

ARCHITECTURAL PATTERN

I processi di sviluppo software

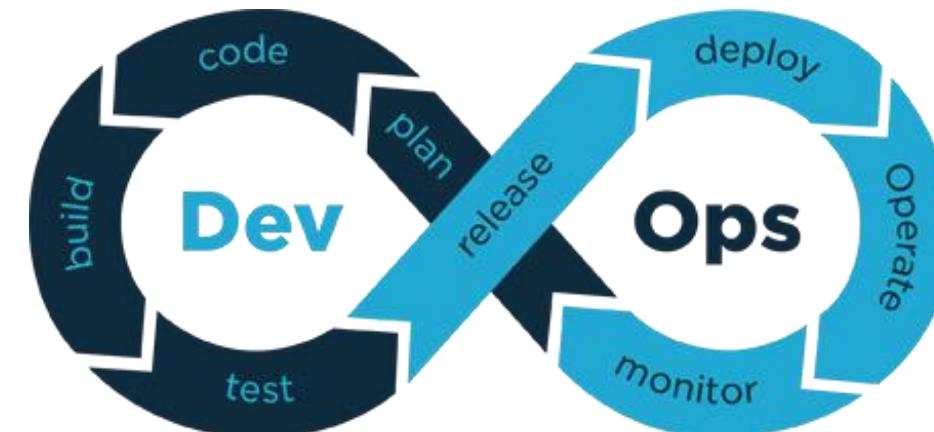
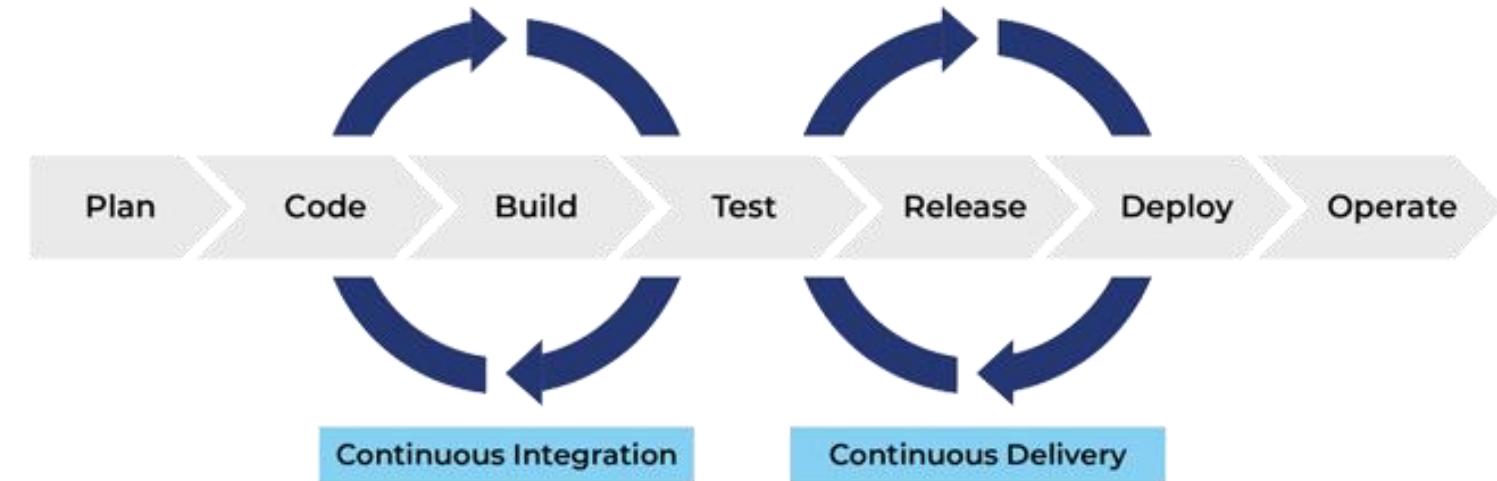
Il «**Mercato IT**» è da sempre caratterizzato dalla continua ricerca di tecniche e paradigmi in grado di **ottimizzare** l'intero processo di gestione dei progetti software (in tutte le fasi del suo ciclo di vita).

L'incredibile pervasività delle **tecniche** IT ha contribuito a rendere tale esigenza ancora più stringente e importante.

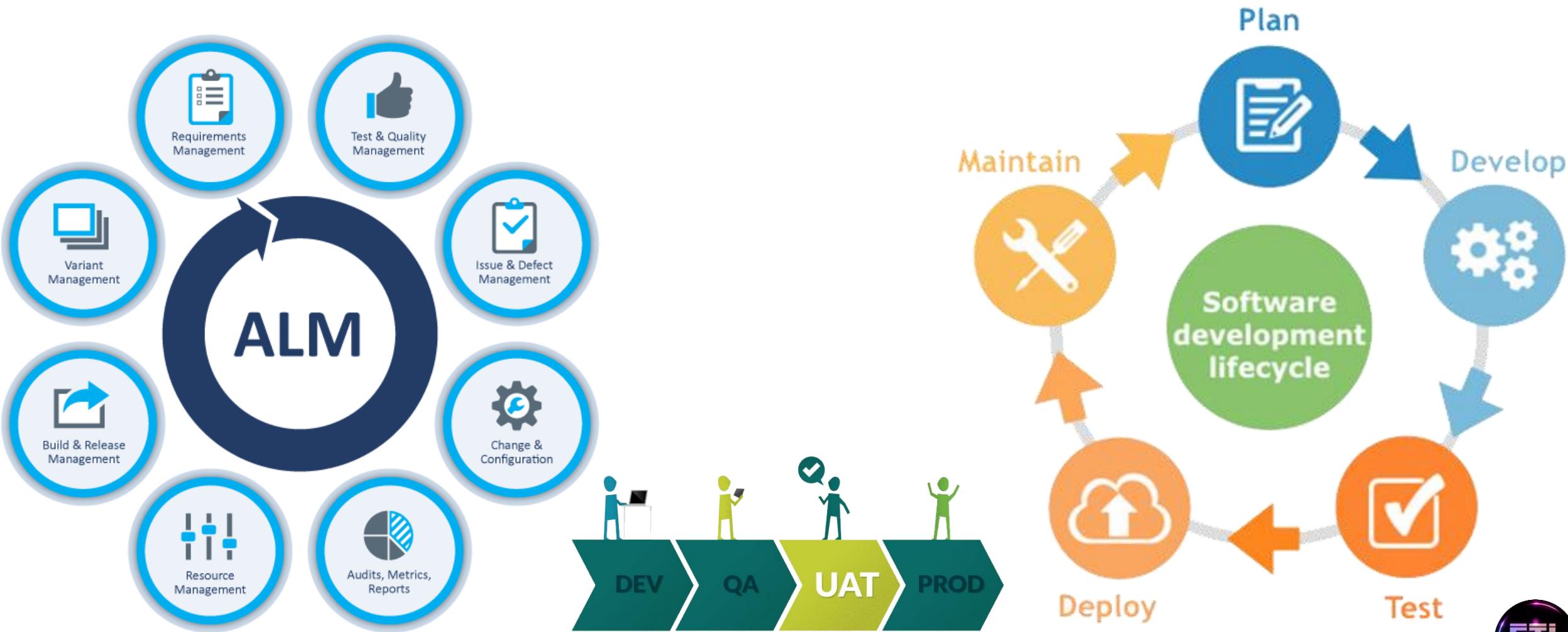


In questo scenario sono nati **paradigmi** come:

- **CI: Continuous Integration**
- **CD: Continuous Delivery**
- **Agile**
- **Modello DevOps**
- Etc.



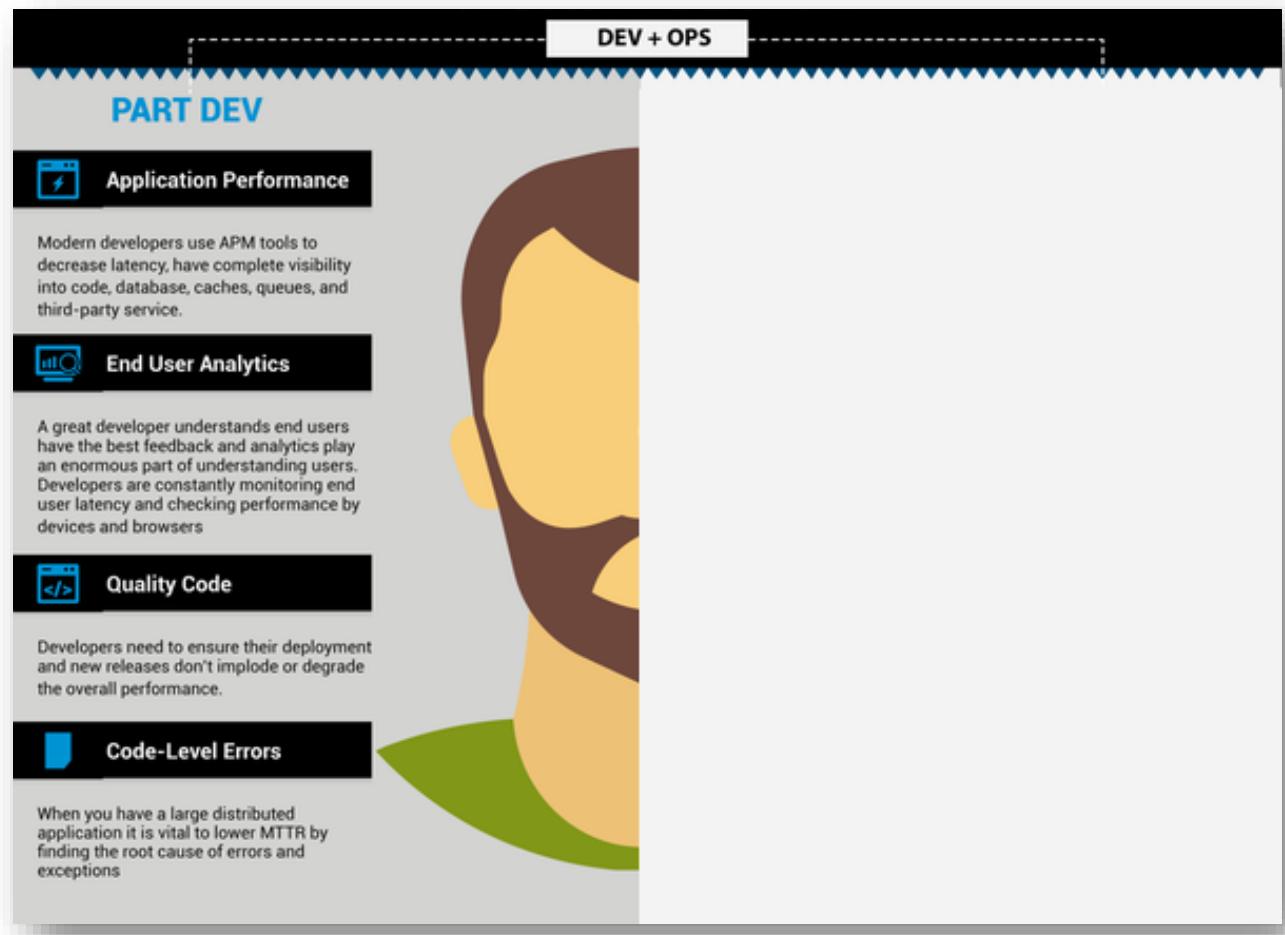
Bisogna facilitare la gestione del **lifecycle** di un software, ovvero facilitare alcune fasi dell'**ALM** (**Application Lifecycle Management**) accelerando, ottimizzando, semplificando aspetti sia tecnici sia gestionali.



Docker è un tool progettato per dare vantaggi sia agli sviluppatori sia agli amministratori di sistema, come parte di numerose DevOps (developers + operations) toolchain.



