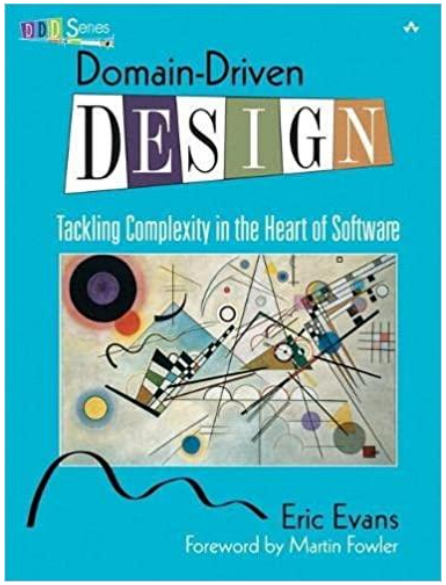
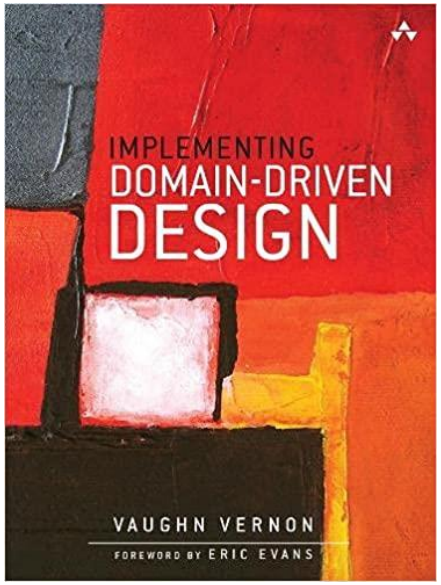
Gilda i3D

Ovvero: Impariamo Domain-Driven Design

INTr3

I Sacri Testi





… e quelli apocrifi

Perché DDD

- La tecnologia è destinata a diventare, inevitabilmente, obsoleta!
- La comprensione del Dominio diventa il punto centrale nello sviluppo software
- I Domain Expert sono la nostra risorsa principale
- Non parlano il nostro linguaggio
- Ci serve un Ubiquitous Language

Capire bene per implementare meglio

- **Comprensione del Dominio**
  - Ubiquitous Language
  - Bounded Context
- **Strategia**
  - Context Mapping
- **Tattica**
  - Entity
  - Value Object
  - Aggregate

Definitions

- **Domain**
  - Rappresenta la sfera di conoscenza relativa all'area per la quale stiamo sviluppando l'applicativo.
  - Ci sono più Domini in un'applicazione
- **Model**
  - Un'astrazione che descrive aspetti selezionati di un Dominio, che viene utilizzato per risolvere problemi relativi al Dominio stesso.

Definitions

- **Ubiquitous Language**
  - Un linguaggio strutturato intorno al modello di dominio e utilizzato da tutti i membri del Team per preparare la documentazione, scrivere il codice, richiedere informazioni nei corridoi. Se questo linguaggio cambia, tutta la catena si deve aggiornare.
- **Content**
  - Il contesto nel quale appare una parola, o un'affermazione, che ne determina il significato. Questo tipo di affermazioni hanno senso solo in quel determinato contesto.

Definitions

- **Bounded Context**
  - La descrizione di un confine (tipicamente una parte sistema gestita da un Team) all'interno del quale vige un Ubiquitous Language ed è definito da un preciso Modello dedicato.
- **Context Mapping**
  - Il modo in cui ogni singolo Bounded Context si relaziona con gli altri. Esistono diversi modelli di relazione, da applicare in base alle necessità. Il modello di comunicazione definito fra due Bounded Context potrebbe cambiare nel tempo.

Strategic Distillation

E’ fondamentale categorizzare le porzioni del nostro dominio sulla base della loro rilevanza nei confronti del business.

DDD definisce tre categorie principali di domini, proprio in base alla loro rilevanza verso il business.

Strategic Distillation

- **Generic Subdomain**
  - Questa porzione di dominio non è specifica del nostro business, pur essendo necessaria. Possiamo pensare ai software per la fatturazione, o per la reportistica generale. Spesso non scriviamo codice di questo tipo, ma ci limitiamo a comprare soluzioni già pronte.

Strategic Distillation

- **Supporting Subdomain**
  - Pur non essendo al centro del nostro business, questa porzione di dominio è sufficientemente specifica per meritare di essere sviluppata ad hoc. Non sarà ciò che ci distinguerà sul mercato, quindi sarà fondamentale tenere i costi di sviluppo sotto controllo.

Strategic Distillation

- **Core Subdomain**
  - E’ la porzione di dominio che ci permette di avere un vantaggio competitivo sulla concorrenza. Quella che ci permette di fare quello che gli altri, la concorrenza, non fanno.
    Se abbiamo chiaro quale è il fattore differenziante per il dominio che stiamo affrontando, questo è il punto, nel nostro software, dove investirci tempo e risorse perché avranno un impatto positivo sul business.

