Alberto Acerbis

Architetture evolutive, DDD, microservizi

Uno sguardo d'insieme

MokaByte

Alberto Acerbis

Architetture evolutive, DDD, microservizi: uno sguardo d'insieme

MokaByte

Alberto Acerbis, *Architetture evolutive, DDD, microservizi:uno sguardo d'insieme*.
MokaByte 2025

Immagine di copertina:
Elaborazione da AI Scribbles con licenza CC BY-SA 4.0

Indice

Introduzione

Se dovessimo ridurre lo sviluppo software a una singola azione, in molti penseremmo alla scrittura del codice. Questo libro, nato dalla rielaborazione di una lunga serie di articoli scritti dall'autore per MokaByte.it, si propone proprio di ripensare questo equivoco ancora troppo comune.

"Oltre il codice: architetture software per un mondo complesso" potrebbe essere una frase che ben sintetizza quanto il lettore troverà in queste pagine.

Nel testo infatti, non mancano certo i riferimenti a tecnologie, codice e pattern architetturali che compongono il bagaglio di conoscenze di ogni buon sviluppatore software. Ma il focus è sempre sul senso del prodotto che si sta progettando, a quanto esso risponde effettivamente a una reale esigenza del cliente o del mercato e al modo in cui l'architettura software sulla quale è costruito è relamente in grado di riflettere la realtà del mondo che cerca di imitare, rivelandosi oltretutto in grado di cambiare per adattarsi alle condizioni sempre più mutevoli e, spesso, imprevedibili, nelle quali dovrà funzionare. Per questo il concetto di *architettura evolutiva* è centrale in questa pubblicazione, a partire dal titolo.

La materia trattata oscilla tra continui riferimenti alle più attuali tendenze dell'architettura software e alcuni principi ormai assodati da tempo ai quali, però, spesso si finisce per non dare l'attenzione che meritano ancora. Numerosi sono gli esempi provenienti dall'esperienza maturata in tanti anni dall'autore. Emerge la consapevolezza che non esistono "proiettili d'argento" o soluzioni pronte all'uso, buone per tutte le stagioni, ma che un buon Software Architect debba interrogarsi continuamente per migliorare quanto già fatto. Anche tecnologie ottime e diffuse, come i microservizi, vengono analizzate nei loro vantaggi e criticità, valutando il loro impiego

in base al progetto che su cui si sta operando e non come riflesso dell'ennesima "moda" tecnologica o metodologica del momento.

Dieci capitoli, di agevole lettura, ci accompagneranno in questo viaggio, affrontando i diversi argomenti in maniera modulare, ma sempre collegata dalla medesima "filosofia" che connota tutto il testo.

Questo libro è rivolto a chi non si accontenta di sapere come implementare una tecnologia, ma vuole capire perché farlo. È per chi è interessato a navigare l'incertezza e a progettare sistemi che durino nel tempo grazie all'aderenza alla realtà che modellano e alla capacità di evolvere.

Benvenuti nello spazio del problema. È tempo di riflettere anzitutto sulle le domande giuste, prima di cercare le soluzioni più adeguate.

L'autore

Alberto Acerbis lavora da anni come Software Architect e si definisce "uno sviluppatore backend". La sua eterna curiosità lo porta comunque a interessarsi anche all'altro lato del codice, consapevole che "scrivere" software significhi principalmente risolvere problemi di business e fornire valore al cliente. In questo, ritiene i pattern del DDD un grande aiuto.

Attualmente ricopre il ruolo di Software Engineer presso Intré, un'azienda che sposa questa ideologia. È co-founder della community DDD Open e Polenta e Deploy, e membro attivo di altre come Blazor Developer Italiani.

Capitolo 1

Quando non trovo le Best Practice

Oltre la scrittura di codice

Sviluppare **software**, di per sé, non è complicato. O meglio… dipende! Se con il termine **sviluppare** ci limitiamo all'azione di scrivere codice, allora l'affermazione è corretta. Per scrivere del buon codice, oltre che imparare un linguaggio e tutte le best practice per applicare le sue regole, mi basta leggere qualche buon libro sul **Clean Code** e magari, visto che sono volenteroso, anche qualche libro sulle pratiche dell'**Extreme Programming**… e il gioco è fatto. Difficilmente mi si potrà dire che ho scritto qualcosa di poco chiaro. Lo si potrà migliorare, senza nessun dubbio, ma non mi si potrà dire che è incomprensibile.

Ma siamo sicuri che sviluppare software sia solo scrivere del buon codice? Il mio docente di ingegneria del software era solito ripetere che la scrittura del codice è solo l'ultima fase nella realizzazione di un sistema. Questa sua affermazione non mi ha mai convinto completamente; all'inizio perché, come ogni novizio introdotto all'arte della scrittura del codice, non vedevo l'ora di veder funzionare su un PC quello che scrivevo, e ogni scoperta, ovviamente, era subito da implementare. Più tardi per altre ragioni, che vedremo più avanti.

Quello che però ho capito è che il professore proprio tutti i torti non li aveva, e che entrambi avevamo un po' ragione e un po' torto. Sviluppare software è qualcosa che va oltre la scrittura di buon codice, l'adozione dei pattern architetturali, e l'applicazione delle pratiche di **Extreme Programming**. Sviluppare software è **risolvere problemi** di **business**, e per fare questo dobbiamo prima conoscere il problema; potremmo dire che sviluppare software è un **continuo apprendimento**, portato avanti da un gruppo di persone: cliente, stakholder, utenti e sviluppatori, di cui il codice è solo l'effetto secondario.

Il senso delle architetture

Sviluppare software dunque è difficile, e lo è maggiormente ora che la parola d'ordine è **Distributed Systems**. Già, perché già era difficile sviluppare un'applicazione da installare su di un server in

locale, figuriamoci sviluppare **applicazioni distribuite** in cui entrano in gioco attori come **microservizi**, Cloud, consistenza, elasticità, affidabilità, etc. Tutte queste voci ricorrono sotto la voce "**architettura**", e quando sentiamo questa parola generalmente abbiamo due reazioni.

La prima che ci porta a pensare alle architetture più adottate, **hexagonal architecture, onion architecture e clean architecture**, che è la più rassicurante, perché ci proietta nel mondo delle **best-practice**, quindi qualcosa di replicabile. La seconda, un po' meno rassicurante, è quella che ci fa realizzare che è difficile trovare conferenze, o libri, che parlano esplicitamente di architetture, di come configurare l'ambiente per ottenere la massima affidabilità, o il massimo delle prestazioni, senza dover costruire qualcosa di troppo **ingombrante** sin dall'inizio, e che ci insegni a mantenere l'integrità strutturale nella nostra **codebase**, senza dover partire con i massimi sistemi sin dall'inizio.

Tra best practice e compromessi

Non esistono le **best-practice** per questi problemi, ma c'è solo una serie infinita di compromessi, di **trade-off.** Quello che è buono per la soluzione al problema di oggi, non è detto che sia valido per il problema che dovrò affrontare domani; dipenderà molto da altri fattori, che accompagnano le richieste, decidere quali soluzioni adottare; e in questo caso l'esperienza conta tantissimo. Intendiamoci, è molto più esperto uno sviluppatore che in un anno ha affrontato dieci problemi diversi, rispetto a uno sviluppatore che da dieci anni lavora sullo stesso **dominio**!

Vent'anni fa, Eric Evans, con il suo iconico libro *Domain Driven DesignI* [1] ci ha presentato una serie di **pattern**, tutt'ora validissimi, per affrontare **problemi complessi**, e non solo. Ci ha mostrato un nuovo modo di affrontare lo sviluppo del software, meno data-centrico e più orientato alla risoluzione dei problemi di business. Pur essendo, come detto, un libro iconico, il testo di Evas dimostra ormai anche tutti gli anni che ha.

Introdurre il Bounded Context

Prendiamo ad esempio uno dei pattern più famosi, e forse più abusati, da lui presentato, il **Bounded Context**, di cui parleremo in maniera approfondita più avanti. Questo pattern è da molti anni utilizzato come riferimento per la realizzazione di **microservizi**. Se riesco a individuare una porzione del business **isolata**, che posso portare avanti indipendentemente dalle altre, magari con un team dedicato, allora ci realizzo un microservizio attorno, e il gioco è fatto. In fondo questo è l'approccio suggerito da Evans: suddividere il **dominio** in tanti sottodomini indipendenti e, soprattutto, ben chiusi all'interno di un **confine invalicabile**, dove ognuno è padrone a casa sua. Una delle frasi che da sempre mi gira in testa, facendo riferimento proprio a questo libro, e a questo pattern, è che il dominio è **ignorante alla persistenza**, ed è anche uno dei concetti che mi impegna di più quando spiego a un team che il pattern del **Bounded Context** è quello da affiancare al concetto di microservizio.

La pallottola d'argento... non esiste

L'idea, ovviamente, è assolutamente valida. Un conto è occuparsi di problemi di **business**, e cercare la soluzione software a questi problemi, un altro è occuparsi di problemi di **architettura**, o **infrastruttura**, quali database, reti, tempi di risposta, elasticità, etc. In un articolo lungimirante [2], Fred Brooks parla di **complessità essenziale** vs **complessità accidentale**, per differenziare appunto le complessità proprie legate al problema di business che intendiamo risolvere, in confronto alle complessità legate all'ambiente in cui distribuiremo l'applicativo, ma anche legate al linguaggio e ai framework che utilizzeremo per svilupparlo.

In questo articolo, Fred Brooks sottolinea che la complessità accidentale è destinata a diminuire, e che gli sviluppatori si possono dedicare maggiormente alla complessità essenziale. Sebbene, insiste Brooks, non esista una **Silver Bullet**, comunque una serie di innovazioni che affrontano la complessità essenziale può portare miglioramenti significativi nella produzione di software di valore. Anche

questo concetto contribuisce a rendere difficoltosa la comprensione moderna di realizzare software di valore; ciò che un tempo era considerato come complessità accidentale, oggi è diventato essenziale, ed altre complessità accidentali si sono aggiunte al nostro lavoro.

Qualche esempio con le architetture

Uno dei compiti di un software **architect** è quello di mantenere la **codebase** scalabile, al pari dell'architettura, e per assolvere a questo compito un **architect** non deve necessariamente conoscere i dettagli del dominio. Ma allora perché, negli ultimi anni, è così difficile realizzare software di valore? Perché fatichiamo a costruire sistemi distribuiti? Perché la complessità è aumentata così tanto con i microservizi? In fondo, nel mondo del software, si fanno sistemi distribuiti dal secolo scorso, non sono una novità portata dal Cloud. Proviamo ad analizzare il problema più da vicino, osservando figura 1.

Mi immagino già l'espressione di molti "Si tratta di un monolite distribuito, un'implementazione sbagliata dei microservizi"; e in

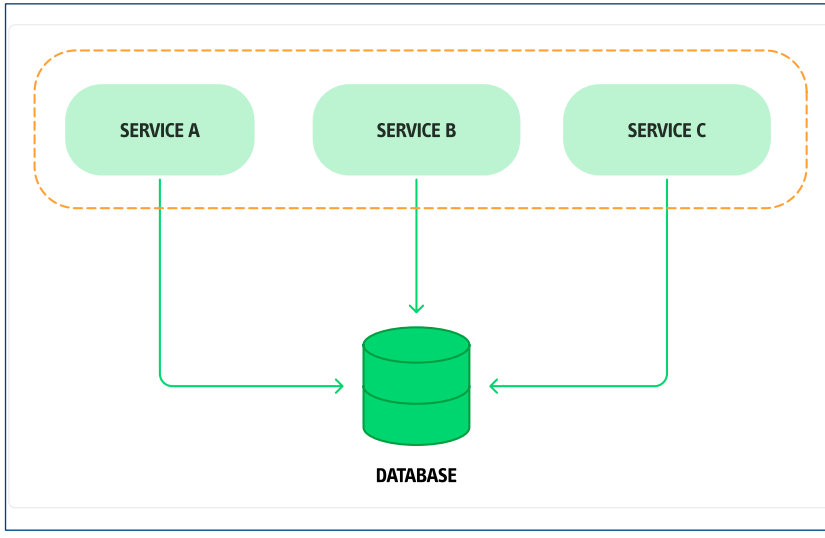

Figura 1.1 – Un'architettura a "monolite distribuito".

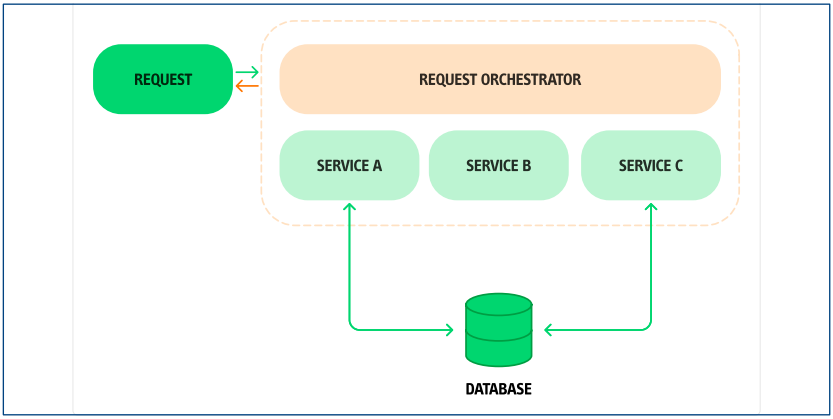

Figura 1.2 – L'architettura si arricchisce di un orchestratore di richieste, aumentando il llivello di disaccoppiamento.

effetti, per certi versi, lo è. Ma sono altrettanto certo che in molti siamo passati da questa implementazione, da questa architettura **Service-Based**. Sono molteplici i benefici che si possono ottenere quando si deve affrontare il **refactor** di un'applicazione monolite se si adotta questa architettura. Permette a un **architect** di determinare quali domini necessitano di ulteriore livello di granularità per essere eventualmente trasformati in **microservizi** e quali sono pronti così come sono.

Non è richiesta la frammentazione del database, almeno non fin da subito, perché non è ancora chiaro quali, e quanti, sottodomini avremo, e come dialogheranno fra loro, e per il momento è molto comodo lasciare l'onere della gestione delle transazioni al nostro database relazionale. Ultimo, ma non meno importante, questo approccio è un approccio di tipo tecnico, che generalmente non richiede il coinvolgimento dei **business expert**, o degli **stakeholder**, ma ci consente di intervenire velocemente sulla codebase per ringiovanirla e predisporla a ulteriori miglioramenti.

Il passo successivo verso la frammentazione del nostro monolite potrebbe essere l'individuazione di operazioni **cross-domain**, ciò

che normalmente rappresenta l'accoppiamento fra i servizi, che noi dobbiamo rendere dinamico se vogliamo poi suddividerli, e per questo dovremo aggiungere un **orchestratore**; magari, in un primo momento, sarà ancora un servizio condiviso fra tutti, e poi potrebbe essere sostituito da un broker di messaggi che gestisce sia le comunicazioni con il Client, sia quelle fra i servizi stessi, garantendo un buon livello di disaccoppiamento, spostandoci verso un'architettura a eventi piuttosto diffusa, **EDA Architecture,** che rispecchia il pattern del bounded context proposto da Evans, che volutamente ignorava la persistenza dei dati.

Nel 2003 tutto questo era sicuramente rivoluzionario, e per certi versi lo è tutt'oggi; ma ora, nell'asserzione moderna di microservizio, c'è un elemento che cambia completamente le regole del gioco, e che rende **essenziale** ciò che prima era **accidentale**, ossia il database. Nei moderni **microservizi** il database è una parte integrante, è una dipendenza interna al **bounded context**, perché sappiamo benissimo che per considerare completamente indipendente un microservizio questi deve avere la sovranità assoluta anche dei dati, in modo da poter liberamente modificare la logica di persistenza senza compromettere il funzionamento del sistema nel suo insieme.

Questo è il vero **game changer**: spostare il database all'interno del perimetro del servizio comporta spostare i problemi relativi ai dati in quelli relativi all'architettura, e l'affermazione "**ignorante alla persistenza**" cambia di significato. Tutto questo porta altra complessità, perché per garantire una totale autonomia fra i vari microservizi la comunicazione fra di loro dovrà essere asincrona, e questo comporta la gestione di un service bus come dipendenza interna al servizio stesso.

Ma se la comunicazione è asincrona, allora anche la gestione del frontend ne risentirà e si dovrà adeguare a questo nuovo pattern; ecco allora che forse, parlare solo di Bounded Context è diventato quantomeno riduttivo; probabilmente ha più senso parlare di **Quantum Architecture**, intendendo con questo termine un pezzo

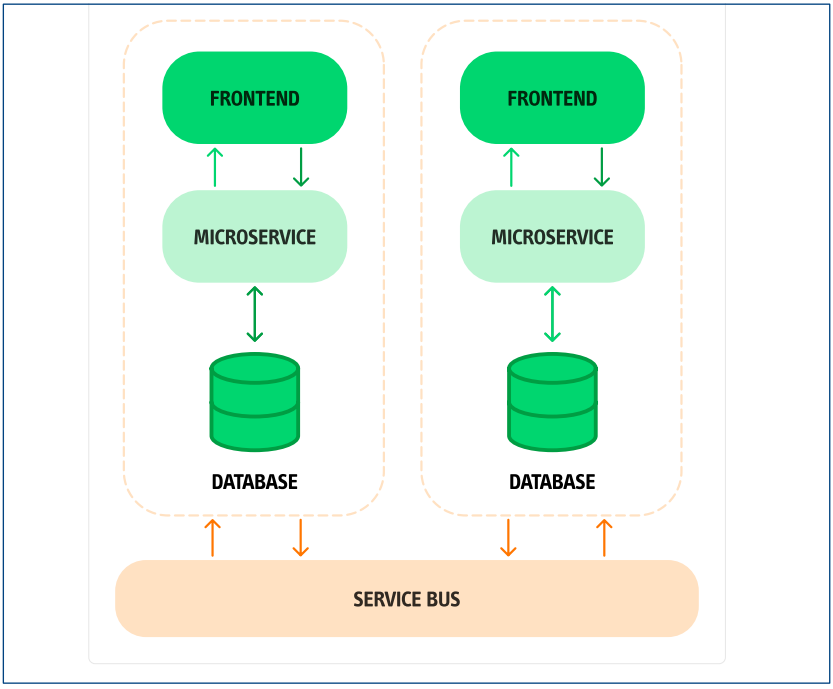

Figura 1.3 – Una "Quantum Architecture".

di architettura che sia veramente deployabile autonomamente, che sia più simile alla rappresentazione di figura 1.3.

A questo punto l'analisi si fa più ampia. Ci troviamo di fronte a due problemi strettamente accoppiati, ma che richiedono approcci diversi e, come detto, all'inizio, un continuo scambio di compromessi per adeguare la nostra soluzione, in modo che possa veramente risolvere i problemi di business per cui ci è stata commissionata.

Da una parte dovremo affrontare i problemi legati proprio al dominio, quelli che **Fred Brooks**, nel suo famoso articolo definisce appunto problemi **essenziali**, e quindi ci dovremo avvalere dei pattern forniti proprio dal Domain-Driven Design; dall'altra, come architetti, dovremo mantenere l'integrità strutturale nella **codebase**, e

affrontare i problemi **accidentali**, e magari verificare se nel frattempo sono diventati **essenziali**.

Da qui

Questo è solo l'inizio del discorso. Dal punto di vista del **dominio** partiremo facendo chiarezza su **problem space** vs **solution space**, e vedremo di capirne le differenze e le modalità di approccio alla questione. Dal punto di vista **architetturale** ci scontreremo con i problemi legati alle dipendenze che esistono all'interno del perimetro del servizio, ossia come comunicano fra di loro i componenti della nostra **Quantum Architecture**, quelle che ricadono nell'ambito delle **Static Coupling,** e delle dipendenze esterne, ossia come comunicano fra loro i vari **quanta** del nostro sistema, ossia le **Dynamic Coupling.**

Avremo come fedele alleato il **Domain-Driven Design**, perché tratta sia i pattern strategici che i pattern tattici, e a noi servono entrambi, ma dovremo andare a bussare anche a qualche altra porta per essere sicuri di realizzare un buon lavoro, per fornire un'applicazione software che risolva veramente il problema del business.

Riferimenti

[1] Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003

[2] Frederick Brooks, *No Silver Bullet. Essence and Accident in Software Engineering*. Proceedings of the IFIP Tenth World Computing Conference, pp. 1069–1076.

`http://worrydream.com/refs/Brooks-NoSilverBullet.pdf`

Capitolo 2

Problem Space vs. Solution Space

Capire i problemi e/o trovare le soluzioni

Tra le tante frasi attribuite in modo apocrifo ad Albert Einstein, c'è quella che dice: "Se avessi a disposizione un'ora per salvare il mondo, utilizzerei 55 minuti per definire il problema e solo 5 minuti per trovare le soluzioni".

Volendo "tradurre" in poche parole questa affermazione, potremmo dire che il **problem space** è l'insieme di tutti i problemi che il nostro applicativo software dovrebbe **risolvere**, mentre il **solution space** è l'insieme di tutte le soluzioni che risolvono ogni dato problema.

Se il mio docente di informatica sosteneva che la scrittura del codice era l'ultima fase — e su questo non concordo pienamente nemmeno oggi — la mia docente di matematica sosteneva che, per risolvere un problema, bisogna prima comprenderlo a fondo: inutile studiare a memoria tutto. Era talmente avanti che ti permetteva di consultare il libro durante l'esame di analisi; ma, nonostante questo, passare il suo esame era un incubo!

Problem space

Come esseri umani abbiamo la tendenza a essere dei *problem solver*: così sostengono Allen Newell e Herbert Simon nel loro libro *Human Problem Solving* [1], e questo lo vediamo quotidianamente nel lavoro di team mentre sviluppiamo software. Abbiamo la tendenza a cercare, e a trovare, una soluzione il più velocemente possibile; tendiamo a uscire dal **problem space** nel minor tempo, come se ci trovassimo a disagio stando in quella zona. Il **problem space** è il luogo in cui risiedono le necessità dei nostri clienti, è la zona che ci permette di acquisire maggiori informazioni sui problemi che i nostri clienti ci chiedono di risolvere tramite il software che svilupperemo per loro. Essenzialmente, questo spazio costituisce le fondamenta sulle quali poggerà lo spazio delle soluzioni. Restando nell'ambito della matematica, il **problem space** è sempre un sottoinsieme del **solution space**: quindi, non dobbiamo avere fretta di abbandonare il primo per approdare, con una scarsa conoscenza del dominio, nel

secondo. Non dobbiamo sorprenderci se, come sviluppatori, tendiamo a fornire una soluzione in fretta: è la nostra natura che ci porta ad agire in questo modo.

E allora non ci importa se non utilizziamo a sufficienza del tempo per capire a fondo il problema, per restare a lungo fuori dalla nostra zona di comfort, col risultato di produrre dei requisiti incompleti per la soluzione completa del problema. Quello che importa è che il risultato così ottenuto sarà sempre incompleto o, peggio ancora, sbagliato; ed è in questo spazio che iniziano i fallimenti dei nostri progetti software.

Restiamo nel Problem Space

Come mi piace spesso ripetere, realizzare software significa risolvere problemi di business e, trattandosi di problemi, questa volta concordo pienamente con la mia docente di matematica: bisogna comprenderli a fondo per poterli risolvere.

Non fornire una soluzione adeguata sin dall'inizio significa partire zoppi, e sappiamo tutti che le nostre applicazioni nascono piccole, ma sono destinate a crescere nel tempo, alcune anche molto in fretta: man mano che cresceranno le richieste del cliente, una piccola deviazione all'inizio del percorso ci porterà a una meta diversa da quella immaginata, perché sarà sempre più difficoltoso adeguare la nostra soluzione alla soluzione del problema stesso.

Come dice E. Evans, man mano che procediamo, come **team**, nello sviluppo dell'applicazione, ci avviciniamo sempre più ai **Business Expert** come conoscenza del dominio, alla comprensione del problema, e questo ci porterà a trovare soluzioni decisamente più adeguate; ma, se il punto di partenza diverge troppo dalla giusta direzione, sarà difficile, se non impossibile, fare coincidere le nuove soluzioni con quelle iniziali.

