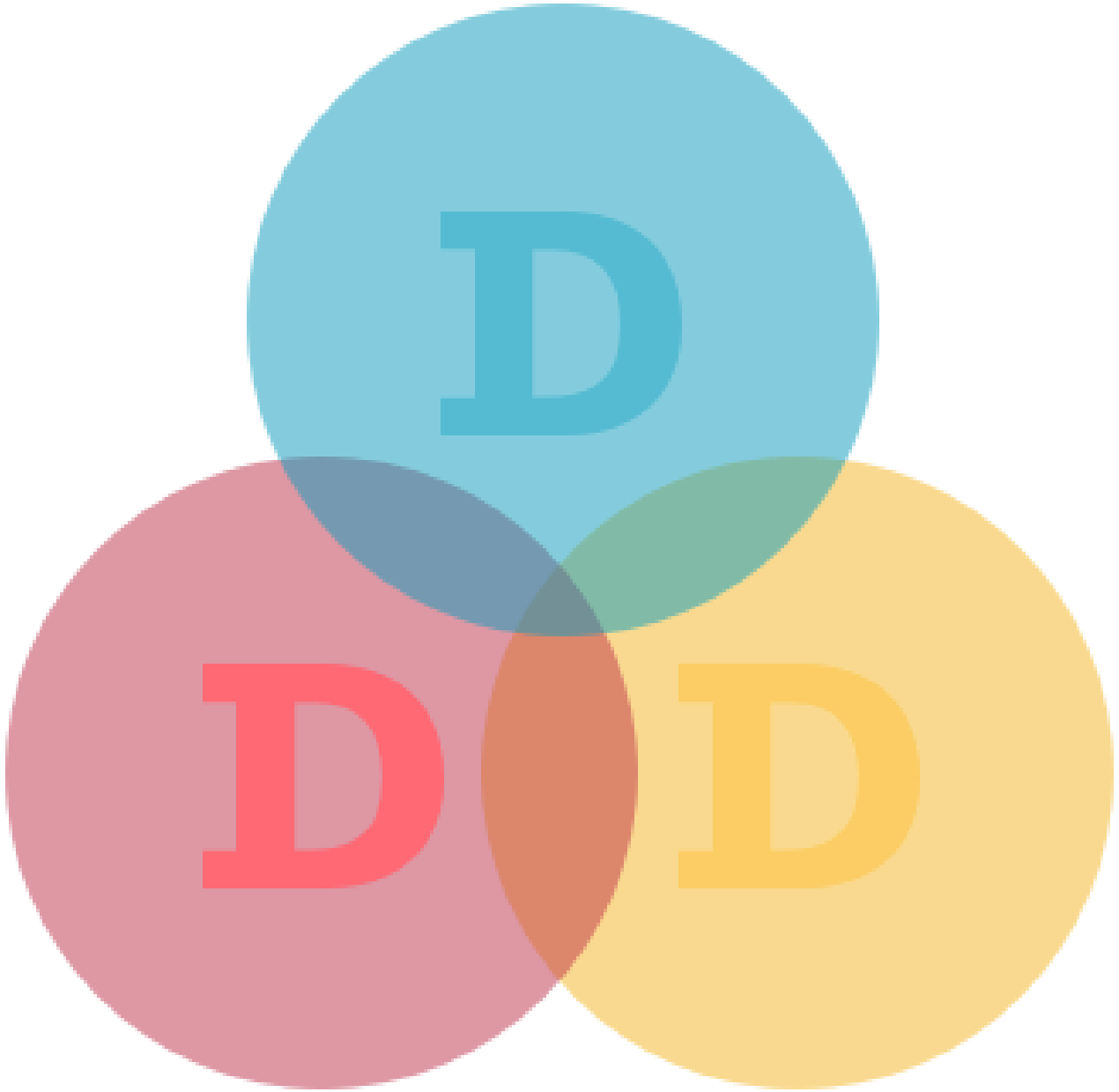
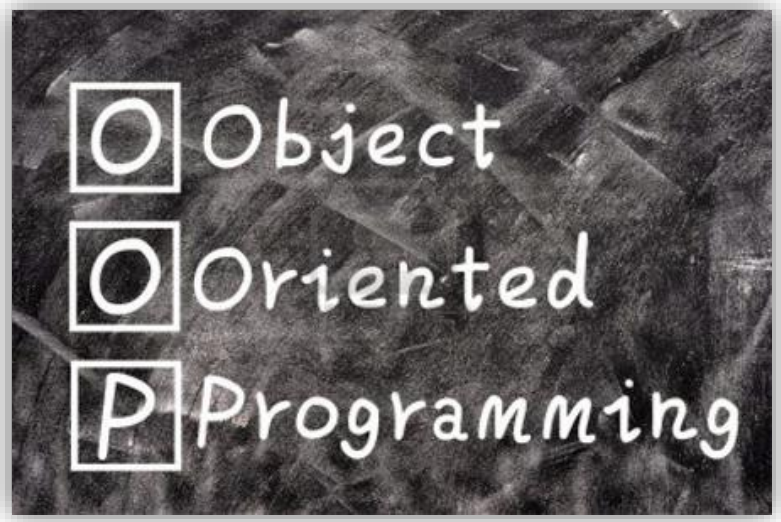

D
D
D

Introduzione

La OOP

«Il fatto che si conosca bene un linguaggio **OOP** non significa che si abbia la capacità di trasformare tale conoscenza in **sistemi orientati agli oggetti ben progettati**»



La padronanza della sintassi di un linguaggio di programmazione OOP, anche fosse perfetta, **non basta** per organizzare e strutturare sistemi e programmi che rispecchino il **paradigma** OOP.

La OOP ha una sua **complessità** che richiede sia la conoscenza di nozioni specifiche (alcune delle quali squisitamente teoriche), sia un opportuno metodo nonché disciplina.

Design a Oggetti vs Structured-design

Già dalla fine degli anni 90, quando gran parte della teoria OO è stata messa a punto, è emerso come analisi e design orientati agli oggetti fossero fondamentalmente differenti dalla tradizionale metodologia del design strutturato.

Il focus sugli oggetti ha bisogno di un diverso approccio alla decomposizione dei problemi e non deve sorprendere quindi che produca architetture software molto dissimili da quelle realizzate per mezzo dello structured-design:

- **Con l'approccio strutturale ci si concentra sulla scomposizione di algoritmi in procedure.**
- **Nell'approccio «a oggetti» ci si focalizza sull'interazione di elementi (oggetti) che comunicano (scambiano messaggi) tra loro.**

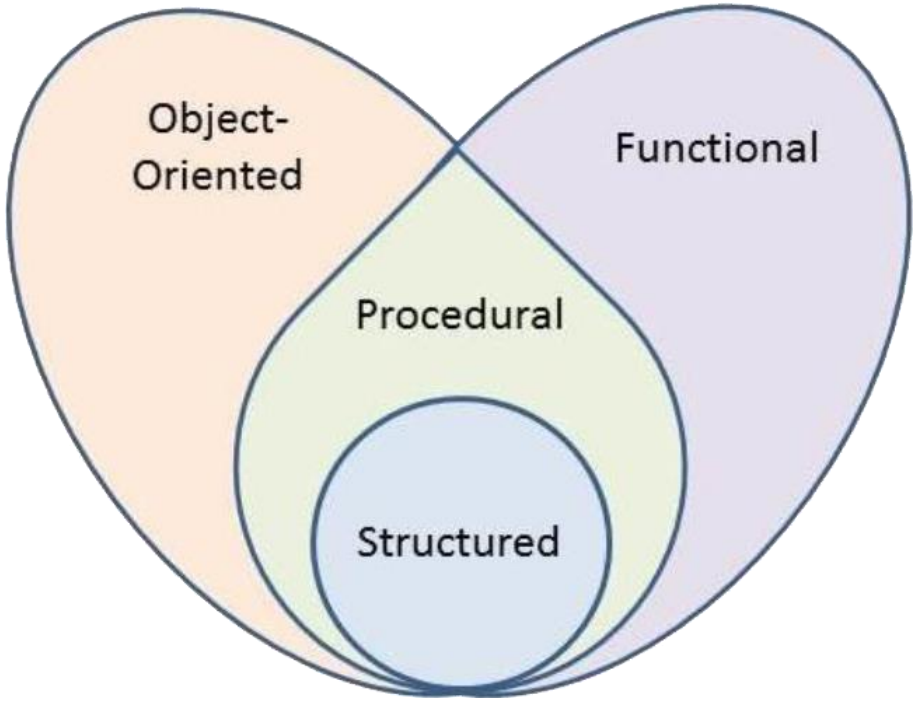
Design a Oggetti vs Structured-design

La OOP consente la descrizione di dinamiche complesse e la realizzazione di «procedure» più facilmente implementabili (e mantenibili) favorendo le interazioni tra oggetti sostanzialmente semplici.

«La complessità dei problemi non deve essere rappresentata spesso da oggetti complessi ma da interazioni complesse tra oggetti semplici.»

Il design strutturale si è sviluppato in modo da riuscire a costruire sistemi complessi utilizzando gli **algoritmi** come loro elemento.

L'object-oriented design si è evoluto in modo da dare supporto agli sviluppatori che utilizzano le **classi** e gli **oggetti** come loro elemento base nella fase di sviluppo di un progetto.

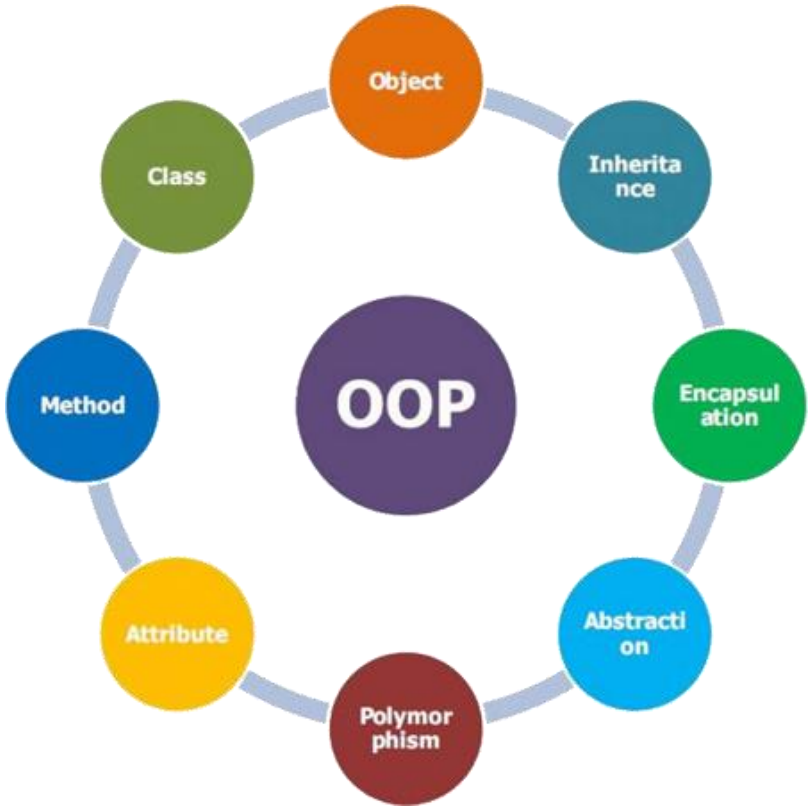


OOP, OOD & OOA

Il modello ad oggetti è di importanza capitale nella descrizione di numerose dinamiche anche non legate allo sviluppo del software e ha quindi subito influenze da diversi ambiti.