Strategic Distillation

- **Cosa non è**
  - La componente strategica dei componenti software è qualcosa che non è immutabile, ma cambia nel tempo. Ciò che è strategico oggi, per il mercato su cui stiamo lavorando, potrebbe diventare la prassi fra qualche anno, obbligandoci a cambiare strategia.
    La differenza fra una componente core ed una di supporto, è che la prima non si deve rompere, ne fermare, se vogliamo essere competitivi. La seconda, è sì importante, ma se si guasta, o si ferma, ci basta sistemarla. E' critica, ma non fondamentale

Classificazione dello Sviluppo

Dobbiamo classificare le parti del nostro dominio in maniera chiara perché sviluppare software nel core domain è un mestiere totalmente diverso da quello di chi sviluppa nell'area di supporting.

Nel Core Domain si ragiona per esperimenti, si accetta la possibilità di riscrivere più volte la soluzione, di seguire una traiettoria non propriamente chiara, perché questa è la strategia giusta da seguire, non perché vogliamo essere fighi!

Per questo è importante adottare metodologie agili che ci consentano di cambiare rotta in tranquillità. (Testing, SOLID principles, etc)

Context Mapping

E’ lo strumento che ci permette di governare la complessità, visualizzando la presenza e le modalità di collaborazione tra i differenti modelli all’interno del dominio.

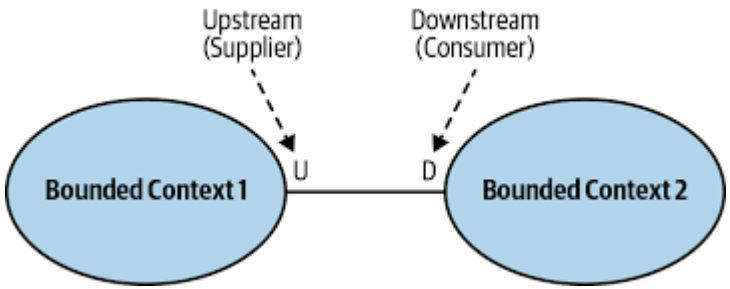
Si tratta di una combinazione di linguaggio grafico per disegnare le mappe e di alcuni principi tipici dell’investigazione per porre le domande giuste

Context Mapping: Brownfield

E’ una mappa dello scenario «as is». Un modello brownfield non deve necessariamente essere perfetto, ma sufficientemente onesto da farci capire il prima possibile cosa non torna.
Possiamo seguire delle semplici linee guide per ottenerlo

- Individuiamo i modelli all’interno e all’esterno del nostro raggio d’azione. Assegniamo loro dei nomi sulla mappa
- I confini sono precisi? Allora sono dei Bounded Context; in caso contrario delle Big Ball of Mud
- Evidenziamo le relazioni fra i modelli
- Quale modello è upstream (dominante) e quale downstream
- Quali pattern utilizziamo per descrivere le relazioni?
- Evidenziamo le condizioni di rischio sulla mappa

Upstream e Downstream



Upstream e Downstream ci permettono di visualizzare sulla mappa i rapporti di forza fra i contesti.

Rendono visibile la politica all'interno degli artefatti di progetto. Non è una questione tecnica fra chi è Client e chi è Server, ma piuttosto un'informazione chiave su chi comanda e chi subisce!

Context Mapping: Pattern

- Partnership. Due Bounded Context che evolvono in armonia. Si verifica soprattutto nelle prime fasi dello sviluppo
- Customer-Supplier. Il rapporto asimmetrico di collaborazione è più marcato, servono maggiori momenti di condivisione e pianificazione
- Conformist. Ci adattiamo ad un modello già esistente per il nostro scopo. Siamo fortemente downstream.
- Shared Kernel. Si tratta di un piccolo modello condiviso. Il rischio, in questo caso, è legato al cambiamento di questo modello, perché coinvolgerà tutto il sistema. Non esageriamo con le dimensioni dello Shared Kernel.

Context Mapping: Pattern

- Open Host. Come Team dobbiamo rendere accessibile il nostro modello tramite API, o meccanismi simili. Siamo upstream, ma dobbiamo dedicare tempo e risorse alla documentazione e alla retro compatibilità.
- Published Language. Ogni volta che due modelli interagiscono hanno bisogno di un modello comune, quello della comunicazione. Possiamo adottarne uno esistente, o rendere pubblico il nostro.
- Anti-Corruption Layer. Nel caso in cui siamo downstream, ma vogliamo proteggere il nostro modello dalle modifiche che ci vengono imposte.
- Separate Ways. Onde evitare problemi di integrazione, non ci integriamo con nessuno! Ci copiamo i dati che ci servono.

Big Ball of Mud

- Se non proteggiamo I confine del nostro modello, se non garantiamo l'integrità concettuale, se dobbiamo incrociare le dita ogni volta che rilasciamo una modifica, se abbiamo una complessità accidentale che ci obbliga ad investire mesi per fare onboarding … allora, senza ombra di dubbio, siamo in una **Big Ball of Mud**.
- Nessuno, deliberatamente, disegna una Big Ball of Mud, ma se non sfruttiamo i pattern del Context Mapping per categorizzare rigorosamente ogni contest così com'è, è molto probabile che prima o poi ci troveremo in questa situazione.