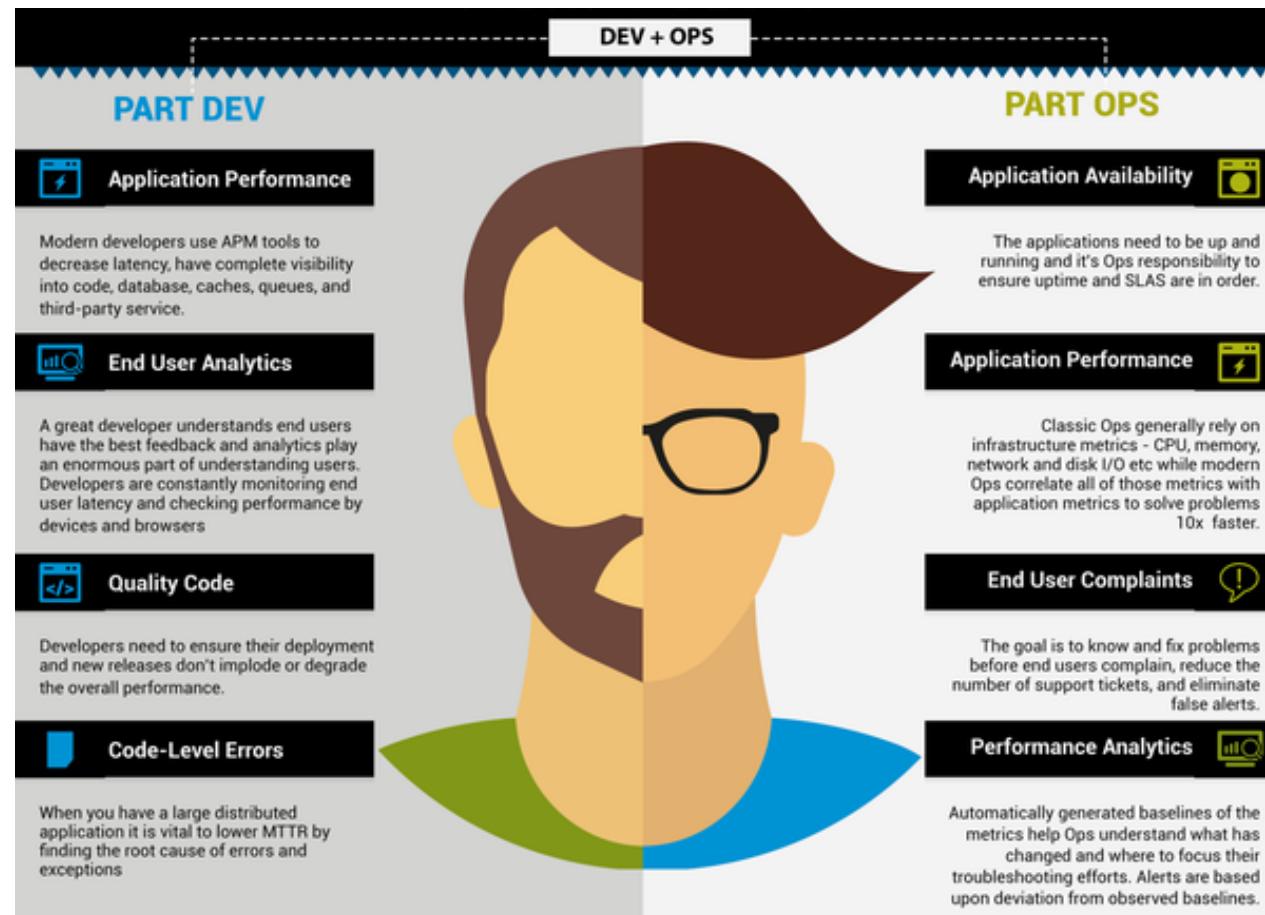
Docker permette agli sviluppatori di implementare codice «senza preoccuparsi del sistema su cui l'applicazione girerà»; Ciò crea indipendenza e disaccoppiamento tra gli ambienti di produzione e le applicazioni «rilasciate**» su questi ultimi.**



Il campo **Operations** ottiene da Docker maggiore flessibilità; inoltre Docker riduce potenzialmente il numero di sistemi necessari, grazie alla sua impronta minimale e meno invasiva.



Tutto ciò ha portato nel tempo alla nascita di nuove figure professionali ovvero i «**Devops**» per l'appunto.





Le evoluzioni dei
sistemi software

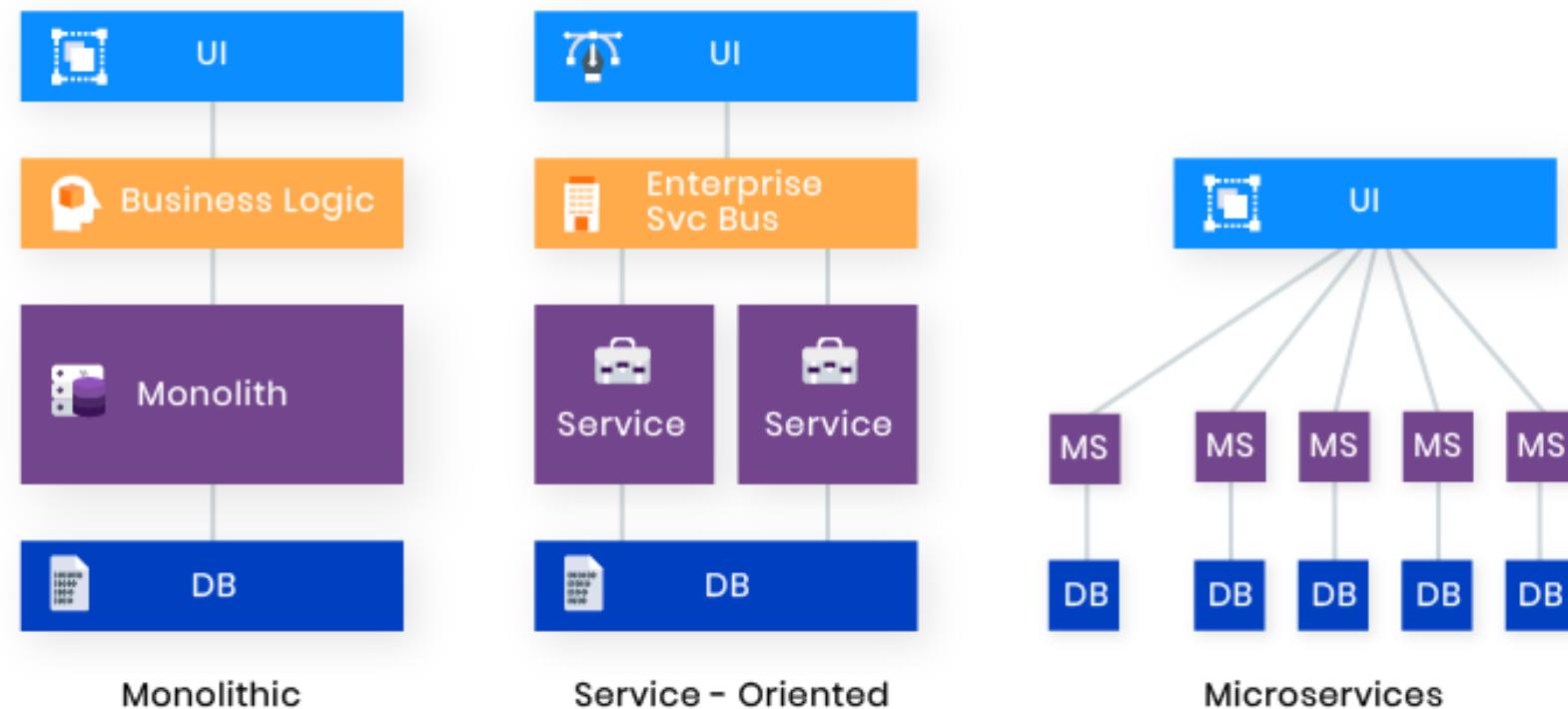
Alla fine degli anni 90 i siti erano per lo più **statici**, con qualche **CGI** all'occorrenza, «**Applet Java**» o altre soluzioni oggi considerate poco eleganti.

L'avvento di **ASP** e **PHP** cambiò successivamente lo scenario dello sviluppo di «**Applicativi Web**» che sempre di più avevano esigenza di assumere caratteri **dinamici** sia dal punto di vista dei contenuti sia dal punto di vista delle **funzionalità esposte**.



Venti anni dopo gli scenari sono drasticamente diversi, le «**Architetture SOA**» hanno preso piede **evolvendosi** poi nelle «**Architetture a Microservice**» e tutto è diventato più complesso.

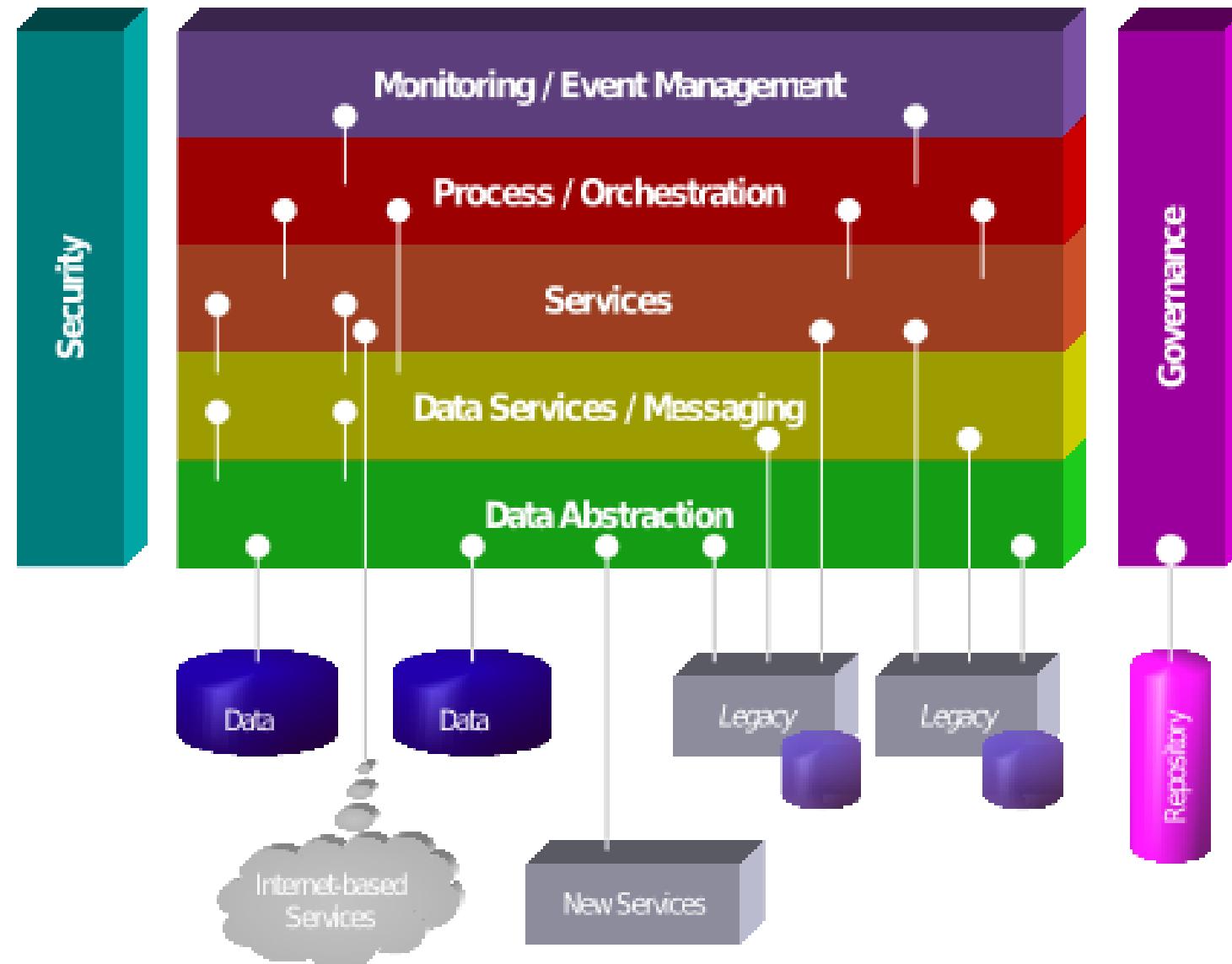
Un'applicazione web, soprattutto in ambito **enterprise** oramai è un **insieme di componenti autonomi**, ognuno col suo ciclo di vita **indipendente**, rilasciati su macchine diverse e a **ritmi sempre più crescenti**.