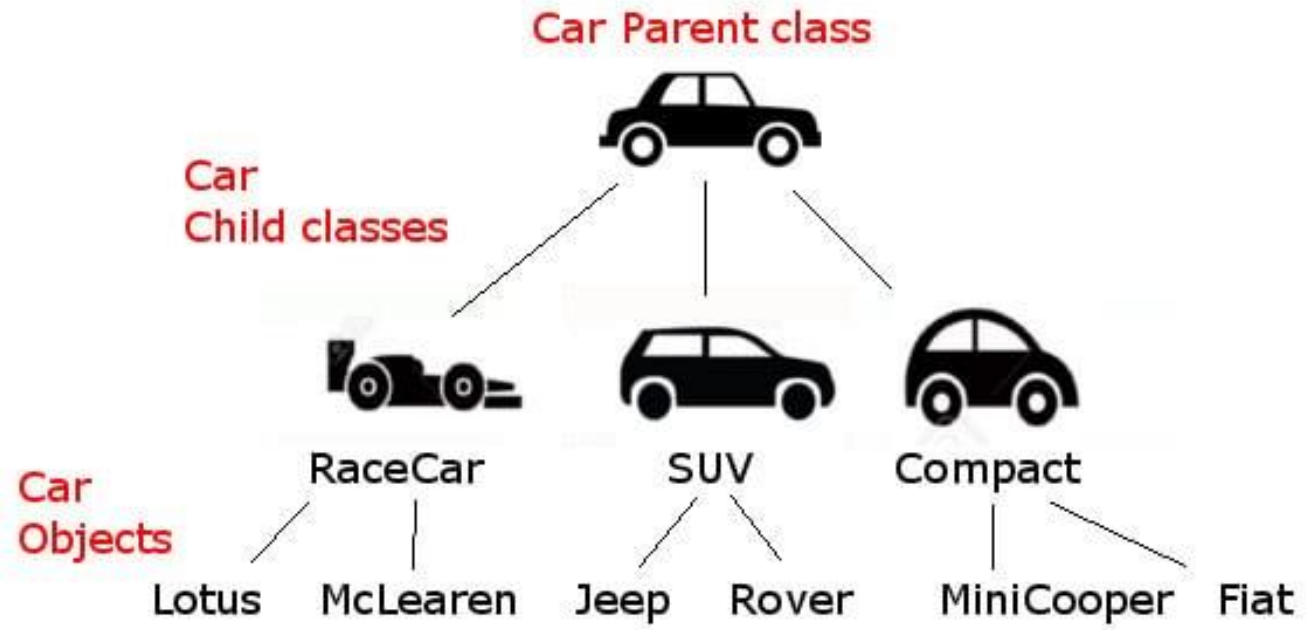
Perciò il concetto di «modello ad oggetti» finale unifica visioni provenienti da mondi diversi:

- Programmazione
- Strutturazione dei database
- Costruzione e design di interfacce
- Etc.



OOP: Object Oriented Programming

Si definisce «Object Oriented Programming» (o semplicemente OOP) un metodo di implementazione in cui i programmi sono organizzati attraverso un insieme di oggetti, ognuno dei quali è un'istanza di una classe, e queste classi sono tutte parte di una gerarchia di entità unite fra di loro da una relazione di ereditarietà.



OOP: Object Oriented Programming

Nella definizione data della OOP, vi sono tre aspetti particolarmente importanti:

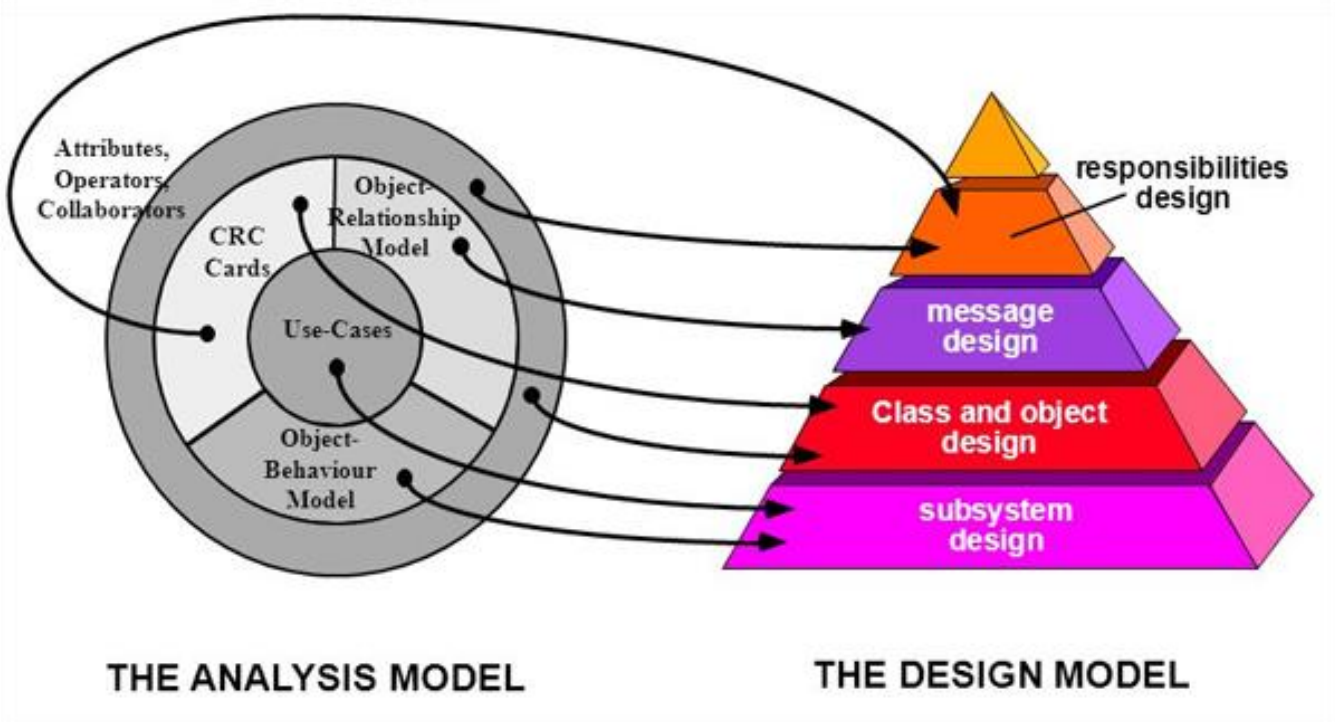
- **OOP utilizza un insieme di oggetti e non algoritmi, e gli oggetti sono la parte fondamentale della costruzione logica**
- **Ogni oggetto è istanza di una classe**
- **Ogni classe è legata alle altre attraverso una relazione detta eredità.**

Se in un programma manca anche solo una di queste caratteristiche, non lo si può definire object-oriented.

OOD: Object-Oriented Design

Possiamo definire l'object-oriented design come una metodologia di progettazione che comprende:

- **il processo di decomposizione ad oggetti**
- **una notazione per rappresentare modelli di logica e fisica dei sistemi**
- **una notazione per rappresentare aspetti statici e dinamici dei sistemaiin fase di progettazione**



OOD e UML

La **OOD** prevede l'adozione di una **notazione** atta a descrivere le interazioni tra gli oggetti e a comunicare e modellare le peculiarità del sistema in esame.

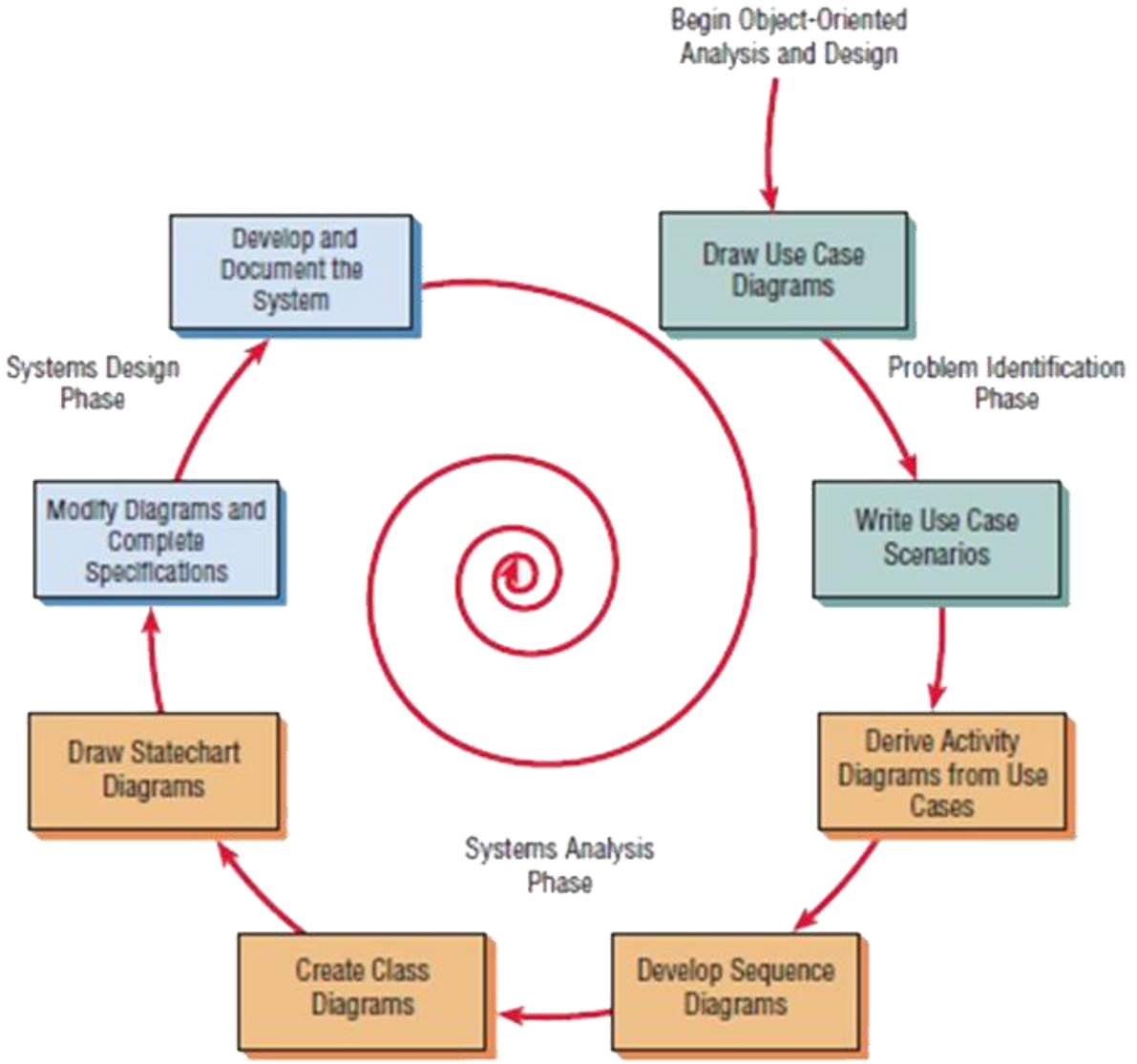
Tale notazione è ovviamente **astratta** ovvero slegata dal particolare linguaggio di programmazione che si intenderà utilizzare per l'implementazione del design.

Ad oggi la notazione universalmente riconosciuta per la modellazione OOD è l'**UML** acronimo di «**Unified Modelling Language**».

OOA: Object-Oriented Analysis

«**Object-oriented analysis**» è l'analisi di un problema e la costruzione di modelli del mondo reale con una visione orientata agli oggetti.

In altre parole è una metodologia di analisi che esamina le necessità di un problema dal punto di vista delle classi e degli oggetti.



Quale relazione lega OOA, OOD e OOP?

Fondamentalmente, i prodotti dell'analisi orientata agli oggetti servono come modelli da cui si possa avviare una progettazione orientata agli oggetti.

I prodotti di progettazione orientata agli oggetti vengono utilizzati come modelli per l'attuazione completa di un sistema utilizzando metodi di programmazione orientati agli oggetti.

Anche se in teoria questo approccio che prevede OOA seguita da OOD (di solito accorpati entrambi sotto l'acronimo OOAD) e OOP (con successive iterazioni e revisioni) sembra una colossale (e burocratica) complicazione ed un inutile allungamento dei tempi di sviluppo, l'esperienza conferma che seguendo buone pratiche OO il software prodotto risulta migliore e soprattutto mantenibile e modificabile.

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.