Come possiamo evitare tutto questo? Restando il più a lungo possibile nello **spazio del problema**. Proprio come nella citazione di A. Einstein. Anche lo stesso E. Evans, con termini più vicini al nostro mondo, sostiene che la prima versione del nostro **modello di**

dominio sarà sicuramente sbagliata e che, se riteniamo di aver trovato la soluzione al primo giro, sicuramente abbiamo sbagliato qualcosa, non abbiamo considerato tutti i fattori del problema. Non a caso ci propone un modello esplorativo, fatto di iterazioni continue di analisi, proposte e feedback

Il blocco delle soluzioni precoci

Lo stesso Evans definisce l'azione del trovare una soluzione sin dall'inizio all'intero problema come ciò che in gergo chiamiamo **big front design**: un blocco ("locking in our ignorance"). Il problema

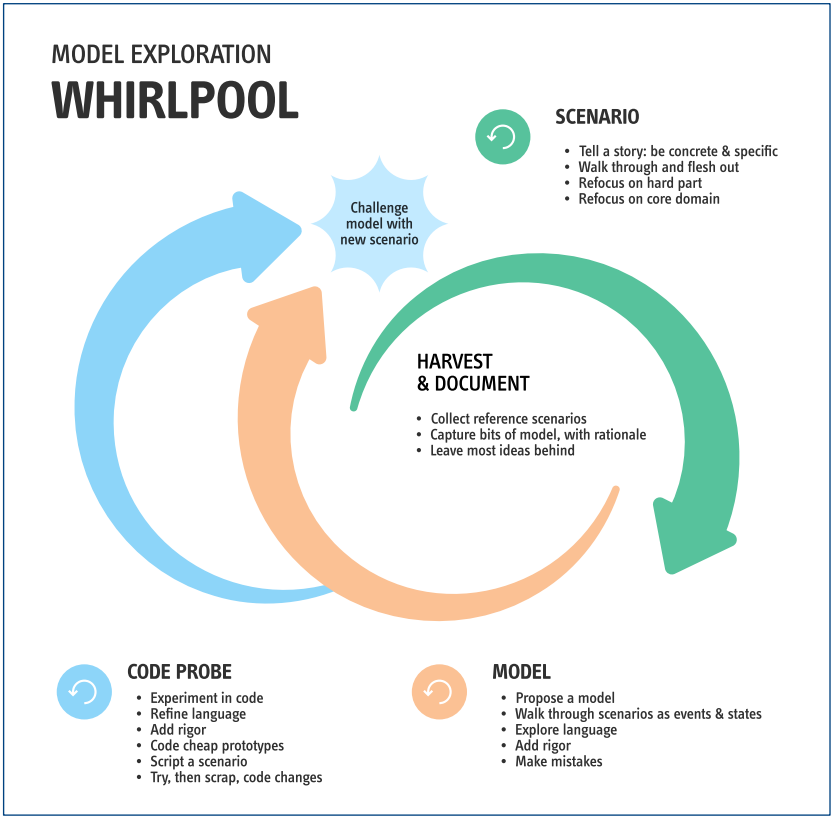

Figura 2.1 –Il modello esplorativo proposto da Evans.

con questo tipo di approccio sta nel fatto che ci troviamo costretti a prendere un sacco di decisioni importanti riguardo la nostra **soluzione** proprio nel momento in cui abbiamo minore conoscenza del problema stesso.

Anche Dan North ci suggerisce massima cautela nel procedere verso una soluzione definitiva. Nel suo articolo *Introducing Deliberate Discovery* [2] suggerisce di mantenere un approccio di mancanza di consapevolezza nei confronti della conoscenza del dominio che stiamo affrontando. La mancanza di consapevolezza, per chi volesse approfondire, è il secondo livello di ignoranza secondo Philip Armour, che proprio a tal proposito, nel 2000, scrisse un articolo sui cinque ordini di ignoranza nell'ambito dello sviluppo software [3].

Qualche indicazione per non trovare soluzioni frettolose

Ma, se l'ignoranza è l'opposto della conoscenza, come possiamo rimanere nello spazio del problema per cercare di colmare la prima e accrescere la seconda? Come si controlla la nostra propensione a proporre una soluzione troppo rapidamente? Possiamo considerare alcuni consigli utili.

Mantenere il controllo della discussione. Non significa che dovete parlare in continuazione, ma che dovete evitare che la discussione converga troppo rapidamente a una conclusione. Spesso un segnale a un vostro collega è sufficiente per far cambiare rotta ai membri del team. Il cliente, o chi per lui, dà per scontato un sacco di informazioni mentre ci racconta del suo problema, e tocca a noi farle emergere in maniera palese nella discussione.

Acoltare le diverse soluzioni. Durante la discussione del problema, vinti i primi momenti di imbarazzo, normalmente emergono molte idee di soluzione, a volte contrastanti fra loro. È importante mantenere un ambiente sano e positivo inserendo i suggerimenti di tutti nello spazio delle soluzioni.

Tenere nascoste le soluzioni principali. Come moderatori dell'esplorazione del problema, il vostro compito è mantenere il più a lungo possibile nascoste le soluzioni che man mano si dimostrano

valide. Se, per esempio, utilizzate EventStorming come tool di esplorazione, lasciate che i partecipanti continuino ad attaccare Post-it sul muro, e magari guidateli verso una direzione, in modo che questa emerga in maniera autonoma dall'esplorazione stessa. Evidenziate le soluzioni che sembrano simili e fate convergere su di esse l'attenzione, per capire se sono modi diversi di esprimere la stessa soluzione, oppure se sono proprio due soluzioni diverse da esplorare.

Conoscere la situazione del cliente

Restare nello **spazio del problema** è sicuramente scomodo per entrambe le parti, cliente e fornitore. Il primo si sente un po' sotto pressione per le innumerevoli domande che gli continuano ad arrivare, il secondo — vale a dire, spesso, noi — non vede l'ora di proporre una soluzione. Affrontare completamente il problema del cliente richiede tempo e non sempre è conveniente insistere più del necessario al primo incontro. L'importante è individuare i confini del problema e cominciare a proporre una soluzione, sulla quale ci si potrà ritrovare per avere feedback e continuare, dopo aver eventualmente aggiustato il tiro.

Il nostro dominio sarà sempre in **continua evoluzione**, se non altro per tutto ciò che appartiene al livello del *unknow unknows*, ossia le cose che ancora non sappiamo di non sapere, ma che scopriremo a mano a mano che diventeremo sempre più esperti del dominio su cui stiamo lavorando. A questo punto dobbiamo capire come si affronta, dal punto di vista architetturale, una soluzione simile.

Evolutionary Architecture

Anche l'**architettura**, così come il software che stiamo implementando, dovrà essere in grado di **evolvere**; è finito il tempo della similitudine fra architettura di un edificio e architettura del software.

Non possiamo permetterci di creare un'applicazione che poi andremo ad abbandonare perché imbrigliata in un'architettura rigida che non consente di adeguarsi alle esigenze del mercato, quali, per esempio, **scalabilità**, **resilienza**, **elasticità**.

Pro e contro dei microservizi e dei monoliti

Se partiamo con una **soluzione** a **microservizi**, quasi certamente troveremo facilmente risposta a questi problemi, ma dovremo pagare lo scotto di portarci in casa parecchia **complessità** accidentale all'inizio del progetto stesso. I microservizi comportano infatti:

- decomposizione del database
- workflow distribuiti
- transazioni distribuite
- automazione dei processi
- distribuzione tramite container e orchestratori

Quelle appena citate sono tutte questioni che comportano anche oneri economici non indifferenti, per un progetto che potrebbe continuare come anche interrompersi per svariate ragioni.

Una soluzione che contempli il buon vecchio **monolite** sembra più abbordabile; il monolite è infatti

- facile da implementare
- facile da testare
- facile da distribuire
- veloce da realizzare

Ovviamente, oltre a questi pregi, ha i suoi bei **difetti**, guarda caso, esattamente quelli che noi vorremmo evitare e che abbiamo citato sopra.

Architettura modulare

Negli ultimi anni, una nuova soluzione architetturale sta prendendo piede: la **Modular Architecture** è una via di mezzo tra microservizi e monolite. Lo scopo di questa architettura è creare una soluzione **production ready** a tutti gli effetti, non un semplice **PoC** (*Proof of Concept*); l'idea di base è individuare i moduli all'interno della soluzione, e, per farl, abbiamo a disposizione gli **Strategic Pattern** del **DDD**, quali **Bounded Context**, **Context Mapping** e, ovviamente, **Ubiquitous Language**. Spesso questi pattern vengono associati ai **microservizi**, proprio perché ci aiutano a definire i confini delle **singole responsabilità** di ogni sotto dominio del nostro sistema.

Contrariamente ai microservizi però, in questo caso non si progettano soluzioni diverse per ogni **Bounded Context** indivuato, ma bensì un modulo all'interno di un monolite. Questo ci servirà nel caso il nostro confine iniziale non sia stato individuato proprio correttamente, e avrà bisogno di qualche aggiustamento; dover fare queste operazioni di sistemazione del codice all'interno di un monolite è sicuramente più semplice che non dover rimettere mano a diversi microservizi.

Conclusioni

Abbiamo visto in questo capitolo che rimanere nello "spazio dei problemi", per quanto scomodo e tendenzialmente innaturale, ci aiuta a entrare nello "spazio della soluzione" con una migliore conoscenza del dominio e una maggiore capacità di assumere le decisioni giuste. Abbiamo anche parlato brevemente di vantaggi e svantaggi delle soluzioni architetturali a monolite e a microservizi, introducendo il concetto di **architettura modulare**.

Vedremo nel prossimo capitolo cosa si intende esattamente per **modulo**, e quali vantaggi ci porta una soluzione architetturale simile.

Riferimenti

[1] A. Newell – H. Simon, *Human Problem Solving*. Echo Point Books & Media, 2019

[2] Dan North, *Introducing Deliberate Discovery*
`https://dannorth.net/introducing-deliberate-discovery/`

[3] Phillip G. Armour, *Five Orders of Ignorance. Viewing software development as knowledge acquisition and ignorance reduction*. "Communications of the ACM", October 2000, vol. 43, n. 10, pp. 17–20
`https://cacm.acm.org/magazines/2000/10/7556-the-five-orders-of-ignorance/abstract`

Capitolo 3

Pattern strategici

Architettura modulare

Abbiamo visto come i microservizi rappresentino spesso un costo eccessivo per un nuovo progetto, o anche per il refactor di un vecchio progetto, e come il monolite può rappresentare un limite alla scalabilità, qualora il nostro sistema ne avesse bisogno.

Una soluzione che sembra contemplare i vantaggi dell'una e dell'altra architettura è la *modular architecture.* Prima di capire come impostare un progetto seguendo questo tipo di architettura vediamo di capire cosa si intende per *modulo* e come questo sia conforme ai pattern espressi dal Domain-Driven Design.

Module

Un modulo rappresenta un insieme di oggetti che esprimo e risolvono un concetto di business. Questi oggetti sono altamente coesi fra loro, il che significa che esiste un alto grado di accoppiamento,

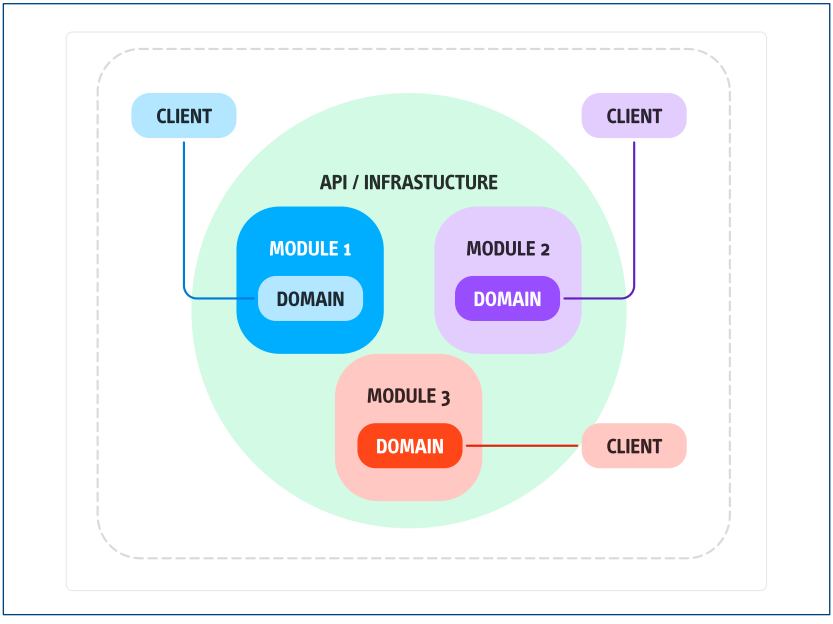

Figura 3.1 – Module.

ma in questo caso non rappresenta un problema, perché non andremo mai a separare questi oggetti gli uni dagli altri. L'importante però è che ogni modulo sia indipendente dagli altri che risolvono altri problemi di business, legati sì allo stesso nostro sistema, ma in un altro *contesto*. Quindi è fondamentale che sia un basso livello di accoppiamento fra i vari moduli del nostro monolite, in modo da permetterci in futuro, se necessario, di suddividerli in altrettanti microservizi, senza dover riprogettare l'intero sistema.

Modular Architecture e Domain-Driven Design

A questo punto è lecito chiedersi come questa architettura e il DDD si intrecciano, ed allora dobbiamo fermarci un attimo e vedere quali pattern offre DDD e quali stiamo applicando in questo specifico contesto.

Eric Evans, nel suo iconico libro, presenta diversi pattern per lo sviluppo del software orientato al business, più che al dato, e li suddivide in due grandi categorie: **Pattern strategici** e **Pattern tattici**. I primi ci servono per disegnare quella che ormai tutti conosciamo come la *Big Picture* del nostro sistema, e ci aiutano a capire come suddividere l'intero **Dominio** in tanti **sottodomini**.

Business Domain & Ubiquitous Language

Che cos'è un Dominio? Intanto togliamoci il dubbio principale, non è necessariamente qualcosa legato al mondo del software, anzi, nella maggior parte dei casi non lo è affatto! Il dominio definisce l'area di attività principale di un'azienda, è il servizio che l'azienda fornisce ai suoi clienti, quella che la rende unica di fronte ai suoi concorrenti. Prendiamo ad esempio Uber, il suo dominio è fornire un servizio di trasporto privato facile da accedere e da consumare, sia per chi lo fornisce, sia per chi ne usufruisce. Non è un problema prettamente legato al software, ma è grazie ad un'app ben progettata che noi ne possiamo cogliere il suo valore … ovviamente se siamo all'estero, ma questa è un'altra storia. Volendo ricorrere alla definizione che spesso si trova di dominio legata al Domain-Driven

Design possiamo dire che "*il dominio è la sfera di conoscenza e attività attorno alla quale ruota la logica applicativa*".

Com'è possibile far combaciare un problema di business, con un buon sviluppo software, sapendo che appartengono a due mondi completamente diversi fra loro? Sapendo che hanno due modelli rappresentativi totalmente diversi? L'idea di Evans è stata proprio quella di creare un modello comune in cui entrambe le parti potessero trovarsi per trovare possibili soluzioni al problema, un modello descritto con termini comprensibili sia dai tecnici che dovranno sviluppare il software, sia dagli stakeholder, dal Cliente e dai suoi utilizzatori, in modo da non creare fraintendimenti nella comunicazione che possono dare origine a soluzioni divergenti.

Un modello non è la copia del mondo reale, come spesso noi sviluppatori tendiamo a sottintendere, ma è un costrutto che ci aiuta a dare un senso ad un dominio complesso; tanto più questa copia è vicina alla realtà, tanto più le decisioni che prenderemo basandoci su di esso saranno corrette e contribuiranno a risolvere il problema. Per poter arrivare ad avere un modello comune fra il mondo del software e quello del business è necessario che entrambi si intendano sui termini utilizzati, senza che ci sia ambiguità sul significato e sul comportamento di ciò che esprimono; questo linguaggio è il primo pattern strategico proposto nel *Blue Book*, ed è l'Ubiquitous Language.

Ubiquitous Language

Durante il tradizionale processo di sviluppo di un'applicazione software il linguaggio utilizzato dagli esperti del dominio, che rappresenta la conoscenza del dominio stesso, viene "traslato" in un linguaggio tecnico più comprensibile agli sviluppatori, in quella che viene definita come la descrizione dei requisiti del sistema, che essendo appunto il frutto di una traslazione, non rappresenta la comprensione del business e dei suoi problemi. Ogni volta che trasliamo il significato di un termine da un contesto ad un altro perdiamo parte del suo significato, non solo semantico, ma anche

comportamentale. L'intenzione, intendiamoci, è buona, ma non il risultato, e ciò che otteniamo è un modello impoverito della realtà, che non potrà che portare a soluzione software incomplete, nelle migliori delle ipotesi, sbagliate nelle peggiori. Trovare un linguaggio comune, che esprima senza ambiguità cosa intendiamo per *caffè*, ad esempio, è la chiave per un software di successo. Perché ho utilizzato il termine *caffè?* Perché quando, da buoni italiani, andiamo all'estero ed ordiniamo un caffè, noi già sappiamo che il barista ci ha capiti sul termine, ma non certo sul suo più profondo significato. Per noi il caffè dev'essere ristretto, forte al punto giusto, servito in una tazzina, e per i più accaniti amanti, amaro. Invece il barista ci prepara una tazza alta cinque centimetri, piena di una bevanda dal colore del nostro amato caffè, ma di tutt'altra sostanza. Non è solo il nome che contribuisce a creare l'ubiquitous language, non si tratta di un dizionario, ma di un linguaggio vero proprio, rigoroso quanto un linguaggio tecnico, ma comprensibile a tutti.

Quando diciamo che bisogna restare il più a lungo possibile nel *problem space* per cercare di comprenderlo sino in fondo, intendiamo proprio questo; riuscire a descriverlo con parole che tutti noi, che saremo chiamati a risolverlo, lo sapremo descrivere in un linguaggio chiaro a tutti, senza ambiguità! Ma allora è sufficiente, per noi sviluppatori, imparare il linguaggio del Dominio ed il gioco è fatto. No!

Domain Language vs Ubiquitous Language

In molte interpretazioni del Domain-Driven Design spesso il linguaggio del dominio, il *Domain Language* è confuso con *l'Ubiquitous Language*.

Il *domain language* è un linguaggio naturale, utilizzato dalle persone che quotidianamente lavorano nell'ambito del dominio stesso. Come recita un brano di Giorgio Gaber

"Le parole definiscono il mondo, se non ci fossero le parole, non avremmo la possibilità di parlare di niente. Ma il mondo gira, e le parole stanno ferme, si

logorano, invecchiano, perdono di significato, perdono di senso, e tutti noi continuiamo ad usarle, senza accorgerci di parlare di niente".

Sicuramente non ricorderemo Gaber per il suo contributo allo sviluppo del software, ma il concetto espresso è molto chiaro a noi che ogni giorno ci dobbiamo confrontare con chi il software ce lo ha commissionato.

Un linguaggio di questo tipo, per noi che siamo abituati ad essere rigorosi quando utilizziamo i nostri linguaggi di sviluppo preferiti, è un grosso problema. A noi serve un linguaggio che esprima chiaramente i concetti, che renda esplicito tutto ciò che può essere ritenuto implicito, e questo linguaggio è proprio l'Ubiquitous Language. Un linguaggio costruito e formalizzato da stakeholders e designers per rispondere alle esigenze della progettazione del nostro sistema. Il livello di precisione e di formalità adottati in questo linguaggio dipendono, ovviamente, dal contesto in cui ci stiamo muovendo, progettare un'applicazione per l'acquisto e la vendita di titoli finanziari è diverso dallo sviluppare un'applicazione a supporto di decisioni riguardanti la salute di un paziente.

L'Ubiquitous Language deve essere preciso e coerente. Deve eliminare la necessità di fare ipotesi e rendere esplicita la logica del dominio aziendale. Non si tratta solo di chiarirci sulle proprietà di un oggetto, ma anche sui suoi comportamenti. L'ambiguità ostacola la comunicazione, per questo motivo ogni termine dell'ubiquitous language deve avere un significato unico.

Linguaggi specifici per determinati domini

Ora che abbiamo chiarito cos'è l'Ubiquitous Language proviamo a metterlo in pratica.

Rebecca Wirfs-Brock afferma

"A model is a simplified representation of a thing or phenomenon that intenionally emphasizes certain aspects while ignoring others. Abstractions with a specific use in mind".

Se volessimo enfatizzare maggiormente il concetto potremmo vederla come George Box, il quale affermava che

"All models are wrong, but some are useful".

Quello che voglio dire è che non esiste un modello universale e nemmeno un linguaggio universale per l'intero dominio, ma solo linguaggi specifici per quel determinato problema di business. L'esempio classico è la mappa; ne esistono di diversi tipi, ognuno per soddisfare uno specifico bisogno (fig. 3.2).

È chiaro che nessuna di queste mappe è in grado di rappresentare il pianeta Terra, viceversa, ognuna di esse contiene solo ed esclusivamente i dettagli necessari al suo scopo, al problema che deve risolvere, ed ovviamente, per ognuna di esse, adotteremo un linguaggio specifico, adatto al caso.

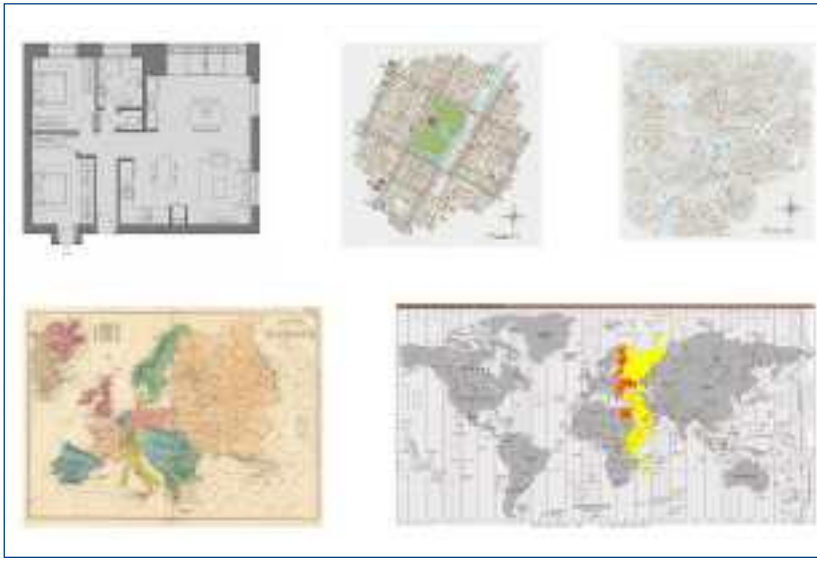

Figura 3.2 – Mappe a scale diverse e con tematismi differenti soddisfano una specifica necessità di visualizzazione.

Abbiamo bisogno di creare modelli astratti della realtà perché solo tramite l'astrazione riusciamo a gestire la complessità, omettendo appunto i dettagli non necessari.

Cosa ha a che fare questo con l'ubiquitous language? Quando costruiamo il nostro linguaggio specifico per il problema che dobbiamo risolvere, stiamo effettivamente costruendo un modello del dominio aziendale, precisamente di un sotto dominio aziendale. L'intento, ripeto, è proprio quello di carpire i modelli mentali degli esperti di dominio, i loro processi per implementarli nella nostra soluzione software. Il linguaggio, e di conseguenza il modello, devono riflettere le entità aziendali coinvolte ed i loro comportamenti, le relazioni che intercorrono fra di esse, le cause e gli effetti e gli invarianti. Ma di questo parleremo in maniera più approfondita nel seguito.

Un altro aspetto fondamentale, proprio come ricordava Gaber, è che anche l'ubiquitous language, essendo un linguaggio, non è statico, ma evolve. Le ragioni sono infinite: la prima il mercato stesso, che si adegua alle nuove esigenze, plasma il significato dei termini per restare nel presente; pertanto, è fondamentale mantenere un'interazione continua con gli esperti di dominio, ed evitare sempre le supposizioni. Rafforzare il linguaggio, e la sua comprensione, significa abbattere il debito tecnico che abbiamo nei confronti del dominio e realizzare soluzioni sempre più appropriate. Una volta stilato un linguaggio comenu, questo deve essere utilizzato in tutti gli artefatti del progetto, dalla documentazione, al codice, proprio per evitare la nascita di ambiguità o fraintendimenti.

A questo punto dovrebbe essere chiaro che una volta trovato l'ubiquitous language del nostro dominio, non ci resterà che suddividerlo in tanti piccoli linguaggi, ognuno specifico per un determinato contesto del business, ed avremo trovato il modo per suddividere il dominio in tanti sottodomini, in tanti *Bounded Context,* o se preferite, visto da dove siamo partiti, in tanti *Moduli.* Come individuare questi confini? Trovando i punti in cui uno stesso termine rappresenta lo stesso oggetto, ma dal punto di vista di due contesti differenti. Ad esempio, posso parlare di un *articolo* dal punto di

vista della logistica, quindi mi interessano dove verrà ubicato, i limiti di scorta e riordino, e cose simili, mentre, sempre lo stesso *articolo*, per un commerciale, avrà bisogno di altre proprietà come il prezzo di vendita, il margine di guadagno, i tempi di consegna e cose simili. Stiamo parlando dello stesso oggetto, ma da due *Bounded Context* diversi.

