le proprie sfide e complessità.



Le applicazioni aziendali («Enterprise Application») hanno spesso a che fare con:

- dati **complessi** (a volte di grandi dimensioni)
- numerose «**business rules**» (molte delle quali a volte neanche tanto chiare o «**logiche**»)

Sebbene vi siano modelli e tecniche di programmazione rilevanti per tutti i tipi di software, molti lo sono solo per alcuni di essi.

Nella maggior parte dei casi, le applicazioni prevedono la «**persistenza dei dati**».

I dati sono persistenti perché devono essere disponibili tra più istanze del programma (a volte devono persistere per diversi anni).

I sistemi moderni utilizzano oramai sia database relazionali che database NoSql.

La **progettazione e l'alimentazione** dei database si è trasformata in una propria professione.



L'accesso ai dati può avvenire in modo «contemporaneo» da parte di molti utenti; per alcuni sistemi si potrebbe trattare di decine di utenti, per altri, come per i sistemi software web-based, potrebbero essere migliaia se non centinaia di migliaia.

Con così tanti utenti, vi sono precise problematiche da affrontare nel garantire che tutti possano accedere correttamente al sistema.

Al di là della numerosità, bisogna comunque assicurare e garantire che l'accesso contemporaneo anche di soli due utenti non comporti errori o comportamenti inaspettati da parte del sistema software.



Tanti dati, di solito, presuppongono anche tante schermate a livello di **interfaccia utente** per gestirli.

Non è insolito avere centinaia di «schermate» distinte (a volte addirittura sugli stessi dati).

Gli utenti stessi possono essere di diversi tipi pertanto, i dati, devono essere presentati in molti modi diversi per gli scopi più disparati.



«Le applicazioni aziendali raramente vivono su un'isola».

Di solito devono integrarsi con altre applicazioni «sparse» nell'ecosistema software dell'azienda.

Spesso capita che nel disegnare le architetture ci si concentra sui casi d'uso che stressano e interessano l'interazione con l'utente e si dimenticano aspetti come le elaborazioni batch, integrazioni dirette con altri sistemi etc.

Gli ambienti che abitualmente ritroviamo in ecosistemi software medio grandi sono caratterizzati da sistemi costruiti in tempi diversi con tecniche differenti (molte volte obsolete) per i quali i meccanismi di «**collaborazione**» saranno differenti.

A volte le aziende intraprendono progetti per integrare diversi sistemi cercando di utilizzare una comune tecnologia di comunicazione.

Gli scenari peggiorano ulteriormente quando si cerca di integrare anche i propri partner commerciali.

Anche se un'azienda unifica la tecnologia per l'integrazione, incontra problemi con le «differenze nei processi aziendali» e «dissonanze concettuali con i dati» (**eterogeneità dei dati**) che implicano trasformazioni ed elaborazioni aggiuntive.

Come diretta conseguenza, i dati devono essere costantemente letti, modificati e scritti in tutti i tipi di formati sintattici e semantici «necessari per il dialogo» tra le varie applicazioni.

Ad aggiungere criticità c'è poi la questione di cosa rientri nel termine «**logica aziendale**» (Business Logic).

Per esperienza sappiamo che la «**Business Logic**» molte volte non ha a che fare con la «**logica**» in quanto quest'ultima è «**piegata alle esigenze dell'azienda**» ovvero ai suoi «processi funzionali».

Le regole di business ci vengono imposte e nella maggioranza dei casi non possiamo fare nulla per cambiarle.

Ciò ci costringe ad affrontare una serie casuale di condizioni «strane» che spesso interagiscono tra loro in modi «sorprendenti».



Queste regole, a volte del tutto «**illogiche**», sono quelle che rendono complesso e difficile lo sviluppare i sistemi software.

Se a ciò si aggiunge che le stesse logiche non sono immutabili nel tempo ecco che ci ritroviamo a dover «pensare» e «sviluppare» i nostri software in modo tale che siano «**predisposti al cambiamento**» in quanto devono adeguarsi a scenari e contesti che possono variare nel tempo.