Context Mapping: Goal

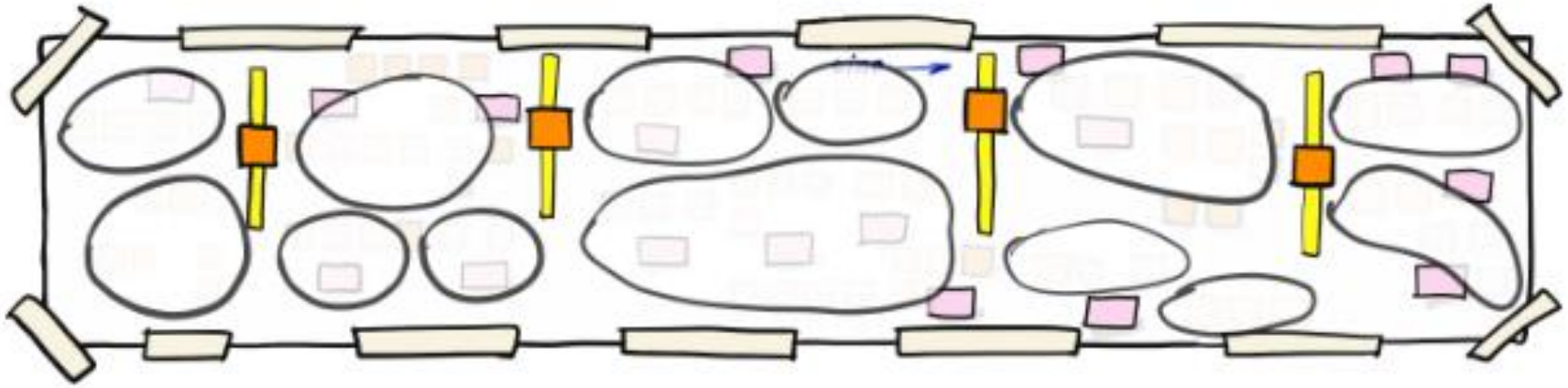
Context Mapping ci obbliga a definire i rapporti fra i Team, a governare le relazioni che ci sono, o che sono destinate a venire. Non è raro sentirsi dire «non abbiamo consegnato in tempo perché ci mancavano le API dell'altro Team» (Leggere *Team Topologies* al riguardo).

Il valore aggiunto di Context Mapping non sono le bolle che definiscono i contesti, ma le linee che tracciamo per definire le relazioni.

In definitiva Context Mapping ci permette di colmare il gap tra architettura e politiche aziendali

Context Mapping: Greenfield

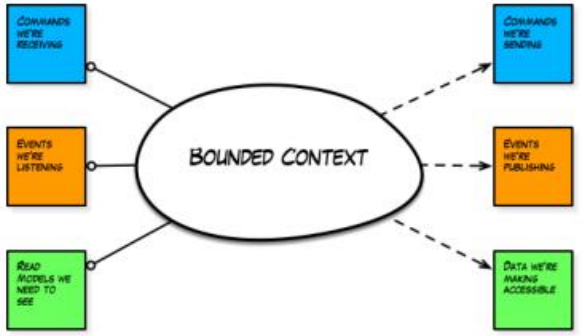
- Quando ci capita il privilegio di partire da zero possiamo utilizzare Context Mapping per tracciare i confini *naturali* del nostro dominio.
- In questo contesto le persone, l'analisi delle fasi di avanzamento di un processo che attraversa più dipartimenti, sono gli indicatori che più ci mostrano la mappa del sistema.
- Big Picture EventStorming è uno strumento molto efficace per questo tipo di indagini



Durante l'esplorazione possiamo "vedere" i confini naturali dei bounded contexts, osservando le persone, gli eventi chiave, processi paralleli, ecc.

Bounded Context Canvas

- Quando I nostri Bounded Context sono pronti per essere implementati nel codice allora possono essere visti come componenti che si scambiano informazioni con l'esterno, che semanticamente parlando, sono **Comandi, Eventi, Query**.
- In questo fase è molto utile appoggiarsi al *Bounded Context Canvas (Nick Tune)*
- Altri spunti molto interessanti sulle relazioni Bounded Context vs Team è il libro *Team Topologies di Matthew Skelton e Manuel Pais*



Un modello semplice ma efficace per vedere i nostri bounded contexts

EventStorming

Si tratta di una famiglia di workshop con un solo scopo «rappresentare la complessità dei flussi chiave delle organizzazioni utilizzando gli Eventi come strumento di esplorazione»

Gli Eventi sono rappresentati da post-it arancioni, e dato che rappresentano un fatto già successo, sono scritti al passato.

E' fondamentale coinvolgere tutti gli attori coinvolti nel progetto, business expert, utilizzatori, sviluppatori.