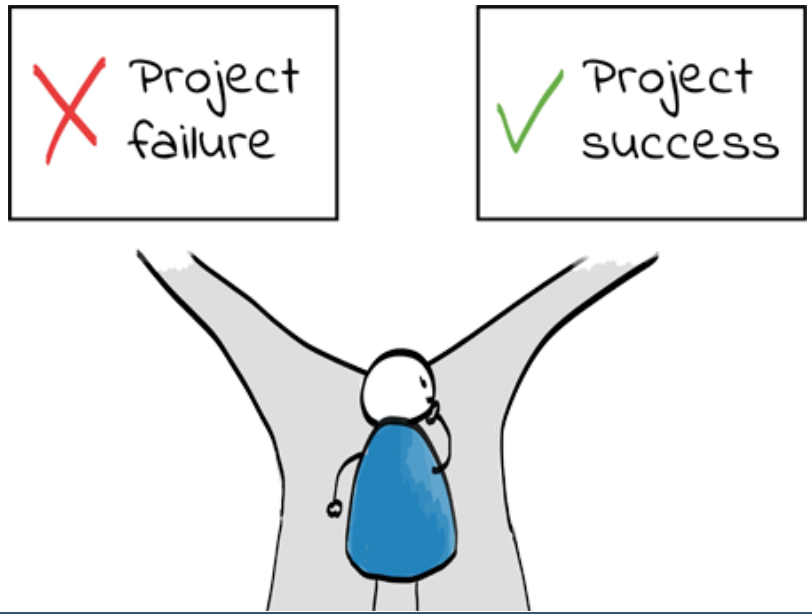
Tra questi uno dei più rilevanti è l'**approccio alla progettazione**; ciò perché ha effetti diretti sulla **complessità** stessa del software.

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.

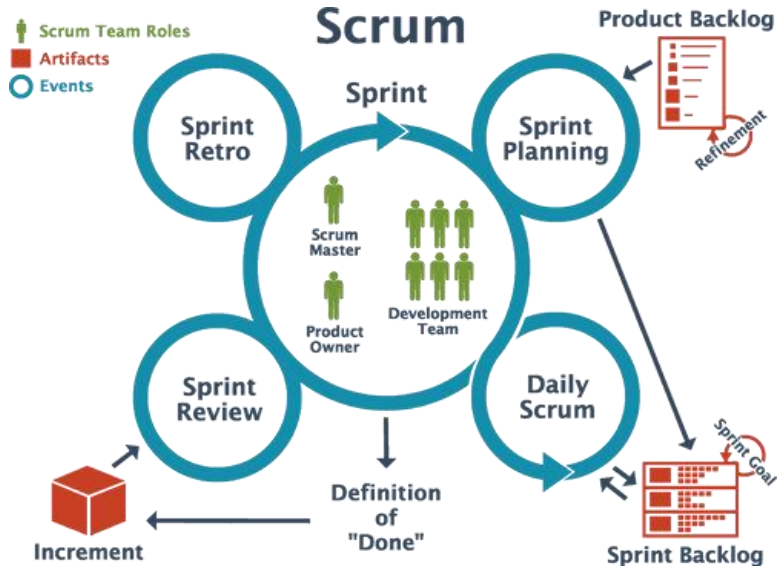


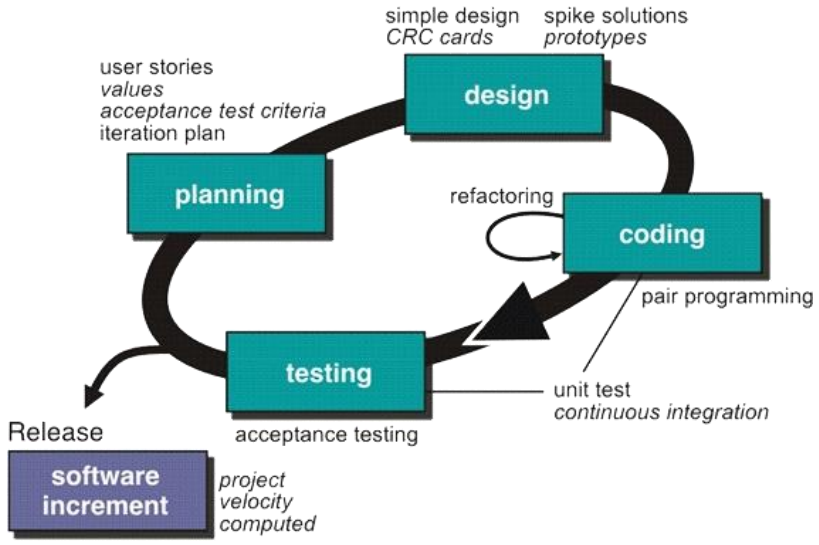
La comprensione dei problemi e le metodologie di sviluppo

Uno dei fattori critici che definiscono il successo di qualsiasi progetto IT è la comprensione 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.

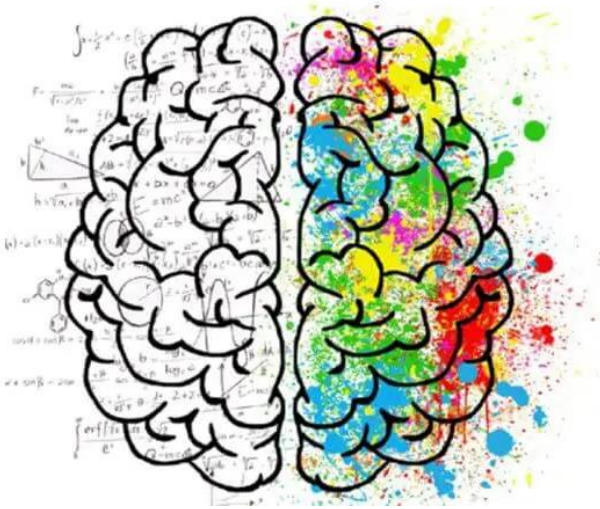
Metodologie di sviluppo software come SCRUM e XP abbracciano l'interazione con gli utenti e la comprensione dei loro problemi.





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.

Problem Space & Solution Space

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.»

Problem Space & Solution Space

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.

Problem Space & Solution Space

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**.

Trovare la soluzione al problema

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.

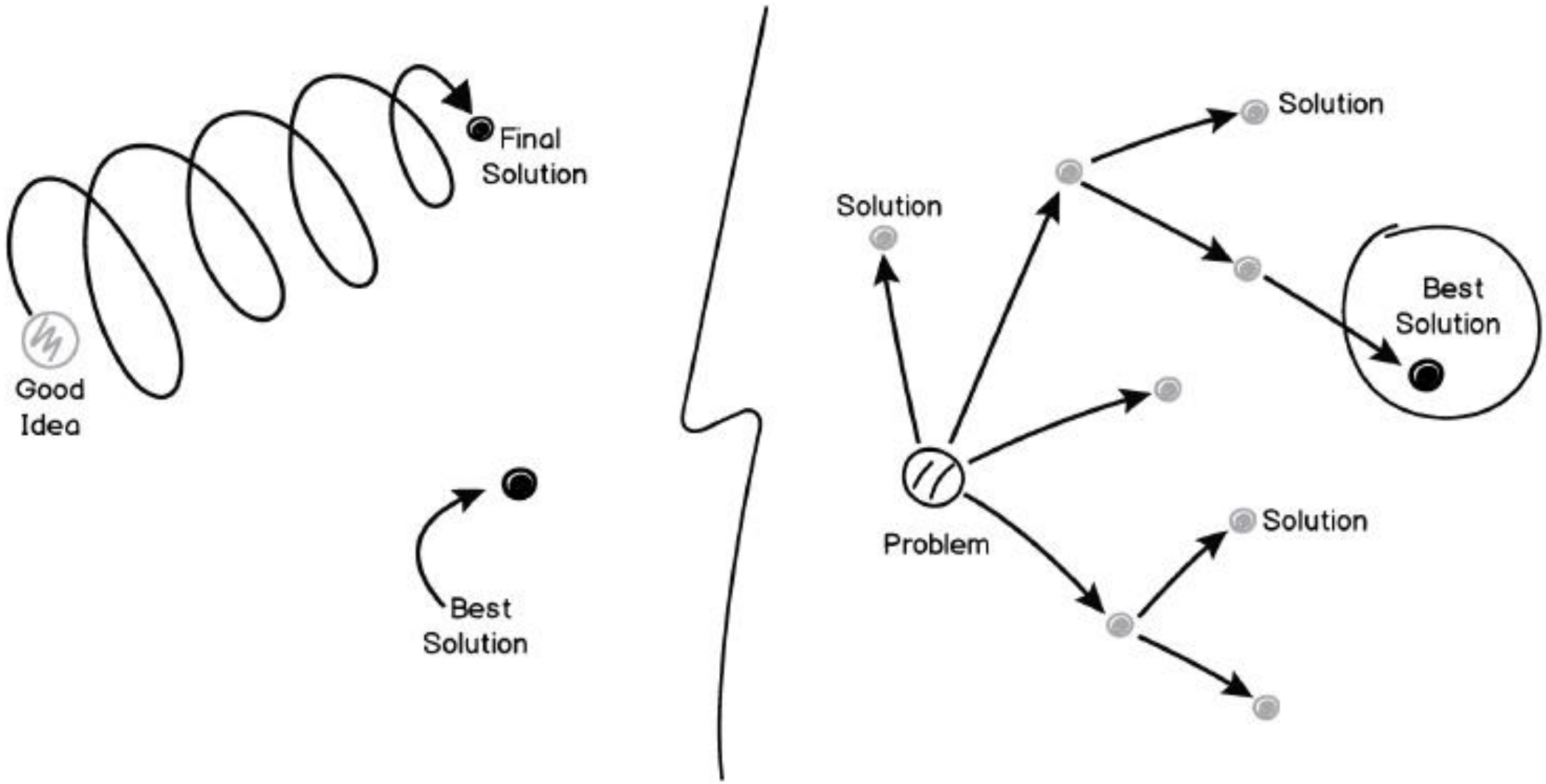
C'è il pericolo di concentrare tutta la propria attenzione su una particolare soluzione, che potrebbe non essere affatto la migliore, ma è stata la prima a venire in mente.

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

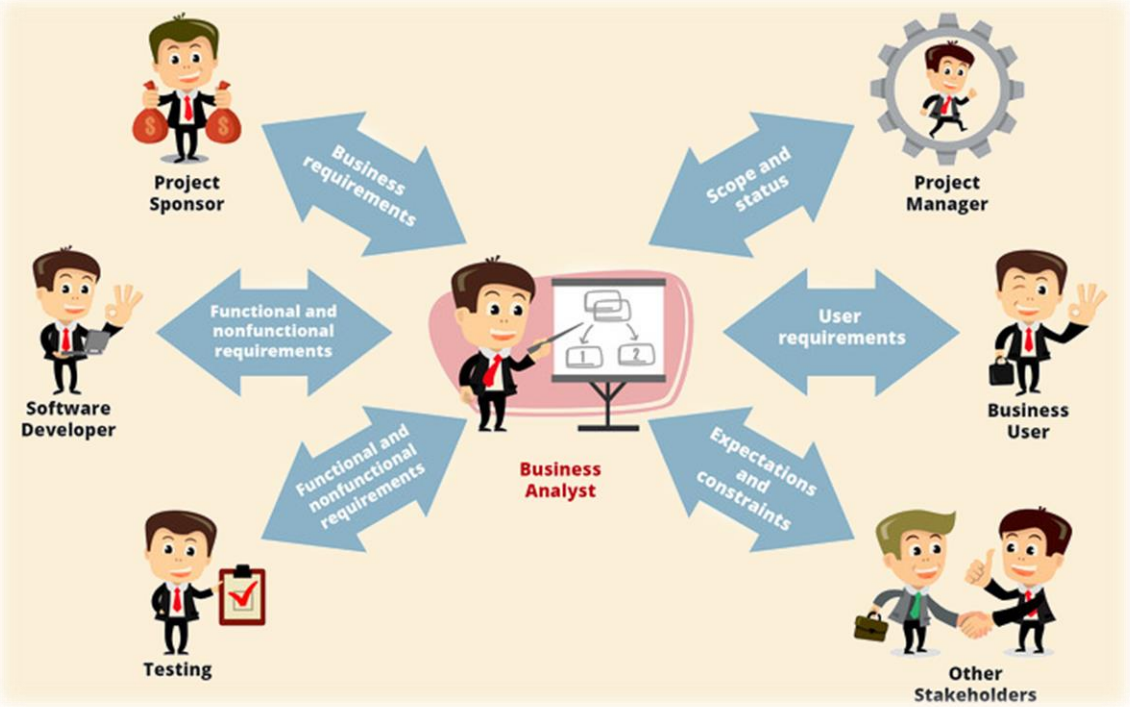


Cosa è andato storto con i requisiti

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.