Bounded Context

Lo abbiamo già detto in precedenza, il modello che utilizziamo per sviluppare il nostro sistema non è una copia della realtà, ma un costrutto che ci supporta a dare un senso a un sistema complesso. Questo modello, è giusto chiarirlo sin da subito, non potrà andare bene per sempre, per tutta la durata del nostro sistema. Le esigenze cambieranno, nuove richieste arriveranno dal mercato e nessuno, all'inizio del progetto, le può prevedere. Non le possiamo prevedere noi, come sviluppatori, basandoci "sulla nostra esperienza" perché ogni progetto è una storia a sé stante, e non le può prevedere il cliente che ci chiede di realizzare il software. Inutile aggiungere proprietà a un modello solo perché pensiamo che prima o poi serviranno. Questa è solo una brutta abitudine che abbiamo noi che scriviamo software, e tanto più siamo *senior*, tanto più ci sentiamo autorizzati a farlo.

No! Tutte le buone pratiche di sviluppo ci insegnano che dobbiamo fare il minimo indispensabile per risolvere il problema attuale; tutto il resto, tutto ciò che va oltre, è solo un accoppiamento verso un modello, sbagliato, che prima o poi pagheremo caro! Quindi?

Quindi rassegniamoci: il modello universale, la *Silver Bullet*, non esiste. Un modello non può esistere senza dei confini ben precisi, oltre i quali la rappresentazione semplificata della realtà, per cui era stato progettato, non è più valida. Con termine mutuato dagli studi fisolofici, è stato definito *Platonic fold*.

"La 'piega platonica' è il confine esplosivo dove il modello platonico entra in contatto con la realtà disordinata, dove il divario tra ciò che si sa e ciò che si pensa di sapere diventa pericolosamente ampio".

I *Bounded Context* definiscono l'applicabilità del sottoinsieme del nostro ubiquitous language e del modello che rappresenta. Ci permettono di definire modelli distinti in base a diversi domini di problemi. La terminologia, i principi e le regole di business espresse da un linguaggio, sono coerenti ed hanno valore, solo ed esclusivamente all'interno di questo contesto delimitato. Proprio come nell'esempio dell'articolo citato precedentemente.

Bounded Context vs. sottodomini

Merita una precisione, a mio avviso, la distinzione fra Bounded Context e **sottodominio**. Apparentemente possono sembrare la stessa cosa, ma nuovamente, la distinzione è fra il concetto di business e la progettazione del sistema.

Per comprendere la strategia di business di un'azienda, abbiamo detto più volte, dobbiamo analizzare il suo dominio, e secondo i pattern espressi dal Domain-Driven Design, questa analisi prevede l'identificazione di diversi sottodomini. Così facendo scopriamo come l'azienda lavora e pianifica la propria strategia di mercato.

I Bounded Context sono la rappresentazione, nel nostro sistema, di questo modello, vengono progettati da noi sviluppatori e rappresentano una decisione strategica, ed infatti appartengono ai pattern strategici del DDD. Decidiamo come dividere il modello di dominio aziendale in tanti piccoli modelli meno problematici da gestire. Decidere se suddividere ogni sottodominio in uno, o più, Bounded Context, dipende dalla complessità del modello.

Avere una relazione uno-a-uno tra bounded context e sottodomini può essere perfettamente ragionevole in alcuni scenari, mentre in altri possono essere adatte diverse strategie di decomposizione. Nell'architettura del software, purtroppo o per fortuna, non esiste una regola, ma tutto è un compromesso, tutte le decisioni che prendiamo hanno vantaggi e svantaggi, sta a noi decidere quale strategia scegliere in base alla situazione. Non è importante **come** progettiamo una soluzione, ma **perché** scegliamo una strategia piuttosto di un'altra.

Laws of Software Architecture

First law: Everything in software architecture is a trade-off

Corollary 1: If an architect thinks they have discovered something that is not a trade-off, more likely they just have not yet identified the trade-off

Second law: "Why" is more important than "how"

Tipi di sottodominio

Progettare un sistema orientandoci sul business aziendale comporta, così come nella realtà, la distinzione dei tipi di sottodominio identificati in fase di esplorazione. In un'azienda non tutti i reparti sono ugualmente strategici: dipende dal tipo di business dell'azienda stessa, e così dovrà essere anche nel nostro sistema software. In Domain-Driven Design vengono classificati tre tipi di sottodominio.

Core subdomains

È il tipo di sottodominio che differenzia l'azienda dalle altre, dove, strategicamente, si cerca di distinguersi sul mercato; conseguentemente, a livello di sviluppo, qui si concentreranno gli sforzi maggiori, si applicheranno tutti i pattern della buona programmazione e, molto probabilmente, il team sarà formato dai migliori elementi. La spiegazione è semplice: un *core subdomain* facile da implementare può fornire un vantaggio competitivo di breve durata, motivo per cui, questo tipo di sottodomini, sono naturalmente complessi.

Generic subdomains

In questa categoria ricadono tutti i sottodomini che tutte le aziende sviluppano allo stesso modo e che non portano nessun vantaggio competitivo al business dell'azienda. Esempi tipici sono i meccanismi di autenticazione e autorizzazione, per cui vale spesso la pena affidarsi a provider esterni, ampiamente testati.

Supporting subdomains

Questa tipologia di sottodomini, così come suggerito dal nome stesso, servono a supporto del business. Non comportano alcun

vantaggio competitivo, ma difficilmente si possono acquistare da uno scaffale; perciò, sono spesso implementati internamente.

Integration Patterns

Abbiamo visto come la suddivisione in sottodomini di un modello di business ci consenta di ottenere diversi modelli che possono evolvere indipendentemente gli uni dagli altri, al di là del fatto che ci lavori un solo team o diversi team. Questa proprietà dei Bounded Context è ciò che li ha eletti a pattern per la definizione dello scopo di un microservizio. Detto questo i bounded context non sono di per sé indipendenti.

Esattamente allo stesso modo per cui un sistema non è l'insieme dei singoli componenti, ma il modo in cui questi sono assemblati e interagiscono fra loro, allo stesso modo il nostro software non è semplicemente l'insieme dei bounded context che abbiamo identificato, ma il modo in cui questi comunicano, e si scambiano informazioni fra loro. Di conseguenza ci saranno sempre dei punti di contatto tra un bounded context e un altro, e questi punti di contatto, così come nella vita reale, saranno governati da *contratti*.

La ragione per cui abbiamo bisogno di definire dei contratti per far dialogare fra loro diversi bounded context è semplice: in ogni bounded context è valido uno specifico ubiquitous language, che non è valido per gli altri, altrimenti non li avremmo divisi. Nel momento in cui sorge la necessità di integrare delle informazioni, è fondamentale decidere quale linguaggio adottare per l'integrazione stessa.

Nel Domain-Driven Design questa problematica è affrontata dal pattern strategico *Context Mapping*, che suddivide le tipologie di comunicazione in tre gruppi, che rappresentano il tipo di collaborazione in essere: **Cooperation**, **Customer-supplier**, **Separate Ways**.

Cooperation

Questo tipo di collaborazione si riferisce a bounded context che hanno una comunicazione ben consolidata, dove il successo dell'uno dipende da quello dell'altro e viceversa.

In questo gruppo troviamo i seguenti pattern.

Partnership

In questo modello l'integrazione è coordinata in modo ad hoc. Un team notifica ad un altro le modifiche apportate al proprio modello ed il secondo si adatterà, senza drammi o conflitti, alle nuove modifiche. L'integrazione, in questo caso, è a doppio senso, non esiste un team, o in generale, un bounded context, che detta le regole. Ognuno lavora in maniera indipendente dall'altro, ma in perfetta armonia per quanto riguarda lo scambio di informazioni. Questo tipo di collaborazione richiede una costante sincronizzazione ed un alto livello di comunicazione, motivo per cui è valido se i team sono vicini, geograficamente parlando, fra loro.

Shared Kernel

Quando una parte di un sottodominio è comune a più sottodomini allora è conveniente creare un modello condiviso fra più bounded context. È fondamentale accettare il fatto che il modello condiviso deve essere consistente attraverso tutti i bounded context interessati; questo rappresenta sicuramente un accoppiamento, e come sempre, bisogna valutarne pro e contro prima di implementarlo. Esistono delle linee guida per aiutarci a prendere questa decisione che possiamo riassumere in questo breve elenco.

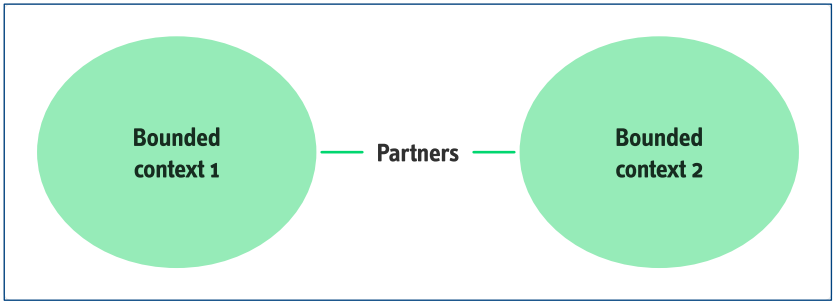

Figura 3.3 – Partnership.

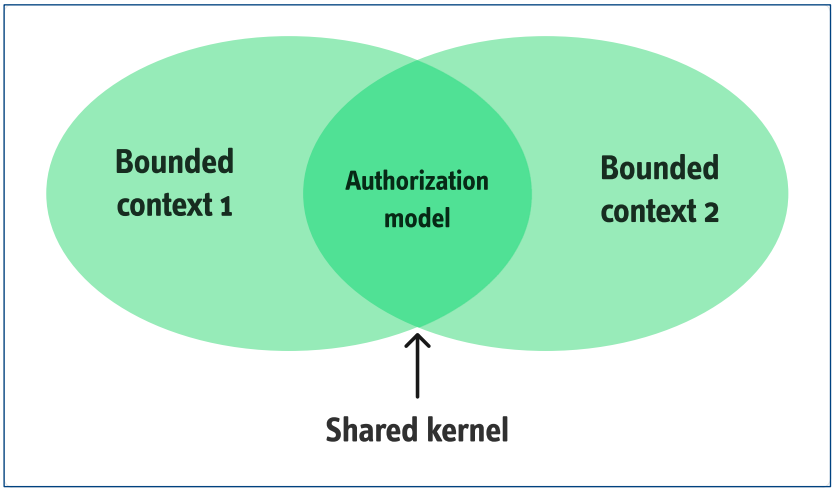

Figura 3.4 – Shared Kernel.

I consumatori del modello condiviso

- Perderanno delle capacità?
- Acquisteranno delle capacità?
- Potranno concentrarsi maggiormente sulle loro scelte strategiche?
- Saranno rallentati da questa dipendenza?
- Potranno rifiutare la migrazione al modello condiviso?
- Il team di sviluppo del modello condiviso sarà reattivo?
- Il numero di modelli dipendenti sarà problematico?
- Il costo di migrazione sarà eccessivo?

Come al solito, non esite una legge, ma dipende dal contesto.

Customer-Supplier

Il secondo gruppo di pattern di collaborazione prende il nome di *customer-supplier,* e come evidenziato nell'immagine, uno dei bounded context, il *supplier*, fornisce un servizio per il suo *customer*. In questo gruppo, a differenza del precedente che prevedeva una collaborazione, entrambi i team (*upstream e downstream*) possono avere successo indipendentemente l'uno dall'altro. Per questo motivo

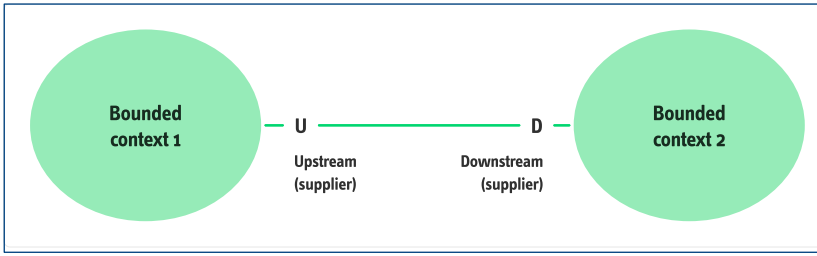

Figura 3.5 – Customer Supplier.

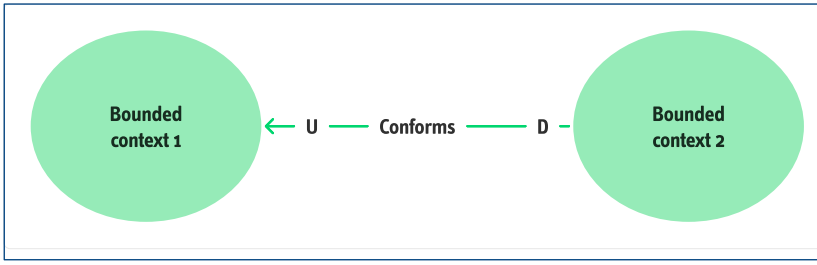

Figura 3.6 – Conformist.

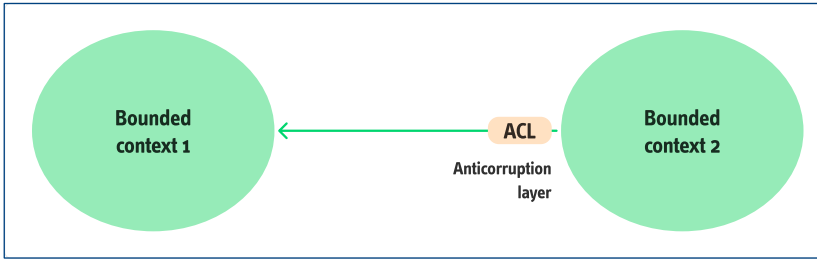

Figura 3.7 – Anticorruption Layer.

spesso si verifica uno squilibrio di potere ed uno dei due team può dettare le regole per il contratto di integrazione. I pattern che appartengono a questo gruppo sono riportati nei paragrafi che seguono

Conformist

In alcuni casi è il team upstream a non avere alcuna motivazione per supportare i team downstream, e si limita a fornire un contratto di integrazione che agli altri non resta che accettare (prendere o lasciare). È il caso in cui ci dobbiamo integrare con qualche servizio esterno, su cui non abbiamo nessun potere di contrattazione. Se il team downstream può accettare il contratto di integrazione così come fornito, allora il rapporto è definito *conformist.*

Anticorruption Layer

Anche in questo è il team *upstream* a forzare le regole del contratto di integrazione, ma a differenza del pattern conformist in questo caso chi si trova a valle, in posizione *downstream*, non è in grado di accettare il contratto per svariate ragioni: cambiamenti frequenti, inefficienza nella gestione del contratto, protezione del bounded context che si trova in downstream. L'*Anticorruption Layer funge da interprete* fra il contratto esterno ed il nostro bounded context.

Open-Host Service

Anche in questo caso il potere decisionale è sbilanciato verso chi sta a monte, ma a differenza del pattern precedente, è proprio chi sta a monte che protegge i propri consumatori separando l'implementazione interna del contratto da quella esterna. Questo disaccoppiamento consente al bounded context a monte di evolvere il proprio modello di implementazione e quello esterno di integrazione indipendentemente.

Può succedere di dover mantenere più di una versione del contratto esterno, per dare il tempo ai vari consumatori di adeguarsi alle nuove specifiche. In questo caso il bounded context a monte può esporre più versioni del contratto di integrazione. Un esempio

classico è rappresentato dall'esposizione di API REST da parte di un Bounded Context, delle quali possono esistere più versioni (fig. 3.9).

Separate Ways

Esiste un terzo gruppo di collaborazione, ed è quello che non prevede collaborazione affatto. Vediamo quali sono i casi che rientrano in questo gruppo.

Communication Issues

Quando i team hanno difficoltà a comunicare oppure a collaborare, anche per ragioni politiche interne all'organizzazione, può essere conveniente che ognuno vada per la propria strada, e quindi si preferisca duplicare alcune funzionalità in più bounded context.

Generic Subdomains

Esistono funzionalità che sono più facili da duplicare che non da integrare, soprattutto nei sottodomini generici, che normalmente sono molto facili da sviluppare, oppure da acquistare ed integrare. Il logging è una di queste funzionalità.

Model Differences

Anche in questo caso, in cui le differenze fra i modelli di diversi bounded context sono talmente evidenti, conviene duplicare le funzionalità piuttosto che investire tempo nell'integrazione. Non è consigliabile prendere questa strada nel caso i sottodomini siano dei *core subdomains* e la ragione dovrebbe essere abbastanza chiara: duplicando funzionalità a questo livello si rischierebbe di andare in contrasto con la strategia aziendale che tende ad ottimizzare il processo concentrando gli sforzi maggiori proprio a questo livello.

Context Map

Il *context mapping* è una rappresentazione visiva dei bounded context a delle relazioni che intercorrono fra di loro, e ci fornisce preziose indicazioni strategiche. Ad esempio, ci fornisce una panoramica

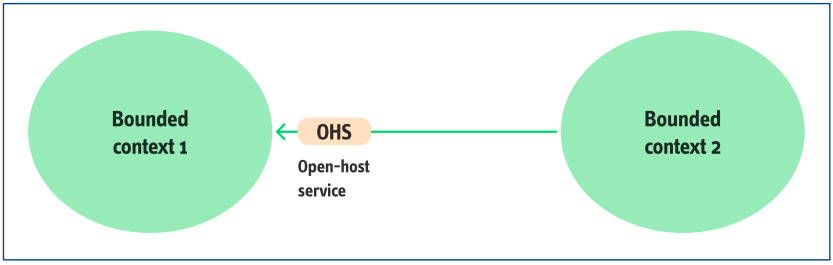

Figura 3.8 – Open Host Service.

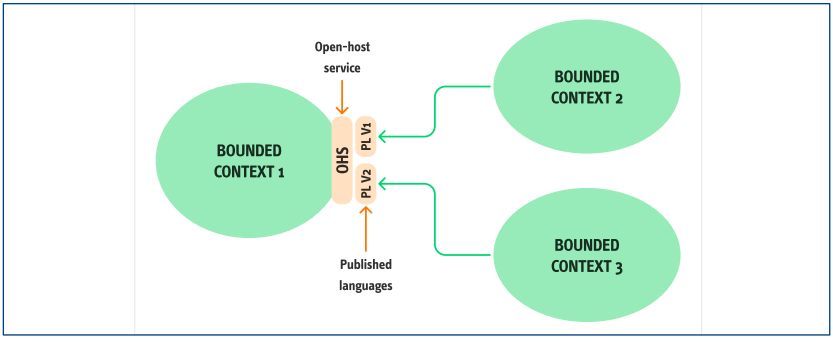

Figura 3.9 – Open Host Sevice con due Published Languages diversi.

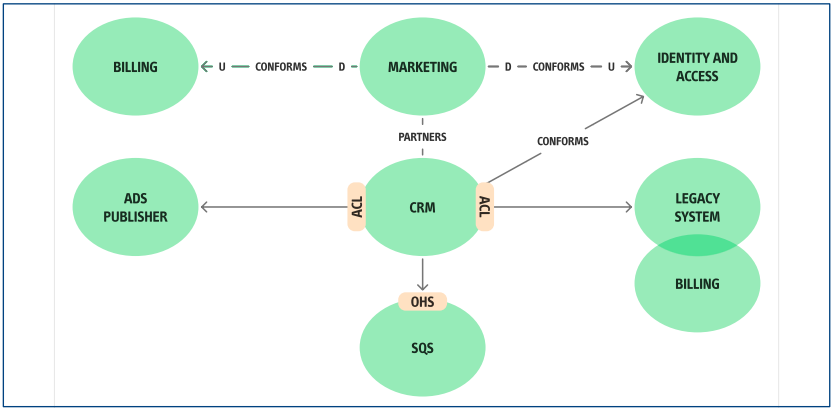

Figura 3.10 – Context Mapping.

dei componenti e dei modelli che implementano (*High-level design)*, ma anche il modello di collaborazione in essere fra i vari team, evidenziando quelli che collaborano e quelli che preferiscono prendere strade autonome (*Communication patterns),* infine, ma non certo meno importante, ci fornisce uno specchio dei problemi organizzativi che esistono all'interno dell'organizzazione (*Organizational issues)*, quando, ad esempio, ci accorgiamo che tutti i rapporti di collaborazione sono di tipo *Separate Ways* o *Anticorruption Layer.*

Conclusioni

Partendo dal pretesto della spiegazione dell'architettura modulare abbiamo visto come Domain-Driven Design ci supporta nell'aspetto strategico dello sviluppo del software. Spesso l'applicazione di questi pattern è vista come la strada per la realizzazione di sistemi a microservizi, ma come abbiamo visto non è scritto da nessuna parte che questo debba essere necessario. Come spesso ripeto, nel 2003, anno di pubblicazione del libro di Eric Evans, non solo non si parlava di microservizi, ma l'unica architettura conosciuta era la *Layer Architecture,* eppure tutti i pattern visti qui erano applicabili.

Un esempio pratico di come applicare questi concetti ad un software reale lo potete trovare in Modular Architecture [1].

Riferimenti

[1] Modular Architecture
`https://github.com/BrewUp/BrewUp.ModularArchitecture`

Capitolo 4

Evolutionary Architecture

Architettura evolutiva

Che cos'è **un'architettura evolutiva**? Troppo spesso l'architettura del nostro applicativo software viene paragonata all'architettura di un edificio. Come a dire, se non hai le giuste e solide fondamenta, non puoi costruirci sopra quello che vuoi; devi decidere prima di partire con i lavori se intendi costruire una casa su di un solo piano, oppure un condominio, oppure ancora un capannone. Non puoi partire con una casetta e finire con un edificio a 5 piani!

Allo stesso modo ci sentiamo ripetere questo discorso all'inizio di un progetto. Il mio applicativo dovrà essere in grado di gestire migliaia di connessioni contemporaneamente, ma non mi dovrà costare un patrimonio in risorse nei giorni in cui nessuno lo usa. Insomma, dovremmo progettare una casetta, ma all'occorrenza trasformarla in un grattacielo, e poi, nuovamente in una casetta se tutti escono. È possibile?

La risposta ovviamente è sì, altrimenti di cosa staremmo parlando? Una *Evolutionary Architecture* è un tipo di architettura che supporta il cambiamento **guidato**, **incrementale** e soprattutto **su più dimensioni**. Detto così vuol dire tutto e niente, ma proviamo ad analizzare le proprietà che abbiamo appena elencato e vediamo che soluzioni potremmo mettere sul tavolo.

Per proporre una **evoluzione guidata** abbiamo bisogno necessariamente dei **test**, che ci garantiscano che, ad ogni evoluzione, il tutto continui a funzionare, e non c'è dubbio che fare test in un **monolite** è nettamente più semplice che testare un sistema **distribuito** a **microservizi**.

Già, ma così facendo, poi come posso garantire la **scalabilità** e la **crescita** su più dimensioni? Allora partiamo con i **microservizi**. In questo caso però ci portiamo in casa, sin dall'inizio, parecchia complessità accidentale, come direbbe il buon Fred Brooks [1] e non vogliamo partire con un carico aggiuntivo sin dall'inizio. Mai come in questo momento la frase di Warren Buffet

Risk comes from not knowing what you are doing

ci piove addosso come un macigno, e se pensate che investire tempo per cercare una buona architettura sia costoso, provate a iniziare con una pessima architettura…

Ma procediamo con ordine.

Cambiamento guidato

Come abbiamo detto, se vogliamo essere certi di poter continuamente apportare modifiche alla nostra soluzione, per adattarla alle nuove esigenze e/o specifiche, allora dobbiamo prima attrezzarci con una seria batteria di **test** che ci possa garantire che, ad ogni cambiamento, non introduciamo errori.

Ma che tipo di test può essere quello in grado di garantire che a livello architetturale, i miei cambiamenti non saranno distruttivi? Non posso certo fare affidamento solo agli **unit test**: quelli mi servono per le funzionalità di business, e vanno benissimo per quelle, ma non per il resto.

Posso forse affidarmi al **Behavior-Driven Development** (BDD)? No! Con questo tipo di test posso testare il comportamento verso l'esterno, garantire che il mio sistema si comporti esattamente come previsto a livello di funzionalità; ma, per quanto riguarda la dimensione architetturale, non mi garantiscono alcun tipo di copertura. Esiste un altro tipo di test, nato prima in ambito sistemistico, rivolto agli operatori **DevOps** più cha noi sviluppatori, per testare che le prestazioni del sistema restino garantite al crescere del suo utilizzo, per monitorare lo stato di salute tramite log centralizzati e alert su eventuali malfunzionamenti, e cose simili. Questo tipo di test ricorre sotto il nome di **Fitness Functions**.

Fitness Functions

Dal punto di vista di uno sviluppatore, una **fitness function** verifica quanto la nostra soluzione sia vicina a quanto desiderato, quanto "**fitta**" le specifiche che un solution architect ha previsto. Ma perché abbiamo bisogno di un test per questo? Sarà compito del **solution architect** monitorare questi aspetti ed eventualmente richiamare il

Team se qualcosa non viene rispettato. No! Se vogliamo garantire la massima flessibilità nell'evoluzione di un sistema, l'unico modo è automatizzare il più possibile tutte le fasi che precedono il suo rilascio in produzione.