Tradotto, gli strati software che implementano la «business logic» devono essere quanto più efficaci ed efficienti possibile, perché l'unica cosa certa è che la logica cambierà nel tempo.

Per alcuni il termine «Enterprise Application» implica un sistema software di grandi dimensioni ma ciò non è del tutto vero.

Le tendenze di oggi vergono verso architetture composte da una miriade di piccole applicazioni che «collaborano» tra di loro ove se un piccolo sistema si guasta non inficia il funzionamento dell'intero «grande sistema».

Ciò porta ovvi vantaggi in un'ottica di «continuous integration» e «continuous delivery».

Tipi di Enterprise Architecture

Quando discutiamo sul come progettare le applicazioni e di quali schemi utilizzare, è importante rendersi conto che le applicazioni stesse sono tutte diverse e che «**problemI diversi portano a modi diversi di fare le cose**».

Gran parte della sfida nel «design» sta nel conoscere le alternative e nel giudicare i compromessi dell'utilizzo di una soluzione rispetto a un'altra.

C'è una vasta gamma di alternative tra cui scegliere a seconda degli scenari e delle esigenze, tutto sta nello scegliere la migliore possibile per il proprio contesto.

Le **metriche** utili per capire quale o quali architetture usare sono molteplici; ecco alcuni esempi:

- **Complessità della base dati**
 - **Complessità dell'interazione con l'utente**
 - **Complessità della UI**
 - **Complessità delle regole di business**
 - **Complessità del sistema hardware/software già esistente (e con cui bisognerà integrarsi)**
 - **Complessità del **Dominio** di applicazione**
- ... e tanti altri aspetti ancora.**

In questo mare di «**fattori**» di incidenza bisogna essere anche molto «**pragmatici**».

Non bisogna ricercare con esasperazione la perfezione !!!

Se, ad esempio, si aggiunge una flessibilità forzata (ed estrema) e si sbaglia, la complessità aggiunta per amore della flessibilità potrebbe ritorcersi contro perdendo di fatto quei vantaggi che, nelle buone intenzioni iniziali, si intendeva avere come requisito.

Scegliere un'architettura significa comprendere i problemi particolari del sistema e scegliere un design appropriato basato su tale comprensione.

Conoscere molti modelli aiuta a scegliere e ad avere alternative.

L'adozione di uno specifico pattern non implica che debba essere applicato pedissequamente; dobbiamo imparare anche a modificarlo per soddisfare le esigenze del dominio di applicazione.

Non si può creare software aziendale senza riflettere sul «come implementarlo**» e su quali «**modelli architettonici**» adottare.**

Scopo di questo corso è fornire quelle informazioni alla base delle «decisioni**» che bisogna prendere quando si intraprende un progetto di un sistema software.**



Le Performances

«Pensare alle prestazioni».

Molte decisioni architettoniche riguardano proprio l'aspetto delle prestazioni.

Per la maggior parte dei problemi prestazionali, l'approccio preferibile è quello di avere un sistema attivo e funzionante, «strumentarlo» e quindi utilizzare un processo di ottimizzazione disciplinato basato sulla misurazione.

Tuttavia, alcune decisioni architettoniche **influiscono sulle prestazioni** in un modo difficile da correggere con l'ottimizzazione post sviluppo.

Anche quando il problema è facile da risolvere, le persone coinvolte in un progetto dovrebbero preoccuparsene sin da subito e prendere le eventuali decisioni.

Ma è sempre difficile parlare, progettare e sviluppare in termini di performance.

Il motivo per cui è così difficile è che qualsiasi consiglio sulle **prestazioni** non dovrebbe essere considerato come un dato di fatto finché non viene misurato sulla propria configurazione.

Troppo spesso i progetti su cui sono state fatte «**troppe**» considerazioni sulle prestazioni, alla fine rivelano una realtà diversa una volta che vengono eseguite effettivamente misurazioni.