EventStorming

Ci sono tre formati di EventStorming

- Big Picture. Discovery di domini complessi, e delle possibili incoerenze nella comprensione del problema, o della soluzione
- Process Modelling. Permette di modellare in dettaglio i processi e le loro dinamiche rappresentando gli attori, gli step e gli interessi in gioco
- Software Design. E' il formato orientato alla progettazione del software a supporto dei processi chiave. Con questo formato si approfondisce l'esplorazione iniziata con il Process Modeling individuando i confini fra i vari contesti, le responsabilità e lo scambio di informazioni, sempre con uno sguardo al business.

Architetture DDD

Chi si illude di scrivere il codice una sola volta, trovando subito la strada maestra, probabilmente o non ha mai sviluppato in contesti complessi, oppure ha creato grandi Big Ball of Mud.

La necessità di riscrivere frequentemente il codice deriva principalmente da

- Domini troppi complessi per poter essere implementati correttamente al primo giro (nel mio caso le applicazioni finanziarie e IoT)
- Il Dominio non è un elemento immutabile, è in continua evoluzione, ed il codice si deve adeguare a questo cambiamento.
- Vogliamo poter aggiungere o togliere funzionalità senza paura. Qualsiasi architettura ci permetta di farlo è la benvenuta

Architetture: Blue Book

Vaughn Vernon ha più volte ribadito il concetto che il primo libro che parla dichiaratamente di architetture DDD è il suo *Implementing Domain-Driven Design* (aka Red Book).

In effetti, semplicemente per ragioni storiche, il Blue Book propone una soluzione architetturale a dir poco obsoleta, ma l'unica veramente possibile nel 2004!

L'architettura a Layer proposta da Eric Evans era assolutamente in linea con il modo di sviluppare software partendo dal database, assolutamente relazionale.

Event-Driven era ancora tutto da definire.

Tactical Patterns

- Si tratta di una collezione di pattern suggeriti da Eric Evans, all'interno del Blue Book, con lo scopo di aiutarci nell'implementazione robusta del nostro domain model.
- Va evidenziato che nel Blue Book non si parla di Domain Event, perché il concetto non era ancora pervenuto nel 2004, così come manca tutta la parte architetturale basata sugli eventi, proprio perché non ancora presenti.

Aggregate

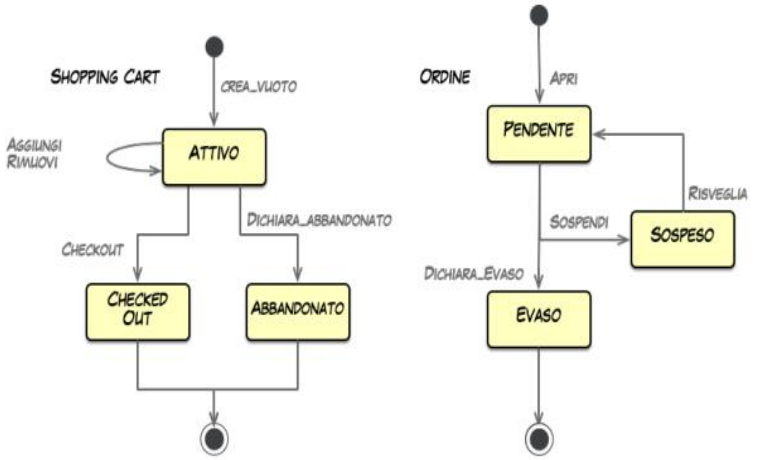
- Si tratta di un insieme di oggetti il cui comportamento è consistente nel suo insieme. L'aggregato si appoggia ad un gruppo di classi progettate per lavorare insieme, di cui alcune hanno un ruolo architetturale ben definito (*EntityRoot*, o *AggregateRoot*)
  In questo contesto assumono particolare significato i pattern della OOP (vedere *Refactoring di Martin Fowler),* dove il ruolo di coordinamento e presidio delle *invarianti (*condizioni che devono sempre essere vere *)* è lasciato alla *root,* mentre i dati sono implementati tramite *Value Object*.

Aggregate Root

- Se dobbiamo garantire lo stato consistente dell'aggregato è buona norma concedere i privilegi di poter modificare i dati ad un unico soggetto. Tutti i metodi che attivano una trasformazione dell'aggregato – *i mutators* – sono gestiti dalla Root.
- La Root è il rappresentante sindacale dell'aggregato. Nessuno può accedere direttamente agli internals dell'aggregato, e lo stesso vale per gli internals, non possono parlare direttamente con gli internals di altri aggregati, ma devono passare per la Root.
- Dentro l'aggregato le responsabilità, in pieno accordo con OOP, sono distribuite fra le varie entities.