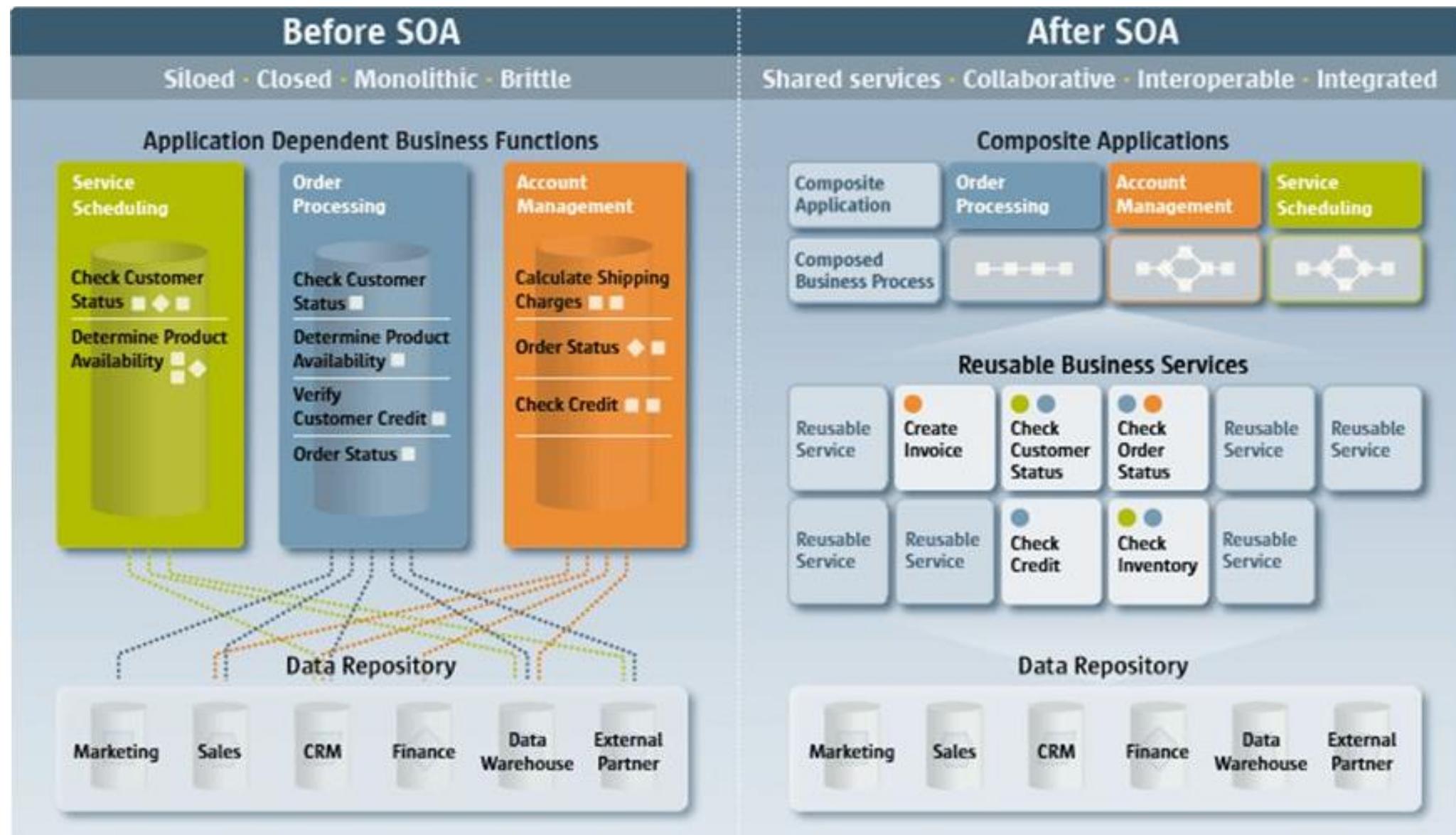
Timeline Trasformazione Delle Applicazioni