Si possono sviluppare strategie, utilizzare strumenti e metodologie per cercare di centrare l'obiettivo.



Attenzione però, perché c'è un importante corollario da tenere a mente:

«Un cambiamento significativo nella configurazione può invalidare qualsiasi dato sulle prestazioni».

Un altro problema nel parlare di prestazioni è il fatto che molti termini vengono utilizzati in modo «incoerente».

L'esempio lampante è: «scalabilità».

Fattori Performances - Response time

E' il tempo impiegato dal sistema per elaborare una richiesta dall'esterno.

Può trattarsi di un'azione dell'interfaccia utente, come la pressione di un pulsante o una chiamata API del server.



La **reattività** riguarda la velocità con cui il sistema **riconosce** una richiesta anziché elaborarla.



Responsiveness

E' un aspetto importante in molti sistemi perché gli utenti possono sentirsì frustrati se si ha una bassa reattività nonostante il tempo di risposta sia buono.

Se il sistema attende durante l'intera richiesta, la velocità di risposta e il tempo di risposta sono i medesimi.

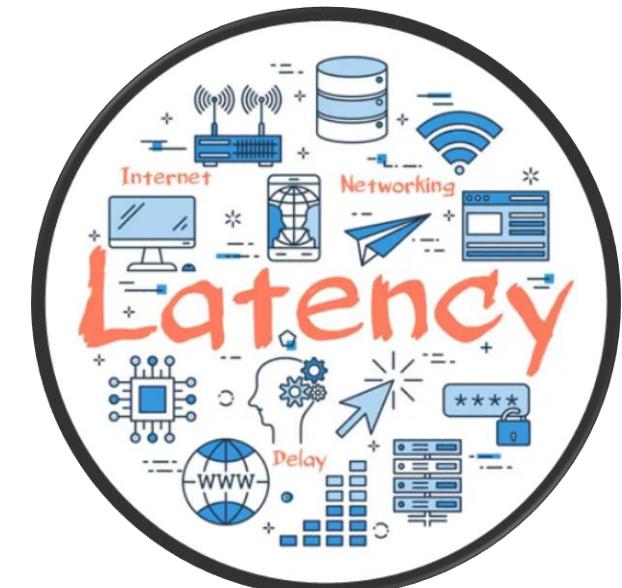
Tuttavia, se viene restituito un **feedback** immediato all'utente inerente all'aver ricevuto la richiesta prima di completarla, la capacità di risposta è migliore e la «**User Experience**» ne guadagna.

Fornire una barra di avanzamento durante una copia di file migliora la reattività dell'interfaccia utente, anche se non migliora i tempi di risposta.

La latenza è il tempo minimo richiesto per ottenere qualsiasi forma di risposta; è solitamente il più grande problema nei sistemi remoti.

In qualità di sviluppatori di applicazioni, non possiamo fare quasi nulla per migliorare la latenza se non, ad esempio, ridurre al minimo le chiamate remote o razionalizzare i payload delle chiamate (quando possibile).

In qualità di sistemisti invece possiamo dare il nostro contributo intervenendo, li dove possibile, anche a basso livello sui sistemi interessati.