Nessuno vuol fare il guardiano del faro. Gli sviluppatori devono sentirsi liberi di apportare le modifiche necessarie al sistema, senza che nessuno stia tutto il giorno ad osservare il loro lavoro; saranno i test nelle pipeline di compilazione e rilascio che diranno loro se hanno svolto un buon lavoro, oppure se devono rivedere qualche parte perché ha violato le specifiche architetturali previste.

Ricordiamoci che in architettura è più importante il **perché** di una scelta, e non il **come** la si implementa. Chi si occupa dell'architettura deve porre l'attenzione, ed essere in grado di giustificare, perché ha fatto certe scelte; come queste verranno implementate sarà una scelta del Team di sviluppo, a patto che questo si possa muovere in un terreno protetto da test.

Scendendo nel concreto, a livello di codice, ci possono aiutare due librerie di test. A chi utilizza il framework **.NET**, Ben Norris mette a disposizione NetArchTest [2]; un pacchetto che ci permette di scrivere test sfruttando delle fluent API. Un esempio di come applicare questi test alla soluzione a moduli vista in precedenza lo potete trovare su GitHub [3]. Questa libreria è ispirata a ArchUnit [4], che è invece pensata per coloro che sviluppano in Java.

Un esempio

Di seguito un esempio, molto semplice ma altrettanto efficace, di implementazione di un test di questo tipo. Vogliamo garantire che il progetto **Sales.Facade**, punto di ingresso al nostro modulo **Sales** non abbia dipendenze dirette con gli altri moduli del sistema. Perché abbiamo bisogno di questo test? Perché vogliamo essere certi che, se un domani il nostro modulo **Sales** evolverà in un microservizio autonomo, non avremo problemi ad estrarlo da questa soluzione, proprio perché non avrà nessuna dipendenza diretta con il resto dei progetti presenti nella soluzione.

```
[Fact]
 public void Should_SalesArchitecture_BeCompliant()
 {
  var types = Types.InAssembly(typeof(ISalesFacade).
Assembly);

  var forbiddenAssemblies = new List<string>
  {
   “BrewUp.Sagas”,
   “BrewUp.Purchases.Facade”,
   “BrewUp.Purchases.Domain”,
   “BrewUp.Purchases.Messages”,
   “BrewUp.Purchases.ReadModel”,
   “BrewUp.Purchases.SharedKernel”,
   “BrewUp.Warehouses.Facade”,
   “BrewUp.Warehouses.Domain”,
   “BrewUp.Warehouses.Messages”,
   “BrewUp.Warehouses.Infrastructures”,
   “BrewUp.Warehouses.ReadModel”,
   “BrewUp.Warehouses.SharedKernel”,
   “BrewUp.Production.Facade”,
   “BrewUp.Production.Domain”,
   “BrewUp.Production.Messages”,
   “BrewUp.Production.Infrastructures”,
   “BrewUp.Production.ReadModel”,
   “BrewUp.Production.SharedKernel”,
   “BrewUp.Purchases.Facade”,
   “BrewUp.Purchases.Domain”,
   “BrewUp.Purchases.Messages”,
   “BrewUp.Purchases.Infrastructures”,
   “BrewUp.Purchases.ReadModel”,
   “BrewUp.Purchases.SharedKernel”
  };
  var result = types
```

```
    .ShouldNot()
    .HaveDependencyOnAny(forbiddenAssemblies.ToArray())
    .GetResult()
    .IsSuccessful;

  Assert.True(result);
}
```

Incremental

Ora che abbiamo risolto il nodo della **crescita guidata**, vediamo di capire come risolvere la questione della **crescita incrementale**. Ad essere onesti non abbiamo molto da dire, in quanto il tema è ben sviluppato parlando di **architetture modulari**.

Intanto bisogna precisare che le **architetture modulari**, di cui tanto si parla ultimamente, non sono affatto una novità. Nel lontano 1971, infatti, David Lorge Parmas [5], della Carnegie Mellon University, descriveva dettagliatamente i vantaggi di un'**architettura a moduli**, riassumendo in tre principali aree: quella **manageriale**, quella della **flessibilità** del **prodotto** e quella della **comprensibilità**.

Manageriale: i tempi di sviluppo dell'intero sistema potrebbero essere ridotti perché team separati potrebbero lavorare a moduli diversi con poca necessità di comunicare.

Flessibilità del prodotto: è molto più semplice, e soprattutto meno invasivo, apportare modifiche a un singolo modulo, indipendente dagli altri, che non all'intero sistema. Su questo punto vale la pena ricordare cosa si intende per sistema **legacy**. Non si tratta di un sistema scritto con framework e/o linguaggi vecchi, ma di un sistema che è ancora molto utilizzato in azienda, (ossia "rende soldi"), ed è però complicato da evolvere o modificare perché il codice al suo interno è fortemente accoppiato e non testato.

Comprensibilità: poter affrontare il sistema un modulo alla volta aiuta a comprenderlo meglio, quindi a proporre una soluzione migliore.

Tutto questo ben prima di **Domain-Driven Design** e di tutti gli approcci "moderni" ora conosciuti.

Coesione vs. accoppiamento

In un'architettura a moduli ogni modulo deve essere assolutamente **indipendente** dagli altri, il che significa che non ci deve essere nessun tipo di accoppiamento statico fra i moduli stessi. Se devo far dialogare fra loro due moduli, lo farò tramite **messaggi**. All'interno di ogni modulo invece gli oggetti possono anche essere dipendenti gli uni dagli altri; in questo caso parleremo di **coesione** più che di accoppiamento, questo perché tutti i componenti del modulo si suppone servano alla soluzione di quella parte di problema del sistema.

Le caratteristiche dell'architettura modulare

Osservando l'immagine possiamo capire alcuni vantaggi che un'architettura simile ci offre rispetto a una soluzione a **microservizi**. Intanto il **Service Bus** per la comunicazione fra i moduli può essere in memoria. Questo ci toglie la necessità di dover subito affrontare l'inserimento di un **Broker** esterno per lo scambio delle informazioni,

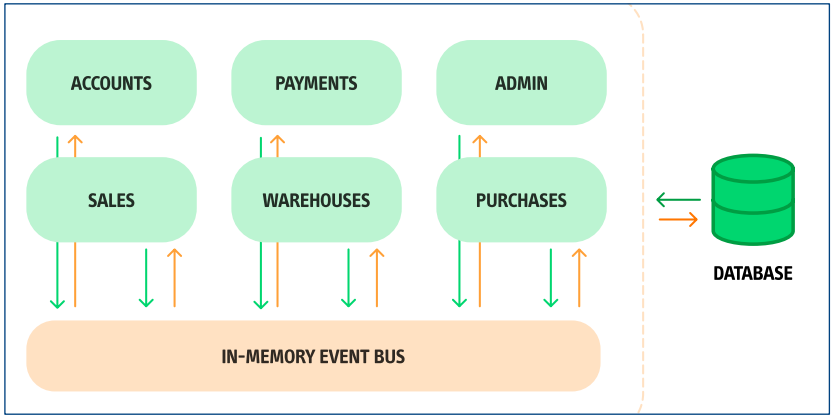

Figura 4.1 – Un'architettura modulare.

ma allo stesso tempo predispone il sistema a utilizzare questo pattern per mantenere il corretto isolamento. Non dobbiamo nemmeno pensare a diverse istanze di database; possiamo utilizzare un solo motore per il database, ma all'interno di esso avere **uno schema per ogni modulo**. Ancora una volta, meno complessità accidentale, in favore di una soluzione pragmatica alla complessità essenziale.

Non sto dicendo che passare da questa soluzione a quella a microservizi non comporti nessun lavoro; sto dicendo che sarà innanzitutto **possibile**, perché appunto ogni modulo è separato logicamente dagli altri, sia per quanto riguarda la comunicazione, che la persistenza dei dati. Certo, ci dovremo attrezzare con **unBroker** per i messaggi, perché la soluzione in memoria non sarà più possibile; ma in fase iniziale saremo molto più veloci a fornire al cliente una soluzione funzionante.

Non dimentichiamo che testare un monolite è decisamente più facile che non testare un sistema distribuito e, se dobbiamo modificare il contenuto dei moduli, spostando oggetti da un modulo a un altro a seguito di una nuova scoperta sul dominio complessivo, lo potremo fare con un po' più di serenità in un monolite rispetto a un sistema distribuito.

Come realizziamo una soluzione a moduli nella pratica? Un esempio scritto in C# lo trovate su GitHub [6].

Multiple Dimensions

Vediamo di analizzare anche l'ultima delle proprietà di un'architettura evolutiva, ossia la crescita in più direzioni. Perché ogni sviluppatore serio ha paura dei sistemi distribuiti?

La risposta semplice è: perché, se cercate, su un motore di ricerca oppure su ChatGPT, risposte riguardanti un'architettura software che faccia al caso vostro, non troverete risposte, se non una serie di pattern applicabili. Già, ma **quali** applico nel mio caso? E soprattutto **come** li applico?

Ricordate la seconda legge delle architetture software? Il "**perché**" è molto più importante di "**come**". Ebbene, se c'è una seconda

legge, significa che ci deve essere anche una prima legge. E infatti esiste e recita che, quando si tratta di architettura software, tutto è un compromesso, e se qualcuno pensa di aver trovato una soluzione che non ha compromessi… semplicemente non ha ancora trovato il compromesso.

Leggi dell'architettura del software

Prima legge: Nell'architettura del software, tutto è un compromesso.

Corollario 1: Se un architetto del software pensa di aver scoperto qualcosa che non è un compromesso, molto probabilmente dipende solo dal fatto che non ha ancora identificato il compromesso.

Seconda legge: Il "perché" è più importante del "come".

Scalare: come e in che direzione?

Garantire la crescita su più dimensioni comporta non solo dover affrontare il tema di una codebase scalabile e facilmente mantenibile. Per questo sappiamo di avere a disposizione i **test**, a tutti i livelli, dagli **Unit Test** sino ai test **End-to-End**.

Ma un sistema distribuito degno di questo nome deve poter **scalare** se le circostanze lo richiedono, e quando diciamo scalare intendiamo che deve essere quanto meno **elastico**, ossia deve scalare verso l'alto, ma anche verso il basso quando non è sotto stress. Trattandosi poi di sistema distribuito avremo bisogno di monitorare tutti i servizi, per garantire che siano in salute e che non compromettano il funzionamento dell'intero sistema.

E qui scoperchiamo un'altra pentola. Cosa succede se il rilascio di un microservizio danneggia l'intero sistema? Non dico nel caso di un monolite distribuito, ossia la peggior soluzione che possiamo adottare. Supponiamo pure che ogni microservizio sia indipendente, implementi soltanto una parte dell'intera logica di business, abbia il proprio database per la persistenza e comunichi solo scambiando messaggi. Se uno di questi microservizi cadesse, gli utenti potrebbero continuare a utilizzare la nostra applicazione? Oppure il tutto crollerebbe?

Gli impatti delle modifiche

Se ripensiamo ora all'articolo di Fred Brooks [1], forse cominceremmo a pensare che la complessità accidentale non è proprio destinata a diminuire. Potrà commutare, ma sicuramente occuperà sempre una parte importante nel nostro sistema.

Per essere certi che la parte di business su cui stiamo lavorando come Team sia effettivamente autonoma dovremmo essere certi che anche l'impatto a livello applicativo lo sia. Ma allora, se ci pensate bene, il solo Bounded Context non è più sufficiente, abbiamo bisogno di espandere il concetto di isolamento. Come recitano gli autori di *Software Architecture: The Hard Parts* [7]:

Bounded Context is not Enough!

Nelle applicazioni distribuite, la garanzia che il rilascio di un microservizio non sia d'ostacolo al resto del sistema è fondamentale. Provate a pensare all'applicazione di *Netflix*. Può capitare che all'avvio non ci appaia la riga delle serie che stavamo guardando, oppure dei film consigliati in base alla nostra esperienza, ma non che l'applicazione non funzioni. Magari certi servizi si caricano in ritardo e appaiono in un secondo momento, ma non si blocca l'intera applicazione.

Questo risultato è il frutto di anni di applicazione di **fitness functions** come **Chaos Engineering** al ciclo di sviluppo. In questo modo siamo in grado di rispettare la crescita del nostro sistema in termini di **funzionalità**, ma anche in termini di **scalabilità multidimensionale**, appunto. In questi termini il **Bounded Context** da solo non è sufficiente a garantire l'indipendenza di un microservizio, perché ci sono aspetti che esulano dai pattern del **Domain-Driven Design**, ma che non possiamo trascurare quando ci occupiamo di sistemi distribuiti.

Architecture Quantum

Di **Architecture Quantum** ho sentito parlare per la prima volta nel libro *Building Evolutionary Architecture* [8] . "Quantum" si riferisce a

un "manufatto" distribuibile in modo indipendente, con un'alta **coesione** funzionale, un alto **accoppiamento statico** e un **accoppiamento dinamico** sincrono. In poche parole, un **microservizio** ben strutturato, dalla **persistenza** alla **UI**, entro un flusso di lavoro.

L'idea è proprio quella di poter distribuire un *artifact* che sia completamente indipendente dagli altri e, nel caso di malfunzionamento, non impedisca all'intero sistema di continuare a funzionare. Magari non avremo accesso ad alcune funzionalità, ma di certo non saremo completamente bloccati. Un **architecture quantum** misura diversi aspetti della topologia e del comportamento dell'architettura software, proprio come richiesto da un'architettura evolutiva, relativi al modo in cui le parti in gioco si collegano e comunicano fra loro.

Static Coupling

Rappresenta il modo in cui le dipendenze statiche si risolvono all'interno dell'architettura tramite contratti. Queste dipendenze includono il sistema operativo, i framework e/o le librerie utilizzate per lo sviluppo del sistema, e qualsiasi altro requisito operativo necessario al funzionamento del quantum.

Dynamic Coupling

Rappresentano il modo in cui i **quanta** comunicano a runtime, in modalità sia sincrona che asincrona. Per questo tipo di accoppiamento si applicano **fitness function** di monitoring delle prestazioni e dello stato di salute.

Princìpi

Ci sono dei principi da rispettare che ci supportino nello sviluppo delle **evolutionary architectures**? Ovviamente ci sono, e proviamo a riassumerli di seguito.

Last Responsible Moment

Per quanto procrastinare sia un'operazione difficile in certi casi, in questo caso non solo è consigliabile, ma è anche salutare. Dobbiamo

rimandare la decisione sull'architettura, ma in genere su qualsiasi punto critico del nostro sistema, all'ultimo momento utile. Dobbiamo cercare di raccogliere quante più informazioni possiamo, dobbiamo ridurre al minimo l'ignoranza verso il sottodominio che stiamo affrontando, prima di decidere perché prendere quella decisione.

Architect and develop for evolvability

Se il principio è progettare prima, e sviluppare dopo, con un occhio di riguardo alle possibilità evolutive del nostro sistema, allora prima lo dobbiamo capire. Non possiamo cambiare un sistema che non si capisce. Quindi torniamo al punto "Problem Space vs Solution Space": non dobbiamo aver fretta di abbandonare lo spazio del problema!

Postel's Law

Il "principio di robustezza", o legge di Postel, afferma

"Sii conservatore in ciò che invii, sii liberale in ciò che accetti"

In pratica ci dice di essere molto cauti nel cambiare il formato della struttura dei dati che inviamo verso l'esterno, per non dover obbligare il mondo a seguire le nostre evoluzioni, e al contempo ad essere un po' più flessibili in ciò che invece ci viene inviato.

Conway's Law

Forse una delle leggi più conosciute.

"Le organizzazioni che progettano sistemi sono costrette a produrre progetti che sono copie delle strutture di comunicazione delle organizzazioni stesse".

Conclusioni

Le **Evolutionary Architecture** sono un argomento molto vasto, di cui abbiamo solo scalfito la superficie, giusto per capire come strutturare il refactoring di un'applicazione esistente, o l'inizio di un

nuovo progetto. I libri che ho citato sono una fonte inesauribile di pattern e dettagli sull'argomento, e rappresentano, a mio avviso, un *must-to-read* per chiunque voglia approfondire il discorso.

Una domanda, a questo punto, sorge spontanea. Un monolite può essere considerato come Architecture Quantum? Per alcuni aspetti sì, siamo in presenza di un forte accoppiamento statico, caratteristica richiesta, ma meno di una alta coesione funzionale. Difficilmente, all'interno di un monolite, troviamo la divisione delle responsabilità attorno ai problemi di business; quasi sempre la parte di business logic è condivisa da tutti i sottodomini.

Un monolite modulare si avvicina molto a un "quantum", ma attenzione alla dipendenza verso il database: se è unico allora non lo è; se ogni modulo ha il proprio schema avremo più possibilità di essere autonomi nelle scelte progettuali, ma attenzione alle dipendenze verso la UI. In breve, un sistema distribuito si presta meglio a essere organizzato a *quanta* rispetto a un monolite.

Riferimenti

[1] Frederick P. Brooks Jr., *No Silver Bullet. Essence and Accident in Software Engineering*. 1986
`https://worrydream.com/refs/Brooks_1986_-_No_Silver_Bullet.pdf`

[2] NetArchTest
`https://github.com/BenMorris/NetArchTest`

[3] I test applicati, su GitHub
`https://github.com/BrewUp/BrewUp.ModularArchitecture/tree/Day1-StrategicPatterns-WithTests`

[4] ArchUnit
`https://www.archunit.org/`

[5] David Lorge Parnas, *On the criteria to be used in decomposing systems into modules*. CMU-CS-71-101, 1971
https://prl.khoury.northeastern.edu/img/p-tr-1971.pdf

[6] Modular Architecture su GitHub
https://github.com/BrewUp/BrewUp.ModularArchitecture

[7] N. Ford – M. Richards – P. Sadalage – Z. Dehghani, *Software Architecture: The Hard Parts*. O'Reilly Media, 2021
https://bit.ly/3T6ptJP

[8] N.Ford – R. Parsons – P. Kua – P. Sadalage, *Building Evolutionary Architectures*, 2nd edition. O'Reilly Media, 2022
https://www.oreilly.com/library/view/building-evolutionary-architectures/9781492097532/

Capitolo 5

Perché non devi condividere i tuoi Domain Events

Progettare per l'evoluzione

Il primo passo per poter progettare un'architettura evolutiva è disegnare il nostro software in modo che possa evolvere facilmente. Già, facile a dirsi, non proprio facile a farsi.

Cosa significa disegnare un sistema che possa evolvere facilmente con le continue richieste del cliente? Proviamo a partire dal classico codice *legacy.* Spesso sento associare il termine "codice legacy" a un applicativo vecchio, che ha bisogno di essere riscritto con un framework nuovo, con nuove tecnologie e linguaggi attuali per essere adattato alle nuove richieste. Niente di più sbagliato. Un codice può essere **legacy** anche se scritto la **settimana** scorsa.

Ciò che caratterizza un codice legacy è il fatto che l'applicativo in questione è importante per l'azienda, è profittevole, ma la sua manutenzione, o evoluzione, è complicata a causa dell'alta dipendenza degli oggetti al suo interno, al punto che modificarne uno comporta il rischio di dover modificare l'intero progetto, e questo, ovviamente, spaventa chiunque.

Come sempre, Domain-Driven Design ci viene incontro. Il modo migliore per rendere un applicativo semplice da manutenere è dividerlo in tanti **piccoli moduli**, non necessariamente **microservizi**, ognuno **associato** a un'**area** del business **specifica**; e per questo abbiamo visto come i pattern strategici, Ubiquitous **Language** e **Bounded Context** in particolare, ci siano di grande supporto. Ma sono sufficienti?

Il **Bounded Context** rappresenta comunque un'area abbastanza grande del nostro modello di business; quindi, il rischio di trovarci a maneggiare qualcosa di complesso rimane. E infatti, non a caso, E. Evans ha presentato anche una serie di **pattern tattici**, molto più orientati allo sviluppo dell'applicativo, che non alla sua comprensione e suddivisione. Non staremo a elencare tutti i pattern tattici, che potete trovare nel libro di Evans, ma ci focalizzeremo su quelli che ci interessano di più per realizzare un applicativo che possa durare nel tempo, ed essere manutenuto senza troppi patemi.

Aggregate

Fra tutti i pattern tattici, quello che ci interessa analizzare è l'**aggregato.** L'aggregato è la rappresentazione semplificata del modello di business che ci prefiggiamo di risolvere. È un pattern complesso e fondamentale, all'interno del Domain-Driven Design. È composto da una, o più, **Entity**, e da uno, o più, **Value Object**.

Detto così sembra quasi semplice disegnare un **Aggregato**, in realtà è una delle fasi più complesse del processo di sviluppo. Essendo una **rappresentazione semplificata** del **modello** di **business**, dovrà mantenersi il più possibile vicino al modello di business reale; ora, senza entrare troppo in discorsi filosofici, il nostro software riuscirà a risolvere i problemi del business in questione fintanto che i due modelli non divergono troppo. Vale a dire, fino a quando, qualsiasi richiesta che arriva dal nostro cliente può essere implementata nel nostro codice.

Ma quindi siamo tornati al punto di partenza? Assolutamente no. L'aggregato non è solo un contenitore di proprietà che descrivono la realtà, ma implementa tutti i metodi necessari per descriverne il comportamento, e tutto ciò che riguarda il particolare problema che risolve passa da lui, e da nessun altro punto all'interno della nostra applicazione. A questo punto è facile capire che, qualsiasi modifica fatta su questo aspetto del business all'interno del nostro codice coinvolge soltanto questo aggregato, senza andare a influenzare altri parti del codice. Ogni aggregato è un piccolo mondo isolato, che dovrà certo comunicare con gli altri, e vedremo tra poco in che modo, ma non dipende da altri per prendere le sue decisioni e validarle secondo le regole di business implementate, ossia i suoi invarianti.

Le comunicazioni all'interno dell'Aggregate

Per garantire questo isolamento l'unico modo che abbiamo per **comunicare** con gli oggetti all'interno di un aggregato è quello di **passare** da una **Entità** molto particolare, che prende il nome di **Aggregate Root,** o **Entity Root.** Nel classico esempio di un ordine di

vendita, composto dalla testa e dalle righe, non potremo accedere liberamente alle righe per modificarle, ma dovremo passare dalla testa dell'ordine, indicata come **Aggregate Root**, che si occuperà di invocare i metodi esposti dalle righe per apportare le modifiche richieste, e al termine delle modifiche, l'aggregate root, ossia la testa dell'ordine, verificherà che le regole di business, o invarianti, siano state rispettate. Ad esempio, che il nuovo totale dell'ordine non superi il budget del cliente.

Facilitare la manutenzione e coerenza dei dati

Concentrare tutti i comportamenti in un solo punto del codice è strategico per la sua manutenzione. Ridurre l'area di intervento dell'aggregato ci aiuta a mantenere uniformità e coerenza attorno al problema che affronta e a ridurre la complessità del nostro codice al solo problema in questione. E cosa dire della coerenza dei dati? Il fatto che l'**aggregate root** verifichi, a ogni variazione di stato dell'aggregato, che le regole di business siano rispettate, ci garantisce **dati coerenti**. Non a caso, l'aggregato deve trovarsi sempre in uno stato *consistent*, sin dalla sua prima istanza. Tecnicamente parlando, un aggregato con una **factory** senza parametri non si può vedere in un dominio degno di questo nome! E qui cominciano i primi problemi di dimensionamento.

Il dimensionamento dell'Aggregate

Se facciamo un aggregato troppo grande, ci portiamo in casa **più complessità** e **più concorrenza**. Sarà più facile, infatti che diverse richieste debbano essere gestite da questo aggregato, che si troverà costretto a gestire molte invarianti, le regole di business; quindi, il codice risultante non sarà facile da manutenere, perché necessariamente complesso. Molto probabilmente però ne guadagneremo in termini di *consistency*: avendo allargato l'area di intervento, molti più dati resteranno coerenti fra loro a ogni cambio di stato.

Contrariamente, se il nostro aggregato sarà piccolo, perderemo in "consistenza", ma guadagneremo in **complessità e concorrenza.**

Un esempio pratico? Provate a pensare alla prenotazione di un posto in una multisala. Se il nostro aggregato è la programmazione giornaliera, avremo una forte "consistenza" dei dati, perché avremo tutto il necessario all'interno dell'aggregato stesso, ma avremo anche molta concorrenza. L'aggregato dovrà occuparsi di gestire le sale con i titoli in programma, i posti disponibili, gli orari, etc.

Se invece il nostro aggregato è la sala, allora il tutto sarà molto più semplice. L'unica serie di comandi che dovremo gestire saranno quelli relativi alla prenotazione dei posti per lo slot di una proiezione. Potremmo scendere ancora? Certamente, potremmo promuovere ad **aggregato** la **fila**.