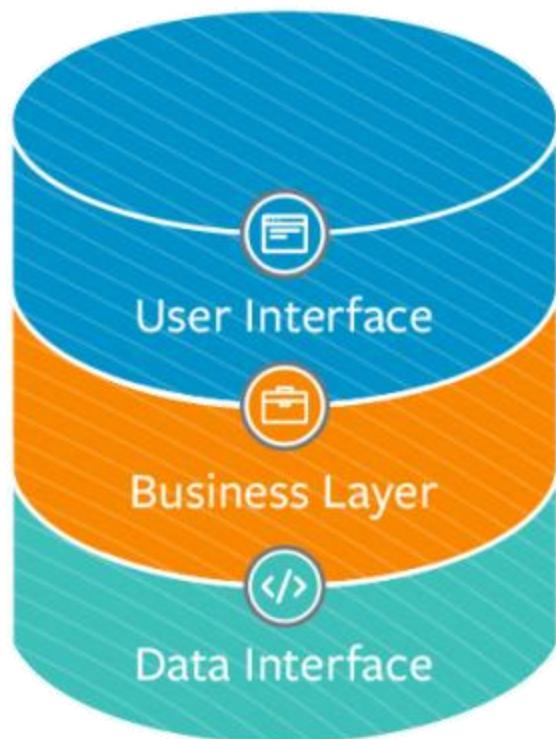
Complessità delle SOA



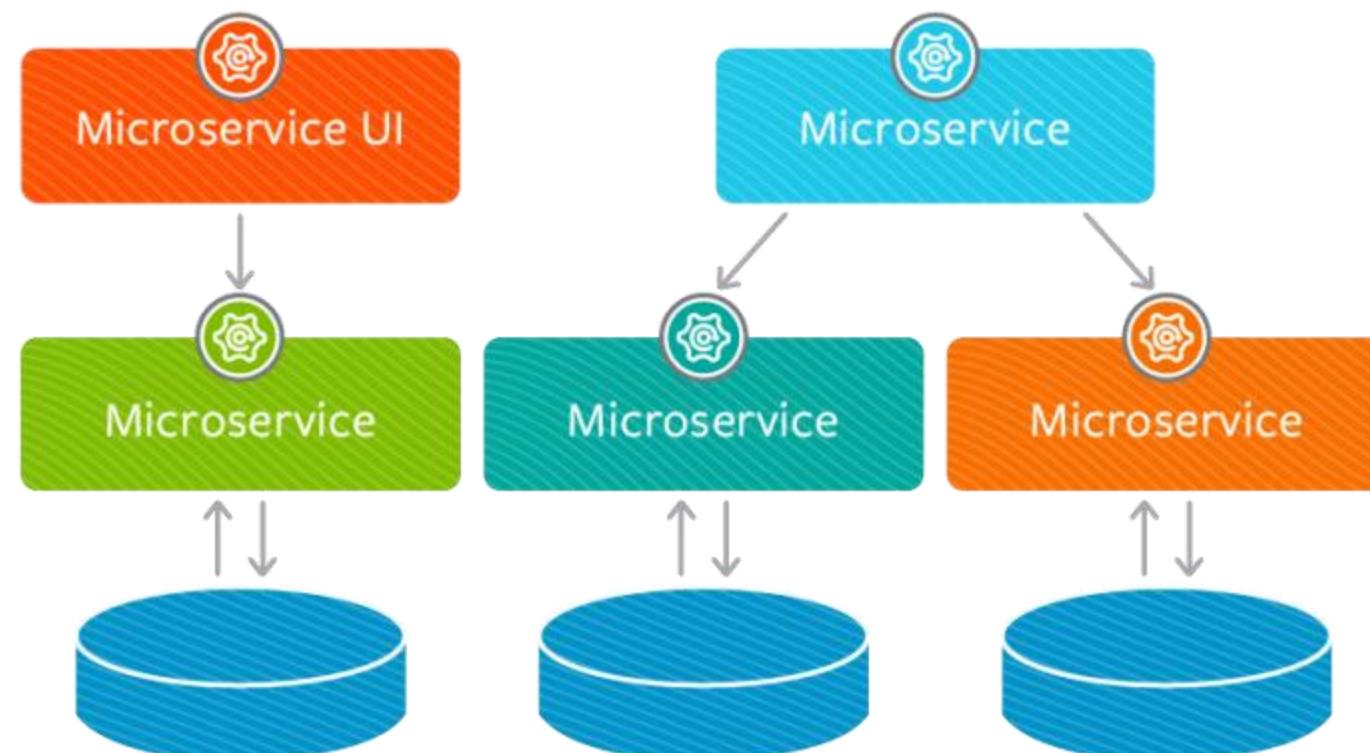
Monolithic vs SOA



Monolithic Architecture

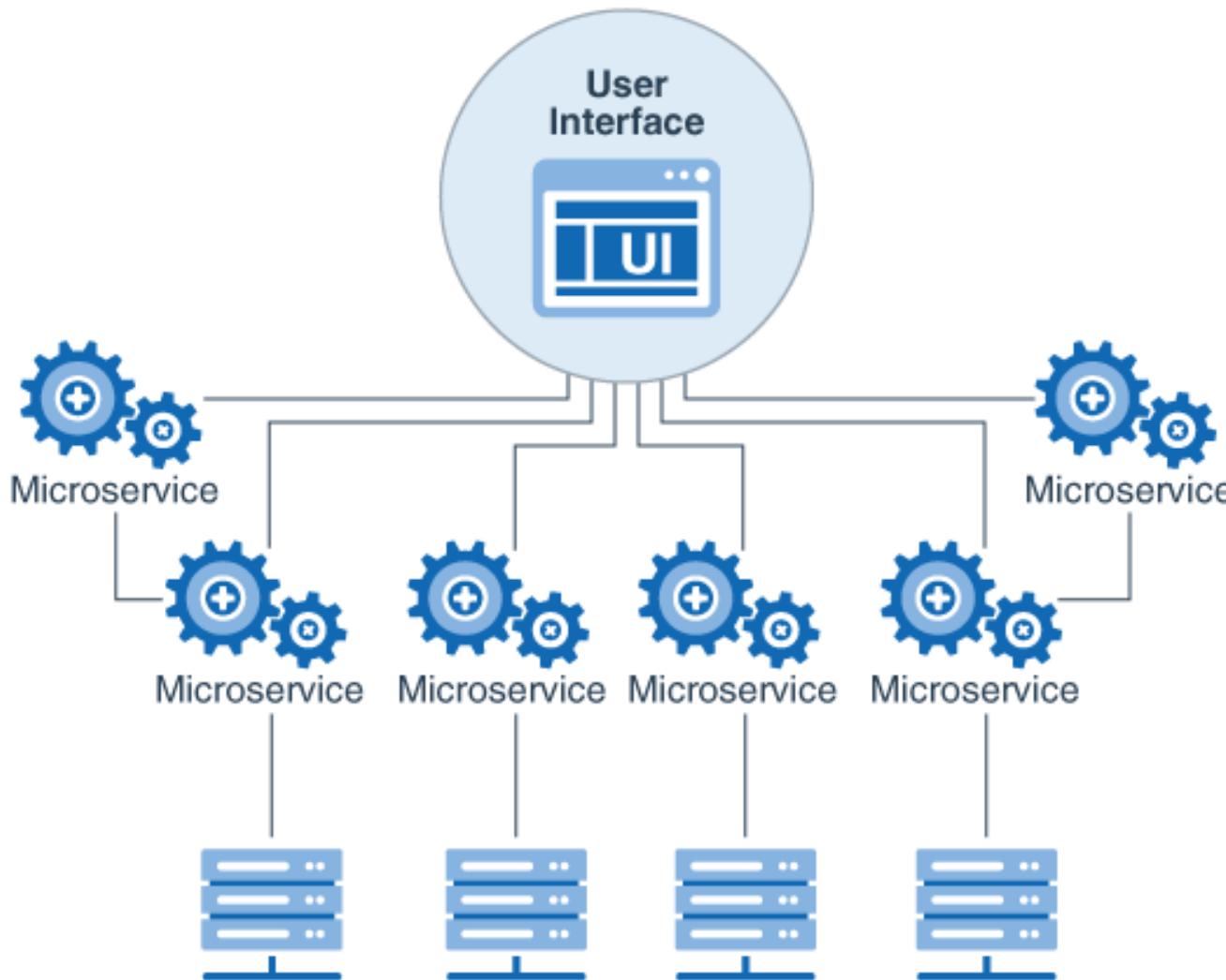


Microservices Architecture

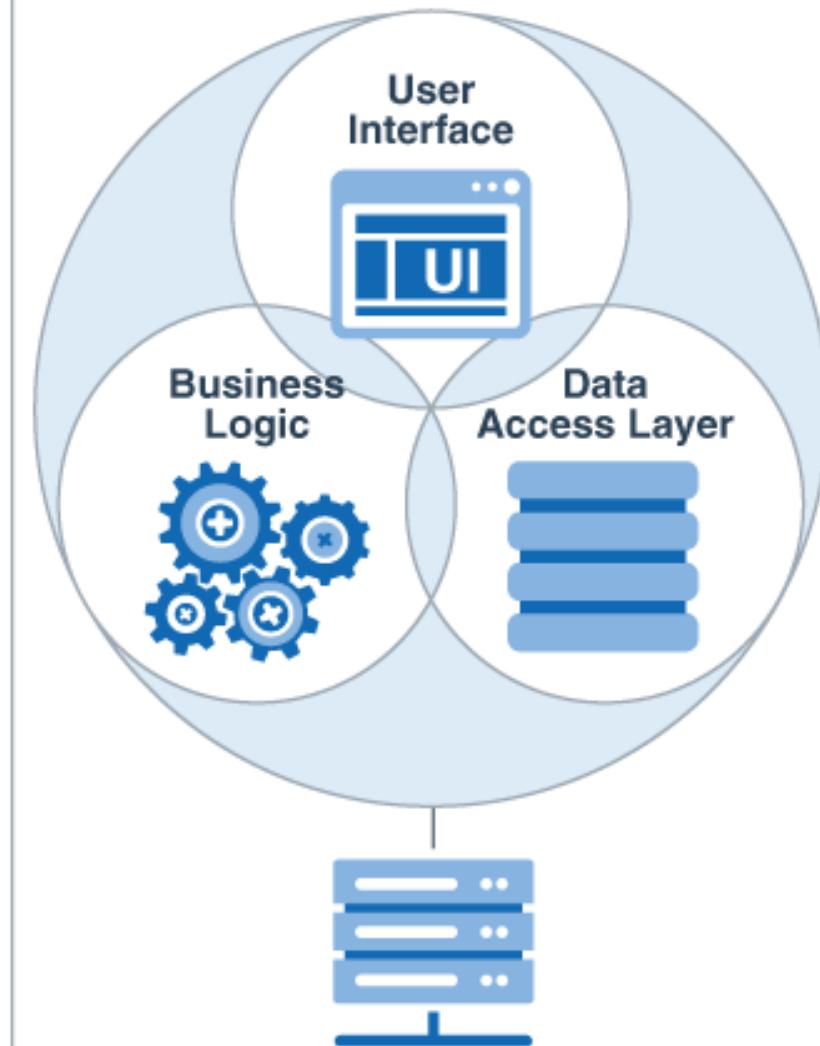


Monolithic vs Microservices Architecture

Microservice Architecture



Monolithic Architecture

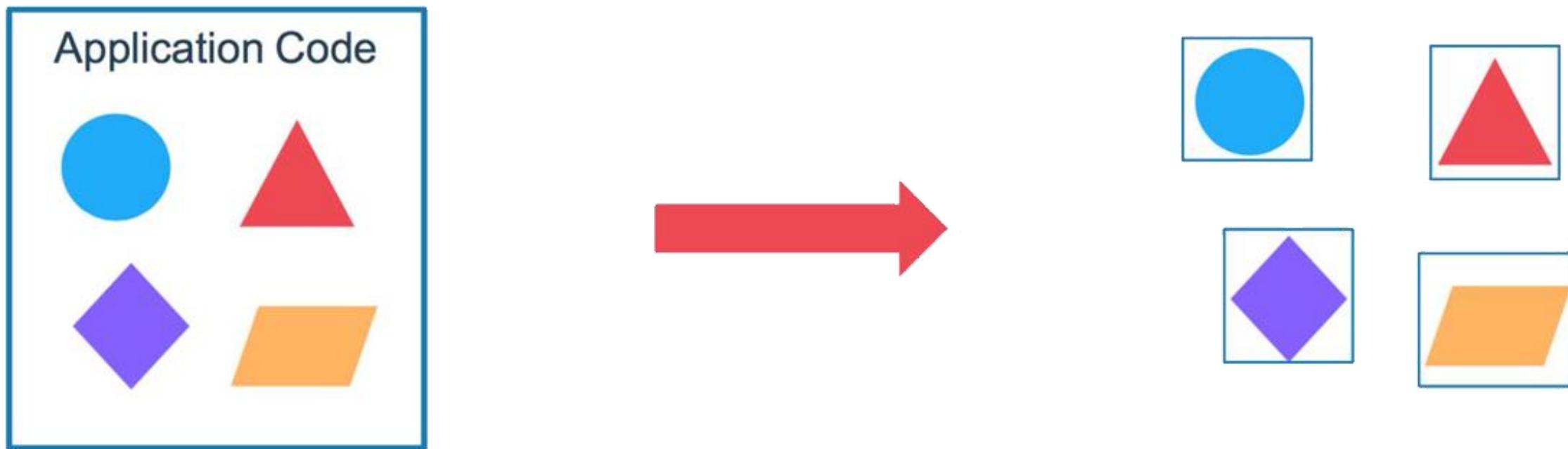


Le «applicazione monolitiche» sono problematiche in quanto difficili da implementare, distribuire e modificare; inoltre, per farle girare, servono server e data-center molto potenti.

Con le «nuove architetture software», le applicazioni permettono di far girare i servizi liberamente, con facilità e con rapidità di gestione delle modifiche e degli aggiornamenti; inoltre sono eseguibili su piccoli server o dispositivi decentrati e portatili.

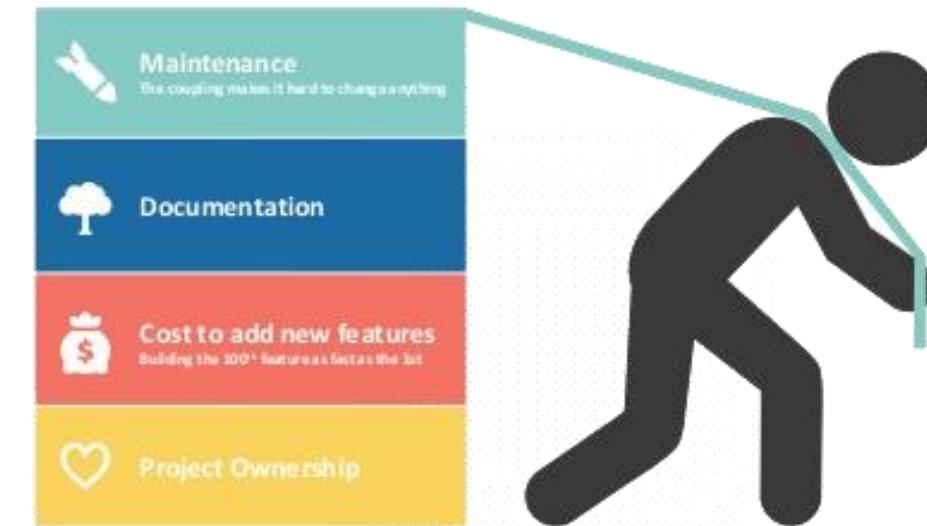
Tutto questo ha comportato di conseguenza una modernizzazione delle applicazioni.

Come schematizzato dalla figura seguente, prima avevamo applicazioni monolitiche e concentrate con molti servizi in un unico contenitore, mentre oggi siamo ad applicazioni composte dall'esecuzione di micro-servizi poco legati tra loro (Forte Disaccoppiamento).



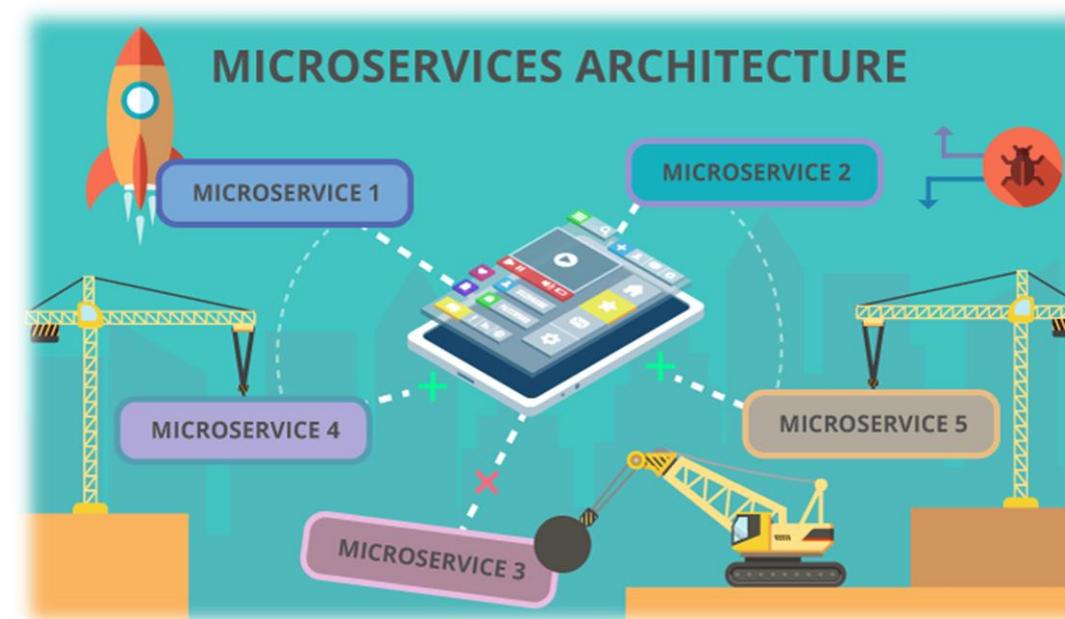
Nelle applicazioni monolitiche, si pongono i seguenti **problemi di sviluppo**:

- Piccole modifiche al codice richiedono il «**re-compile**» e la «**ri-esecuzione**» del test, del debug e del deployment.
- L'applicazione diventa «**single point of failure**» ovvero può andare in «**crash**» per il blocco di una delle sue parti.
- L'applicazione è difficile da **scalare**.



Nei nuovi pattern di sviluppo:

- E' possibile ridimensionare le applicazioni in operazioni separate e isolate tra loro, quindi il «failure di una non influisce sulle altre»
- Si può rendere l'app scalabile in modo indipendente e altamente disponibile in base alla progettazione.



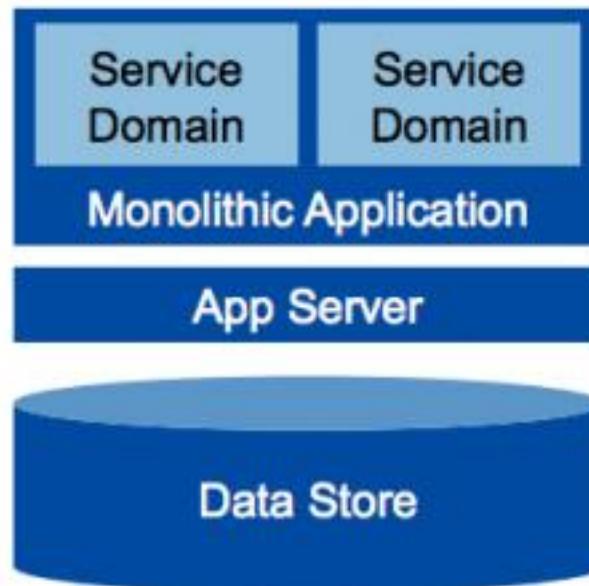
A causa della complessità raggiunta nell'ambito dello sviluppo di applicazioni «**web-oriented**» e di «**Servizi**» (architetture SOA, microservizi, DevOps, etc.) sono sorte nuove necessità a cui, il mondo IT ha cercato di rispondere con la creazione di strumenti che consentano di:

- Aumentare l'**affidabilità**
- Consolidare la **distribuzione delle applicazioni** (limitando errori umani, checklist lunghissime da controllare e tutti i passaggi «snervanti» e carichi di rischi necessari per il deployment.)
- Abbattimento dei **rischi**
- Abbattimento dei **tempi** e dei **costi**

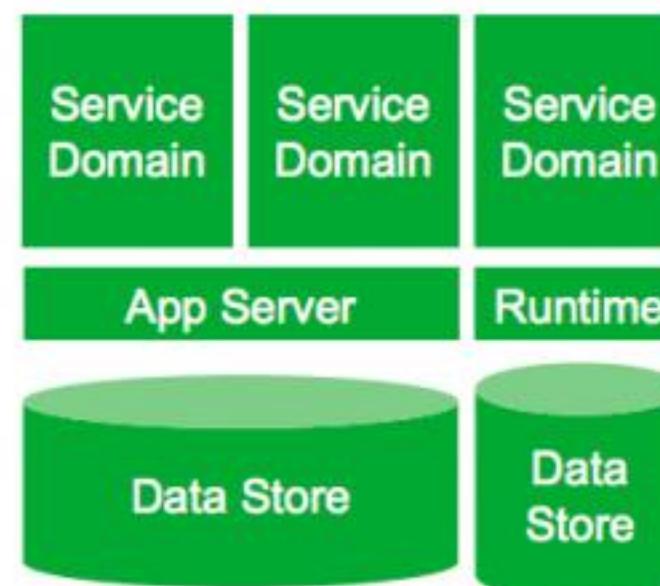


Looser Coupling, Greater Agility

(Macro) Services



Miniservices



Microservices



Lower Complexity, Easier to Run

Layering

La **stratificazione** è una delle tecniche più comuni utilizzate dai progettisti di software per scomporre un sistema software complicato.

Quando si pensa a un sistema in termini di **layers**, si immaginano i principali sottosistemi del software disposti in una qualche forma di «**torta a strati**», in cui ogni strato poggia su uno strato inferiore.

In questo schema il livello superiore utilizza vari servizi definiti dal livello inferiore, ma il livello inferiore non è a conoscenza del livello superiore.

Solitamente queste architetture sono definite «**opache**» perché ogni strato di solito nasconde i suoi strati inferiori dagli strati sovrastanti; quindi lo strato 4 usa i servizi dello strato 3, che usa i servizi dello strato 2, ma lo strato 4 non è a conoscenza dello strato 2.

La scomposizione di un sistema in livelli ha una serie di **importanti vantaggi**:

- Ogni livello può essere sostituito senza che ciò comporti modifiche agli strati sottostanti.
- Possiamo «**comprendere**» il funzionamento di uno strato senza avere necessità di conoscere gli altri.
- Si riducono al minimo le dipendenze tra i layers.
- I layers sono una buona strada per la **standardizzazione**.
- Una volta creato un layer, è possibile **riutilizzarlo** nei layers (o servizi) di livello superiore.