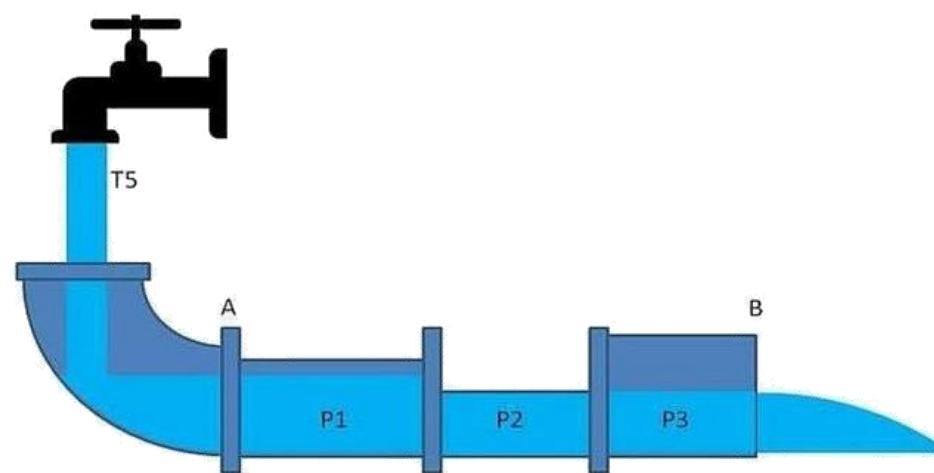


Il **throughput** è quante cose puoi fare in un dato periodo di tempo.

Esempio: se sto eseguendo la copia di un file, la velocità effettiva potrebbe essere misurata in byte al secondo.

Per le applicazioni aziendali una misura tipica sono le «**transazioni al secondo**» (**tps**), ma il problema dipende strettamente dalla complessità della transazione.

Rispetto a questo fattore, le prestazioni sono il **rendimento** o il **tempo di risposta** (a seconda di quale sia più importante per il contesto di riferimento).



Nota: dal punto di vista dell'utente, la reattività può essere più importante del tempo di risposta, quindi, per lui, migliorare la reattività a dispetto del costo del tempo di risposta o della velocità effettiva aumenterà le prestazioni (è un problema in questo caso di «percezione dell'utente»)

La determinazione del **carico (di lavoro)** consiste nell'individuare a quanto **stress** è **sottoposto un sistema**; è una metrica che potrebbe essere calcolata in base al numero di utenti attualmente connessi al sistema stesso.

Esempio: Potremmo dire, dopo una misurazione, che il tempo di risposta per alcune richieste è di 0,5 secondi con 10 utenti e 2 secondi con 20 utenti.



La **sensibilità al carico** è un'espressione di come il tempo di risposta varia con il carico.

Supponiamo che il sistema A abbia un tempo di risposta di 0,5 secondi per 10-20 utenti e il sistema B abbia un tempo di risposta di 0,2 secondi per 10 utenti che sale a 2 secondi per 20 utenti.

In questo caso il sistema A ha una sensibilità al carico inferiore rispetto al sistema B.

Potremmo anche usare il termine «**degradazione**» per indicare che il sistema B degrada più del sistema A.



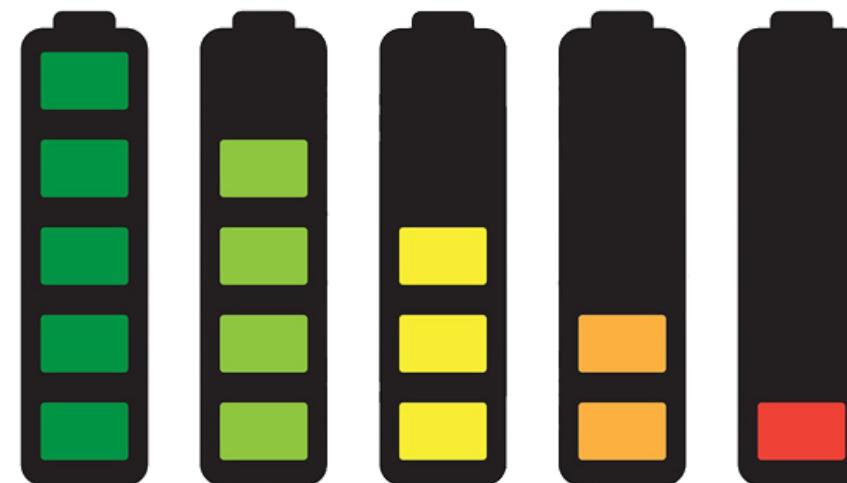
L'efficienza è la prestazione divisa per le risorse a disposizione.

Un sistema che ottiene 30 tps su due CPU è più efficiente di un sistema che ottiene 40 tps su quattro CPU identiche.