Entity

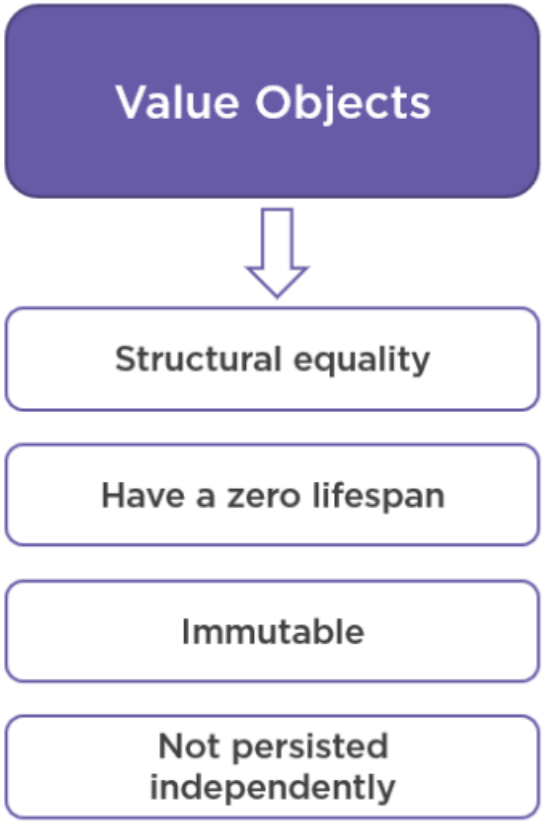
- E’ un oggetto con una identità che ci permette di tracciare le sue evoluzioni. Rappresenta il cuore di un aggregato. E’ fondamentale che la Entity sia ben definita, priva di ambiguità, e il suo comportamento ben governato da una macchina a stati
- La teoria dice che un aggregato può contenere più entity, di cui una root. La pratica suggerisce di avere una sola Entity per Aggregato (ossia avere aggregate piccoli, a responsabilità limitata)



Due esempi di macchina a stati per `ShoppingCart` *ed* `Order`*, semplici perché hanno una sola responsabilità.*

Value Object

- E’ un oggetto definito dal suo valore. A differenza della entity non ha una identità, perchè non abbiamo bisogno di monitorare il cambiamento di stato, in quanto un Value Object è *immutabile*.
- Rendere espliciti i tipi tramite l’utilizzo di Value Object ci assicura chiarezza nel codice, evita ambiguità e, conseguentemente, il nascere di errori semantici. Ridocuno dal basso la complessità della codebase
- Closure of Operations. Se il dominio è complesso, ma in generale come buona pratica di sviluppo, è nostra responsabilità individuare l’astrazione e il linguaggio che lo rendono robusto.



Factory

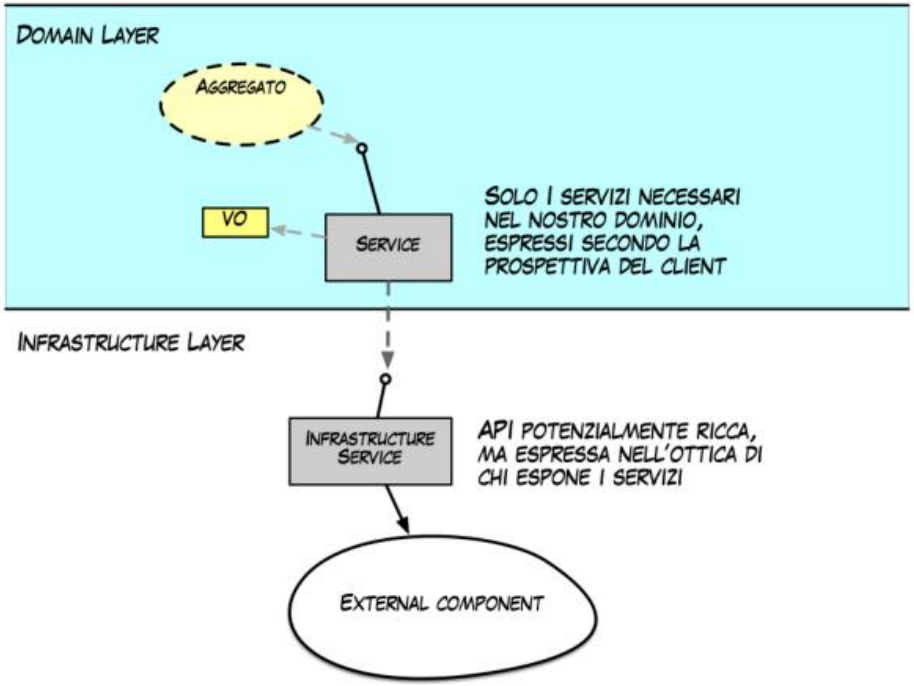
- Non si tratta di un pattern vero e proprio, ma piuttosto di una responsabilità. Dovendo garantire lo stato consistente dell'aggregato in qualsiasi momento, dobbiamo partire con il piede giusto, garantendola sin dall'inizio.
- Il Factory è responsabile del primo step della nostra macchina a stati (dal nulla ad un oggetto consistente), mentre la Root è responsabile di quelli successivi (da un oggetto consistente, ad un oggetto consistente).
- La Factory garantisce che non ci sono modi alternativi per avere un'istanza del nostro aggregato, ma solo quello proposto da noi che ne garantisce l'integrità sin dall'inizio.