Cosa conviene fare? Qual è la scelta migliore? Ovviamente dipende! Fermo restando che non esistono aggregati troppi piccoli, non bisogna nemmeno esagerare e portarsi a casa un sovraccarico di lavoro, spesso non necessario.

Domain Event

Ora che abbiamo capito come isolare i comportamenti del business per realizzare applicazioni più mantenibili, come facciamo a far **comunicare** fra loro i singoli aggregati?

Se gli aggregati appartengono tutti allo stesso **Bounded Context**, la cosa è piuttosto semplice. Ogni aggregato ha il proprio **aggregate root**, incaricato di mantenere i rapporti con il mondo esterno, quindi, un **Domain Service** che si occuperà di farli dialogare sarà più che sufficiente. Ma quando un aggregato deve far sapere ad altri aggregati, al di **fuori** del proprio **Bounded Context**, che il suo stato è cambiato, come fa?

E a questo punto che interviene un altro pattern. Il **Domain Event**. Prima di addentrarci nel parlare di questo pattern, vediamo di capire, in poche parole, di cosa si tratta. Un **Domain Event** ha due caratteristiche principali. Per prima cosa, esprimono fatti che sono già accaduti, qualcosa che è già successo nel passato, e per questo motivo il nome di un domain event è sempre espresso utilizzando il **participio passato**. La seconda caratteristica è che un **domain**

event è immutabile! Non può essere modificato in nessun caso, proprio perché esprime qualcosa che è successo nel passato e, come nella vita reale, ciò che è successo non si può cambiare, al massimo possiamo compensarlo con qualche azione correttiva.

I **domain event** sono eventi che appartengono al **modello** di **dominio**, e qui cominciamo a entrare nel vivo della questione. **Non possono essere distribuiti a chiunque**. Una domanda per capire se un evento è un evento di dominio potrebbe essere: "Questo evento interessa ai domain expert?". Se la risposta è "sì", allora è un evento di dominio.

Ora che abbiamo chiarito le caratteristiche del **domain event**, ci potremmo chiedere come mai non erano presenti nel libro di Evans? La risposta reale, ovviamente, non la conosco! Sicuramente gli eventi hanno importanza nel momento in cui da monolite si passa a sistema distribuito, ossia quando si entra nel mondo delle **architetture** a **eventi**. In un sistema distribuito, dove appunto i vari pezzi del nostro sistema non devono essere reciprocamente accoppiati fra loro, lo **scambio** di **messaggi** diventa la **soluzione**. L'arrivo dei microservizi, e di questa tipologia di architetture, ha decretato il successo dei **Domain Event**. Quando si parla di eventi all'interno di Domain-Drive Design non si può non pensare al pattern **CQRS+ES** introdotto da Greg Young.

CQRS

CQRS (Command Query Responsibility Segregation) è l'evoluzione del pattern CQS (Command Query Separation) di Bertrand Meyer, autore del linguaggio Eiffel, e uno dei padri della OOP, che aveva appunto teorizzato la netta **separazione** fra un'azione che **modifica** i **dati** all'interno del database (Command), da una che **legge** i **dati** presenti nel database stesso.

La prima è autorizzata a modificare lo stato di un record, ma non necessariamente deve restituire un risultato, se non il fatto che l'operazione è andata, o meno, a buon fine. La seconda, al contrario, non deve assolutamente apportare modifiche al dato, ma deve limitarsi

a restituire i dati che corrispondono alla query di interrogazione. Se nessuno apporta modifiche, la query deve sempre restituire lo stesso risultato (idempotente).

L'innovazione di Greg Young a questo pattern è stata la netta separazione del database nelle due fasi, di scrittura e di lettura, del dato. Il punto di partenza è che se un database è ottimizzato per la scrittura, allora non lo sarà per la lettura, e viceversa. Come spesso accade, un'immagine vale più di mille parole.

Per essere certi di essere allineati sui principi base di questo pattern, vediamo di esplicitare il significato di figura 5,1 prima di proseguire.

Dal **Client** viene inviato un **Command** per la richiesta di una variazione di stato del nostro **Aggregato**. Un esempio potrebbe essere "**CreaOrdineDiAcquistoDaPortale**". Il nome del comando dovrebbe esplicitare chiaramente l'intento del business.

Il **Command** viene gestito appunto da un **CommandHandler**. Questo componente si occupa di invocare i metodi descritti all'interno del nostro Domain Model (in pratica il nostro Aggregato). Ricordiamoci che il **Domain** Model non contiene soltanto le proprietà del nostro oggetto (saremmo di fronte a un Dominio Anemico), ma anche l'implementazione dei comportamenti di business, concretamente, i **metodi** per gestire le sue variazioni di stato.

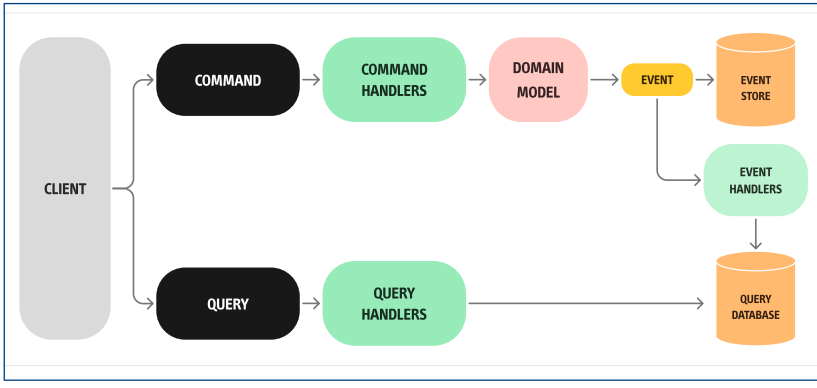

Figura 5.1 – Lo schema di CQRS.

Il **Domain Model** si prende in carico il **Command** e, se l'azione richiesta viola le regole di business, ha la facoltà di respingerlo. Ad esempio, il totale dell'ordine viola il budget del Cliente. Al termine dell'azione associata al Command il Domain Model emette un **Domain Event**. Ricordate che se il vostro sistema è totalmente asincrono, il Domain Model deve emettere un **Domain Event** sia in caso di successo, sia in caso di fallimento, altrimenti la vostra UI non verrà mai avvertita dell'accaduto.

Del **Domain Event** emesso viene effettuata la persistenza nell'**EventStore**. Non è strettamente obbligatorio salvare l'evento nell'**EventStore** ma, se vogliamo lavorare con aggregati basati sugli eventi, lo diventa; altrimenti non potremmo più ricostruire lo stato dello stesso a ogni richiesta di variazione di stato. Così come non è necessario avere due database diversi per eventi e **Read Model**. Per restare semplici è possibile utilizzare un solo database per entrambi gli scopi.

Lo stesso Domain Event viene pubblicato in modo che tutti i servizi, all'interno del Bounded Context, che sono interessati a ricevere notifica dalla variazione di questo **Domain Model**, possano effettuare l'iscrizione per per aggiornare il **ReadModel**, ossia il modello che serve per le interrogazioni, query, del Client.

Qualsiasi query che arriva dal Client andrà a agire sul **ReadModel**, che conterrà diverse *projection* in base alle necessità della nostra UI.

Domain Event

Il **Domain Event** è parte, a tutti gli effetti, del modello di dominio, ed è una rappresentazione di ciò che è successo nel dominio stesso. In pratica, anziché limitarci a salvare lo stato del nostro aggregato, ne **salviamo le singole variazioni di stato**, espresse appunto tramite il **Domain Event**. Ogni volta che dovremo intervenire sull'aggregato per applicare un nuovo comando, o per decidere di rifiutarlo perché in contrasto con le regole di business, andremo a rileggere tutti gli eventi relativi a questo aggregato, li applicheremo in maniera sequenziale, sino a ottenere la versione attuale dello stesso. Come mi piace spesso ripetere, è un pò come passare dall'osservare

una fotografia del nostro aggregato — lo stato attuale — a osservare il suo film, che ci racconta come è arrivato a trovarsi in questo stato.

Non a caso i Domain Event, come abbiamo già detto, vengono identificati con **nomi** al **passato**, proprio perché ciò che esprimono è **già successo**! Nessun altro elemento all'interno del nostro sistema può permettersi di questionare al riguardo. Questo comportamento ci garantisce flessibilità, e tranquillità, quando andremo a implementare nuovi comportamenti, o a modificare quelli esistenti, all'interno del nostro codice. Tutta la logica di business è racchiusa nel nostro **Aggregato**, e non distribuita in diversi layer, o servizi, sparsi nella nostra codebase. Perciò, una volta apportate le modifiche, e verificato che i nostri test continuino a essere verdi, potremo tranquillamente rilasciare la nuova versione, con la certezza che non andremo a rompere nulla al di fuori del nostro **Bounded Context**.

Chiunque sottoscrive un Domain Event lo accetta così com'è, senza inserire nel codice clausole decisionali.

Come è fatto un Domain Event

Ma com'è fatto un **Domain Event**? Si tratta di un DTO che contiene le proprietà che sono cambiate all'interno del nostro aggregato, e una sua implementazione in C# potrebbe essere questa.

```
1 reference
public sealed class DiscountApplied : DomainEvent
{
 1 reference
 public readonly OrderNumber OrderNumber;
 1 reference
 public readonly Discount Discount;
 1 reference
 public readonly Total Total;
 0 references
 public DiscountApplied(OrderNumber orderNumber,
 Discount discount, Total total)
```

```
  {
    OrderNumber = orderNumber;
    Discount = discount;
    Total = total;
  }
}
```

Proviamo a analizzare questo codice. Innanzitutto, è una classe **sealed**, il che significa che nessuno la può ereditare, e quindi espandere, all'interno del codice. Tutte le sue proprietà sono **readonly**. Il motivo è molto semplice. Per essere inviato alle parti interessate, il nostro **Domain Event** verrà serializzato, in fase di invio, e poi deserializzato, in fase di ricezione. Durante queste fasi, nessuno può intervenire a modificarne il suo contenuto. Dobbiamo essere certi che ciò che esce dal nostro dominio arrivi integro a destinazione.

I più attenti si saranno accorti di un'altra importante caratteristica: la **semantica** utilizzata all'interno del **Domain Event**. È possibile notare che, per esprimere le grandezze delle proprietà, non vengono utilizzati tipi primitivi, ma tipi customizzati che rispecchiano il linguaggio del business in questione, ossia l'**Ubiquitous Language**. Se definisco il numero di ordine con un tipo **string**, o **UUID**, solo i tecnici saranno in grado di capirlo, e quindi di validare, o meno, il mio **Domain Event**. Se invece utilizzo il tipo custom **Order Number**, tutti i membri del team capiranno di cosa stiamo parlando, abbattendo tutte le barriere di comunicazione fra persone di business e persone tecniche, riducendo la possibilità di fraintendimento, e quindi di produzione di codice non esattamente conforme alle specifiche, al minimo.

Modellare l'Aggregato in relazione alla realtà

Quando modelliamo il nostro Aggregato, ossia il modello di business che ci accingiamo a risolvere, vogliamo restare il più conformi possibile alla realtà, e avere un linguaggio comune, lo abbiamo già detto e ripetuto, è fondamentale. Ovviamente dovremo semplificare

dei concetti, perché è impossibile replicare un modello reale all'interno del nostro codice. Per non rischiare di divergere troppo è fondamentale creare **aggregati** il più **piccoli** possibile, per rimanere il più vicino possibile alla realtà. Un po' come considerare l'intero processo di business una figura geometrica complessa, di cui non siamo in grado di calcolarne la superficie, se non suddividendola in infinite parti che **integreremo** al fine di ottenere un risultato finale non proprio esatto (non riusciremo mai a replicare la realtà), ma sicuramente molto, molto simile. Sufficientemente simile per schematizzare e risolvere il problema stesso.

Infine, ma non meno importante, ogni **Domain Event** contiene l'**ID** dell'**aggregato** a cui appartiene, e l'eventuale **ID** delle operazioni a esso correlate; essendo queste proprietà ripetute per ogni **Domain Event** è bene raggrupparle in una classe **DomainEvent** che ci aiuta sia a semplificare la scrittura di tutti i domain event, sia a capire, solo guardando il codice, se l'oggetto che stiamo osservando è un **Domain Event**!

Chi può sottoscrivere un Domain Event?

A questo punto dobbiamo chiederci quali sono le parti interessate a sottoscrivere un **Domain Event**, chi ne è legittimato. Trattandosi di un elemento del Dominio stesso, non può, e **non deve**, uscire dal **nostro Bounded Context**! Nell'immagine relativa al pattern CQRS/ES, abbiamo visto che il Domain Event serve per aggiornare il nostro Read Model, ossia la parte di dati che poi gli utenti della nostra applicazione andranno a interrogare per prendere decisioni. In un sistema distribuito, queste informazioni possono essere ridondate e replicate in diversi Read Model, ognuno in Bounded Context differenti, e potenzialmente, in microservizi differenti, ossia, database differenti. Quale strumento utilizziamo per aggiornare tutti questi Read Model che non appartengono al nostro Bounded Context? La risposta del programmatore pigro è: "Il Domain Event", ed è qui che il nostro sistema, in un batter d'occhio, si trasforma da Distribuito, a Big Ball of Mud.

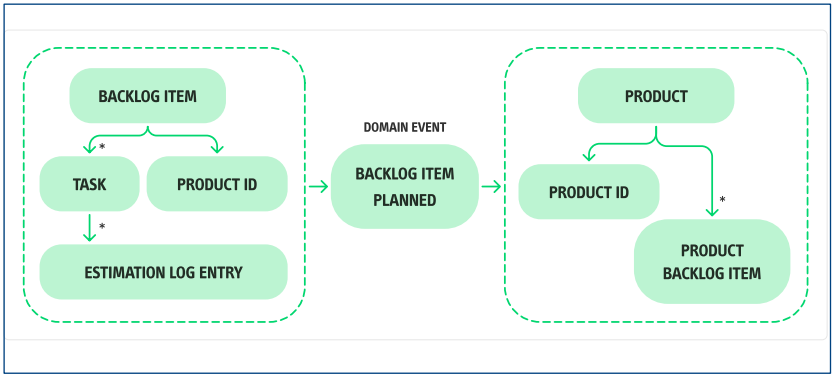

Figura 5.2 – Schema del Domain Event (da Vernon [1])

Condividere un **Domain Event** con altri **Bounded Context** comporta un errore dal punto di vista semantico, ossia sto condividendo informazioni espresse in un linguaggio tipico di un Bounded Context con un altro che non è detto condivida lo stesso linguaggio, fermo restando che nel Domain Event ci potrebbero essere informazioni che non devono uscire proprio da questo contesto per essere condivise.

Ma c'è un errore anche dal punto di vista più implementativo e, rifacendoci a quanto detto nella parte 4 di questa serie a proposito delle **Evolutionary Architecture**, nel momento in cui due parti di codice condividono qualcosa, ci troviamo di fronte a un **accoppiamento**, e quindi possiamo dire addio alla facoltà di evolvere in maniera indipendente a ognuno di esse. Gli aggregati, come già detto, sono una rappresentazione della realtà, uno schema semplificato del problema di business da risolvere.

I problemi di business, essendo problemi legati agli esseri umani, sono per loro natura mutevoli, e in quanto tali, prima o poi, evolveranno in modo tale da non poter più essere rappresentati dal nostro aggregato, che dovremo far evolvere per soddisfare il nuovo modello. Non si tratta di un errore di modellazione iniziale, o di una prematura ottimizzazione! Semplicemente dobbiamo accettare che **le**

cose cambiano! Lo so che molti di voi stanno cercando riferimenti all'interno del concetto di **antifragilità** dei sistemi di Taleb, e in effetti è proprio così.

Ne abbiamo parlato quando abbiamo introdotto appunto le Architetture Evolutive; la sfida non è costruire un sistema perfetto, ma un sistema in grado di evolvere, di adeguarsi ai cambiamenti continui, di modificare anche una sua piccola parte per adeguarsi alle nuove necessità. Forse risulta molto più evidente perché condividere il nostro Domain Event va esattamente nella direzione opposta a questa.

Prima di essere considerato un errore semantico, va analizzato come un errore architetturale. Se il contratto con cui scambio informazioni con gli altri modelli è lo stesso che utilizzo al mio interno per mantenere la coerenza fra **Domain Model** e **Read Model**, allora non mi potrò permettere il lusso di poter modificare il mio aggregato a piacimento, perché così facendo andrei a modificare il contratto di comunicazioni con gli altri Bounded Context, che sarei costretto ad avvisare, e che dovranno essere aggiornati e pubblicati, in caso di architettura a microservizi, in concomitanza con il Bounded Context modificato, pena l'incapacità di comunicare a causa proprio della variazione del Domain Event condiviso. E di colpo ci siamo portati in casa la complessità dei sistemi distribuiti, con i limiti del monolite accoppiato, in sintesi, abbiamo creato la perfetta **Big Ball of Mud**.

Integration Event

Ma allora come avviso il "resto del mondo" che lo **stato** del mio **Bounded Context** è stato **modificato**? Utilizzando appunto un Integration Event, ossia un evento che dal punto di vista tecnico è assolutamente identico al Domain Event, ma che conterrà **informazioni** che **possono essere condivise**, espresse in un linguaggio comune a tutti i Bounded Context del mio sistema.

Ma se i dati del Domain Event e dell'Integration Event sono esattamente gli stessi? Non importa, pubblico ugualmente un **Integration**

Event, perché sicuramente, prima o poi, il Domain Event cambierà, e noi potremo permetterci di farlo perché saremo certi che viene consumato solamente all'interno del nostro Bounded Context, e potremo continuare a emettere lo stesso **Integration** Event all'esterno, senza interrompere la condivisione di informazioni con il resto del sistema.

In questo modo avremo garantito l'indipendenza delle parti del nostro sistema, che potranno evolvere liberamente, senza attendere il benestare delle altre.

Versioning

Cosa succede se, a causa appunto di una nuova richiesta, il nostro **Domain Event** deve cambiare? Se, ad esempio, dobbiamo aggiungere, o togliere, una delle proprietà?

Innanzitutto, dobbiamo capire se si tratta veramente di una variazione, e quindi è sufficiente aggiungere delle proprietà, ma il significato espresso del nostro Domain Event resterà invariato; in questo caso saremo veramente di fronte a una **modifica** dello stesso, a una **nuova versione**. Se invece le modifiche saranno talmente invasive, da modificare il significato semantico del nostro Domain Event, allora saremo di fronte a un **nuovo Domain** Event, probabilmente con un **nome diverso**, in grado di comunicare il nuovo intento del business.

Fatte le dovute premesse, va comunque detto che un Domain Event non deve mai essere modificato! Nuovamente, le ragioni sono sia semantiche che tecniche. Dal punto di vista semantico, il Domain Event, ormai lo abbiamo imparato, esprime un concetto di business, e noi vogliamo che questo concetto resti invariato nel nostro EventStore e nel nostro Domain Model. Dal punto di vista tecnico, abbiamo detto che il Domain Event viene prima serializzato, e poi deserializzato; quindi, se modifichiamo la sua struttura dopo averlo salvato rischiamo di non essere più in grado di recuperare gli eventi dall'EventStore, quindi di ricostruire lo stato del nostro Aggregato, quando dovremo deserializzare gli eventi salvati.

Come si risolve questa situazione? **Versionando** gli eventi. Avremo quindi la versione V1, V2, … Vn del Domain Event che è cambiato nel tempo. Attenzione, ricordiamoci, a patto che il suo significato, dal punto di vista del business, resti invariato, altrimenti dovremo proprio rinominarlo. Nel caso in cui avremo più versioni dello stesso evento, dovremo prevedere nel nostro codice tutti gli **handler** per tutte le versioni, oppure un **handler** in grado di catturarli tutti, valorizzando, laddove necessario, le proprietà mancanti con valori standard. La gestione del **versioning** è una questione complessa, un ottimo libro [2] al riguardo lo sta scrivendo, da anni, proprio Greg Young, e vale certamente la pena di essere letto.

Conclusioni

Da programmatore pigro, se devo condividere informazioni fra Bounded Context diversi, appartenenti a microservizi diversi, sono molto tentato di farlo tramite un qualcosa che nel mio codice esiste già, in questo caso un Domain Event. Non devo fare altro che aggiungere un altro **eventhandler** allo stesso topic e il gioco è fatto.

Purtroppo, non sto commettendo solamente un errore semantico, ossia sto condividendo informazioni scritte in un linguaggio tipico di un Bounded Context, con altro Bounded Context che non necessariamente conosce questo linguaggio ... altrimenti non parleremmo di Ubiquitous Language! Senza trascurare il fatto che in un Domain Event, essendo un oggetto che appartiene al Domain Model, ci possono essere informazioni riservate, che non devono uscire dal confine del Bounded Context stesso. Perché non si tratta solamente di un errore semantico? Perché nel momento stesso in cui due funzioni condividono un oggetto, un qualsiasi oggetto, queste ultime risultano essere accoppiate fra loro, e nel momento in cui dovrò modificarne una, sarò costretto a modificare anche l'altra, con buona pace all'indipendenza dei microservizi che le ospitano. Provate voi, ora, a pubblicare il vostro microservizio, in seguito all'implementazione di una nuova feature che ha modificato il Domain Event, senza avvisare tutti i relativi sottoscrittori allo stesso!

Anche se le informazioni da scambiare, in un primo momento, sono esattamente le stesse contenute nel Domain Event, create un Integration Event e condividete quello. Con un piccolo sforzo avrete guadagnato la libertà di intervenire sul vostro Domain Model come vorrete, senza dover renderne conto a nessuno, e si sa, la libertà non ha prezzo!

Riferimenti

[1] Vaughn Vernon, *Implementing Domain-Driven Design*. ⁋ Addison-Wesley Professional, 2013

[2] Greg Young, *Versioning in an Event Sourced System*
`https://leanpub.com/esversioning/read`

Capitolo 6

Perché Event Driven?

Un po' di storia

Abbiamo già enunciato le leggi che regolano le architetture software, e la seconda di esse recita:

Why is more important than how

Perciò vediamo di capire perché dovremmo sempre aspirare a implementare soluzioni **event-driven.**

Correvano gli anni Settanta, proprio all'inizio di quel decennio, quando Alan Kay, uno dei padri della **programmazione a oggetti**, insieme ad altri illustri colleghi dello Xerox PARC, creò **SmallTalk**, un linguaggio orientato agli oggetti. In molti, se non tutti, già sapevano di questa storia, ma quello che spesso tutti ci dimentichiamo, sono i principi che lo stesso Alan Kay enunciò a proposito di OOP:

- ogni cosa è un **oggetto**;
- gli oggetti comunicano fra di loro **inviando** e **ricevendo messaggi**;
- più gli gli oggetti contengono **dati** e **comportamenti**;
- più gli altri punti che potete trovare comodamente in rete.

È curioso scoprire che anche Carl Herwitt, autore del modello **Actor Model** [2], arrivò alle stesse conclusioni, sostituendo l'**oggetto** con il concetto di "**attore**", pur senza conoscere né la persona, né le ricerche di Alan Kay.

Se poi volessimo andare ancora un po' più a ritroso nella storia, scopriremmo di questo trattato, di cui riporto solo alcuni punti, scritto nel 1922:

- il mondo è tutto ciò che si verifica;
- il mondo è la totalità dei fatti, non delle cose;
- il mondo è determinato dai fatti e dal loro essere **tutti** i fatti;
- il mondo si divide in fatti;
- qualcosa può verificarsi o non verificarsi e tutto il resto rimane uguale.

Se vi state chiedendo quale genio dell'informatica fosse così avanti nei tempi tanto da avere anticipato persino il concetto di **Event Sourcing**, fermatevi… si tratta di un trattato filosofico [3].

Già queste informazioni cominciano a rispondere alla domanda del "**perché dovremmo creare sistemi event-driven?**".

Lo scenario attuale

Tornando a tempi più recenti, provate a pensare come i **sistemi distribuiti** stanno plasmando il nostro modo di creare applicazioni. Il **cloud computing**, in maniera particolare, ha influenzato profondamente il modo di consumare informazioni, non solo per i nostri sistemi, ma anche per noi stessi.

Non esiste business che non sia a portata di **app** oggigiorno. Abbiamo bisogno di scaricare e inviare dati non solo all'interno dei nostri applicativi, ma dobbiamo essere aperti anche al resto del mondo se veramente vogliamo che il nostro sistema abbia successo. Dobbiamo fare in modo che i nostri dati siano fruibili, "**consumabilii**" anche da altri, con le dovute precauzioni, ma dobbiamo essere aperti. Tutto questo ha spostato il modo di progettare il software, sdoganando definitivamente la programmazione **asincrona**, e mandando in pensione il concetto di programmazione **sincrona**, se non per piccole applicazioni local, sempre che ancora esistano…

Dobbiamo avere la certezza di poter consumare le informazioni nel momento in cui ci servono, non quando vengono prodotte, e questo comporta che tutti i messaggi che produciamo, o a cui facciamo un **subscribe** per riceverli, devono avere la possibilità di **persistenza**, affinché sia possibile consumarli al bisogno. Del resto, è proprio quello che accade nella vita reale fra persone reali, esattamente come descritto da Alan Kay e Carl Herwitt nelle loro ricerche.