La capacità di un sistema è un'indicazione della portata o del carico massimo effettivo.

Esso potrebbe, ad esempio, corrispondere a un massimo assoluto o un punto in cui le prestazioni scendono al di sotto di una soglia accettabile.



La **scalabilità** è una misura di come l'aggiunta di risorse (solitamente hardware) influisca sulle prestazioni.

Un **sistema scalabile** è quello che consente di aggiungere hardware e ottenere un miglioramento delle prestazioni proporzionato.

Esempio: il raddoppio del numero di server a disposizione per raddoppiare il throughput.



La scalabilità di carico può esplicarsi in:

- **Scalabilità Verticale:** relativa all'aumento della capacità di elaborazione di una singola macchina
- **Scalabilità Orizzontale:** relativa all'aggiunta di altre macchine in distribuzione di carico

Il problema di fondo, sul quale ragionare è che le decisioni di progettazione non influenzano allo stesso modo tutti i fattori di prestazione fin qui analizzate.

Portiamo un esempio e supponiamo di avere due sistemi software in esecuzione ognuno su di un server.

La capacità del sistema A è di 20 tps mentre la capacità del sistema B è di 40 tps.

Tra A e B quale presenta prestazioni migliori ?

Tra A e B quale risulta essere più scalabile ?

Non possiamo rispondere alla domanda sulla scalabilità da questi dati e possiamo solo dire che B è più efficiente su un singolo server.

Se aggiungiamo un altro server, notiamo che A ora gestisce 35 tps e B gestisce 50 tps.

La capacità di B è ancora migliore, ma sembra che A possa ridimensionarsi meglio.

Se continuiamo ad aggiungere server, scopriremo che A ottiene 15 tps per server extra e B ne ottiene 10.

Dati alla mano, alla conclusione delle nostre misurazioni e osservazioni, possiamo dire che A ha una migliore scalabilità orizzontale, anche se B è più efficiente per meno di cinque server.

Quando si progettano e creano sistemi software aziendali spesso ha più senso pensare alla **scalabilità hardware** piuttosto che ragionare sulla capacità o addirittura sull'efficienza.

La scalabilità offre prestazioni migliori quando occorre e potrebbe essere anche più facile da implementare.

Spesso i progettisti fanno cose complicate che migliorano la capacità su una particolare piattaforma hardware quando in realtà potrebbe essere più economico acquistare più hardware.

Se un sistema A ha un costo maggiore di un sistema B e tale costo maggiore equivale a un paio di server, B finisce per essere più economico anche se hai bisogno solo di 40 tps.

È di moda lamentarsi di dover fare affidamento su hardware migliore per far funzionare correttamente il nostro software.

Ma acquistare hardware più recente è spesso più economico che far funzionare il software su sistemi meno potenti.

Allo stesso modo, aggiungere più server è spesso più economico che aggiungere più programmati, a condizione ovviamente che il sistema software sia scalabile.

Patterns

I **pattern** esistono da molto tempo quindi è inutile disquisire sul perché esistano e si applichino.

Tuttavia, questa è un'opportunità per fornire una visione diversa dei modelli e ciò che li rende un approccio utile alla descrizione del design.

Non esiste una definizione generalmente accettata di pattern, ma forse il miglior punto di partenza è un'osservazione di «Christopher Alexander», fonte di ispirazione per molti appassionati di pattern:

«Ogni schema descrive un problema che si verifica più e più volte nel nostro ambiente, quindi descrive il nucleo della soluzione a quel problema, in modo tale da poter utilizzare questa soluzione un milione di volte, senza mai farlo allo stesso modo due volte»

Alexander è un architetto, quindi parlava di edifici, ma la definizione funziona abbastanza bene anche per il software.

Il fulcro del pattern è una soluzione particolare, comune ed efficace nell'affrontare uno o più problemi ricorrenti.