La stratificazione è una tecnica importante, ma vi sono degli svantaggi.

- I livelli **incapsulano** bene alcune cose ma non tutte; uno scenario di esempio sono le «**modifiche a cascata**» (classico è il caso di un'applicazione aziendale a più livelli dove l'aggiunta di un campo che debba essere visualizzato sulla UI, si ripercuote come modifica sul database e su ogni layer intermedio).
- Strati **aggiuntivi possono pregiudicare le prestazioni**; ciò può dipendere, ad esempio, dal fatto che ad ogni livello le informazioni che passano debbano essere trasformate/elaborate.

La parte più difficile nel design di un'architettura stratificata è decidere quali livelli debba avere e quale dovrebbe essere la responsabilità di ognuno di essi.

Quando si sviluppava su sistemi batch i developers non pensavano molto ai livelli.

La nozione di livelli è diventata più evidente negli anni '90 con l'ascesa dei sistemi client-server ovvero di sistemi a due livelli: il client conteneva l'interfaccia utente e altro codice dell'applicazione e il server era solitamente un database relazionale.

Gli strumenti di sviluppo (ad esempio: VB, Powerbuilder e Delphi) rendevano particolarmente facile lo sviluppo (era possibile creare una schermata trascinando i controlli su un'area di progettazione e configurandoli tramite schede delle proprietà per collegarli al database).

Questi sistemi client-server funzionavano molto bene se l'applicazione riguardava solo la visualizzazione e il semplice aggiornamento di dati relazionali.

I primi problemi sono arrivati con le **logiche del dominio**:

- **Regole aziendali**
- **Convalide**
- **Calcoli**
- **Etc.**

Solitamente i developer le implementavano nel codice del client e direttamente nelle schermate dell'interfaccia utente (cosa che generava ridondanza e duplicazione del codice).

Poiché la logica del dominio nel tempo è diventata sempre più complessa, è diventato molto difficile lavorare con questo approccio.

Un'alternativa era inserire la logica del dominio nel database nelle «**stored procedure**».

Tuttavia, tale approccio forniva meccanismi di strutturazione del codice limitati.

Come se non bastasse tutto ciò rendeva difficile eventuali sostituzioni del motore database.

Successivamente ha preso piede la **programmazione orientata agli oggetti (OOP)** la quale forniva una risposta valida al problema della logica di dominio.

A questo punto si è passati a un sistema a tre livelli.

- un livello di **presentazione** per l'interfaccia utente
- un livello per la **logica del dominio**
- un'origine **dati**.

Ciò ha portato alla organizzazione in layer degli strati software nonchè alla specializzazione dei layer stessi.

Il vero «shock sismico» si è avuto però con l'ascesa del Web.

All'improvviso le persone volevano distribuire le proprie applicazioni client-server spostando:

- le **interfacce su di un browser Web**
- le **logiche di dominio e l'origine dati sui server**

Un sistema a tre livelli ben progettato ☺ !!!

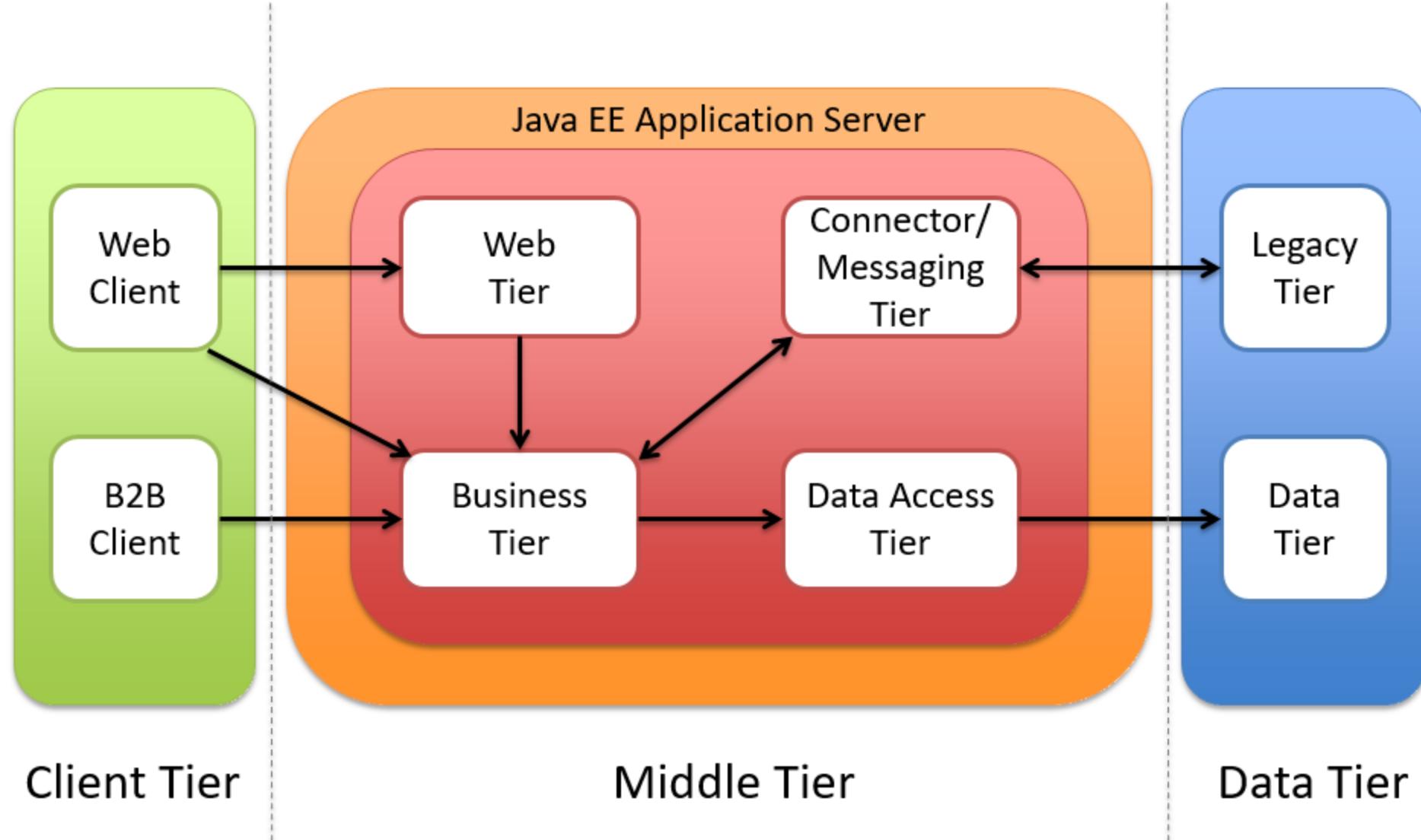
Finalmente si incomincia a parlare di architetture «multi-tier».

E' un modello architettonico «**astratto**» per applicazioni enterprise indipendente dalle scelte tecnologiche (linguaggio, piattaforma, etc.)

Le funzionalità dell'applicazione sono suddivise in «**3 livelli isolati**» (detti Tiers):

- **Client Tier:** esegue richieste al Middle-tier
- **Middle Tier:** gestisce le richieste provenienti dai clients e processa i dati dell'applicazione
- **Data Tier:** mantiene i dati in strutture di memorizzazione permanenti





Quando le persone discutono della stratificazione, c'è spesso una certa confusione sui termini «layer» e «tier».

Spesso i due sono usati come sinonimi, ma tanti vedono il «tier» come se implicasse una separazione fisica (esecuzione su server diversi).

I sistemi client-server sono spesso descritti come sistemi «two-tier» ove la separazione è fisica (es. il client è un pc desktop e il server è un pc server).

I «layer» sono invece visti come strati software indipendenti dal «dove» risiedano.

Ma questa è pura e mera «speculazione filosofica» 😊.

Responsabilità dei livelli:

- **Presentation Logic:** View, Servizi et.c
- **Domain Logic:** Logica (cuore del sistema)
- **Data Source Logic:** Comunicazione con database, sistemi di messaggistica, gestori di transazioni, altro

La logica di presentazione riguarda come gestire l'interazione tra l'utente e il software.

Può essere semplice come una riga di comando o un sistema di menu basato su testo, o più probabile un'interfaccia utente grafica o basata su HTML

Le responsabilità principali del livello di presentazione sono:

- **visualizzare le informazioni all'utente**
- **interpretare i comandi dell'utente in azioni sul dominio e sull'origine dati.**

La logica dell'origine dati riguarda la comunicazione con altri sistemi che eseguono attività per conto dell'applicazione.

Questi possono essere monitor delle transazioni, altre applicazioni, sistemi di messaggistica e così via.

Per la maggior parte delle applicazioni aziendali, la più grande parte della logica dell'origine dati è un database che è principalmente responsabile dell'archiviazione dei dati persistenti.

Il «pezzo» rimanente è la logica di dominio, nota anche come «logica di business».

Qui ha sede il lavoro che un' applicazione deve svolgere per il dominio con cui stiamo lavorando; comprende

- **Calcoli basati su input**
- **Gestione dei «dati memorizzati»**
- **Convalida di tutti i dati che provengono dalla presentazione**
- **Capire esattamente quale logica di origine dati inviare, a seconda dei comandi ricevuti dalla presentazione**
- **Etc.**

A volte i livelli sono disposti in modo che il livello del dominio nasconde completamente l'origine dati dalla presentazione.

Altre volte la presentazione accede direttamente all'archivio dati.

Sebbene questo sia un approccio meno purista, tende a funzionare meglio nella pratica.

La presentazione può interpretare un comando dell'utente, utilizzare l'origine dati per estrarre i dati rilevanti dal database e quindi lasciare che la logica del dominio manipoli tali dati prima di presentarli a video.

Una singola applicazione può spesso avere più «packages» per ciascun layer.

Un'applicazione può presentare ad esempio più layer di presentazione:

- **Un layer di presentazione per il web**
- **Un layer di presentazione per il mobile**

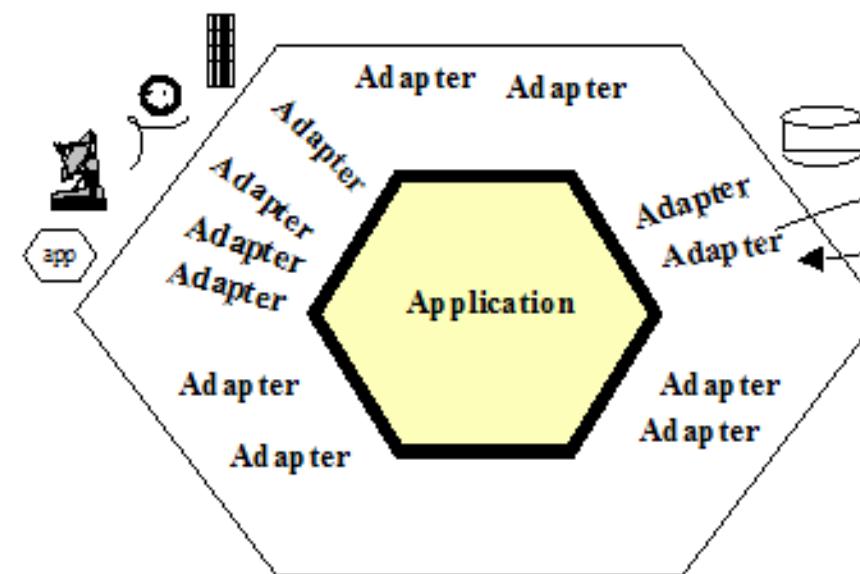
Anche la logica dell'origine dati potrebbe essere organizzata in più «packages» (ad esempio uno per ogni database se vi sono più database in gioco).

Analogamente la logica di dominio può essere suddivisa in aree distinte relativamente separate l'una dall'altra ognuna specializzata nell'assolvere specifici e determinati compiti.