Questo cambiamento nel modo di gestire i dati ha provocato radicali cambiamenti a livello di **architetture** software, ma soprattutto a livello di **relazioni** e di business nella società. Aspettate… stiamo forse dicendo che Conway [4] aveva ragione?

Che cosa sono i microservizi Event-Driven

I **microservizi** e le **architetture a microservizi** esistono da molti anni, intendiamoci, in diverse forme e sotto differenti nomi. I

"diversamente giovani" ricorderanno **SOA**, Service-Oriented Architectures, in cui regnava la comunicazione sincrona fra diversi servizi e i messaggi venivano scambiati in una comunicazione diretta fra due specifici servizi distribuiti.

La comunicazione **event-driven** non è quindi la novità, ma le esigenze sono cambiate: ora sono intervenuti termini come **big data, real-time, scaling** e la comunicazione sincrona è diventata insostenibile; non possiamo pensare di avere un insieme di sistemi, tutti contemporaneamente e costantemente, sincronizzati, basta pensare alle **Fallacies of distributed computing** [5] per averne la prova.

In una moderna **architettura** a **microservizi** basata sugli **eventi**, questi ultimi **non** vengono distrutti dopo essere stati consumati, ma vengono **persistiti** per poter essere consumati da altri servizi che, per qualsiasi motivo, potrebbero non essere disponibili al momento in cui gli eventi stessi sono stati emessi.

Questa distinzione è fondamentale per capire come si sono evolute le architetture dei sistemi distribuiti, e come sono cambiate le tecnologie a corredo o, facendo riferimento a alle definizioni di Fred Brooks [6], come certe **complessità accidentali** siano diventati **essenziali**.

Certamente il primo cambiamento lo notiamo a livello tecnologico, perché è il primo che, da tecnici, siamo chiamati a risolvere; ma è sempre il motivo dietro le quinte, il famoso "**perché lo fai?**" che ci indica la direzione. Le **architetture a microservizi basate su eventi** portano con loro un cambiamento di paradigma. Ora scriviamo i microservizi attorno alle necessità del business, e facciamo in modo che ogni microservizio si occupi di quel pezzo di business della nostra applicazione, concetti che in SOA non erano nemmeno considerati, se non da pochi illuminati.

Altra questione importante: ora ci preoccupiamo che la nostra applicazione, nel suo insieme, non si blocchi se, per qualsiasi motivo, una parte di essa non è disponibile e non può ricevere messaggi. Accettiamo il fatto che alcune funzionalità non siano disponibili, ma non che l'intero sistemi sia offline, e questo è possibile solo ed

esclusivamente se le comunicazioni fra le varie parti del sistema stesso sono **asincrone**.

Domain-Driven Design, Bounded Context e microservizi

Sembra quasi il preludio di un film tipo "I Tre Moschettieri". In effetti **Domain-Driven Design** ha avuto il suo più grande momento di gloria e di avvio al successo proprio con l'arrivo delle architetture a microservizi. L'ho già detto che non esiste una relazione uno-a-uno fra **Bounded Context** e **microservizi**, ma è altrettanto innegabile che questi due pattern hanno parecchio in comune, e che questo pattern del **DDD** è sicuramente il più utilizzato per l'isolamento del problema di business che poi viene implementato nel rispettivo microservizio.

Proviamo a pensare a un'azienda, di un qualsiasi settore, e sicuramente troveremo un reparto commerciale, un reparto **vendite**, un reparto **supporto** clienti,e così via. Questi **reparti** sono spesso associati, nell'operazione di **Context Mapping**, ai rispettivi **Bounded Context** nel sistema che dovrà gestire l'azienda stessa. Questa frammentazione in sottodomini ovviamente può continuare ulteriormente dopo questo primo passaggio, per creare ulteriori sottodomini, più granulari, che ci aiuteranno a isolare i vari problemi di business che dovremo risolvere, e a creare quindi team indipendenti per la loro gestione, che creeranno **microservizi indipendenti** gli uni dagli altri, che potranno evolvere in maniera autonoma secondo le necessità dei singoli contesti. Ma siamo certi che i nostri microservizi siano **realmente indipendenti** gli uni dagli altri?

Coupling vs Cohesion

Quando suddividiamo il **dominio** della nostra applicazione in tanti sottodomini, l'operazione più difficile è sempre quella di individuare prima, e gestire dopo, le **dipendenze** fra di essi. Ricordiamoci che il fatto di suddividere un dominio **non** ci autorizza a creare tante **applicazioni isolate** l'una dalle altre: questo è quello che facciamo noi sviluppatori dietro le quinte, ma l'utente che utilizzerà

la nostra applicazione si aspetta di usare appunto **una** applicazione, **non tante** applicazioni! In un modo o nell'altro, questi **microservizi**, come abbiamo visto prima, devono **comunicare**.

Microservizi indipendenti

Intanto chiariamo che, per essere realmente indipendente, un **microservizio** deve avere il **proprio database**, o i propri database, se distinguiamo un database per le normali operazioni di lettura e uno per la persistenza degli eventi, nel caso volessimo implementare il pattern CQRS-ES. Dovrà avere accesso a un sistema di **trasporto** dei **messaggi** per comunicare con l'esterno, e dovrà, possibilmente, rispondere a una parte della **UI** a esso **dedicata**.

Soltanto se siamo in grado di garantire tutto questo isolamento potremo veramente affermare che la nostra architettura è **un'architettura** a **microservizi**, in caso contrario, staremo implementando un "monolite distribuito", che può anche essere un passaggio intermedio della nostra evoluzione, ma non l'obiettivo finale.

Quindi? Tutti i microservizi sono disaccoppiati fra loro? Non esattamente. All'interno di ogni microservizio ogni oggetto può tranquillamente dipendere da un altro appartenente allo stesso microservizio. Questa situazione è totalmente accettabile, in quanto stiamo parlando di un componente, il microservizio, che quando cambia, lo fa nel suo insieme; e tutti i test implementati al suo interno ci garantiscono che tutte le modifiche che apportiamo per implementare nuove **feuture** non rompono il suo funzionamento. Il discorso cambia quando parliamo di relazioni con gli altri microservizi.

Comunicazione asincrona

Tornando a quanto detto riguardo a SOA e alla comunicazione sincrona fra i vari servizi, ora ci appare chiaro perché, in una moderna visione di un **sistema distribuito**, questo tipo di comunicazione non può più essere accettato. Se la comunicazione fra due microservizi fosse sincrona, significherebbe avere una dipendenza fra i due, il che ci porterebbe ad affermare che ogni cambiamento apportato a

uno dei due, quasi certamente, porterebbe alla necessità di cambiare anche l'altro, impedendo così la possibilità di rilasci indipendenti.

Ricordate quando abbiamo trattato la necessità di **separare** gli **eventi** che vengono consumati all'interno di un **Bounded Context** (**Domain Event**) da quelli che vengono condivisi con gli altri **Bounded Context** (**Integration Event**)? Bene, ora sostituite **Bounded Context** con **microservizio** e tutto sarà ancora più chiaro. Se l'evento che un microservizio emette verso l'esterno è diverso da quello che consuma internamente, allora avrò sempre la possibilità di apportare modifiche, ossia cambiare versione, a quello interno, senza la necessità di avvisare chi consuma quello esterno del cambiamento, perché quest'ultimo potrà restare invariato.

In questo modo abbiamo rimosso la dipendenza fra due, o più, microservizi. Ricordate il Principio di Robustezza, o legge di Postel? Dobbiamo essere rigidi nel cambiare i contratti verso l'esterno, perché questi rappresentano la dipendenza verso altri sistemi, e se li cambiamo, dobbiamo necessariamente avvisare chi li consuma, dobbiamo riscrivere i nostri **contract test** e riallineare l'intero sistema; in pratica dobbiamo dire addio all'indipendenza del rilascio!

E nel caso invece dovessimo apportare modifiche anche all'evento di integrazione? Nessun problema, una delle caratteristiche delle Architetture Evolutive è proprio quella di considerare normale mettere a disposizione versioni diverse dello stesso servizio, quindi potremo tranquillamente emettere due integration event, uno per chi ci ha chiesto la nuova versione, e l'altro per chi non ne ha bisogno, in modo che possa continuare a consumare la versione precedente.

Conway's Law e strutture di comunicazione

La cosiddetta legge di Conway [4] recita:

> Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.

Questa famosa citazione di Melvin Conway, risalente al 1968, è spesso citata quando si parla di **sistemi distribuiti** e, pur essendo stata scritta in tempi in cui architetture simili non erano nemmeno lontanamente pensabili, è assolutamente adeguata.

Nell'esempio che ho fatto prima della divisione in sottodomini, ho descritto delle **aree** di business all'interno di un'azienda generica; quando si organizzano i team attorno a queste aree di business, questi team finiscono col produrre servizi che sono delimitati dai loro confini di business, ossia, tendono a condividere informazioni tipiche del loro **contesto**, nel bene e nel male. Qui si potrebbero aprire interi capitoli su quali informazioni condividere e quali no, ma non è il nostro scopo.

Informazioni condivise e livelli di indipendenza

Sicuramente le **informazioni** che ogni Team condivide con gli altri determinano il livello di **indipendenza** che ogni Team sarà in grado di costruirsi. Proviamo a fare un esempio pratico.

Se il eam che si occupa del sotto dominio del magazzino non condivide le giacenze dei prodotti, sarà impossibile per gli altri team costruirsi una copia delle stesse nei propri database, in modo da non dipendere sempre da questo microservizio ogni qualvolta si avrà la necessità di conoscere la disponibilità, o meno, di un prodotto. Se, ogni volta che il team del reparto vendite deve implementare una funzionalità sugli **ordini di vendita**, deve chiedere al team — leggi **microservizio** — del magazzino se il prodotto è disponibile, significa che fra i due microservizi esiste una **dipendenza**. La potremo risolvere in tanti modi diversi, non necessariamente con una comunicazione sincrona, ma la dovremo **risolvere** per fornire al nostro cliente una data di consegna prevista.

Inverse Conway Maneuver

Un'interessante esplorazione di questo problema è stata fatta da Martin Fowler, che tratta il problema e propone una soluzione interessante denominata **Inverse Conway Maneuver** [7].

Responses to Conway's Law	
Ignore	Don't take Conway's Law into account, because you've never heard of it, or you don't think it applies (narrator: it does)
Accept	Recognize the impact of Conway's Law, and ensure your architecture doesn't clash with designers' communication patterns.
Inverse Conway Maneuvre	Change the communication patterns of the designers to encourage the desired software architecture.

Esplorare nuove modalità all'interno dell'azienda

Cambiare i pattern di comunicazione all'interno di un'azienda non è affatto banale. Significa cambiare equilibri delicati costruiti negli anni, significa abbracciare un nuovo modo di affrontare i problemi, in cui, quasi sicuramente alcune figure vedono diminuire il loro **potere** all'interno del palinsesto aziendale.

Se fino a ieri, per poter confermare una data di consegna, dovevi chiedere a me, responsabile del magazzino, la disponibilità del prodotto, e ora invece hai la tua autonomia, e puoi chiudere l'ordine autonomamente, significa che io ho perso valore?

No, significa che è cambiato il modo di comunicare, significa che stiamo esplorando nuovi modi, stiamo esplorando, e magari scopriremo che non è la soluzione migliore quella implementata, ma saremo pronti a percorrere nuove strade.

Ricordate quando abbiamo parlato di **Architetture Evolutive**? Una delle sfide è proprio quella di **abbracciare** una **cultura** della **sperimentazione**, in cui **fallire** viene letto come **provare**, e non ci fa paura.

In ogni caso non sempre è facile ignorare i confini esistenti all'interno delle organizzazioni, e non è detto che farlo sia un bene.

Procedere per piccoli passi durante un refactoring supporta non solo l'apprendimento del dominio su cui stiamo lavorando, ma permette a chi ci chiede supporto, di accettare nuovi modelli di comunicazione e di condivisione.

Conclusioni

La **comunicazione** è la base delle relazioni, non solo umane, ma anche dei **sistemi** che sviluppiamo, e senza di essa sia i primi che i secondi, sono destinati a fallire, sia pure in termini diversi.

Le strutture di comunicazione determinano il modo in cui il software viene creato e gestito durante il suo ciclo di vita, e del ciclo di vita di chi lo utilizza. Le informazioni che decidiamo di condividere fra i nostri sistemi, così come nella vita reale, determinano il grado di **indipendenza**, o di **dipendenza**, di ogni servizio rispetto agli altri; e questo è un dato di fatto di cui tener conto durante le operazioni di ristrutturazione di un applicativo esistente. Non possiamo pensare di rimodellare tutto in un colpo solo: i piccoli cambiamenti sono più facili da accettare.

Riferimenti

[1] Adam Bellemare, *Building Event-Driven Microservices*. O'Reilly, 2020

[2] Actor Model
`https://en.wikipedia.org/wiki/Actor_model`

[3] Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*. Kegan Paul, Trench, Trubner & Co., 1922
`https://t.ly/6XHaZ`

[4] Legge di Conway
`https://en.wikipedia.org/wiki/Conway%27s_law`

[5] Fallacies of distributed compunting

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[6] Frederick P. Brooks Jr., No Silver Bullet. Essence and Accident in Software Engineering. 1986
https://worrydream.com/refs/Brooks_1986_-_No_Silver_Bullet.pdf

[7] Martin Fowler, *Conway's Law*
https://martinfowler.com/bliki/ConwaysLaw.html

Capitolo 7

Architetture antifragili

Agile & Architecture: un rapporto complicato

In un articolo [1] su MokaByte si è parlato del Manifesto Antifragile; in particolare ho apprezzato il paragrafo che affronta la misurazione del rischio. Riprendendo il pensiero di Nicolas Taleb ne *Il Cigno Nero*, Giovanni ci ricorda che lo studio della gestione del rischio non è lo studio di un evento prevedibile e collocabile nel futuro, come invece succede nella realtà, e soprattutto nel mondo esplorato da Taleb. La sua esplorazione è rivolta in particolare all'applicazione dei principi dell'Antifragilità nelle organizzazioni; possiamo estendere tutto questo a livello architetturale?

Quando si parla di Agilità, il ruolo che ricopre — o dovrebbe ricoprire — l'architettura non è più un semplice argomento di discussione, ma parte del motivo per cui le iniziative Agile falliscono e i software architect sono sempre meno credibili. Un sondaggio condotto da IASA Global pochi anni fa su un campione di 260 aziende rivela che più del 75% sta implementando una forma di pratica Agile. Delle aziende interpellate, meno del 50% hanno integrato l'architettura all'interno dei loro processi Agili. Il quadro generale che emerge è che, al momento, **Agile** e **Architettura** stanno ancora lottando per trovare una forma di collaborazione.

Secondo questo paper esiste una terza soluzione, ossia Agile attraverso Antifragile. Vale a dire, piuttosto che sforzarsi di controllare o di rimuovere il controllo stesso, cerchiamo di costruire sistemi, sia tecnici che di business, che puntino a essere Antifragili rispetto al cambiamento. Questo approccio permette la realizzazione di architetture che abilitano l'Agilità attraverso la capacità di essere **antifragili** al cambiamento.

VUCA: serve un nuovo approccio

La situazione in cui ci troviamo oggi sembra essere troppo complicata per essere affrontata con pratiche descritte in manuali datati di oltre vent'anni. Come descrive Taleb nei suoi libri (*Antifragile* e *Il cigno nero*) serve un approccio che crei sistemi in grado di prosperare nel caos, non di controllarlo o dominarlo.

Il mondo è in continua mutazione, il business cambia continuamente, e così le richieste che ci arrivano dai Clienti che ci chiedono continui cambiamenti, o nuove implementazioni, ai software che abbiamo implementato o, peggio, stiamo ancora progettando. Si parla di scenari **VUCA**, acronimo creato a partire dalle parole **Volatility**, **Uncertainty**, **Complexity**, **Ambiguity**. Di questa capacità di prosperare nella **volatilità**, nell'**incertezza** sembra proprio essercene un gran bisogno oggigiorno. Stiamo entrando in una nuova fase della **crisi del software**, che già era stata analizzata in un rapporto della NATO nel 1968 [2], ma le conseguenze sembrano essere più serie.

Come ci suggerisce Nicholas Negroponte [3]

"Computing is not about computers anymore. It's about living."

A fronte di questa analisi, il mondo **VUCA** sembra necessitare di un nuovo approccio.

Accettare la complessità

Nei primi anni di questo secolo, Dave Snowden, durante il suo lavoro alla IBM, sviluppò un'"infrastruttura concettuale" per indagare e classificare la complessità, il **Cynefin** (/kˈnɪvɪn/), termine gallese che significa *habitat*.

I sistemi aziendali contemporanei, secondo questo framework, non sono classificati semplicemente complicati, sono classificati come complessi. Un sistema è **complesso** perché non risponde in modo lineare agli input, ai cambiamenti. Per risolvere un problema di business creiamo un modello del sistema; il modello, per sua stessa natura, è una semplificazione della realtà, perché sarebbe impossibile replicare esattamente il mondo reale; una citazione di Rebecca Wirfs Brook esprime chiaramente il concetto:

"A model is a simplified representation of a thing or phenomenon that intentionally emphasizes certain aspects while ignoring others. Abstraction with a specific use in mind."

Cosa rende fragile un sistema?

I problemi nascono proprio nel momento in cui il modello semplificato della realtà è **troppo distante** dalla stessa, e le risposte che ci fornisce, se ancora è in grado di farlo, non sono più valide! È quanto descritto da **Nassin Nicholas Taleb** nel libro *Il Cigno Nero*. I problemi di business appartengono proprio alla categoria dei **sistemi complessi**, hanno comportamenti imprevedibili, e tentare di risolverli con pratiche inadeguate porta al fallimento dei progetti, proprio a causa della **piega platonica** descritta da Taleb. La soluzione a questo problema è accettare la complessità come qualcosa che non possiamo né predire, né tantomeno controllare: possiamo solo costruire sistemi **antifragili**, in grado di diventare più forti e resilienti al cambiamento del business nel tempo.

Il grado di fragilità di un sistema è generalmente funzione della sua struttura interna, di come è stato progettato. La sua abilità al cambiamento, come abbiamo già visto in precedenti capitoli, dipende dal grado di interconnessione delle sue parti, quanto queste sono **accoppiate** fra loro, più che che **coese**. È fondamentale, per l'evoluzione del sistema, che le singole parti **non** siano **accoppiate** fra loro,

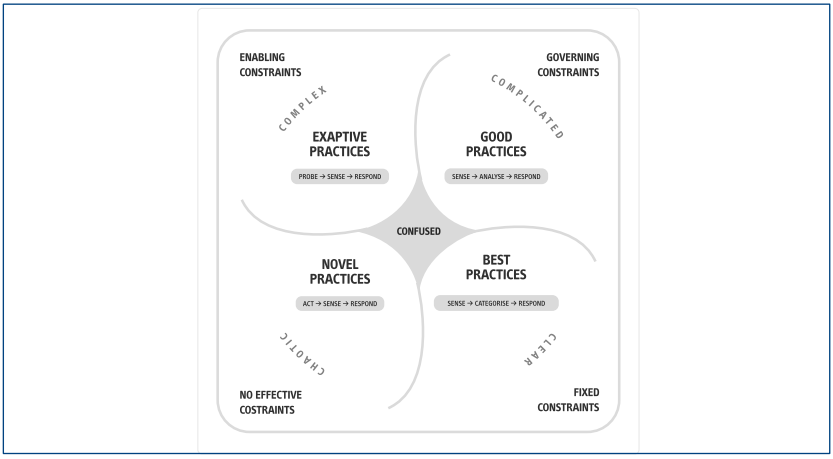

Figura 7.1 – I domini del modello Cynefin.

in modo da poter intervenire con facilità su ognuna di esse senza il timore di fare danni sulle altre.

Approcci orientati al cambiamento

Tuttavia, nel 1972, David Parmas, pioniere dell'ingegneria del software, ha sollevato un ampio dissenso sul concetto di **modularizzazione** del software affermando che è possibile costruire sistemi software migliori focalizzando l'attenzione su cosa cambierà piuttosto che su ciò che accadrà a livello funzionale. Juval Lowy, tramite il suo approccio **IDesign** [4] pone l'attenzione tra decomposizione funzionale e volatilità e propone alcune alternative che ne semplificano la realizzazione.

Questi approcci, a differenza di altri che si focalizzano sulla modularizzazione, si concentrano sugli **elementi** che possono **cambiare**, più che su requisiti concreti che soddisfano bisogni immediati. Il libro *Anti-fragile ICT System* [5] di Kjell Jørgen Hole descrive le proprietà che hanno i sistemi antifragili:

- **modularità:** componenti indipendenti, collegati fra loro;
- **legami deboli**: basso livello di accoppiamento fra i componenti;
- **ridondanza**: presenza di più istanze di un componente, per far fronte ad un guasto;
- **diversità**: la capacità di risolvere un problema in modi diversi con componenti diversi.

Dobbiamo cambiare il modo in cui progettiamo i nostri sistemi se vogliamo affrontare le sfide proposte dall'analisi **VUCA.** Seguendo i suggerimenti di Taleb, Parmas e Lowy, dobbiamo concentrarci su ciò che **non sappiamo**, accettando i nostri limiti e la nostra capacità limitata di prevedere il futuro. Questo è anche l'approccio suggerito da Dan North che sostiene che all'inizio di ogni progetto dovremmo avere l'umiltà di porci al secondo livello di ignoranza, in accordo con Phillip Armour [6] ossia accettare la nostra **mancanza di consapevolezza**: **non sappiamo di non sapere**. Un interessante approfondimento è disponibile nell'articolo *Introducing Deliberate Discovery* [7] di Dan North stesso.

Residuality Theory

Questo nuovo modo di pensare alle architetture software è proprio ciò che propone Barry O'Reilly nel suo libro *Residues* [8].

Secondo l'autore

"the future of a system is a function of its residues – the leftovers of the system after the impact of a stressor"

Barry parte da presupposti filosofici, per dimostrare matematicamente la sua idea. Il presupposto di partenza è proprio il fatto che nessuno degli approcci tradizionali è in grado di misurare **tempo**, **incertezza**, **cambiamento**. Se progettiamo la resilienza delle nostre architetture **basandoci** solo **su ciò** che **conosciamo**, o pensiamo di conoscere, senza pensare fuori dagli schemi, non otterremo mail il risultato sperato.

Quest'idea richiama il "mito della caverna" di Platone o, se preferite un approccio più moderno, ***Matrix***. Nel mito della caverna, raccontato nel VII libro della *Repubblica*, il prigioniero incatenato lì

Figura 7.2 – Un'illustrazione del "mito della caverna" di Platone.

dentro riesce a vedere solo le ombre proiettate sul fondo della grotta di ciò che passa fuori e pensa che sia quella la realtà. Ma, quando esce dalla grotta, può invece vedere la realtà vera, che proiettava quelle ombre sul fondo della caverna.

Dopo un primo momento di rifiuto, con la tentazione di tornare all'interno e di incatenarsi nuovamente nella **sua** comoda realtà, egli vorrebbe liberare tutti i prigionieri, ma non tutti sono disposti ad aprire gli occhi, a soffrire per l'esposizione alla luce del sole, e preferiscono rimanere incatenati…

Capite perché ho citato *Matrix*? Non sembri una mancanza di riguardo verso il filosofo greco: anche questo è un esempio di come si possa pensare oltre gli schemi.

L'importanza degli stressori

O'Reilly spinge il lettore a non focalizzarsi troppo durante la prima progettazione dell'architettura, quella che lui definisce un'**architettura naive**. Piuttosto è interessante analizzare quali sono gli **stressori**, ossia qualsiasi evento capace di alterare il funzionamento dell'architettura stessa. Uno **stressore** è un qualsiasi evento — dal mancato login al crollo del mercato azionario, dal server che non risponde allo scoppio di una guerra che altera tutti gli equilibri — che rompe il modello che noi ci eravamo costruiti per risolvere il problema di business.

Ecco dunque emergere la capacità di **pensare fuori dagli schem**i, l'importanza del **pensiero laterale**, su cui **Edward De Bono** ha scritto già parecchi anni fa, con libri come il celebre *Sei Cappelli per Pensare* [9].

Per individuare quanti più stressori possibile, è fondamentale esplorare a fondo il problema e, lo abbiamo già citato più volte, questo comporta il coinvolgimento di tutti i membri del team. Una volta individuati gli **stressori** il compito del *software architect* è identificare gli **impatti** che questi hanno sui **componenti** dell'**architettura naive**. Per portare a termine questo lavoro, tutto ciò di cui abbiamo bisogno è un foglio Excel.