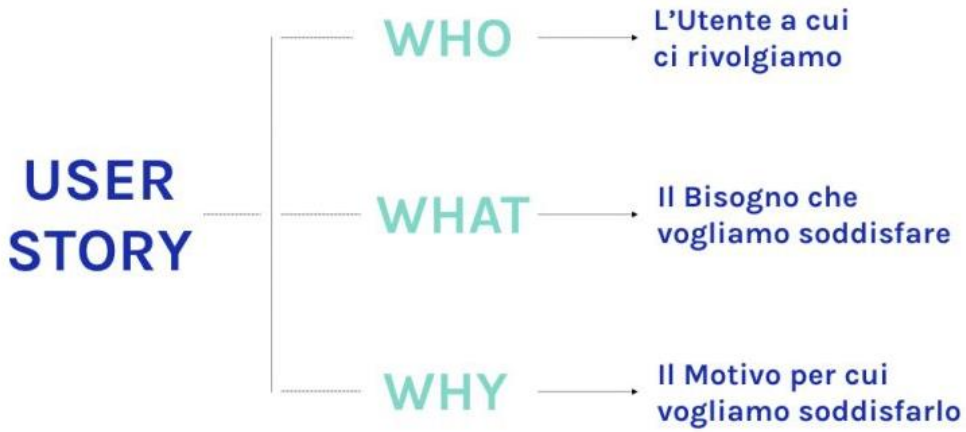
Cosa è andato storto con i requisiti

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».



Cosa è andato storto con i requisiti

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 neessario spendere sempre più tempo e denaro per scrivere requisiti di qualità superiore».

Cosa è andato storto con i requisiti

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 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.

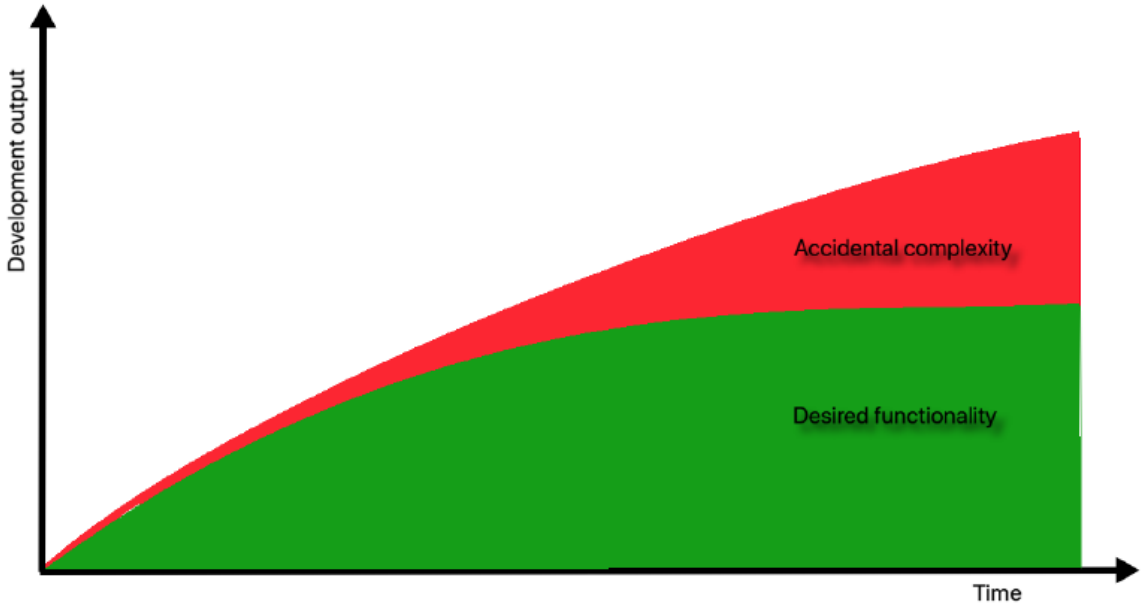
L'industria del software fatica ancora a combattere la «**complessità accidentale**».

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.



Tipo Di Complessità

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 e La Complessità

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.

Abbattere la Complessità

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.

Il Dominio Applicativo

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

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

Il **Framewrok 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.

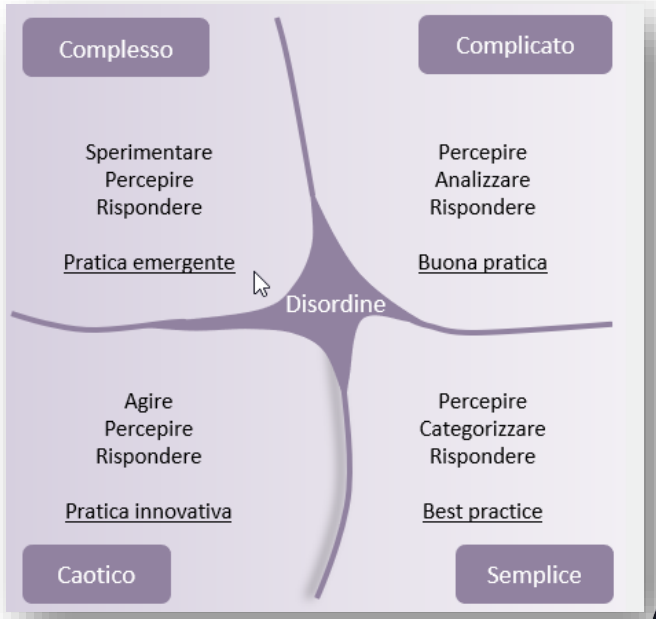
Classificazione Cynefin

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

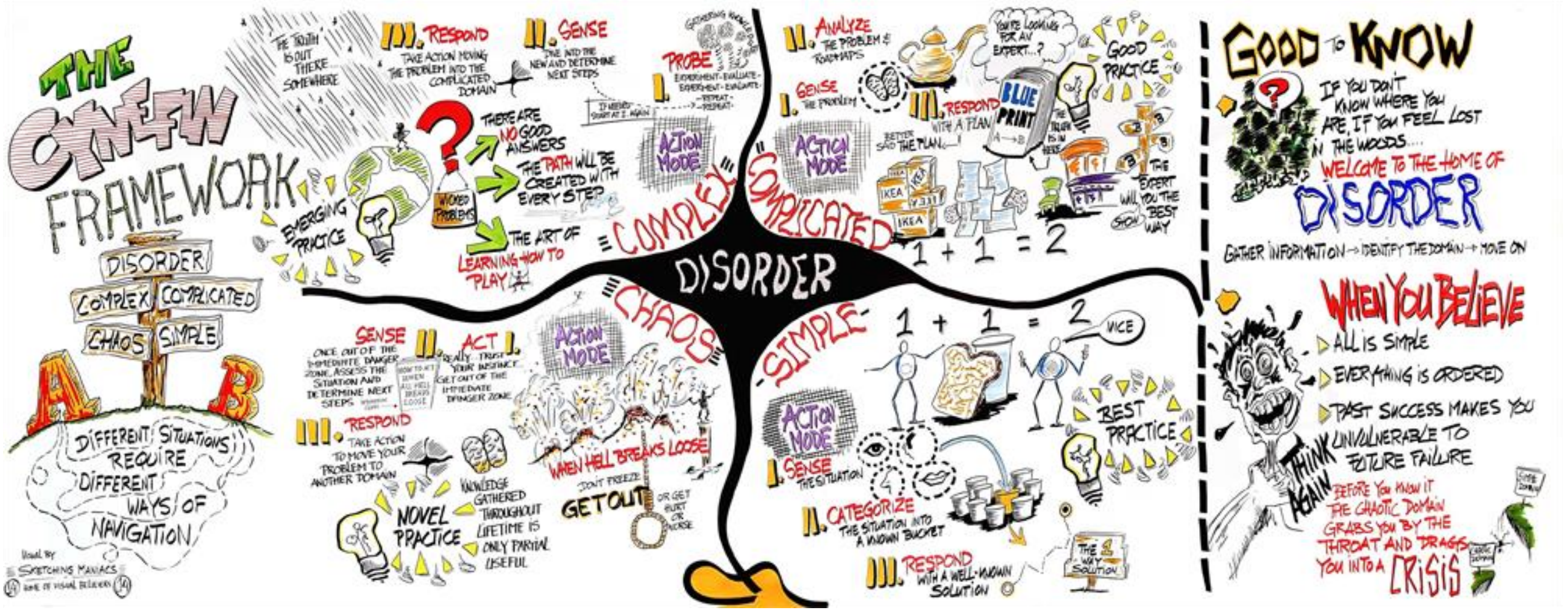
Fornisce un «senso di collocazione» per ogni dato problema attraverso la descrizione delle proprietà dei problemi «generici» che rientrano in ciascun dominio.

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



Il Dominio «Simple»

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).**

Il Dominio «Simple»

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

- **Semplificazione eccessiva** (Oversemplicfication): 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 pratices**» 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.

Il Dominio «Complicated»

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.

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 ambio significa chiaramente «una notevole propensione al rischio».

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

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

Il Dominio «Chaotic»

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»

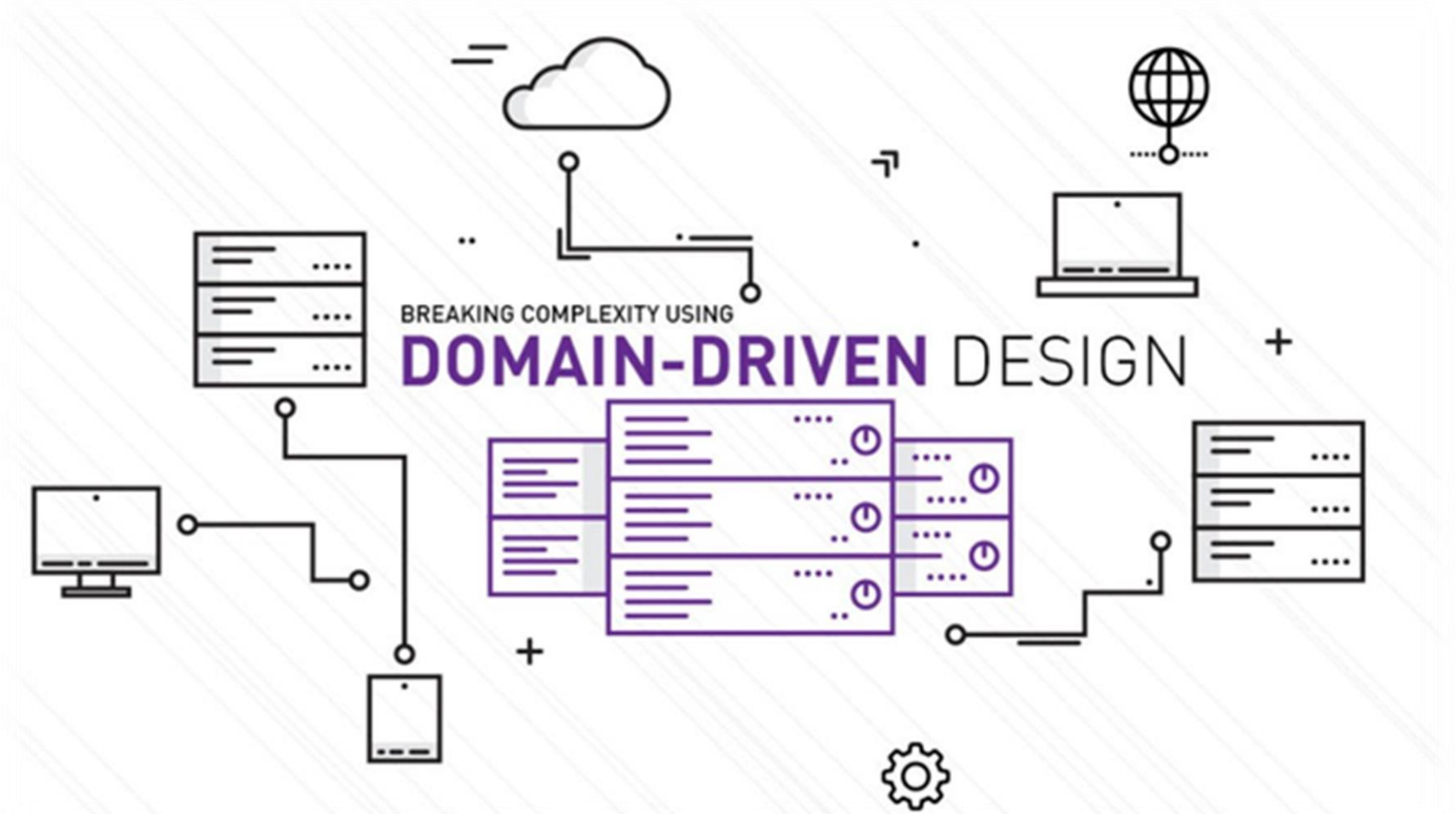
Il disordine è il quinto e ultimo dominio, posto proprio nel mezzo degli altri domini (La terra di Mordor ☺)

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.