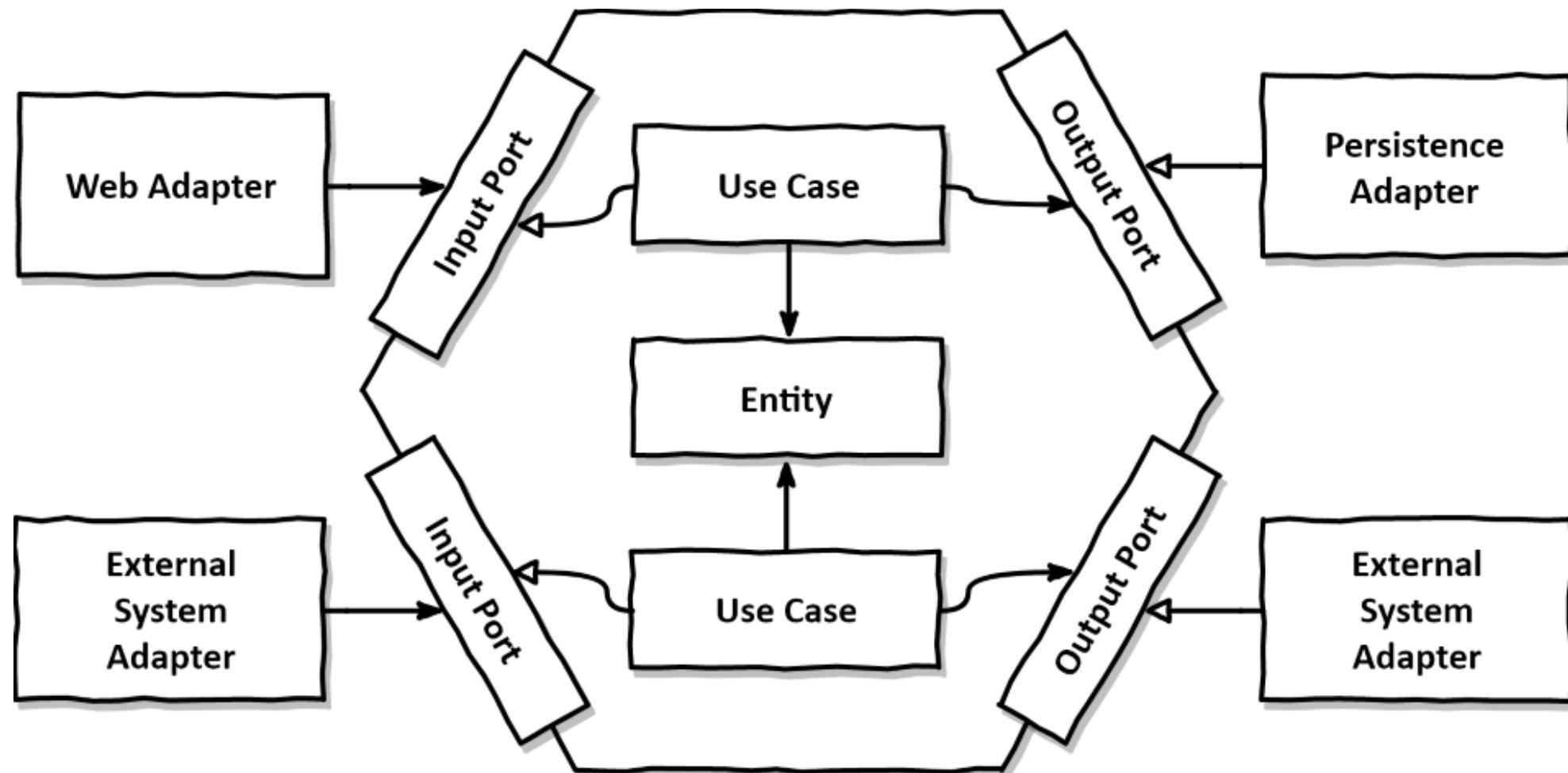
Hexagonal Architecture

Se analizziamo il layer della presentazione e quello dell'origine dati dal punto di vista del layer della logica di dominio appare evidente che i primi due sono la «conessione», il «canale di comunicazione» tra la logica di business con il mondo esterno.

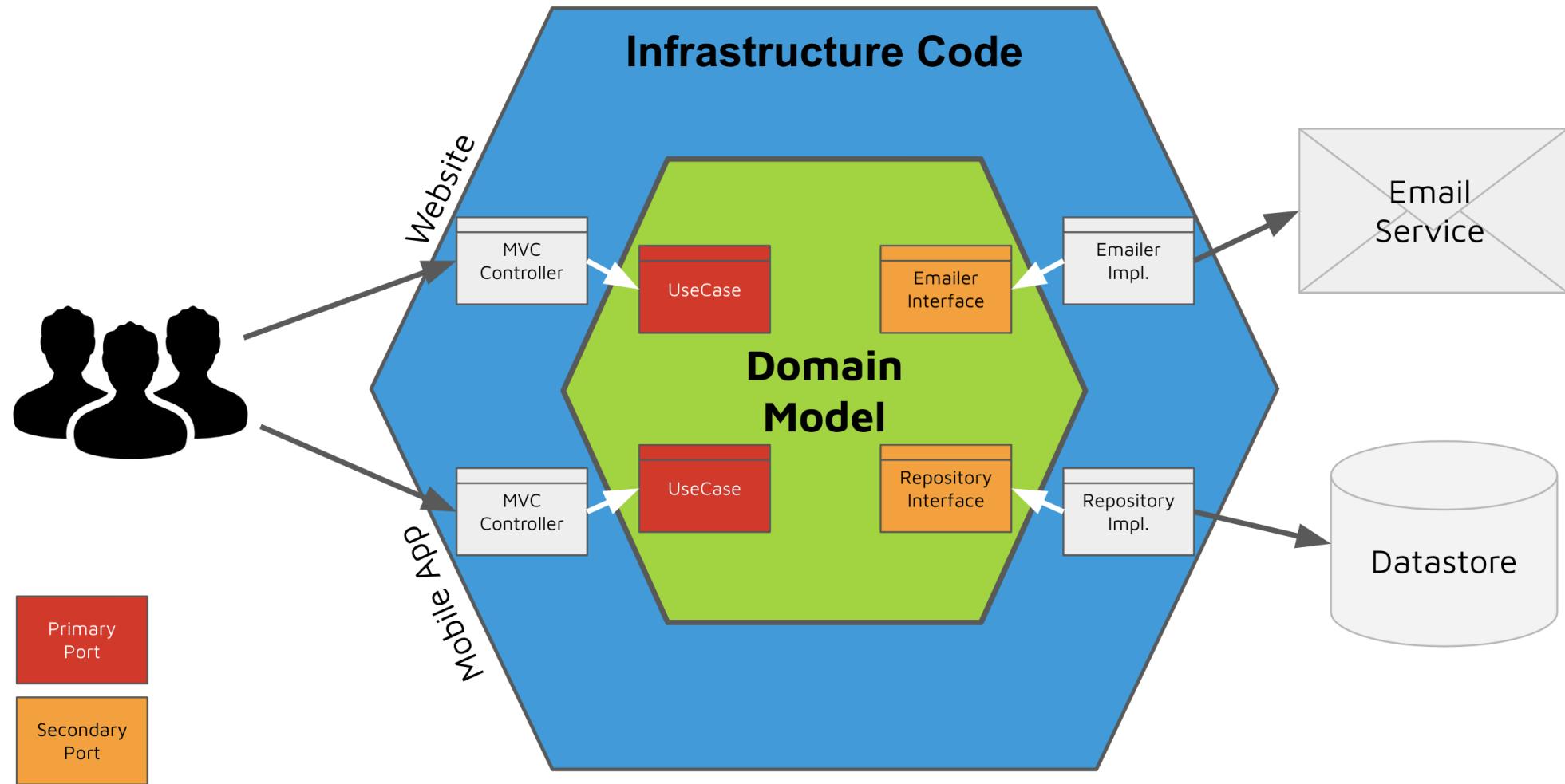
Questo modo di vedere lo strato di «domain» è la logica alla base del cosiddetto modello «**Hexagonal Architecture**» formulato da «**Alistair Cockburn**», ove un'architettura software è visto come un nucleo centrale circondato da interfacce a sistemi esterni.