Metaforicamente, potremmo affermare che un pattern è uno schema inteso come un «consiglio» e l'arte di creare schemi è dividere molti «consigli» in parti relativamente indipendenti in modo che si possa fare riferimento a loro e discuterli più o meno separatamente.

Una parte fondamentale dei modelli è che sono radicati nella pratica (e quindi nell'esperienza).

Possiamo trovare schemi osservando ciò che le persone fanno, osservando le cose che funzionano e poi cercando il «nucleo della soluzione».

Non è un processo facile, ma una volta individuati alcuni buoni schemi, questi diventano preziosi.

Conoscere i Pattern significa saperne abbastanza per avere un'idea di quali sono gli schemi che si ripetono, quali problemi risolvono e come li risolvono.

Non è necessario conoscere tutti i dettagli di un pattern, ma solo quanto basta in modo che, se si incontra un problema, sia possibile identificarlo e associarlo a un pattern per adottare le relative soluzioni (e nel caso approfondirne lo studio ☺).

Solo allora avremo, quindi, bisogno di comprendere veramente in profondità lo schema e quindi il relativo Pattern.

Una volta che si ha bisogno di un Pattern bisogna comprendere come applicarlo nel proprio contesto di interesse.

Una cosa fondamentale dei pattern è che «non si possono mai applicare alla soluzione alla cieca».

Fowler afferma che i pattern sono «semicotti», ciò significa che «bisogna sempre completarne la cottura nel forno» quando li applichiamo ai nostri progetti.

Ogni volta che adottiamo un pattern ci capiterà di modificarlo un «pò qui e un pò là» per adeguarlo al nostro contesto di utilizzo.

«Utilizziamo la stessa soluzione diverse volte, ma alla fine non è mai esattamente la stessa».

Ogni pattern è relativamente indipendente, ma, in generale, non sono isolati l'uno dall'altro.

Spesso uno schema «**traina**» l'altro o uno si «**verifica**» solo se ne esiste un altro.

I **confini** tra i pattern sono naturalmente sfocati.

Chi li studia e li codifica cerca di rendere ogni pattern il più autonomo possibile ma non possiamo non tenere in conto le **relazioni** che possono intercorrere tra di essi.

Come sviluppatori e/o progettisti, probabilmente scopriremo che la maggior parte di questi modelli ci è **familiare**.

I Pattern traggono spunto dall'osservazione e dall'esperienza e quindi da quello che tutti noi, quotidianamente già facciamo quando sviluppiamo sistemi software.

I pattern solitamente non sono idee originali; sono moltissime osservazioni di ciò che accade sul campo.

Non è un caso che molti «autori» di pattern affermano (correttamente) di non aver «inventato» un pattern, ma piuttosto che ne hanno «**scoperto**» uno.

Il nostro ruolo è prendere nota della soluzione comune, cercarne il nucleo e quindi annotare il modello risultante.

Per un designer esperto, il valore del modello non consiste nella «**novità**» ma nella «**standardizzazione della conoscenza e della comunicazione del pattern**».

Esempio: se tutti gli sviluppatori di un team sanno cosa sia un «**Remote Facade**» o un «**Data Transfer Object**» significa non solo parlare la stessa lingua ma anche «**sapere già cosa fare**» o eventualmente «**cosa studiare**» per approfondimenti.

Il risultato è che i modelli creano un vocabolario sul design: questo è il motivo per cui la terminologia è una questione molto importante.

Il mondo dei Pattern è variegato e la letteratura ad essi legata è altrettanto vasta; ecco alcuni esempi (sempre utili e che bisognerebbe almeno leggere):

- *A Pattern Language* (di Christopher Alexander); è un saggio di architettura e urbanistica 😊
- *Design Patterns: Elements of Reusable Object-Oriented Software*; GoF ovvero Gang of Four
- *Pattern-Oriented Software Architecture* (POSA)

I nomi dei pattern sono fondamentali, perché parte del loro scopo è creare un vocabolario che consenta ai designer di comunicare in modo più efficace ed efficiente.