Repository

- Ha lo scopo di costruire un livello di astrazione al di sopra dell'infrastruttura di persistenza (non voglio legarmi alla tecnologia del database che decidi di utilizzare).
- Permettermi di utilizzare il linguaggio del mio dominio quando devo persistere il dato, o reidratarlo dal database.
- Astrarre i problemi di persistenza ci permette di implementare dei test specifici sul Dominio, senza preoccuparci di come il dato venga persistito fra una transazione e l'altra.

Services

- Ci sono funzionalità chiave, all'interno del Dominio, che travalicano il confine del singolo aggregato. Il calcolo dei litri necessari per creare una confezione di bottiglie di birra data la capacità della bottiglia stessa , per esempio. In questo caso, visto che si tratta di un comportamento «generale» all'interno del Dominio, lo possiamo implementare tramite un servizio e renderlo disponibile a più aggregati.



Il service di dominio cattura i bisogni del dominio, e dietro le quinte li mappa con quanto offerto da una libreria o un componente esterno

Bounded Context: Implementazione

Affinchè sia garantito l'assunto che ci lascia la massima libertà di azione all'interno di un Bounded Context, essendo ottimizzato per un singolo scopo, devono essere rispettati alcuni requisiti principali

- Ciascun Bounded Context è libero di adottare la propria architettura, ed ovviamente, il proprio linguaggio
- La persistenza deve essere privata. Qualsiasi forma di condivisione è delegata alla frontiera del nostro Bounded Context
- Incentivare la comunicazione tra diversi Bounded Context tramite l'utilizzo degli eventi
- L'architettura esagonale (Alistair Cockburn), proposta poi con diversi altri acronimi, permette il disaccoppiamento della logica di dominio dall'implementazione dell'infrastruttura

Bounded Context & Microservizi

Common Characteristics
Componentisation via services
Organised around business capabilities
Decentralised data management
Products not projects
Decentralised governance
Smart endpoints and dumb pipes
Evolutionary design
Infrastructure automation
Designed for failure

We cannot say there is a formal definition of the microservices architectural style, but we can attempt to describe what we see as **common characteristics** for architectures that fit the label.

(Martin Fowler, James Lewis)

Bounded Context & Microservizi

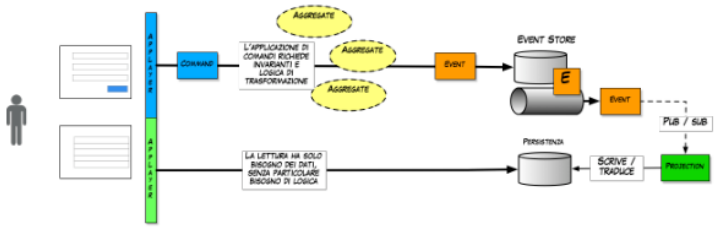
Feature	Bouded Context	Microservice	Compatibility
Organized around Business Capabilities	È implicitamente inteso nel concetto stesso di Ubiquitous Language, che è il pattern principale per identificare un Bounded Context	Teams Cross-Funzionali specifici per una funzionalità di business	Perfetta!
Decentralized Governance	Un modello condiviso per ogni scopo	Vengono favorite/incentivate le scelte locali, che devono essere indipendenti.	Perfetta!
Decentralized Data Management	La persistenza privata è fondamentale per la consistenza del linguaggio, ma soprattutto necessaria per l'evoluzione sicura e indipendente del modello	Ogni microservizio deve persistere i propri dati in un database privato! Pena l'impossibilità di evolvere autonomamente dagli altri	Perfetta!
Evolutionary Design	Ogni modello può, e deve, evolvere indipendente dagli altri	E' una key feature	Perfetta!
Smart endpoints and dumb pipes	Raccomandato come modello strategico	Key feature. SOA docet!	Fattibile
Language Consistency	Ubiquitous Language! Obbligatorio!	Implicito e raccomandato	Nessun problema
Componentization via Services	Context Map	Key feature	Nessun problema
Products not Projects	Raccomandato per la conoscenza approfondita del modello	Key feature	Nessun problema
Design for Failure	DDD incentiva l'evoluzione continua	Key feature	Nessun problema

CQRS

Command-Query Responsability Segregation è un pattern divenuto cool grazie a Greg Young e Udi Dahan.

Nato sulle ceneri di CQS di Bertrand Meyer enfatizza la separazione netta fra il modello utilizzato per scrivere informazione, e quindi apportare modifiche all'aggregato, da quello utilizzato per leggerle.

A differenza di CQS, in CQRS ci sono proprio due modelli di database distinti, appositamente ottimizzati ognuno per il proprio scopo



CQRS/ES.

CQRS

La forte segregazione promossa da questo pattern comporta una radicale revisione nel modo di sviluppare applicazioni.