I Domini e il DDD

Ai fini di questo corso il risultato più importante di quanto fino a qui evidenziato è che il «DDD può essere applicato quasi ovunque ma è praticamente inutile in domini semplici e caotici».



Processi decisionali e pregiudizi.

I processi mentali

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

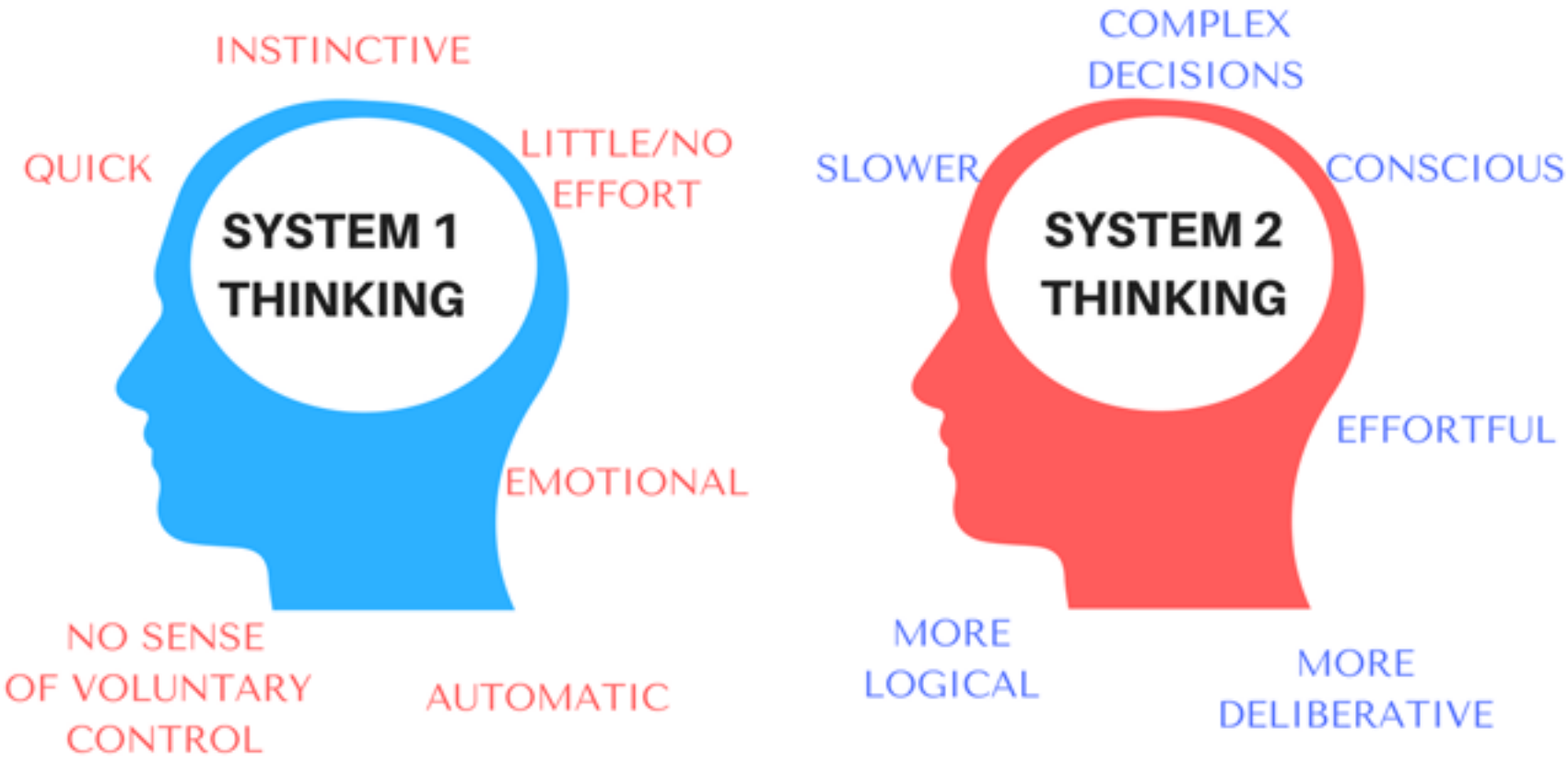
La «**teoria del doppio processo**» in psicologia suggerisce che questi tipi di attività cerebrale sono completamente diversi e divisi in due categorie:

- 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.

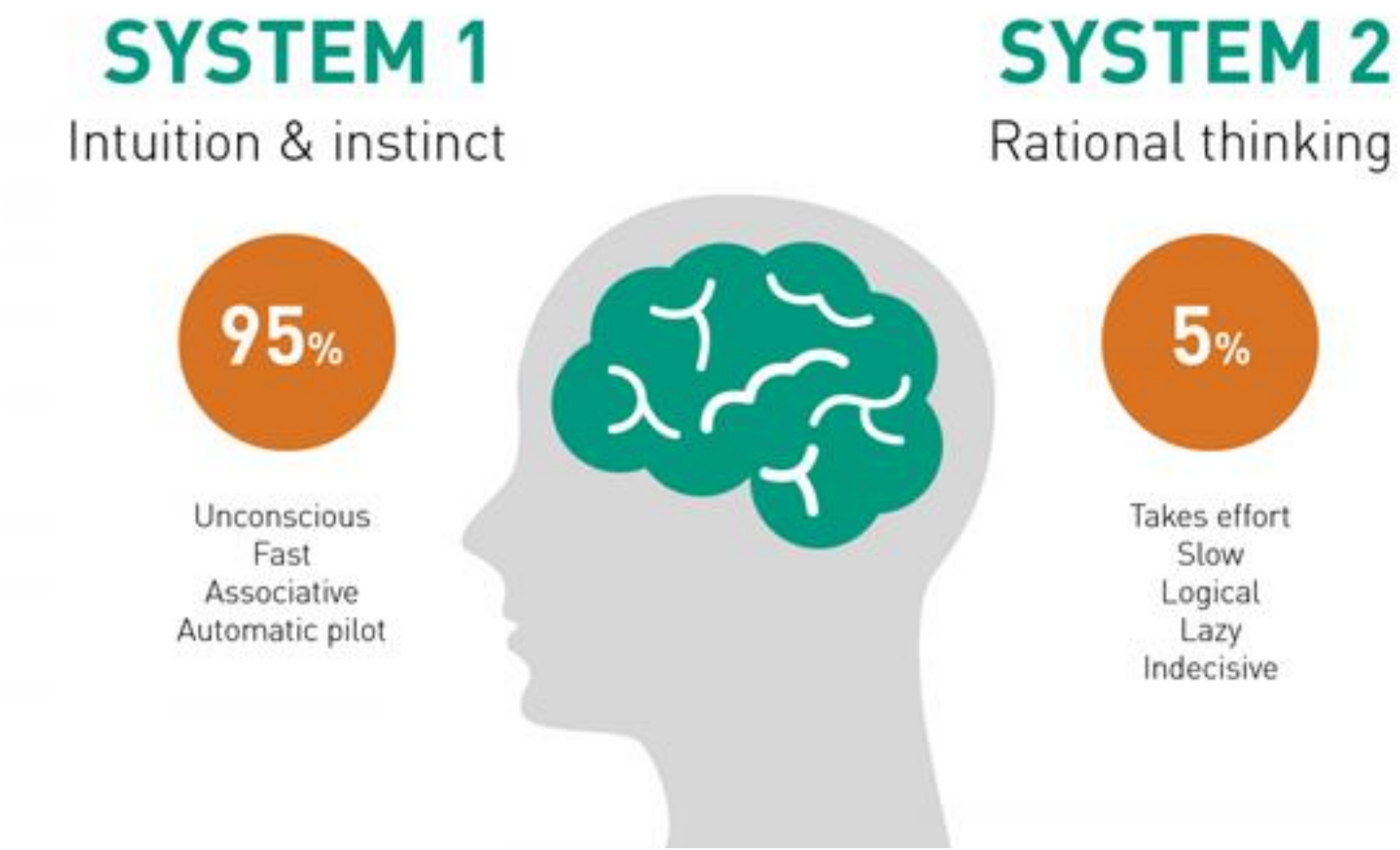
- Il «**processo cosciente esplicito**»: i processi coscienti possono essere alterati attraverso il ragionamento logico e l'educazione.

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

La teoria del doppio processo



La teoria del doppio processo



DDD e i Rapporto tra processi decisionali

Cosa c'entra tutto questo con DDD ?

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).

DDD e i Rapporto tra processi decisionali

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 poggiono esclusivamente su di un'opinione.

Euristiche che influenzano la progettazione

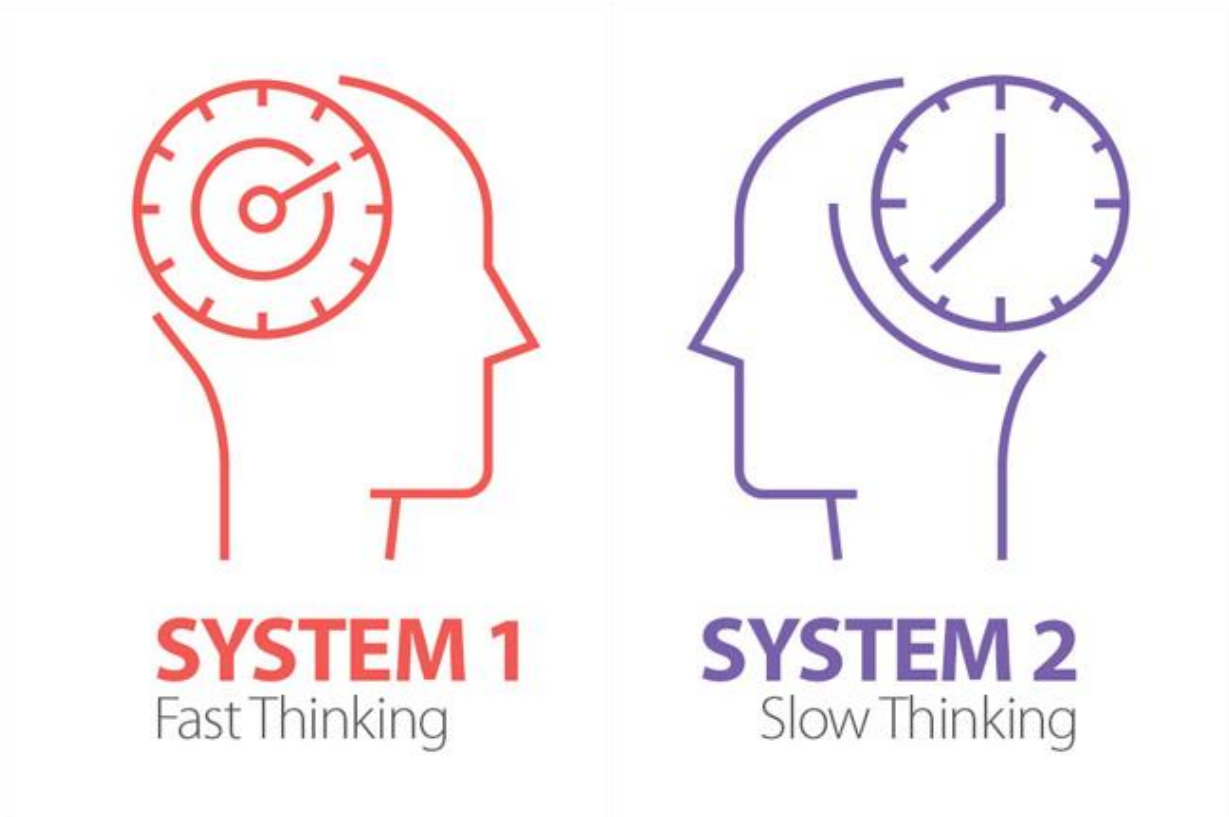
Euristica della disponibilità: le informazioni che abbiamo non è detto che siano sufficienti per conoscere appieno un problema particolare; ciò accade perché le persone tendono a basare le proprie decisioni solo sulle informazioni in mano, senza nemmeno cercare di ottenere maggiori dettagli.