Stressors	Sales	Warehouses	Logistics	Purchases	Payments	Tot
Login fuori servizio	1	0	0	0	0	1
Birra IPA Terminata	1	1	0	0	0	2
Credito Cliente Esaurito	1	0	0	0	1	2
Carta di Credito Scaduta	0	0	0	0	1	1
Indirizzo fuori zona di consegna	0	0	1	0	0	1
Proibizionismo del mercato della Birra	1	1	1	1	1	5
Arrivo sul mercato di un nuovo tipo di birra non previsto dal sistema	1	0	0	0	0	1
Tot	5	2	2	1	3	

Figura 7.3 – Il foglio con gli stressori nel nostro ipotetico ERP di produzione e vendita birre.

Su questo foglio riporteremo tutti gli **stressori** individuati; se prendiamo come esempio il nostro solito ERP di produzione e vendita della birra potremo ritrovarci in questa situazione.

Comepossiamo notare, non ci sono solo stressori **tecnici**, ma anche situazioni inaspettate e imprevedibili che potrebbero irrompere nel nostro mercato.

Reti booleane

Attorno al 1960 Kauffman introdusse il concetto di **Random Boolean Network** (RBN), rete booleana randomica. Si tratta di una matrice in cui gli elementi commutano da 0 a 1 in modo randomico quando stimolati, proprio come può succedere nei sistemi complessi composti da molti componenti. Durante i suoi esperimenti, Kauffman si accorse che connettendo i nodi in modi particolari otteneva di volta in volta maggiore o minore stabilità e influenzava il numero di possibili configurazioni dei loro elementi, chiamate stati di fase, presenti in ogni rete.

Si accorse che il numero di nodi (N), il numero di collegamenti tra di loro (K) e la tendenza dei nodi a comportarsi in modo simile (P), influivano sul numero di stati di fase di un sistema. Inoltre, si

accorse che collegare i nodi tra loro, creando delle dipendenze, riduceva drasticamente il numero di stati di fase, anche di molti ordini di grandezza, e che il sistema sembrava tornare costantemente allo stesso gruppo di stati di fase. Chiamò questi stati di fase **attratori**.

Le reti booleane di Kauffman hanno influenzato molti studi, perché esistono **attrattori** in ogni sistema complesso. Nonostante la molteplice possibilità di combinazione dei nodi nei sistemi complessi, un **ordine** appare **sempre**, e il sistema tende a restituire sempre lo stesso gruppo di attrattori. Nella società questo spiega la ripetibilità di alcuni pattern comportamentali mentre, restando vicini al nostro mondo, i **pattern** che tutti conosciamo e applichiamo nei nostri sistemi possono essere visti come **attrattori**. **Risolvono problemi simili che si ripetono continuamente**. Un attrattore è uno stato particolare a cui il sistema tende continuamente. Volete un esempio? Come esseri umani siamo attratti dal riposo quando siamo stanchi, dal cibo quando siamo affamati. Un buon software architect deve trovare quali sono gli attrattori per il sistema che sta costruendo.

Tra criticità ed equilibrio

Nei suoi esperimenti Kauffman individuò la proprietà di criticità, ovvero, a un certo numero di **nodi** (N) e di **collegamenti** (K) un sistema è resiliente a cambiamenti inaspettati, e allo stesso tempo non così complicato da far collassare le sue risorse. Esempi del tutto simili li troviamo quotidianamente nel nostro lavoro, comparando i sistemi monolitici (basso numero di **N** e **K**) con i sistemi a **microservizi** (**alto numero di K e N**), e sappiamo benissimo quanto sia critico trovare il giusto equilibrio fra queste soluzioni.

Un terzo valore, **P**, descrive la propensione, se preferite i **bias**, di un nodo verso un particolare insieme di comportamenti. Questo valore è applicato alle architetture software proprio per limitare il comportamento dei suoi componenti all'interno del sistema. Non a caso utilizziamo interfacce, e non classi concrete, nella comunicazione fra componenti fra loro coesi. Le metodologie e i principi

che applichiamo nel nostro lavoro — OOP, SOLID, DRY, etc. — servono a trovare il **giusto equilibrio** di **N**, **K** e **P** nei nostri sistemi. La difficoltà sta proprio nel trovare questo **equilibrio**, rendendolo a volte — sempre? — una questione di congetture, esperienza, e confronti.

Questo problema si risolve proprio **simulando** in modo casuale l'**ambiente**, finché l'architettura non mostra segni di **criticità**. Ecco lo scopo del foglio Excel. Riportare il livello di **stress** al nostro sistema e verificare se abbiamo trovato o meno questo equilibrio.

Esiste una grande differenza tra **criticità** e **correttezza**. È impossibile raggiungere la correttezza in un sistema complesso, mentre un sistema critico è auspicabile. L'obiettivo di un software architect è proprio quello di trovare la **criticità** per il sistema che sta costruendo, non l'assoluta correttezza. Quella è l'obiettivo di un matematico, o al più di uno sviluppatore, ma non di un software architect.

Come ricordato nel più volte citato *Software Architecture. The Hard Parts*:

"Don't try to find the best design in software architecture. Instead, strive for the least worst combination of trade-offs"

Alla ricerca della criticità

Il primo foglio Excel ci serve per identificare quali sono gli **stressori** in grado di compromettere il funzionamento del nostro sistema. Una volta trovati, dovremo intervenire — ricordiamo che siamo ancora in fase di progettazione! — e ottenere un nuovo sistema, costruito sui **residui** del precedente.

A questo punto sottoponiamo il nostro **nuovo** sistema a un nuovo test, con nuovi **stressori**. Ricordate? Un sistema raggiuge la **criticità** quando è **stabile** al sopraggiungere di **stressori** inaspettati). Otterremo un nuovo fogio Excel (fig. 7.4).

Nella prima colonna troviamo il nostro sistema precedente, o naive, e nella seconda quello ottenuto dai residui. A questo punto possiamo calcolare l'**indice del residuo**:

$$\mathrm{Ri} = \frac{\mathrm{R} - \mathrm{N}}{\mathrm{R} + \mathrm{N}}$$

Al numeratore troviamo la differenza fra l'architettura residua e quella naive, al denominatore la somma. L'indice sarà compreso fra

$$-1 \leq \mathrm{Ri} \leq +1$$

Se l'indice è **positivo** significa che il nostro sistema è più **resiliente** a stressori inattesi, e possiamo continuare la nostra ricerca. Se è negativo significa che non abbiamo proprio lavorato benissimo…

A un certo punto, ci accorgeremo che l'**indice** è molto vicino allo **0**: abbiamo trovato il giusto equilibrio e che, ragionevolmente, il nostro sistema sarà in grado di reggere a situazioni inaspettate.

Conclusioni

L'**agilità**, ossia l'abilità ad adattarsi al cambiamento, è spesso vista come la soluzione ai rapidi cambiamenti. Tuttavia, e lo stiamo

Stressors	Naive	Residual	Tot
Il servizio vendite non regge il carico di richieste	0	1	1
La birra IPA non è più richiesta	0	1	1
Il cliente ha cambiato Carta di Credito, ma non l'ha registrata	0	1	1
La banca del cliente è fallita	1	1	2
Il cliente ha cambiato indirizzo, ma l'evento di riconciliazione non è arrivato alla Logistica	0	1	1
Tot.	1	5	
Residual Index: 66.67%			

Figura 7.4 – Il foglio di calcolo con gli stressori ottenuti dai residui.

vedendo, non è sufficiente. L'**agilità** è spesso funzione delle circostanze, e noi possiamo adattare le risorse che abbiamo a disposizione.

Un'architettura povera può rapidamente compromettere il sistema e il business, limitando l'agilità al cambiamento. **Residual Architecture** cerca di dare un contributo alla realizzazione di sistemi **antifragili** basando la propria teoria su aspetti filosofici dimostrati poi matematicamente.

Riferimenti

[1] Giovanni Puliti, *Antifragile Manifesto. Un manifesto per gli anni a venire*. MokaByte 305, maggio 2024
https://www.mokabyte.it/2024/05/15/manifestoantifragile/

[2] Peter Naur – Brian Randell (eds.), *Software Engineering*. Report on a conference sponsored by the NATO Science Commitee. Garmisch, Germany, 7th to 11th October 1968. January 1969
`https://www.scrummanager.com/files/nato1968e.pdf`

[3] La pagina Wikipedia su Nicholas Negroponte
`https://it.wikipedia.org/wiki/Nicholas_Negroponte`

[4] IDesign
`https://www.idesign.net/about`

[5] Kjell Jørgen Hole, *Anti-fragile ICT Systems*. Springer, 2016

[6] Phillip Glen Armour , *The Five Orders of Ignorance*. Communications of the ACM 43(10), October 2000
`https://www.researchgate.net/publication/27293624_The_Five_Orders_of_Ignorance`

[7] Dan North, *Introducing Deliberate Discovery*
`https://dannorth.net/introducing-deliberate-discovery/`

[8] Barry O'Reilly, R*esidues. Time, Change, and Uncertainty in Software Architecture*. LeanPub, 2024
`https://leanpub.com/residuality`

[9] Edward De Bono, *Sei cappelli per pensare. Manuale pratico per ragionare con creatività ed efficacia*. Rizzoli, 2013

Capitolo 8

La filosofia dell'architettura del software

La filosofia dell'architettura

L'idea che **filosofia** e **architettura software** possano essere collegate potrebbe sembrare un po' stramba. Tuttavia, qualche mese fa ho avuto l'opportunità di partecipare a un corso sull'**architettura del software** veramente rivoluzionario, che mi ha aperto un nuovo mondo, e un **nuovo modo** di esplorare il **software come supporto** ai processi umani.

Nei precedenti capitoli abbiamo già affrontato la differenza, e l'importanza, del *Problem Space* e del *Solution Space*. Quali sono le domande, implicite ed esplicite, che ci poniamo quando iniziamo un nuovo progetto? Definire cosa costituisce un problema e cosa consideriamo una soluzione implica una visione del mondo, che spesso è inconsapevolmente "filosofica".

Sono certo che molti di voi, così come ho pensato io quando ho scoperto questo studio, stanno pensando come si fa a fare il lavoro che facciamo, ossia progettare architetture software, in un modo completamente diverso.

La risposta è tanto semplice, quanto sbalorditiva: dobbiamo tornare al punto di partenza, porci domande del tipo

- Come vedo il mondo?
- Come risolvo problemi?
- Cosa penso sia un problema?
- Cosa pensiamo sia un problema e cosa pensiamo sia una soluzione? Come li definiamo?

Be', se iniziamo a pensare in questo senso, un po' di filosofia cominciamo a intravvederla…

Una rivoluzione nel pensiero sul design del software

Il corso è stato tenuto da Barry O'Reilly, che è l'autore di questa nuova interpretazione dell'architettura del software che porta il nome di **Residuality Theory** di cui potete trovare un interessante video [1] nei Riferimenti in fondo al capitolo. Ovviamente

Residuality Theory non si occupa solamente di filosofia: ci sono delle solide basi matematiche a supporto del lavoro e vi invito a seguire il video.

Sistemi ordinati e disordinati

L'attuale modo di risolvere problemi e progettare architetture software evidenzia un forte distacco tra ciò che è scritto nei libri, la **teoria**, e ciò che realmente succede nel mondo, la **pratica**. Il **software** e il **business** appartengono a due diverse categorie di sistemi.

Il primo, il software, è un **sistema ordinato**, fatto di regole precise, matematiche, che devono essere rispettate affinché possa funzionare.

Il secondo, il mondo del business, appartiene alla categoria dei **sistemi disordinati**, quelli che non seguono regole precise, ma soprattutto quello in cui le regole del gioco cambiano continuamente.

Questo mi porta alla mente una frase che lessi tempo fa su una rivista di montagna, di cui non era riportato l'autore:

> "Ci sono almeno due tipi di giochi [...] I giochi finiti e quelli infiniti. Un gioco finito si gioca per vincerlo, un gioco infinito per continuare a giocare. I partecipanti a un gioco finito giocano entro certi confini ben precisi, i partecipanti ad un gioco infinito giocano con i confini."

Il software appartiene decisamente alla tipologia dei giochi infiniti, proprio perché con il software noi cerchiamo di risolvere problemi di business, o almeno questo dovrebbe essere l'obiettivo!

Se la nostra soluzione è basata solo su principi tecnologici, che sono certamente importanti, ma non sufficienti, sarà una soluzione valida fintanto che il problema di business che dobbiamo risolvere resta immutato, ma nel momento in cui le esigenze di mercato, le nostre regole del gioco, cambieranno, se il nostro sistema non si saprà adeguare, allora sarà inutile.

Ci sono diversi esempi nella storia che dimostrano come una soluzione e una visione troppo rigide abbiamo fatto fallire colossi che sembravamo incrollabili.

Blockbuster non ha saputo cogliere l'idea rivoluzionaria che sarà poi di Netflix, che abbatteva i costi aggiuntivi degli utenti che riconsegnavano le cassette in ritardo il lunedì, e quando ha visto le sue quote di mercato diminuire e ha cercato di copiare il modello di business, non ha fatto in tempo ed è fallita. Kodak è un altro esempio, e Nokia un altro ancora.

Processi decisionali nell'architettura del software

Proviamo ad esaminare il processo decisionale di un ingegnere del software quando deve prendere decisioni **architetturali**. Ala fonte ci sono tre problemi.

Il problema **tecnico**: come scalerà il mio sistema, quanto sarà resiliente, quanto tempo ho a disposizione per portare a termine la prima versione, e altre domande simili. Un libro che consiglio a questo proposito è *Fundamentals of Software Architecture* [2].

Il problema del **team**: che tipo di team abbiamo a disposizione? È un team cross funzionale indipendente, oppure abbiamo dipendenze da Team esterni?

Il problema **aziendale** del cliente: Come fa questo progetto a generare valore per l'azienda che ce lo ha commissionato? Perché è questo che dobbiamo prima capire, e poi risolvere.

Tutti noi prendiamo continuamente decisioni su questi aspetti, ma non ci poniamo mai la domanda "Perché pensi che questa decisione sia giusta?". Se leggiamo il libro *Software Architecture: The Hard Parts* [3], scopriamo che non esiste **la** risposta giusta ma che si tratta sempre di un compromesso fra diverse soluzioni possibili, tutte ugualmente giuste e sbagliate.

Secondo Barry, è proprio qui che interviene la **filosofia**. Quando ci poniamo queste domande, stiamo chiedendo a noi stessi "Come funziona il mondo?". E questa **è filosofia**. Quindi, quando pensiamo alla filosofia in termini di architettura del software, stiamo riflettendo su "Qual è la nostra filosofia di progettazione?", "Qual è la tua filosofia di team?" e, ancora, "Qual è la tua filosofia per creare valore al cliente?".

La filosofia nell'architettura del software

Da un recente corso di filosofia a cui ho partecipato, ho imparato che in filosofia non è tanto importante ciò che pensi sia vero, quanto perché pensi che sia vero. Questo è assolutamente in linea con quanto scritto anche nei libri sopra citati, che ci trasmettono due leggi fondamentali riguardo l'architettura del software:

Prima legge: Tutto è un compromesso! Corollario: "Se pensi di aver trovato una soluzione che non ha compromessi, semplicemente non hai ancora scoperto il compromesso".

Seconda legge: Il perché è più importante del come!

Quindi, quando parliamo di **architettura del software**, stiamo veramente parlando di come crediamo che i sistemi dovrebbero essere **progettati**, cosa intendiamo per lavoro di **team** e cosa intendiamo per consegnare **valore**, perché, quello che conta è che il nostro software deve, o almeno dovrebbe, consegnare valore, per avere successo.

Ciò che rende affascinante l'architettura del software, almeno per quelli come noi che la masticano tutti i giorni, è che non riguarda solo la scrittura di codice — per quello ormai abbiamo Copilot, o tanti altri strumenti simili — ma si tratta di prendere decisioni, fare compromessi e comprendere come queste decisioni influenzeranno l'intero sistema che stiamo progettando.

Una delle cose che ho imparato da questo breve corso di filosofia è che ci sono sempre **prospettive diverse**, o modi differenti, di guardare allo stesso problema. Dobbiamo imparare a traslare questo modo di pensare nel nostro lavoro quotidiano perché, se riusciamo a comprendere queste diverse prospettive, e come queste interagiscono fra loro, potremo progettare sistemi migliori.

Proviamo a pensare alla prima legge dell'architettura software. Ogni decisione che prendiamo è un compromesso. Se decidiamo di ottimizzare le prestazioni, sacrifichiamo la semplicità, ma anche il tempo di consegna. Se scegliamo una soluzione semplice, ma poco scalabile, terminiamo prima. Quindi **microservizi** o monolite? Ogni decisione che prendiamo come architetti riguarda questi

compromessi, e dobbiamo essere consapevoli di cosa stiamo sacrificando e di cosa stiamo guadagnando, per noi stessi, ma soprattutto per chiarezza nei confronti di chi ci ha commissionato il lavoro.

La filosofia, a tal riguardo, ci insegna che non si tratta tanto di trovare la risposta giusta, ma anzitutto di porre le domande giuste. Non vi ricorda qualcosa? Già Eric Evans, nel suo iconico libro blu, sosteneva l'importanza di trovare un linguaggio e un modello comune fra tecnici ed esperti del business su cui porre domande che fossero comprensibili a tutti. Quando parliamo di architettura del software, non si tratta di identifica il miglior design, la soluzione migliore, ma di trovare le domande migliori che possiamo porci, e possiamo porre, per comprendere il problema ed esplorare lo spazio delle soluzioni. La filosofia ci aiuta a migliorare noi stessi nel formulare queste domande e nell'esplorare le diverse possibilità. Ci insegna a migliorare nella comprensione dei problemi e nell'esplorare i diversi modi in cui possiamo affrontarli.

L'architettura software è affascinante perché non riguarda solo la tecnologia, ma riguarda anche le persone, sia durante la progettazione, ma soprattutto dopo, quando consegniamo il lavoro. Nella prima fase dobbiamo pensare a come unire le persone, come allineare i diversi obiettivi, come aiutarle a lavorare insieme in modo efficace. La filosofia ci aiuta ad esplorare i diversi modi in cui possiamo unire le persone e favorire la loro collaborazione.

La filosofia ci insegna ad abbracciare la complessità. Ci insegna che il mondo non è semplice, che i problemi non sono semplici e che le soluzioni non sono semplici. Dobbiamo saper cogliere questa complessità quando progettiamo sistemi software, dobbiamo capire che non esistono risposte, e soluzioni, semplici, e che ogni decisione che prendiamo comporta compromessi. Non è sufficiente replicare la soluzione che abbiamo adottato in precedenza per un progetto simile! Dobbiamo farci guidare dalla filosofia nella comprensione, e nella gestione, della complessità.

Per concludere, la filosofia ci aiuta a riflettere, a porci domande, a renderci diversi da un qualsiasi Copilot! I nuovi strumenti di

Intelligenza Artificiale scrivono codice migliore del nostro, in un tempo infinitamente inferiore, ma questo codice funziona solo se abbiamo posto le **domande giuste** al generatore di turno; altrimenti sarà soltanto un modo diverso, anche se più veloce, del solito "copy and paste from stackoverlow" con i risultati che tristemente conosciamo.

Filosofia e architettura: le radici del pensiero progettuale

La filosofia è un argomento che mi interessa da un po' di tempo, quindi, quando ho visto un corso che univa l'architettura software con la filosofia, ho deciso di partecipare al corso di Barry O'Reilly. Attraverso il lavoro svolto per il suo dottorato in scienza della complessità e design del software, Barry O'Reilly propone un approccio innovativo che va oltre le metodologie tradizionali, sfidando le concezioni consolidate su come progettiamo i sistemi.

Il punto di partenza proposto è un aspetto sempre ignorato nello sviluppo del software, ossia le basi filosofiche che guidano le nostre decisioni. Quando affrontiamo un problema, ci poniamo implicitamente domande fondamentali

L'influenza del pensiero complesso

Il concetto di **complessità** è centrale quando si parla di architettura software, perché, come già detto più volte, scrivere software non significa semplicemente trovare soluzioni tecniche, ma coinvolge il modo in cui percepiamo e strutturiamo il mondo attorno a noi.

L'attuale approccio, che si concentra nella scomposizione del problema in parti più piccole, non è sempre sufficiente per affrontare i **sistemi complessi** e **interconnessi** di oggi. Serve, secondo Barry O'Reilly, una visione combinata di scienza della **complessità** e riflessione **filosofica** che ci permetta di vedere il software non come insieme di componenti isolati, ma come un sistema **vivente**. L'idea è quella ci creare strutture che possano evolvere in modo **organico**, **adattandosi** ai cambiamenti senza compromettere la stabilità.

Riformulare il ruolo dell'architetto software

Il lavoro di un software architect deve andare oltre la creazione di sistemi meramente tecnici; l'architetto deve diventare un **pensatore strategico** che lavora con **modelli mentali** complessi. Le decisioni architetturali non derivano solo dall'esperienza o dalle *best practice*, ma anche dal modo in cui l'architetto interpreta il contesto e bilancia diversi fattori come flessibilità, scalabilità e semplicità.

Conclusione: filosofia come strumento per il futuro del software

Il messaggio di O'Reilly è chiaro: per progettare sistemi migliori, dobbiamo esplorare le **radici** del nostro **pensiero**. Solo comprendendo il modo in cui percepiamo il mondo e prendiamo decisioni, possiamo creare architetture che non siano solo funzionali, ma anche sostenibili e pronte per il futuro.

La filosofia, quindi, non è un'aggiunta accessoria allo sviluppo software, ma una componente essenziale per affrontare le sfide di un mondo sempre più complesso. Attraverso un approccio che integra logica, intuizione e creatività, gli architetti del software possono ridefinire il loro ruolo e guidare il cambiamento.

Riferimenti

[1] Barry O'Reilly, *An Introduction to Residuality Theory*. NDC Oslo 2023 (video) `https://www.youtube.com/watch?v=0wcUG2EV-7E&t=773s`

[2] Mark Richards – Neal Ford, *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020

[3] N. Ford – M. Richards – P. Sadalage – Z. *Dehghani, Software Architecture: The Hard Parts*. O'Reilly Media, 2021

Capitolo 9

Event Sourcing is not Event Streaming

Le parole sono importanti

Nel mondo dello sviluppo software, i termini di moda spesso generano confusione, soprattutto quando suonano simili ma si riferiscono a concetti completamente diversi. Due di questi termini "confondibili" sono **Event Sourcing** ed **Event Streaming**. Sebbene entrambi siano fondamentali nelle architetture moderne, essi servono scopi distinti e operano in ambiti completamente differenti.

Event Sourcing

Event Sourcing è un modello architetturale in cui ogni cambiamento di **stato** di un sistema, più precisamente di un aggregato nell'ambito di **Domain-Driven Design**, è rappresentato come un **evento** immutabile e memorizzato in **ordine cronologico**. Questo approccio consente di **ricostruire** lo **stato** del sistema in qualsiasi momento semplicemente rileggendo gli eventi.

Un database che supporta una simile architettura deve essere in grado di garantire la concorrenza ottimistica, il che significa che la lettura degli eventi in esso memorizzati è sempre consentita, senza nessuna restrizione: i vincoli riguardano esclusivamente gli aggiornamenti dei record. Già, ma trattandosi di una sequenza di fatti accaduti nel nostro sistema, ed essendo questi fatti immutabili, di fatto i vincoli intervengono solo nel momento in cui si cerca di aggiungere (*append*) un nuovo evento, perché di fatto non è consentito modificare un evento!

Un **Event Store** assegna un numero di versione, un **progressivo**, ad ogni evento che viene memorizzato. Quando due operazioni cercano di appendere contemporaneamente un evento sullo stesso stream dello stesso aggregato, **Event Store** verifica la versione. Se la versione dell'aggregato che sto cercando di salvare coincide con l'ultima versione dell'evento nello stream, allora salva e incrementa la versione. Negli altri casi respinge al mittente la richiesta.

Questa proprietà è fondamentale per garantire la **Strong Consistency** del nostro aggregato. Un sistema a eventi possiede sì la **£ventual Consistency** ("coerenza finale") per quanto riguarda

le repliche — i Read Model per attenerci alla nomenclatura di CQRS+ES — ha anche la **Strong Consistency** ("coerenza forte") nel modello di scrittura.

Riepilogando, un **Event Store** deve poter

- garantire una **cronologia completa** delle modifiche di stato e abilitare funzionalità come audit e retrospettiva;
- memorizzare una sequenza immutabile di eventi.

Un database Event Store è il compagno di viaggio ideale per un'architettura **CQRS** in cui si vogliono salvare gli eventi si sistema, i **Domain Event**, per poter ricostruire lo stato dei nostri aggregati partendo proprio da questi eventi.