- Il write model è uno store di eventi append-only.
- Il read model è la proiezione del Dominio, e come tale ne «subisce» le scelte

Quando si parla di CQRS i *Domain Event* sono la scelta più ovvia in termine di propagazione delle informazioni.

Testing

Posto che i comandi e gli eventi sono oggetti immutabili, e che il Dominio è una scatola al quale inviamo comandi e ci aspettiamo eventi, come testiamo il nostro software?

- I test dovrebbero essere capaci di rappresentare con semplicità qualunque comportamento complesso
- Devono essere comprensibili da tutti
- Rappresentano il pensiero dietro il design dell'aggregato
- "Given, When, Then" con gli eventi, i comandi ed i Command Handler

Testing

L'idea essenziale è quella di suddividere uno scenario in tre sezioni:

Given: descrive lo stato dell'aggregato prima di inviare il comando. Una sorta di prerequisito del test.

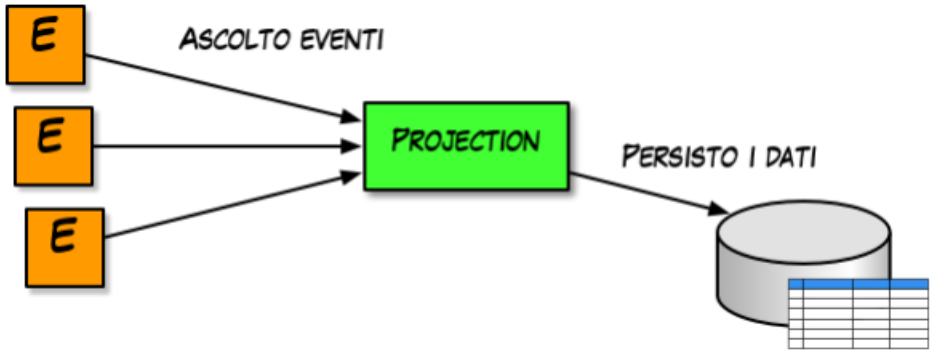
When: rappresenta il comando da inviare all'aggregato.

Then/Expect: descrive i cambiamenti di stato che ci si aspetta in seguito al comando.

Projections

Le Projections sono gli strumenti che ci permettono di trasformare gli eventi in dati consumabili dalle nostre applicazioni.

Il compito, piuttosto semplice, di una projection è aggregare le informazioni che arrivano dagli eventi in strutture dati facilmente, e soprattutto velocemente, consumabili. Per questo motivo i database documentali trovano ampio spazio nella realizzazione dei ReadModel



Il ruolo di una Projection in un'architettura a eventi.

Event Sourcing

E’ ormai consuetudine, quando si utilizza il pattern CQRS, salvare gli eventi generati dal Dominio. EventSourcing è un pattern architetturale che permette la ricostruzione dello stato di un’applicazione partendo proprio dagli eventi salvati. Dal punto di vista architetturale è una delle massime espressioni di applicazione di modello funzionale, infatti, possiamo esprimere lo stato di un aggregato come funzione della sua storia di eventi

$$State = f(events)$$

Passare da un modello CRUD ad un modello ad eventi è un po’ come passare dalla fotografia di un oggetto in un preciso istante, a tutto il film che ne racconta l’intera storia.

L’esempio classico è l’estratto conto della banca, ottenuto leggendo tutti i movimenti sul nostro conto corrente.

Event-Driven Architecture

Questo tipologia di architettura sfrutta proprio la propagazione degli eventi come meccanismo di comunicazione asincrona permettendo, di fatto, un totale disaccoppiamento fra le parti in causa.

Affinché gli eventi propagati portino con sé il maggior quantitativo di informazioni possibili si assiste ad una netta distinzione fra gli eventi tipicamente di Dominio, e legati ad uno specifico Bounded Context, e gli eventi che vengono utilizzati per propagare informazioni al resto del mondo

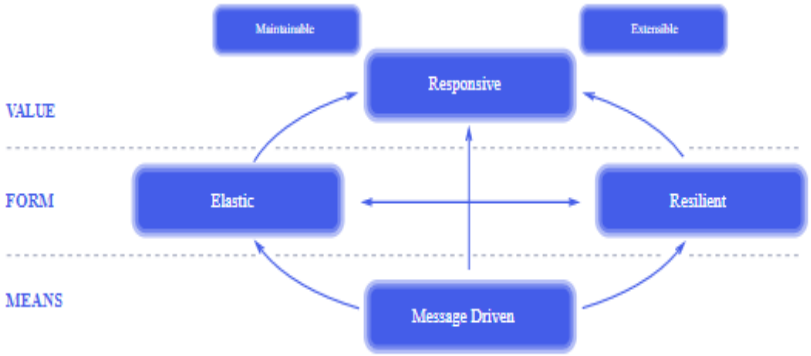
- Domain Event. Eventi locali ad uno specifico Bounded Context
- Integration Event. Eventi pubblici, condivisi con il resto del sistema, in un linguaggio più agnostico rispetto all'Ubiquitous Language del Bounded Context