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.

Conclusioni

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



Conclusioni

Essere ossessionati 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.

Tecniche per acquisire la conoscenza

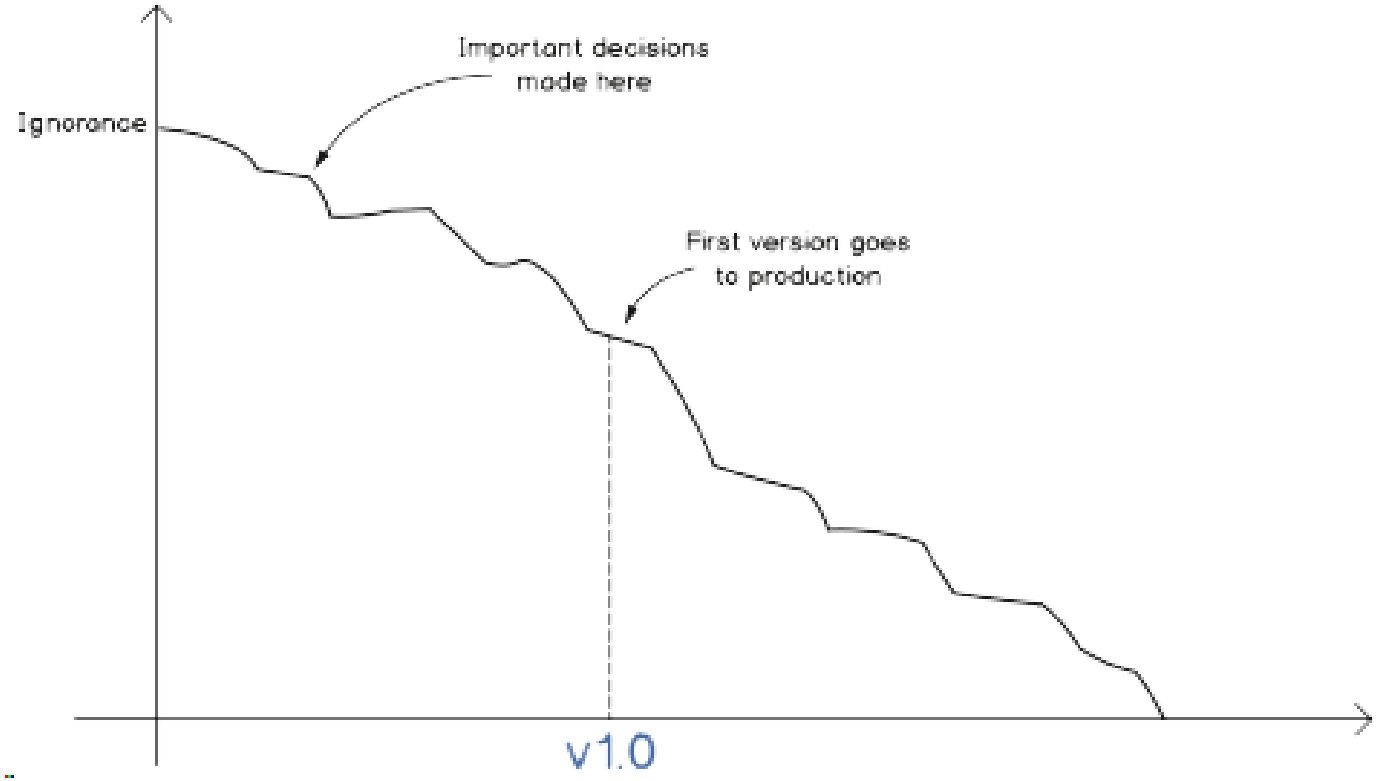
- **Parlare con quante più persone possibili che abbiamo la conoscenza del dominio**
- **Essere buoni osservatori: andare sul campo, vedere gli attori lavorare e le attività svolte e poi effettuare le interviste/riunioni**
- **Usare metodologie per cogliere le azioni e rappresentarle graficamente (pittogrammi 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 (ovvera quella usata nel dominio oggetto di studio)**
- **Etc. Etc.**

Riduzione dell'ignoranza

I principi base su cui poggia il DDD, in generale, hanno come obiettivo 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.

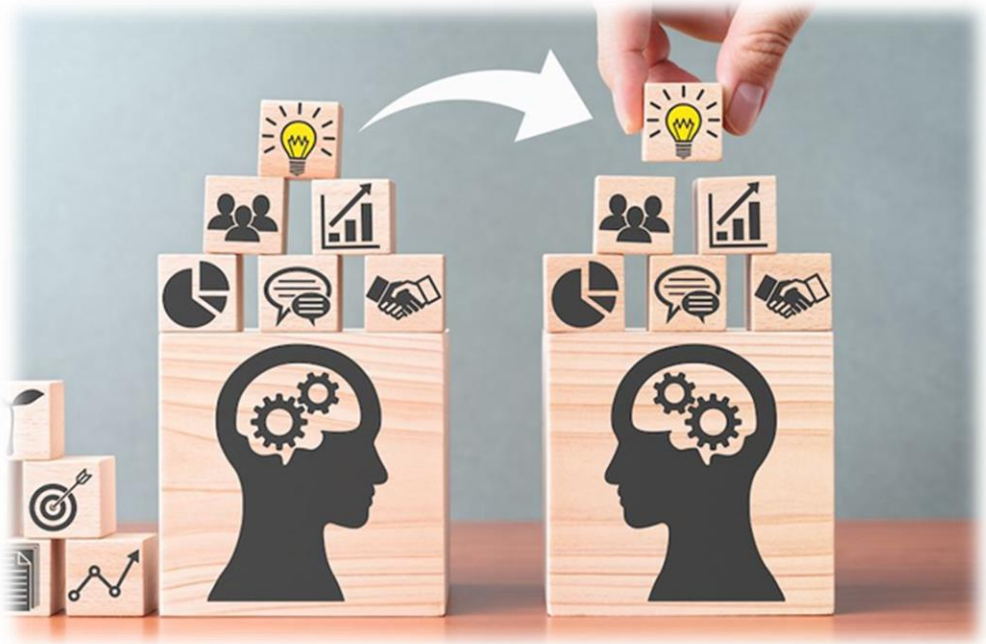
A questo livello si ha la maggior parte della conoscenza, si sa cosa fare e come farlo.

I livelli dell'ignoranza – Livello 1

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.

I livelli dell'ignoranza – Livello 3

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**.

I livelli dell'ignoranza – Livello 4

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, conscia o subconscia, porta ad una mancanza di conoscenza e ad un'errata interpretazione del problema, e quindi aumenta la possibilità di costruire la soluzione sbagliata.

La consapevolezza del rischio

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**

Come mitigare i rischi

- **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.

Il linguaggio e il contesto

L'importanza del linguaggio nel DDD

Nell'industria del software, abbiamo sviluppato questa percezione ingenua che gli unici linguaggi che contano sono i linguaggi di programmazione.

Ecco perché spesso parliamo in modo complesso e senza senso per i nostri colleghi di altri reparti o ai nostri clienti, i quali hanno difficoltà a comprendere cosa stiamo cercando di dire.

Questo problema è reciproco perché molte linee di business hanno sviluppato il loro gergo, che altre persone potrebbero non comprendere completamente.

I principi base nel linguaggio del DDD

Nel DDD, i punti cardine per la definizione del linguaggio li possiamo riassumere così:

- **Ubiquitous Language**: il linguaggio deve essere **onnipresente** ovvero deve essere quello condiviso dal dominio, dal software e dagli attori coinvolti.
- **Why language is important**: è importante parlare la **medesima lingua**
- **Making implicit explicit**: rendere chiaro ciò che ancora non lo è
- **Language in context**: applicare il linguaggio giusto nel contest giusto.
- **Expressing behavior**: esprimere il comportamento

Ubiquitous Language

Come «Developer» cerchiamo di tradurre un linguaggio umano in un linguaggio di programmazione; **questa è l'essenza del nostro lavoro**.

Tuttavia, questa non è la parte essenziale della **routine** quotidiana dello sviluppatore.

Due persone possono capirsi solo se parlano la stessa lingua.

Non è necessario che sia verbale; potrebbe benissimo essere la lingua dei segni o la lingua della musica, ma entrambi gli interlocutori **devono possedere la stessa comprensione** di questo linguaggio universale altrimenti sorgeranno problemi di comunicazione.

Non solo devono parlare la stessa lingua, ma questa lingua deve essere usata in un contesto.

Il linguaggio e la comprensione

Quasi ogni settore ha sviluppato un linguaggio particolare che solo le persone di quel settore comprendono appieno.

Uno degli esempi di tale industria è quella del software.

Quando due programmatori discutono i dettagli di implementazione di alcuni sistemi ragionevolmente complessi, i non programmatori intorno a loro non capiscono molto di questa conversazione e di solito si annoiano.

La mancanza di comprensione si traduce sempre in una mancanza di interesse.

Il Linguaggio e la raccolta dei Requisiti

I requisiti non si concentrano solo sulla **soluzione** e **nascondono i problemi**, ma tendono anche a tradurre un determinato linguaggio in uno più tecnico e considerato come più **adatto** per gli sviluppatori.

In realtà, funziona più come un telefono rotto.

Più livelli di traduzione vengono aggiunti alla linea di trasmissione, meno informazioni rilevanti e precise raggiungono il ricevitore.

Un altro aspetto di tale traduzione è che rallenta la comunicazione.

Se gli sviluppatori richiedono più informazioni agli attori che posseggono la conoscenza ma non sono in grado di comprendere la «**lingua del contesto**» il coinvolgimento dei «traduttori» diventa inevitabile.

Rendere «l'implicito esplicito»

Quando iniziamo a lavorare su un nuovo sistema, dobbiamo imparare molto.

Prima di conoscere la lingua del dominio, utilizziamo la «nostra comprensione del dominio», che spesso si basa su presupposti (se non addirittura sy preconcetti/pregiudizi).

Immaginiamo una situazione in cui partecipiamo a una riunione con esperti di dominio allo start di un progetto; gli esperti tentano di spiegare il loro problema e noi iniziamo lentamente a studiare la loro lingua; a un certo punto «penseremo» di aver avuto un'idea, formuliamo un'ipotesi e più o meno pensiamo di sapere cosa fare.

Il primo e il più ovvio rischio è quello di pensar che: «quello che vedi è tutto ciò che c'è» (what you see is all there is – WYSIATI); ciò va sotto il nome di «euristica della disponibilità».