Un esempio classico, e concreto, potrebbe essere quello di un e-commerce, in cui Event Sourcing potrebbe rappresentare eventi come "OrdineDaCarrelloCreato", "PagamentoTramiteCartaDiCreditoConfermato", "OrdineConPrioritàAltaSpedito", consentendo non solo di ricostruire l'intera cronologia di un ordine, ma anche di capire il processo di business sottostante. Proprio per questo motivo un Event Store è utilizzato come **source-of-truth** del sistema.

In conclusione, l'Event Sourcing è caratterizzato da:

- stato derivato dagli eventi;
- persistenza immutabile;
- audit log naturale;
- riproducibilità dello stato;
- event replay;
- time travel;
- source of truth.

Event Streaming

L'altro protagonista di questo confronto è l'**Event Streaming**. Event Streaming è focalizzato sullo **spostamento** in tempo reale di informazioni— in pratica dati — da un oggetto a un altro all'interno di un sistema distribuito. Strumenti tipici che implementano questo paradigma sono **Kafka**, **RabbitMQ**, **Azure Service Bus**, **AWS SQS** e **SNS**.

Ogni elemento di un sistema distribuito può sottoscriversi a questi flussi e ricevere appunto in tempo reale informazioni che servono ad attivare azioni varie, come aggiornamento di database, attivazione o disattivazione di un'apparecchiatura in una piattaforma IoT, acquisto o vendita di un titolo in un sistema bancario, etc.

I sistemi per l'Event Streaming non sono progettati per essere dei **database**! Ovviamente hanno delle funzionalità di storage, ma non per essere utilizzate per lo scopo di un tradizionale database. Queste funzionalità sono state aggiunte per garantire resilienza e tolleranza ai guasti e, grazie ad esse, scenari come quelli in cui il broker interrompe il proprio servizio e il sistema ricevente non è disponibile sono perfettamente gestiti.

Qualcuno potrebbe obiettare che anche questi sistemi gestiscono un log **append-only**, proprio come un **Event Store**, ma gli eventi in esso contenuti non possono essere utilizzati come **source-of-truth**, almeno non come nel caso di un **Event Store**!

In conclusione, l'Event Streaming è caratterizzato da:

- elaborazione real time;
- pub/sub;
- scalabilità;
- immutabilità degli eventi;
- indipendenza tra produttore e consumatore;
- elaborazione distribuita;
- integrazione tra sistemi *legacy* e "moderni".

Confronto

Gli **eventi** sono la nostra **source-of-truth** solo se li si utilizza nel modello di scrittura come base per la reidratazione dello stato di un aggregato. Se si utilizzano gli eventi per costruire una vista, allora si esternalizza la verità su un altro storage, che paradossalmente, potrebbe ricevere aggiornamenti anche da altri stream, violando completamente la "catena di consistenza" dei fatti accaduti nel nostro aggregato. L'uso di strumenti per l'**Event Streaming** come strumenti di **Event Sourcing** porta proprio a questa conclusione.

Event Sourcing	Event Streaming
Persistenza dello stato	Trasmissione real time di dati
Eventi come cambiamento di stato di un aggregato	Eventi come comunicazione fra sistemi distribuiti
È un pattern architetturale per la gestione dello stato	È un paradigma di integrazione e comunicazione

Tabella 9.1 – Confronto tra Event Sourcing ed Event Streaming.

Un altro aspetto critico è quello di mantenere gli stream brevi. In Kafka, ad esempio, non importa quanti eventi ci sono in un Topic, ossia, non importa quanto sia lungo il Topic stesso, perché si tratta di un canale di comunicazione.

Il discorso è completamente diverso nel caso di un **Event Store**, perché in questo caso stiamo parlando di database, non di canali di comunicazione; quindi, più eventi ci sono nello stream, maggiore è la sua dimensione, peggiori sono le prestazioni. Questo aspetto non impatta solo sulle prestazioni, ma rende difficile il versionamento degli eventi, aspetto cruciale quando si parla di CQRS+ES, ma che esula dagli aspetti di questo capitolo.

La tabella 9.1 prova a riepilogare le maggiori differenze fra Event Sourcing e Event Streaming.

Conclusioni

È facile cadere nella trappola dell'utilizzo di un sistema di **Event Streaming** come **tuttofare**. Le promesse sono molto allettanti, ma alla lunga si ottengono solo problemi. Costruire soluzioni software, lo ripeto, non è un esercizio di stile: è risolvere problemi di business.

I problemi di business mutano continuamente, costringendo i nostri sistemi ad adeguarsi; quindi, utilizziamo gli strumenti per il ruolo per cui sono stati progettati e, dalla lista delle cose da risolvere, toglieremo della complessità accidentale inutile.

Capitolo 10

Il ruolo del Software Architect

Cosa vuoi fare da grande?

Volevo fare il Software Architect, quindi, come per ogni cosa che chiunque vuole fare, ho cominciato a documentarmi sul ruolo di questa figura. E la domanda che mi sono posto è "Qual è il compito di un Software Architect?"

Inizialmente, suppongo come molti, mi sono formato per fornire soluzioni architetturali adeguate, proporre soluzioni con il giusto mix di semplicità e complessità per supportare i progetti di chi si affidava al mio team per costruire la propria soluzione. Poi, frequentando e confrontandomi con le persone giuste, ho scoperto che, forse questa è solo una parte del lavoro del Software Architect.

C'è un'altra domanda, che negli capitoli di questa serie, mi sono già posto, ossia: "Perché sviluppiamo software?". È quello che dovremmo chiederci tutti. Noi sviluppiamo software per risolvere problemi di business. Il nostro lavoro non è scrivere del codice che funziona: per quello ci sono i kata, molto utili per affinare le nostre tecniche, per impararne di nuove, ma non risolvono problemi di business. Il nostro compito è **progettare** sistemi che **funzionino** e **risolvano problemi**.

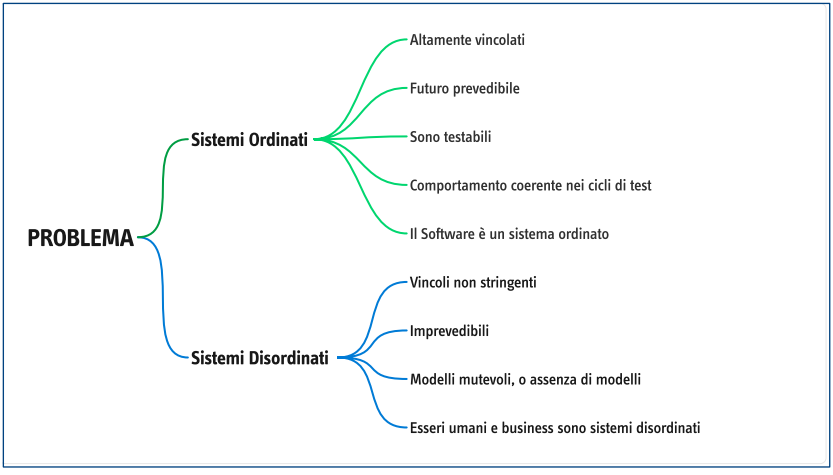

Figura 10.1 – Sistemi ordinati e sistemi disordinati.

Mettere insieme ordine e disordine

Ed è proprio qui che nascono i veri problemi, perché nella realtà esistono due tipi di sistemi, i sistemi **ordinati**, e i sistemi **disordinati** (fig. 10,1).

Il **software**, come potete notare dall'immagine, appartiene ai sistemi **ordinati**, e infatti è costituito da regole matematiche e ingegneristiche rigorose e prevedibili. Il **business**, invece, appartiene alla categoria dei sistemi **disordinati**, così come noi esseri umani.

Il compito del Software Architect, dunque, è quello di progettare un sistema ordinato che risolva i problemi di un sistema disordinato. Avete presente il gioco delle forme, in cui bisogna incastrare le forme nel posto corretto? Provate a incastrare la forma cubo nell'alloggiamento in cui deve entrare la forma cerchio… Ecco, il Software Architect ha questo obiettivo.

Per risolvere problemi di business, dobbiamo imparare il linguaggio del business o, meglio, dobbiamo essere in grado di creare un **linguaggio**, e un **modello**, che sia noi che il business possiamo capire, e su cui possiamo ragionare. Dobbiamo essere in grado di **collegare** fra loro diversi **linguaggi** fra **persone** che hanno diversi background che compongono il team che realizzerà il prodotto. Parafrasando la metafora di Gregor Hohpe, il valore di un software architect, si misura in base ai piani di un grattacielo che è in grado di coprire. Il suo valore sta proprio nell'aiutare tutti i partecipanti a trovare questo linguaggio comune, e di conseguenza, un **modello comune** su cui costruire il prodotto finale.

Linguaggi e modelli

Su questi punti E. Evans non aveva dubbi, e lo abbiamo visto nei dettagli. **Ubiquitous Language** e **Bounded Context** sono due pattern fondamentali per **Domain-Driven Design**, che si basano proprio sul **linguaggio** e sulla **modellazione** della realtà.

Il primo elemento, il **linguaggio**, è necessario per stabilire di cosa stiamo parlando in modo chiaro, senza argomenti o conoscenza implicite. Tutto deve essere chiaramente esplicitato e compreso

dall'intero Team. Anche nelle conversazioni fra persone succede la stessa cosa. Se vi unite ad una conversazione fra amici quando questa è già iniziata, la domanda spontanea che viene da porre è "Di cosa state parlando?". Senza questa richiesta di **identificazione** del **contesto** si rischia seriamente di deragliare, di parlare di un argomento simile, ma non dell'argomento stesso della conversazione. La cosa migliore che ci può capitare è una risata collettiva, la peggiore rispondere in maniera inopportuna a una domanda. Senza conoscere il contesto è impossibile creare un modello mentale su cui ragionare.

Nella creazione di prodotti software si parla molto spesso di **modelli**, e lo stesso E. Evans ne parla dettagliatamente nel suo iconico blue book, ma cosa intendiamo esattamente per modello? L'architettura del software è basata sui modelli, la **programmazione** a **oggetti** ne ha fatto il punto focalizzante della sua teoria. **Modellare** la realtà in **oggetti** ideali. Spesso, però, questi modelli falliscono proprio nel loro intento principale: essere un'immagine coerente della realtà. Ma perché?

I filosofi non erano programmatori, però...

Se torniamo indietro, ai tempi delle grandi discussioni filosofiche del mondo greco, possiamo stare certi che non si trattasse di scegliere fra programmazione funzionale o paradigma a oggetti. Eppure, nelle enunciazioni di due personaggi che non la pensavano esattamente allo stesso modo riguardo alla visione della realtà possiamo riscontrare quasi la stessa divergenza che c'è fra programmazione **funzionale** e **Object** Oriented Programming.

Il primo personaggio è Eraclito (535-475 a.C.) secondo il quale "Nessun uomo entra due volte nello stesso fiume, perché non è lo stesso fiume e lui non è lo stesso uomo". Ossia, il mondo, secondo Eraclito, è un flusso in costante divenire.

Il secondo filosofo è invece Platone (428-348 a.C.) — già citato a per il "mito della caverna", il quale, più di un secolo dopo, sosteneva che "Il mondo sensibile che ci circonda è solo una copia imperfetta delle Forme che rappresentano la vera realtà".

Due visioni opposte del mondo. Non dimentichiamo che, nella filosofia occidentale moderna, il pensiero di Platone, insieme a quello di Aristotele, ha costituito per secoli il nucleo fondamentale delle nostre convinzioni. Molti aspetti del nostro modo di interpretare il mondo e le cose che ci circondando, derivano dal pensiero filosofico di Platone e di coloro che, a più riprese dall'età tardoantica fino al rinascimento, si sono rifatti al pensiero di questo filosofo. Siamo figli di Platone quando adottiamo il paradigma a oggetti dell'OOP.

Domain-Driven Design — che peraltro non rinnega OOP — è invece più vicino al pensiero di Eraclito, perché, pur conscio dell'importanza del modello per capire e affrontare il problema, è altresì convinto convinto che un **modello** è **soggetto** a **mutazioni**, e che è immerso in un fluido divenire. Il che, a guardar bene, è anche alla la base della metodologia Agile.

Il grosso problema, infatti, non è il modello in sé, ma è il modello pensato con l'idea che sia corretto e valido **per sempre**. Questa è la "trappola" reale, e quando scopriamo che ciò che sappiamo — o, peggio, che **pensiamo di sapere** — è pericolosamente distante dalla realtà, allora viene prodotto il **cigno nero**, secondo la celebre metafora di Nassim Taleb.

Lo scopo di un modello

A cosa mi serve un modello? Con un esempio pratico è molto facile capirlo. Io sono un grande appassionato di montagna, in tutte le stagioni. Quando voglio raggiungere un rifugio che non conosco, la prima cosa che faccio è procurarmi le informazioni sul sentiero da seguire e sull'altimetria da superare. Una carta topografica che mi fornisca informazioni sugli elementi naturali che contraddistinguono la zona in modo peculiare (fiumi, laghi, passi, etc.), ma anche le strutture inserite dall'uomo, come appunto i sentieri: è il modello perfetto per risolvere il mio problema.

Se invece conosco bene il modo per raggiungere il rifugio, ma la mia intenzione è andare a ripetere qualche via di arrampicata che non conosco in zona, allora mi serve un altro tipo di mappa. Mi

serve uno schizzo della via di arrampicata con le indicazioni della lunghezza dei tiri di corda, delle protezioni in loco, con un'indicazione delle difficoltà di ogni tiro e altre informazioni simili. Non mi interessa la fotografia della montagna che andrò a scalare, perché, per quanto bella essa possa essere, non mi fornisce le informazioni necessarie.

Un appassionato di fotografia potrebbe obiettare che, in entrambi i casi — carta topografica o schema della via di arrampicata — mi sto perdendo la bellezza del paesaggio perché nessuna mappa da me scelta mi restituisce la bellezza dei luoghi. E dal suo punto di vista ha perfettamente ragione; ma abbiamo obiettivi differenti!

Secondo Rebecca Wirfs-Brock, un **modello** è la rappresentazione semplificata di una cosa, o di un fenomeno, che enfatizza intenzionalmente alcuni aspetti, trascurandone altri. È un'astrazione con un uso specifico in mente. E un'astrazione può essere precisa solo se il contesto è fissato e compreso da tutti i partecipanti alla comunicazione.

Lo schizzo della via di arrampicata è funzionale solo se tutti quelli che verranno con me hanno come obiettivo la ripetizione della via. Se vogliono godersi la passeggiata, hanno fra le mani un modello inadatto.

Ulteriori considerazioni di ordine "filosofico"

Tornando nel nostro mondo, ossia la progettazione di sistemi che risolvono problemi di business, il **linguaggio** diventa uno strumento fondamentale per la comprensione chiara del problema fra tutti i membri del Team. Se pensiamo all'esempio precedente, non possiamo permetterci di mescolare **escursionisti** e **scalatori** e pretendere che utilizzino la stessa mappa (modello) per raggiungere il loro diverso scopo. C'è bisogno di un'intesa sui termini del problema da risolvere e sul conseguente modello da realizzare.

Secondo Wittgenstein, prima ancora di Evans, "I limiti del mio linguaggio significano i limiti del mio mondo": esattamente allo stesso modo, i limiti dell'**Ubiquitous Language** costituiscono i limiti, i

bound, del nostro contesto (**Bounded Context**). Il linguaggio non è solo descrittivo, ma **costruttivo**, e definisce il modo in cui i team comprendono e affrontano il dominio, o una parte di esso.

Ontologia

Un **modello**, in Domain-Driven Design, può essere visto come una **rappresentazione ontologica** parziale, costruita per riflettere aspetti rilevanti di una realtà complessa. L'**ontologia** è la branca della filosofia che studia **ciò che esiste** e il **come possiamo classificare** e **comprendere** le cose che ci circondano. È un modo per organizzare la realtà, stabilendo proprio le categorie di oggetti e comprendendone le relazioni tra di essi. In pratica, è proprio come costruire una "mappa" di ciò che esiste e del modo in cui questi elementi si collegano fra loro. Esattamente ciò che cerchiamo di fare con i pattern **Bounded Context** e **Context Mapping** quando applichiamo DDD nei nostri progetti.

Teleologia

Un'altra domanda fondamentale che ci dobbiamo porre, quando cerchiamo di modellare un problema, è il suo perché, il suo scopo. "Il perché è più importante del come" (seconda Legge dell'Architettura del Software). La **teleologia** è un'altra branca della filosofia che si occupa proprio dello studio degli scopi o delle finalità di un processo o di un'entità. Si concentra proprio sul *perché* qualcosa accade, cercando di capire il fine o l'obiettivo che una determinata cosa, o evento, sta cercando di raggiungere. Qual è lo scopo di questo modello in questa parte del sistema?

Appare chiaro ora perché, quando esploriamo il dominio su cui andremo a lavorare, è importantissimo, se non addirittura fondamentale, che siano presenti tutti i soggetti coinvolti dal progetto.

Fenomenologia

Nell'analisi del dominio che riguarda il prodotto che intendiamo creare, dovrà essere presente non solo chi ha **pensato** di realizzarlo,

ma chi dovrà **utilizzarlo**, chi dovrà **venderlo**, chi dovrà **realizzarlo** e chi dovrà **finanziarlo**. Dobbiamo valutarne, e convalidarne, tutti gli aspetti. Per questo compito ci viene in aiuto la **fenomenologia**, che è lo studio dell'esperienza umana e di come percepiamo e interpretiamo la realtà. Invece di concentrarsi su ciò che esiste **indipendentemente** da noi, la fenomenologia si focalizza su come viviamo, sperimentiamo e comprendiamo le cose nel nostro mondo, su come vediamo e sentiamo le cose. È come un'analisi dettagliata dei fenomeni così come appaiono alla nostra coscienza. Ecco perché è fondamentale il contributo di tutti, perché ognuno ha la propria percezione della realtà che lo circonda; e noi, in qualità di software architect, le dobbiamo considerare tutte.

Bello, sì... ma io volevo fare il Software Architect

Tutto molto bello e interessante, ma perché è importante modellare correttamente? Vediamo di analizzare, nel quotidiano di un architetto software, l'impatto di questo approccio, partendo da una considerazione fra **coupling** & **cohesion**.

Due componenti sono tanto più integrati fra loro, ossia la loro forza di integrazione è molto più forte, quanta più **conoscenza** si **scambiano**. Viceversa, sono tanto più distanti fra loro quanta meno conoscenza hanno uno dell'altro.

In pratica, nel primo caso, se ne modifico uno, quasi certamente devo modificare anche l'altro; viceversa, nel secondo caso, potrei non aver bisogno di modificare entrambi i componenti per

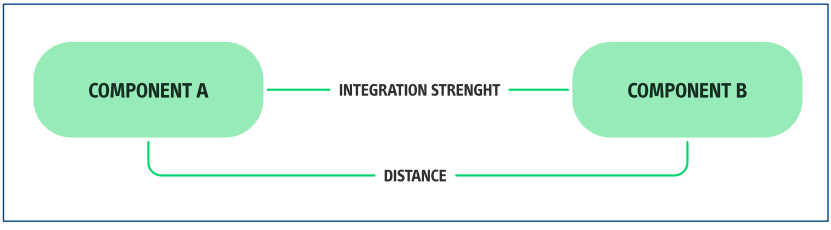

Figura 10.2 – La forza di integrazione di due componenti.

	COESIONE Alta	COESIONE Bassa
DISTANZA Alta	light coupling	loose coupling
DISTANZA Bassa	high cohesion	low cohesion

Tabella 10.1 – Accoppiamento, disaccoppiamento, alta o bassa coesione.

apportare modifiche a uno di essi. È sbagliato, quindi, avere troppo accoppiamento? "Dipende", è la classica risposta del consulente. Ma riassumiamo le possibili combinazioni in una tabella e vediamo di capire quando scegliere la coesione, e quando scegliere il disaccoppiamento.

All'interno di un **determinato Bounded Context**, ossia dentro al modello, mi aspetto una **forte coesione** fra i componenti che lo compongono, uno scambio intenso di informazioni e conoscenza. Non ho paura di questo **forte accoppiamento**: dopotutto, se devo modificare il comportamento di un Bounded Context, accetto di modificare i componenti al suo interno, perché so che l'ho strutturato correttamente, e quindi è isolato dagli altri.

Viceversa, se i due componenti appartengono a **due Bounded Context diversi**, allora, fra i due, ci deve essere **accoppiamento basso** o nullo, perché non voglio doverli modificare entrambi in caso di modifica ad uno solo di essi.

Bounded Context e Microservices

Questo chiarimento è fondamentale perché spesso associamo il pattern del **Bounded Context** ai microservizi, parlando indifferentemente dell'uno o dell'altro come se fossero la stessa cosa, ma **non lo sono affatto**.

I confini del **Bounded Context** sono costruiti attorno allo scopo del modello, quindi, in questo caso, ricerchiamo un'**alta coesione**

fra i suoi componenti. I confini di un **microservizio**, contrariamente, sono costruiti attorno alla dimensione dell'immagine dell'*artifact* da distribuire; per questo motivo cerchiamo una grande distanza dagli altri microservizi, o se preferite, un **bassissimo accoppiamento**.

Bounded Context e microservizi hanno forze trainanti diverse fra loro: ciò che mantiene sana la nostra **codebase** è la chiarezza dei confini semantici fra i Bounded Context; che si tratti di un monolite modulare o di un microservizio poco importa.

La separazione fisica la dobbiamo fare solo se e quando ne vale la pena. Spesso è inutile partire subito con i microservizi, aggiungendo complessità accidentale al progetto; manteniamo la semplicità e vediamo cosa succede, vediamo cosa impariamo a mano a mano che procediamo con l'esplorazione del dominio su cui stiamo lavorando.

Il rischio più grosso in cui possiamo cadere, come software architect, è quello di prendere **decisioni importanti** nel momento in cui la nostra **conoscenza** del **dominio** è **bassa**, proprio all'inizio del lavoro. Dobbiamo prima ridurre il debito tecnico, e poi decidere.

Parafrasando Socrate: "Ciò che devo imparare lo imparo facendolo, conscio che l'unica cosa che so è di non sapere".

Lo stesso Dan North nel suo ormai famoso articolo *Introducing Deliberate Discovery* [1] sostiene questo concetto, aggiungendo che, all'inizio di ogni progetto dovremmo avere l'umiltà di porci almeno al **secondo livello di ignoranza** [2] ossia "*So di non sapere qualcosa*". Troppo spesso ci troviamo forzati a proporre soluzioni in tempi stretti, ma una buona architettura richiede tempo, richiede conoscenza, altrimenti il rischio di sbagliare, di per sé comunque altissimo, è quasi certo. Come dire: se pensi che una buona architettura sia costosa… provane una mediocre e poi fai i conti…

Conclusione

Il ruolo del **Software Architect** va ben oltre la definizione di **strutture** e **pattern tecnici**: è un equilibrio costante tra **logica** e **adattabilità**, tra **rigore** e **flessibilità**. Non si tratta solo di scrivere codice o

progettare soluzioni ottimali, ma di creare un **linguaggio comune** che unisca **tecnologia** e **business**, **team** di **sviluppo** e **stakeholder**.

Un buon Software Architect non cerca risposte definitive; piuttosto, fornisce opzioni, **modelli** che si **evolvano** con la realtà del contesto. Ogni decisione architetturale deve essere presa con consapevolezza, sapendo che il **cambiamento** è **inevitabile** e che la vera sfida sta nella capacità di apprendere, adattarsi e guidare il team attraverso le complessità del dominio.

Alla fine, il valore di un Software Architect non si misura solo nella qualità dell'architettura realizzata, ma nella capacità di rendere il team più forte, più coeso e più capace di affrontare le sfide del futuro.

Riferimenti

[1] Dan North, Introducing Deliberate Discovery. 2010
`https://dannorth.net/introducing-deliberate-discovery/`

[2] I 5 livelli di ignoranza
`https://wiki.c2.com/?OrdersOfIgnorance`

“Oltre il codice: architetture software per un mondo complesso” potrebbe essere un buon modo per sintetizzare quanto il lettore troverà in queste pagine. Un libro rivolto a sviluppatori e architetti del software, essenziale per spostare il focus dalla soluzione allo spazio del problema.
Come si possono realizzare sistemi software capaci di rifettere la realtà del business e durare nel tempo? Che cosa significa creare una architettura evolutiva in grado di adattarsi al mondo complesso e mutevole in cui ci troviamo a operare? Qual è il ruolo del Software Architect nell’attuale panorama dello sviluppo software?
Architetture evolutive, DDD, microservizi. Uno sguardo d’insieme **risponde a queste e ad altre domande con un percorso in dieci capitoli che non si ferma al “come” ma cerca il “perché”, con una precisa filosofia.**

L’autore
Alberto Acerbis lavora da anni come Software Architect e si definisce “uno sviluppatore backend”. La sua eterna curiosità lo porta comunque a interessarsi anche all’altro lato del codice, consapevole che “scrivere” software significhi principalmente risolvere problemi di business e fornire valore al cliente. In questo, ritiene i pattern del DDD un grande aiuto.
Attualmente ricopre il ruolo di Software Engineer presso Intré.