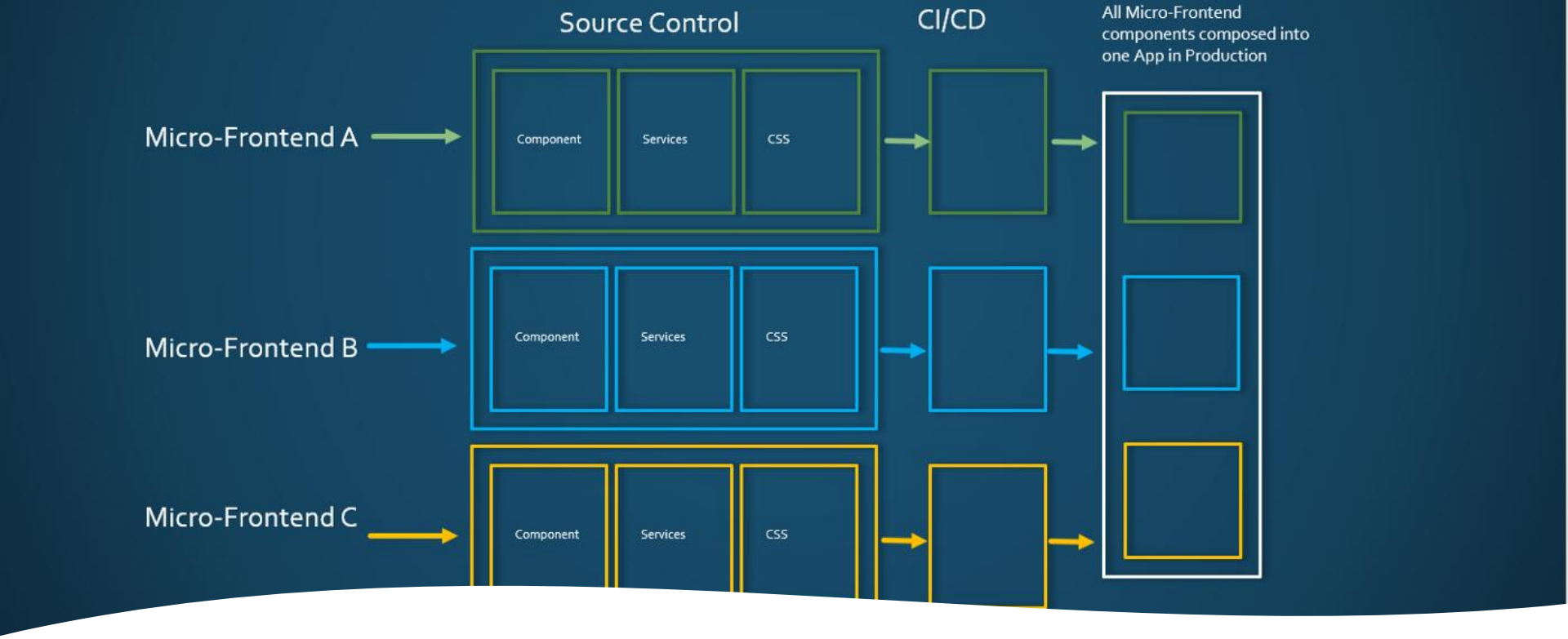
The Reactive Manifesto

Il sistema, se è in generale possibile dare una risposta ai client, la dà in maniera tempestiva. La responsività è la pietra miliare dell'usabilità e dell'utilità del sistema; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito di modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni con il sistema.

The Reactive Manifesto

- Resilient. Il sistema resta responsivo anche in caso di guasti
- Elastic. Il sistema resta responsivo sotto carichi di lavoro variabili nel tempo
- Message Driven. Il sistema è basato sullo scambio asincrono di messaggi per garantire il basso accoppiamento fra i componenti.



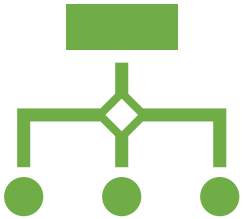


DDD nel Frontend

Dopo aver suddiviso la Big Ball of Mud lato backend perché non fare la stessa cosa lato frontend?

E' possibile applicare gli stessi pattern applicati al backend anche al frontend?

An Old History – Self-Contained System



ThoughtWorks Technology Radar 2016

- Autonomous Web Application
- Each SCS is owned by One Team
- Asynchronous Communication
- Each SCS has its own API
- Each SCS include Data and Logic
- No Shared UI
- No Shared Business Code
- Shared Infrastructure can be minimized

Decision Framework

Prima di partire con la suddivisione del frontend bisogna chiarire alcuni punti

- Definire cos'è un microfrontend nella nostra architettura
- Composizione di microfrontend
- Routing in una soluzione a microfrontend
- Comunicazione fra i microfrontend

Microfrontend Ways

Possiamo implementare una soluzione a microfrontend in quattro modi diversi

- Shared Session and parameters
- Routing
- Components
- Shared Components / Class Library