Rendere «l'implicito esplicito»

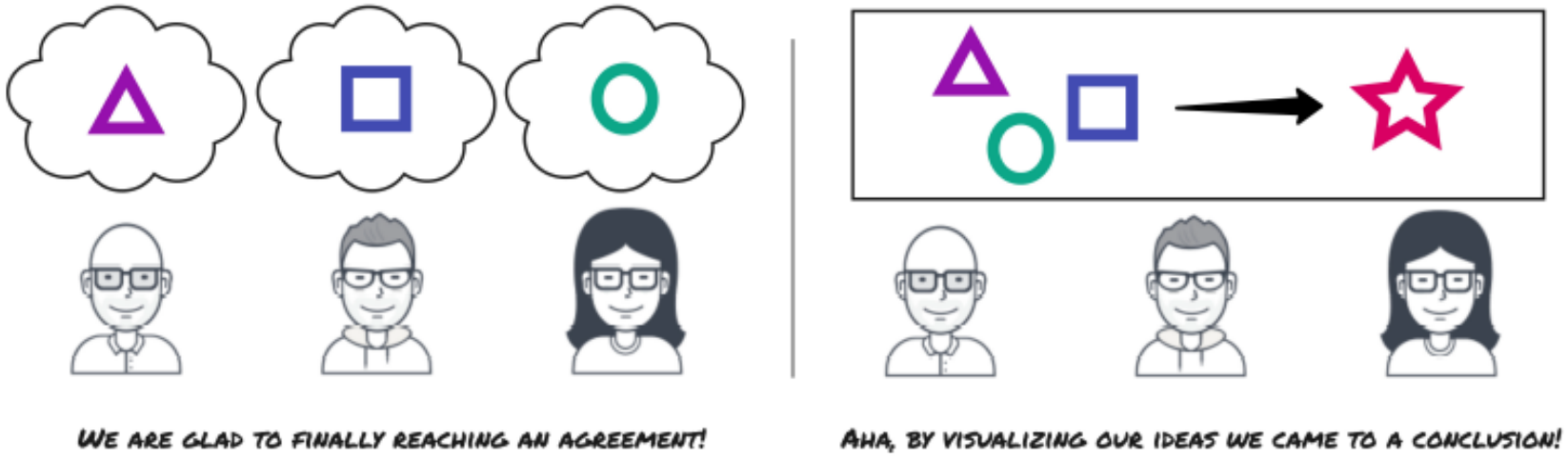
In altre parole applichiamo la nostra «**conoscenza**» (limitata e poggiante sulle esperienze passate) con la convinzione di «**avere la sensazione di aver compreso quanto necessario**».

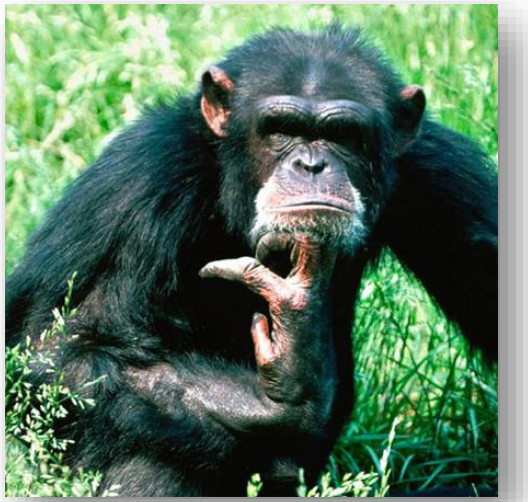
Ciò ci **infonde sicurezza** per cui, con confidenza, passiamo allo step successivo, ovvero al calcolo delle stime e …… «**logicamente falliamo**», e quando ne prendiamo coscienza può essere oramai tardi.

Rendere «l'implicito esplicito»

Altri rischi in cui si incorre nei workshop è «trovarsi già dall'inizio d'accordo su qualcosa». Tutti lasciano la stanza per incontrarsi di nuovo dopo un paio di settimane, magari per discutere di alcune specifiche o addirittura di prototipi.

Il tempo passa, siamo ancora nella stessa stanza, e nessuno è contento poiché, con nostra reciproca insoddisfazione, scopriamo di «essere d'accordo su cose completamente diverse».





Rendere «l'implicito esplicito»

Per risolvere il problema della «falsa comprensione condivisa» dobbiamo:

- **Rimuovere le ipotesi.**
- **Rendere esplicite le cose implicite.**
- **Scrivere requisiti comprensibili per tutti.**
- **Progettare interfacce grafiche (GUI) comprensibili per tutti.**
- **Scrivere «CODICE COMPRENSIBILE».**

Linguaggio e Contesto

Il problema che bisogna affrontare e risolvere è quello delle differenze linguistiche (sia a parità di lingua sia per lingue differenti).

Nel quotidiano dobbiamo tenere in conto che per oggetti e azioni di tutti i giorni, vengono utilizzate parole diverse.

Lo stesso può essere osservato per le lingue che si evolvono all'interno di gruppi professionali in cui le persone sviluppano un proprio gergo.

È importante definire il significato preciso delle parole in quanto evitare la confusione è davvero uno degli obiettivi per trovare e identificare l'«Ubiquitous Language».

Linguaggio e Contesto

È importante rendersi conto che l'«Ubiquitous Language» è valido solo all'interno di uno «specifico contesto».

Un contesto diverso è definito da una lingua diversa.

C'è un'idea sbagliata che «Ubiquitous Language» sia chiamato «onnipresente» perché è l'unica lingua per l'intera azienda, organizzazione o dominio.

Non è così in quanto è onnipresente verticalmente e non orizzontalmente.

Linguaggio e Contesto

In altre parole, all'interno di un azienda (il nostro dominio di riferimento) i vari reparti in cui è suddivisa (i nostri contesti) possono conoscere «un oggetto» (ad esempio il concetto di «prodotto») in modo diverso e con una «profondità» diversa degli «attributi dell'oggetto» (es. le proprietà del prodotto).

Ciò equivale a dire che per lo stesso dominio, esiste un contesto diverso in cui la lingua cambia e talvolta cambia in modo significativo.

Cosa succede se continuiamo a usare lo stesso significato delle parole in tutti i contesti? Ebbene, le «cose» diventano molto meno esplicite.

Linguaggio e Contesto

Il grado di ambiguità aumenta con ogni nuovo contesto che non riusciamo a identificare e separare.

Ciò porta a modelli poco chiari e, di conseguenza, a oscurare il codice, nel quale invece dobbiamo chiarire cosa intendiamo esattamente quando usiamo questa e quella parola.

Mescolare contesti diversi in un ambiente di lavoro porta anche a qualcosa chiamato «cambio di contesto».

Dal punto di vista dell'analisi e del design ciò si traduce nel fatto che la maggior parte del tempo deve essere speso per identificare le «azioni» (o se vogliamo le attività) e comprendere in quali contesti vengano eseguite.

Tradotto, ciò significa che chi fa analisi, design, sviluppo non può pensare di vedere un «**oggetto**» da un'unica prospettiva «**codificata universalmente**».

Bisogna «entrare nel dominio di interesse», comprenderlo, individuare i contesti, dialogare con gli attori che operano nei contesti e **determinare la giusta lingua con cui comunicare in ogni contesto**.

Con una metafora, potremmo affermare che è necessario individuare le giuste finestre del nostro edificio per guardare al suo interno da più prospettive (o per meglio dire da «più punti di vista»).

Linguaggio e Contesto

Non appena il significato delle parole inizia a cambiare tra le diverse parti del sistema, questo dovrebbe far scattare un allarme perché probabilmente si stanno attraversando i confini del contesto.

Il contesto emerge quando si discute degli e con gli utenti.

Ciò che bisogna «imparare» è «comprendere»:

- **Quale ruolo gioca questo utente in quel particolare momento ?**
- **Quale lingua parla quell'utente ?**

EvenStorming

I Tempi della scoperta

La **scoperta** della terminologia del dominio è essenziale e questa terminologia diventa parte dell'«**Ubiquitous language**».

Tuttavia, il processo di scoperta può essere piuttosto lungo e non sempre riesce.

Quando discutiamo di come funziona l'azienda e di quali problemi risolveremo scrivendo software, **troppo spesso la conversazione si riduce alla discussione delle funzionalità che l'azienda desidera implementare**.

Un insieme di funzionalità, ovviamente, può essere chiamato software, ma non costituisce necessariamente un sistema.

I Tempi della scoperta

Inoltre, per costruire una soluzione completa per un problema particolare, è necessario un pensiero più a livello di sistema:

- **Ma chi ci dirà come funziona l'azienda come sistema ?**
- **Con chi dovremmo parlare e quale formato dovrebbe assumere questa conversazione ?**

I Requisiti

Molto spesso gli sviluppatori conoscono il dominio solo sotto forma di requisiti.

Ma, per quanto fin qui detto, i requisiti hanno i loro difetti.

Per migliorare le conoscenze si decide di parlare direttamente con gli esperti di dominio organizzando workshop o meeting con loro.

Alcune persone partecipano e vengono intervistate; molte cose vengono discusse, molte nuove intuizioni vengono a galla, ma c'è un risultato minimo sotto forma di qualsiasi artefatto di modellazione.

Poi si inizierebbe a disegnare diagrammi UML, ma all'infuori dei tecnici, chi li comprenderebbe ?

Si potrebbero prendere appunti per poi scoprire che bisogna effettuare più workshop di chiarimento perché vi sono troppi concetti vaghi e impliciti su punti che potrebbero costituire le fondamenta del sistema software futuro.

Allora cosa bisogna fare per applicare un corretto DDD ?

La visibilità

Fornire visibilità durante la discussione.

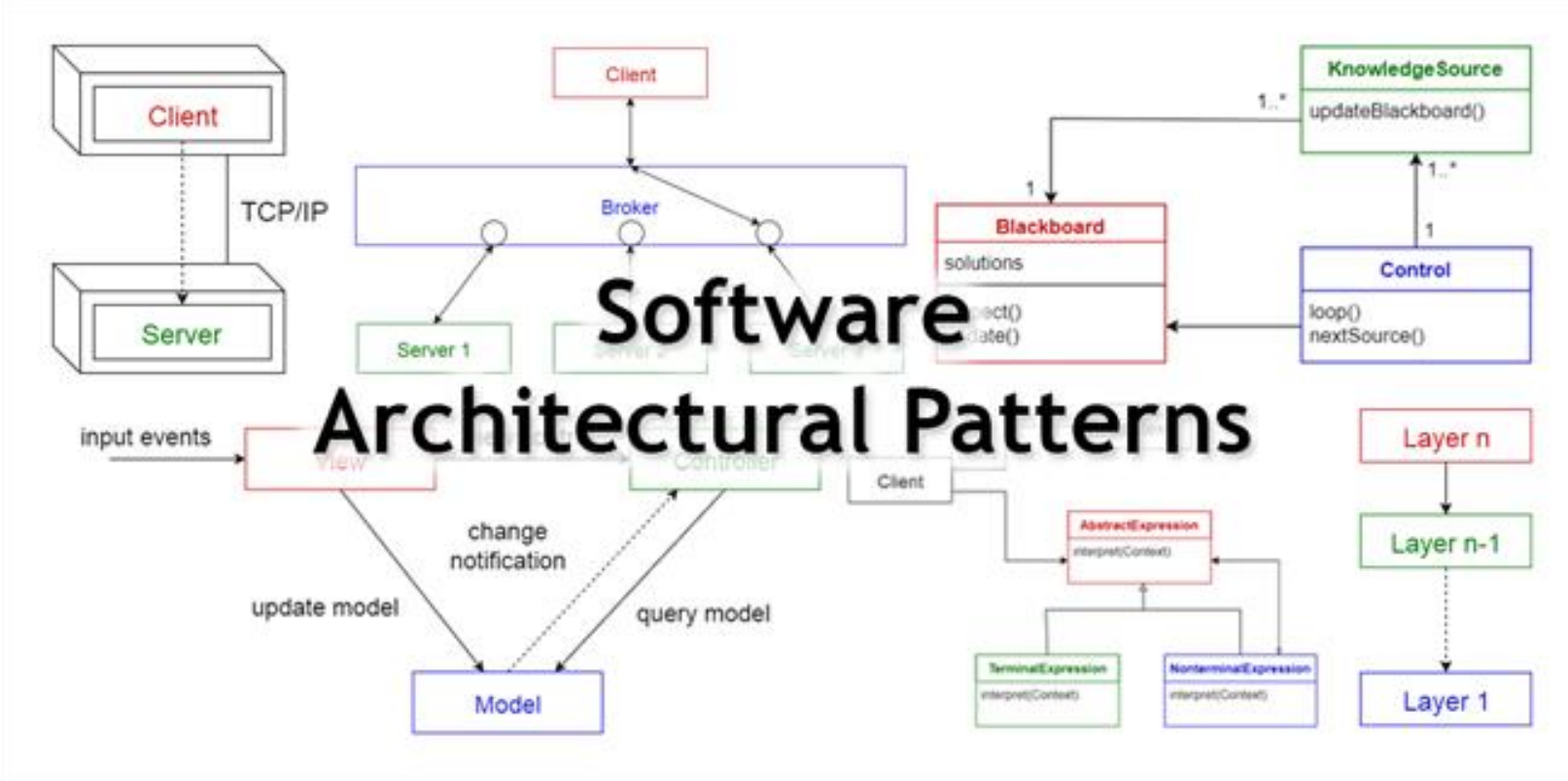
Ciò dovrebbe rimuovere le ipotesi quando molte persone discutono la stessa cosa con termini diversi.