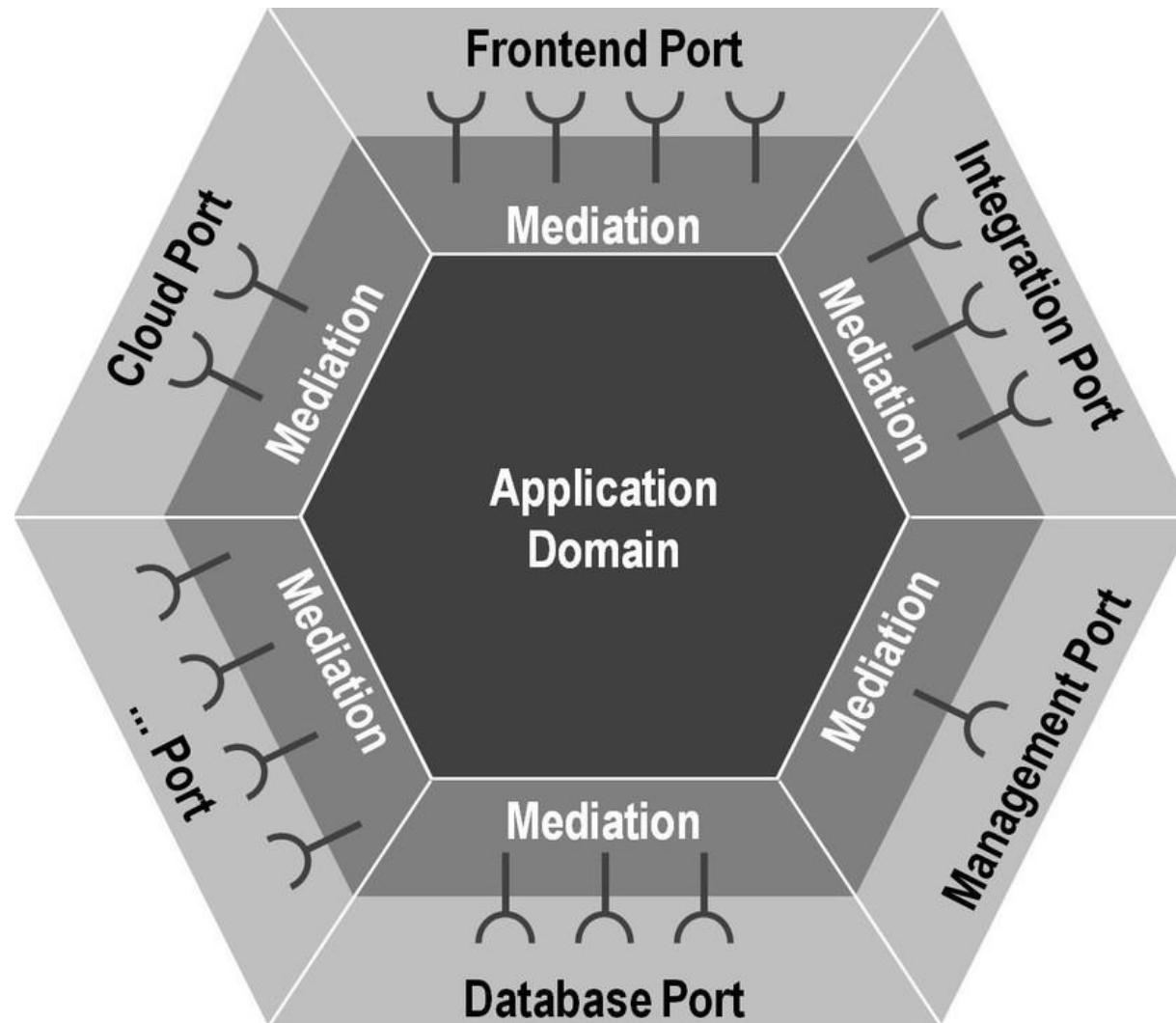
Hexagonal Architecture



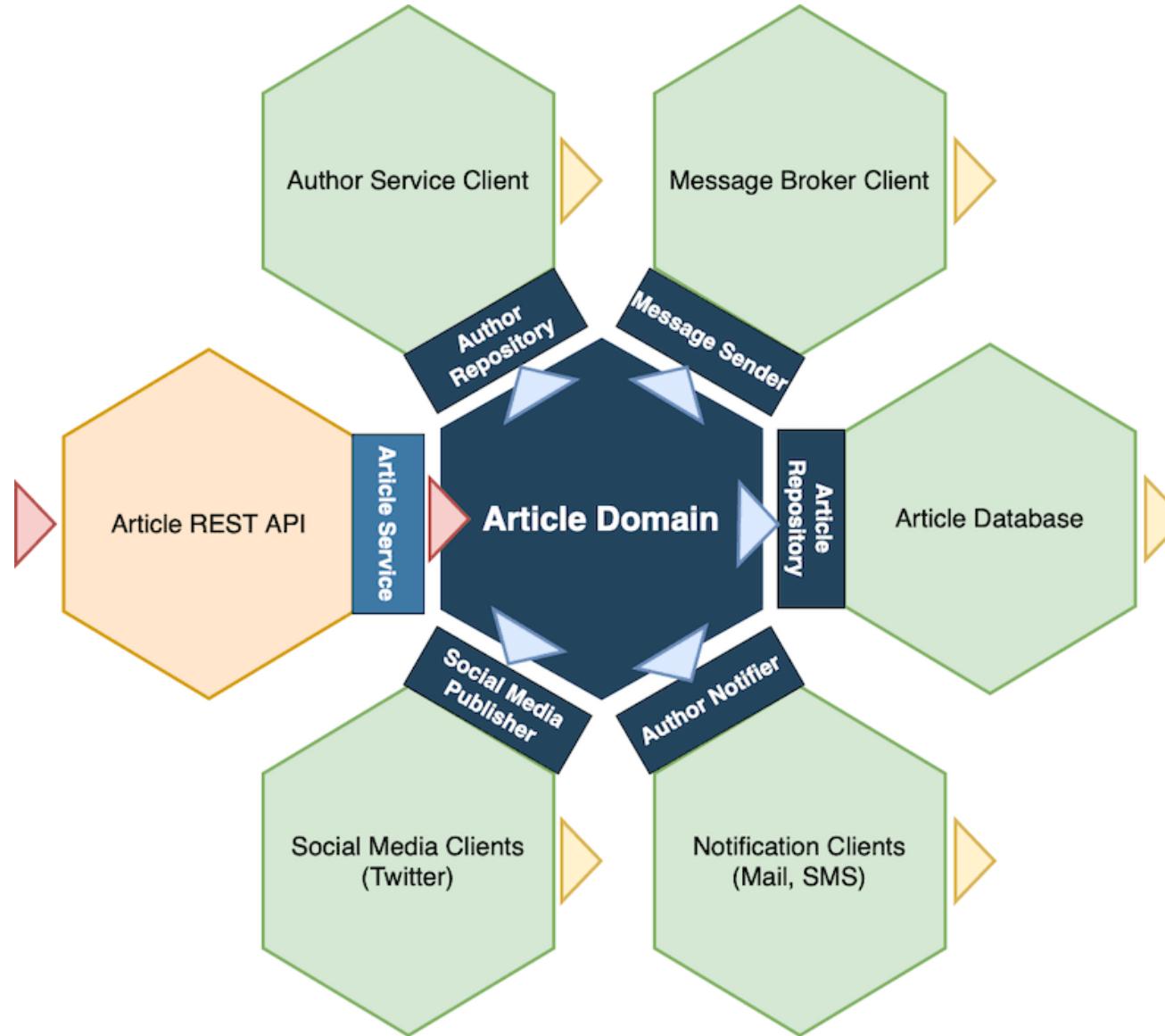
Hexagonal Architecture



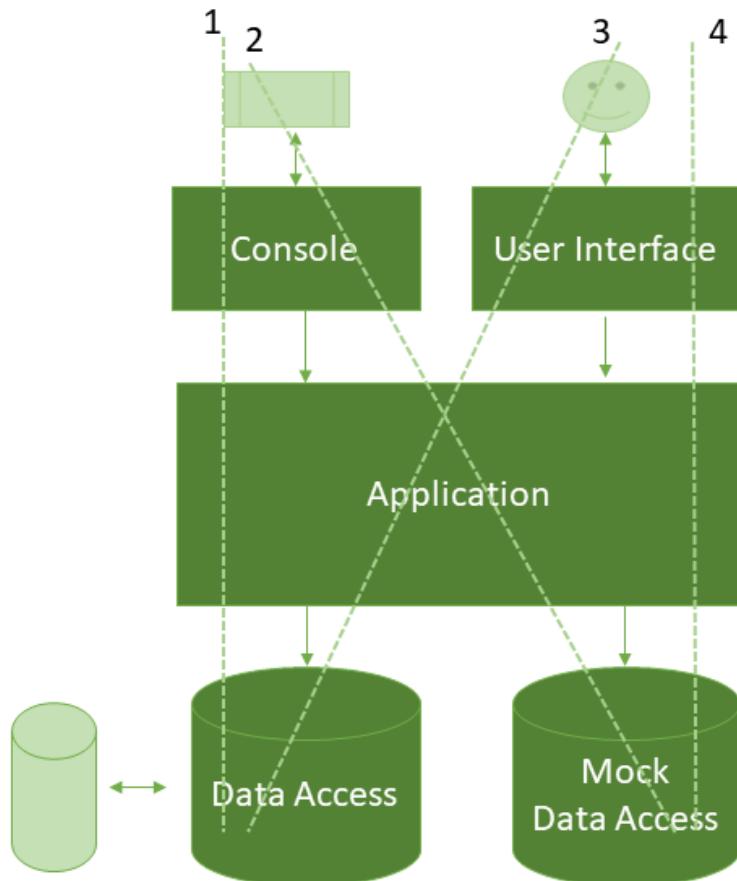
Hexagonal Architecture



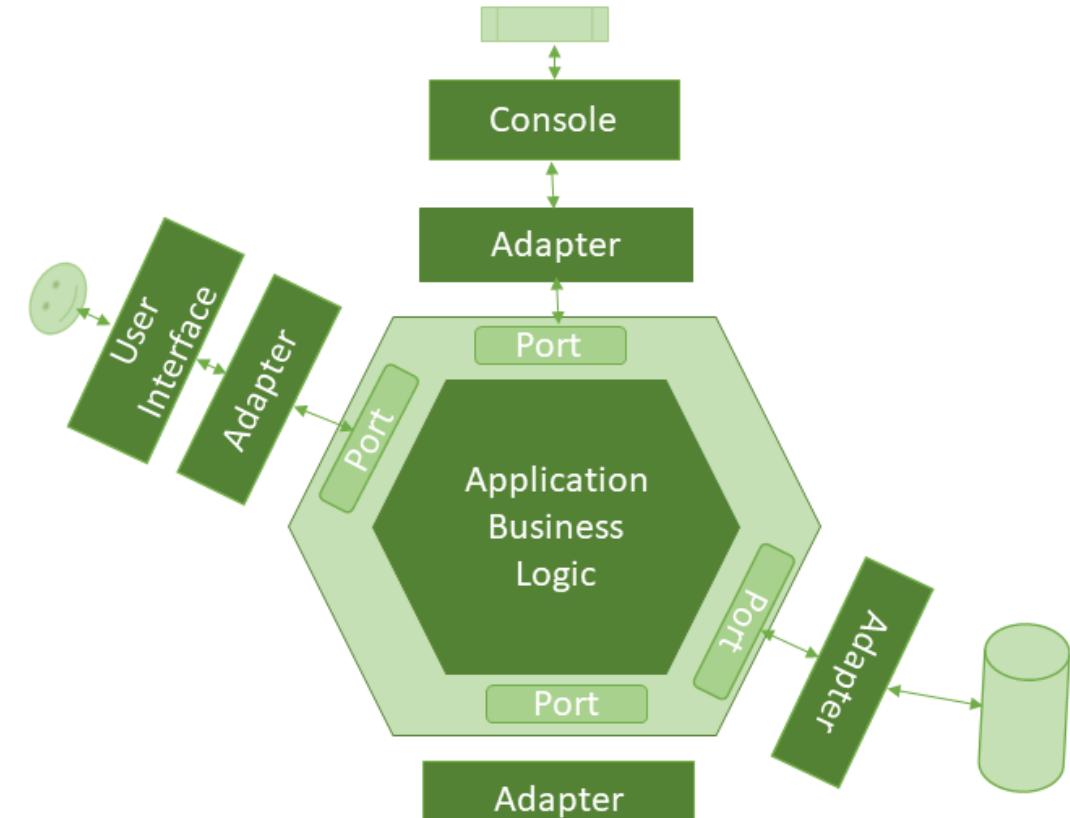
Hexagonal Architecture



Three Layer Architectural vs Hexagonal Architecture



Three Layer Architectural drawing



Hexagonal Architectural drawing

Alla luce di questa nuova prospettiva possiamo pensare che:

- **La presentazione è un'interfaccia esterna per un servizio che il sistema offre a qualcun altro (sia che si tratti di un utente o di un semplice programma remoto)**
- **L'origine dati è l'interfaccia per le «cose» che ti forniscono un servizio.**

Sebbene per ogni applicazione aziendale possiamo identificare i tre livelli di responsabilità comuni di presentazione, dominio e origine dati, il modo in cui separarli dipende dalla complessità dell'applicazione

Il consiglio generale è di scegliere la forma di separazione più appropriata per il problema in esame, o, più in generale, di assicurarsi di operare un qualche tipo di separazione, quanto meno a livello di subroutine.

Insieme alla separazione, esiste anche una regola fissa sulle dipendenze:

«Il dominio e l'origine dati non dovrebbero mai dipendere dalla presentazione».

Cioè, non dovrebbero sussistere chiamate di subroutine dal dominio o dal codice sorgente dei dati verso il codice della presentazione.

Adottare questa regola semplifica, ad esempio, la sostituzione (o l'aggiunta) di presentazioni diverse sulla stessa «base logica» e semplifica la modifica della presentazione senza che vi siano ripercussioni sugli strati più bassi dell'architettura.

Nota: La relazione tra il dominio e l'origine dati è più complessa e dipende dai modelli di architettura utilizzati per l'origine dati.

Uno dei problemi principali legato alla definizione della logica di dominio è la difficoltà nel riconoscere cosa sia «logica di dominio» e cosa siano altre forme di logica.

Un test informale (a titolo di esempio) potrebbe consistere nell'immaginare di aggiungere un livello radicalmente diverso a un'applicazione, come un'interfaccia a riga di comando a un'applicazione Web; se è necessario duplicare una funzionalità per riuscirci ciò è indice che una logica di dominio abbia «sconfinato» nella logica di presentazione.

L'obiettivo, quando si parla di livelli logici, è di dividere un sistema in parti separate per ridurre l'accoppiamento tra le diverse parti di un sistema.

Scegliere dove eseguire i livelli

La separazione tra i livelli è utile anche se i livelli sono tutti in esecuzione su una macchina fisica.

Tuttavia, ci sono luoghi in cui la struttura fisica di un sistema fa la differenza.

Per la maggior parte delle applicazioni IS (information System) la decisione è se eseguire l'elaborazione su un client, su una macchina desktop o su un server.

...e qui le scelte architetturali possono essere molteplici 😊 !!!!

Es.: Desktop Application, RIA Web-App, SPA etc. etc.

Scegliere dove eseguire i livelli

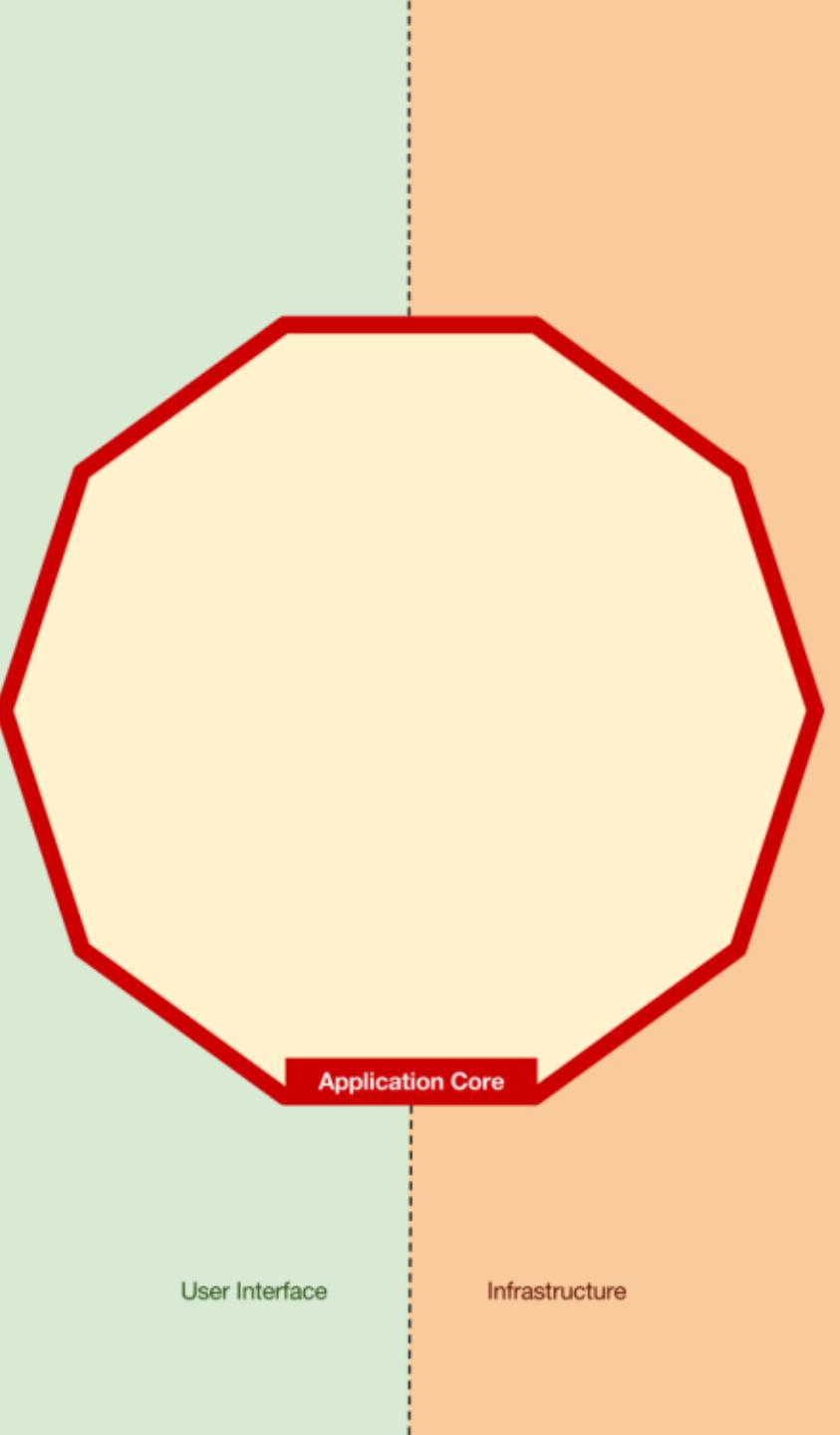
È possibile eseguire la logica di business tutto sul server o tutto sul client, oppure è possibile suddividerlo.