Eliminare le ambiguità facendole emergere in superficie per ulteriori esplorazioni.

Linguaggio di modellazione comprensibile

Avere un linguaggio di modellazione che le persone capiscano.

UML non è un'opzione e le solite caselle e frecce non hanno una vera notazione, quindi le persone possono confondersi e iniziare a passare il tempo cercando di chiarire il significato delle cose.



Coinvolgimento delle persone

Coinvolgere molte persone contemporaneamente.

E' necessario che ogni partecipante a un workshop possa esprimere la propria opinione (ma con criterio e organizzazione) cercando di trasmettere efficacemente il proprio messaggio, mentre tutti gli altri devono stare in ascolto (tanto alla fine ognuno potrà dire la propria).

Ciò è necessario perché supponendo che persone con interessi e background diversi siano presenti in una sessione, queste potrebbero mostrare una mancanza di interesse e annoiarsi in quanto non coinvolte nei processi di «scoperta» e di «comprensione» del dominio e dei contesti.

Uniformità

- Bisogna **trovare un modo per esprimere**:
  - Termini
  - Comportamenti
  - Processi modello
  - Decisioni
- Bisogna **evitare di esprimere**
  - Caratteristiche
  - Dati

Nel tempo è stato formulato un metodo chiamato «EventStorming», con il quale si è cercati di affrontare tutte queste problematiche e che verrà descritto nelle prossime slide.

Modeling Language

L'idea di base dietro EventStorming è fornire una semplice notazione di modellazione utilizzata per visualizzare il comportamento del sistema in un modo che tutti possano comprenderlo.

Questo approccio crea visibilità e aumenta il coinvolgimento delle persone.

Considerando il comportamento come l'aspetto centrale della conoscenza del dominio, l'intero esercizio EventStorming riguarda la scoperta di come funziona l'azienda.

Modeling Language

In generale, potremmo postulare che ogni sistema in un dato momento di tempo si trova in uno stato particolare; questo stato può cambiare quando gli attori che interagiscono con il sistema fanno qualcosa ovvero quando compiono azioni.

Le azioni portano a un cambiamento di stato del sistema il quale quindi dovrebbe scatenare un evento grazie al quale possiamo gestire a affrontare la nuova situazione (ovvero il nuovo stato); tali eventi sono noti come «eventi di dominio».

Ogni evento di dominio rappresenta un fatto, un cambiamento nel sistema che stiamo cercando di modellare.

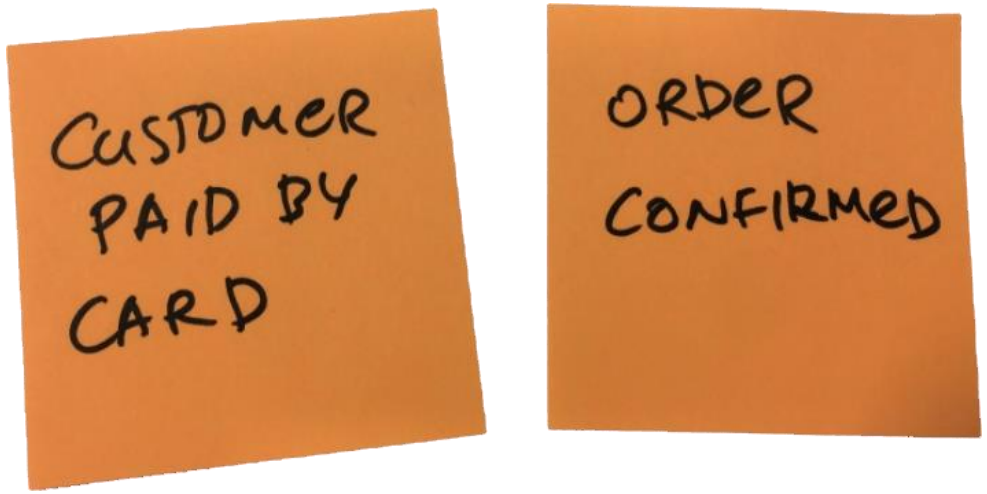
Pertanto, la prima parte del nostro linguaggio di modellazione consiste nel creare il concetto di eventi di dominio.

È il concetto più semplice e anche più importante di cui si occupa «EventStorming» ed il motivo per cui si chiama proprio «EventStorming»

Ogni concetto in EventStorming è rappresentato come una nota adesiva di un colore specifico.

Il colore è essenziale perché, mentre procediamo e portiamo più pensieri al modello, abbiamo bisogno che i colori rappresentino in modo coerente le stesse idee in tutto il modello per evitare confusione.

Quelli indicati in figura sono due eventi di dominio che si sono verificati in una sequenza; gli eventi devono avere un soggetto (sostantivo) e un predicato (verbo).



In primo luogo, un cliente ha pagato utilizzando una carta di credito o di debito; poi il loro ordine è stato confermato.

Possiamo identificarlo come un dominio di e-commerce.

Non c'è niente di speciale nelle frasi scritte su foglietti adesivi, tranne una regola cruciale: **il verbo deve essere al passato, a indicare che qualcosa è accaduto ed è diventato un fatto**.

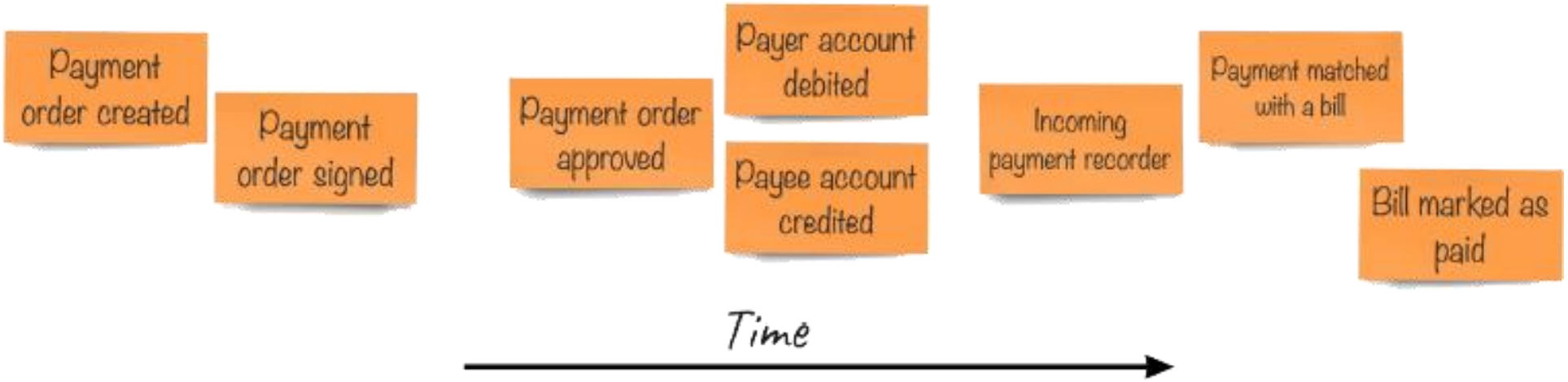
In questo esempio notiamo due cose essenziali.....

Sequenza Temporale

La prima: gli eventi di dominio seguono una sequenza temporale.

È abbastanza logico perché i fatti rappresentano i cambiamenti successivi nel sistema e quindi avvengono in un ordine particolare (ad esempio, il pagamento non viene approvato prima di essere firmato).

Alcune cose possono accadere in parallelo, come addebitare e accreditare conti contemporaneamente, non appena l'ordine di pagamento viene approvato, il che potrebbe significare che la banca è sicura che il pagatore disponga di fondi sufficienti per completare il pagamento.



La seconda: non abbiamo un solo sistema in gioco.

In effetti, stiamo modellando l'intero processo, ma vi potrebbero essere almeno tre parti che possiamo in prima battuta individuare.

- l'utente che (ad esempio) con un canale «banca Internet» crea e firma gli ordini di pagamento;
- il back office bancario, che completa l'operazione
- il sistema di abbinamento pagamento-fattura del beneficiario che, tra l'altro, potrebbe essere completamente manuale

Visualization

Il nostro semplice modello fornisce già «molta conoscenza» alle persone coinvolte nel workshop.

Non solo abbiamo cercato di identificare cosa accade durante il processo di pagamento, ma abbiamo inserito l'intero flusso in una sequenza temporale e siamo stati in grado di identificare con buona approssimazione parti del processo che possono verificarsi in diversi sistemi fisici.

La visualizzazione è uno degli aspetti più potenti di qualsiasi tecnica di modellazione e EventStorming non fa eccezione.

Non appena mettiamo qualcosa sul nostro modello, possiamo ragionarci sopra invece di pronunciare parole e agitare le mani.

Visualization

Quando le persone vedono ciò che è considerato l'intero quadro, alcuni potrebbero iniziare a chiedersi cosa succede se domande.

- **Cosa succede se non ci sono abbastanza soldi nel conto?**
- **Cosa succede se il numero di riferimento della fattura è sbagliato?**
- **Cosa succede se l'account del beneficiario non è corretto?**
- **E se, e se, e se? …**

Visualization

Sembra quindi che il nostro processo non sia così semplice.

Il mondo reale è complicato in quanto, il più delle volte, il numero di casi limite supera quello che è considerato un flusso regolare di eventi.

Ma basta la nostra comprensione iniziale su una visione semplificata del mondo per carpirne la complessità ?

Le persone che hanno interesse di raggiungere un obiettivo non hanno intenzione di commettere errori, ma in questo primo approccio che abbiam dato c'è un problema di fondo che può rendere un disservizio a coloro che stanno facendo del loro meglio per creare un modello di eventi adeguato.

Visualization

In un **workshop** di solito le persone si siedono intorno a un tavolo e parlano ma come abbiamo già suggerito, non è così che funziona EventStorming.

Ci aspettiamo che le persone si muovano nella stanza e siano attivamente coinvolte nelle conversazioni, che potrebbero avvenire simultaneamente in diversi lati della stanza.

Quindi, **abbiamo bisogno di spazio**, ma con precisione di quanto ?

Pensiamo ai post-it usati nel precedente esempio per disegnare un caso semplice e pensiamo a tutti i casi limite/eccezioni che potremmo rilevare. Quanto dovrà essere grande la lavagna/parete etc. che utilizzeremo per modellare ?

Ricordiamoci che, per questo tipo di approccio, è essenziale avere una visione d'insieme altrimenti «**limitato sarà lo spazio e limitato sarà il modello disegnato**».

Domain Model

Domain Model

I modelli rappresentano oggetti della vita reale.

Alcuni modelli potrebbero essere abbastanza astratti altri invece possono dare una visione più dettagliata di ciò che rappresentano.

In generale, anche se cercano di rappresentano il mondo reale hanno uno scope limitato e raggiungono solo un certo livello di dettaglio.

Un modello potrebbe ignorare degli elementi poiché ritenuti non necessari, ma ciò non significa che quegli aspetti trascurati non siano importanti ma solo che non sono cruciali per quel particolare spazio del problema.

Domain Model

In quanto tali, i modelli di dominio nel software devono rappresentare quegli aspetti del dominio aziendale che sono essenziali per risolvere il problema in questione.

A volte vi è la tentazione di scendere nel dettaglio ma aggiungerebbe complessità non necessaria non aiuterà a risolvere il problema.

Gli oggetti di un modello di dominio rappresentano i dati e il comportamento del dominio.

Il comportamento e i dati nel modello sono interconnessi.

Domain Model

Il **comportamento** del modello ne **manipola i dati** e poiché questi ultimi rappresentano ciò a cui il modello stesso è interessato e su cui opera, tali dati sono anche noti come **stato**.

Lo stato sono i dati che descrivono l'aspetto del nostro sistema a un determinato instante e ogni comportamento del modello cambia lo stato.

Lo stato è ciò che persistiamo nel database e che possiamo ripristinare in qualsiasi momento prima di applicare un nuovo comportamento.

Ogni comportamento del modello di dominio **causa una transizione nello stato del domain model**.

Tutto ciò che documenta come cambia lo stato del dominio dovrebbe far parte del modello di dominio.