Sul server è la scelta migliore per facilità di manutenzione e scalabilità.

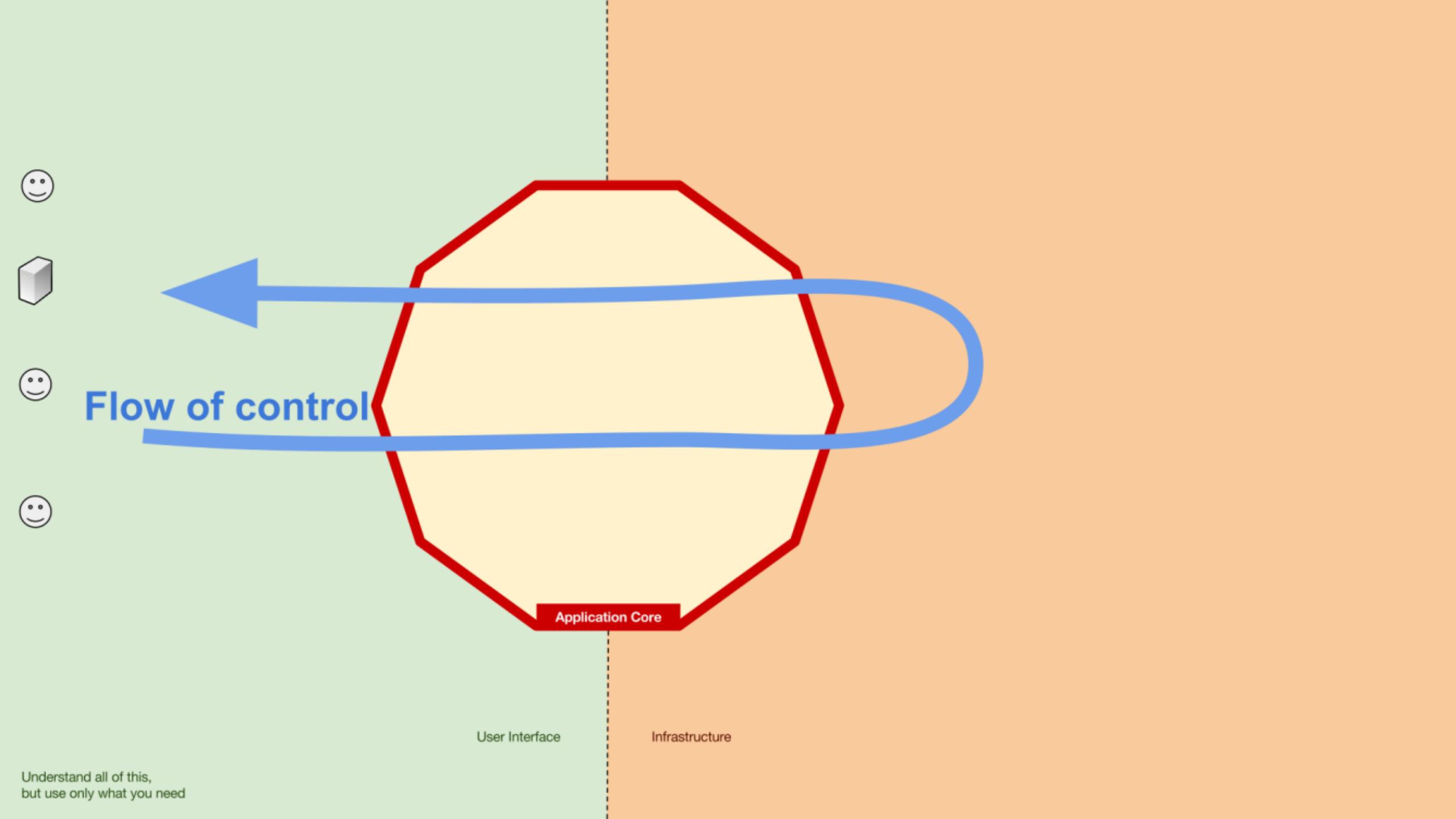
La richiesta di spostarlo sul client è per esigenza di reattività o per un utilizzo disconnesso.

Se è necessario eseguire un po' di logica sul client, si può considerare di eseguirla tutta lì, almeno in questo modo è tutto in un unico posto anche se ovviamente non è una scelta ottimale.

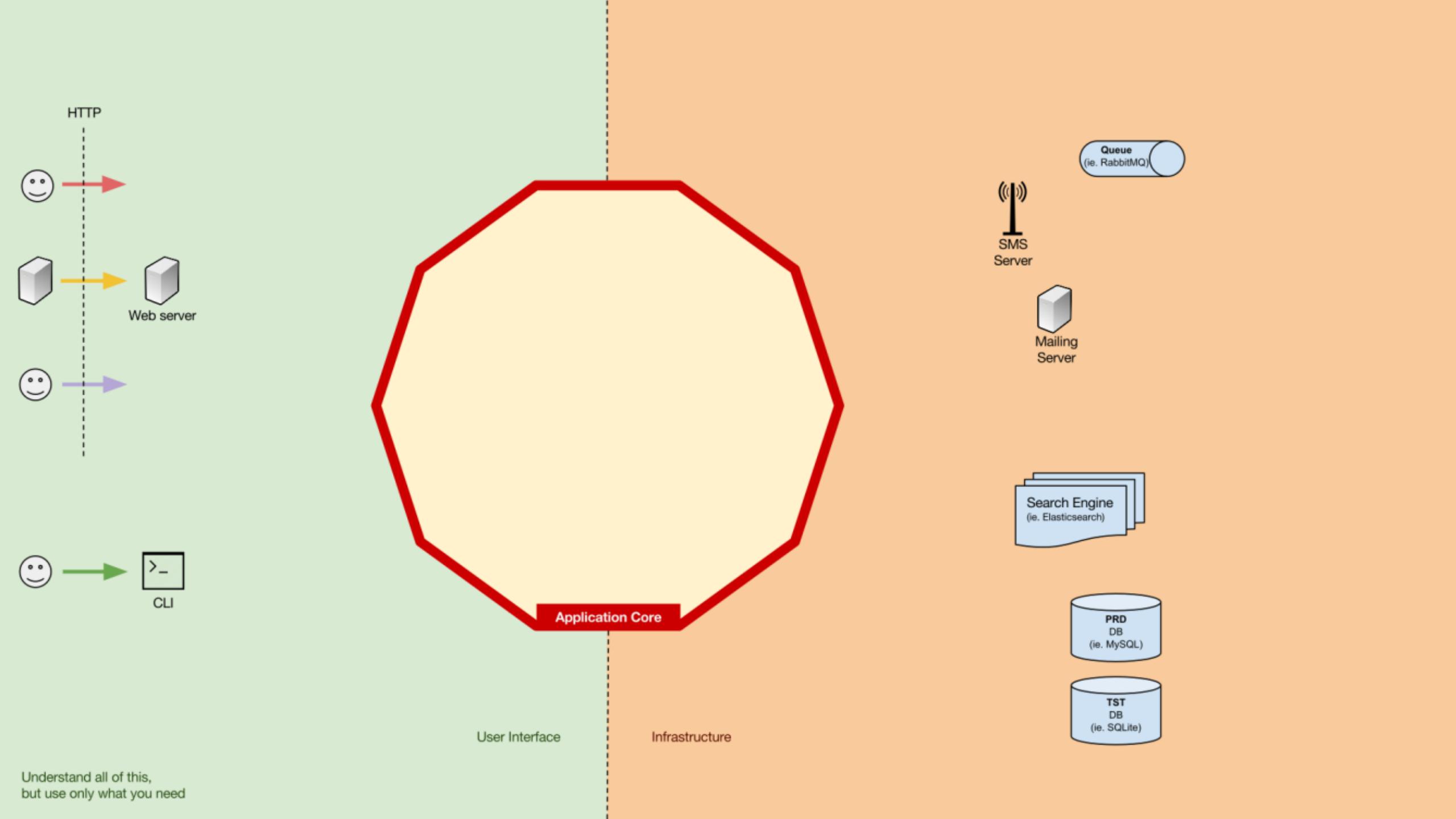
Design di
un'architettura
Esagonale



Understand all of this,
but use only what you need

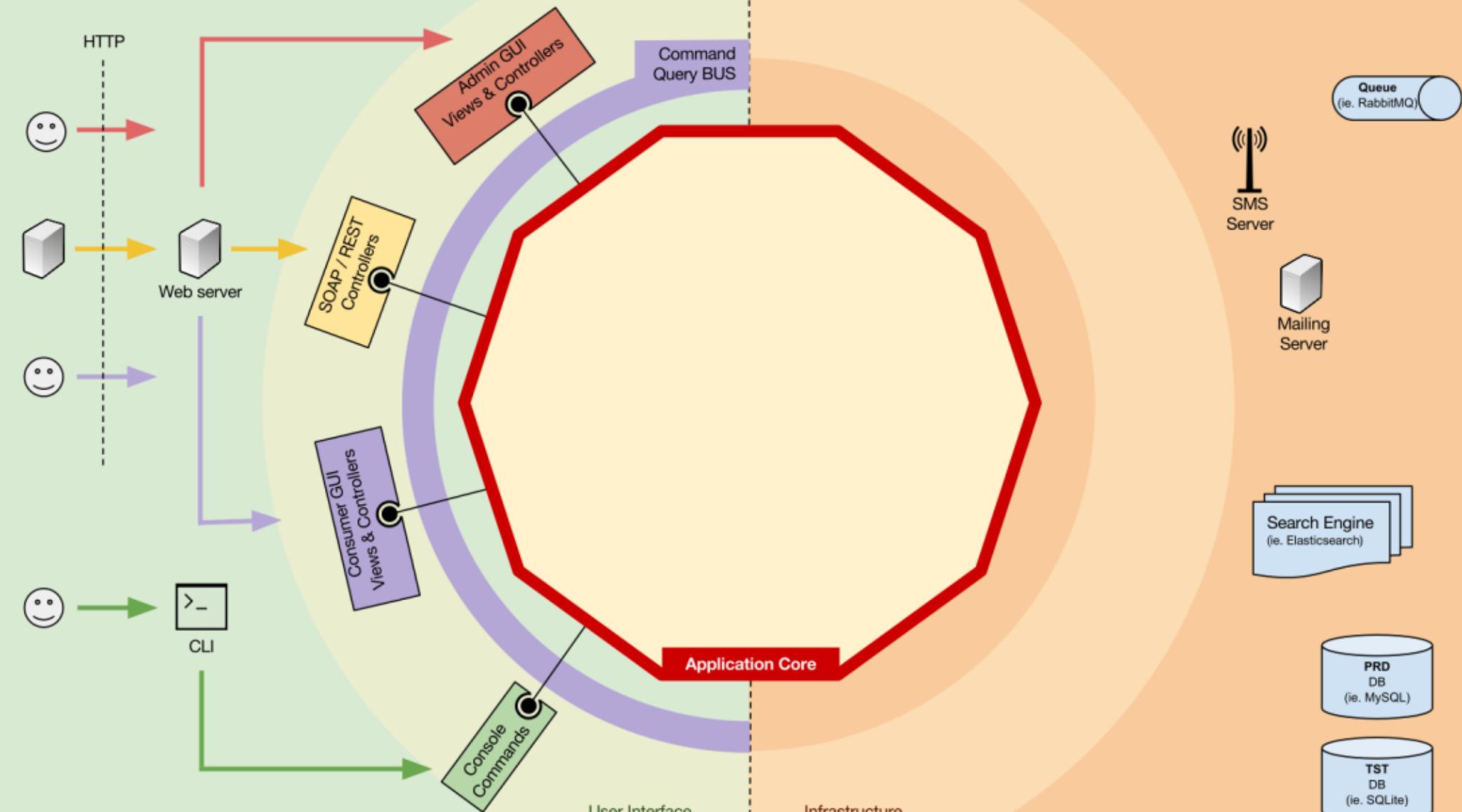


Understand all of this,
but use only what you need



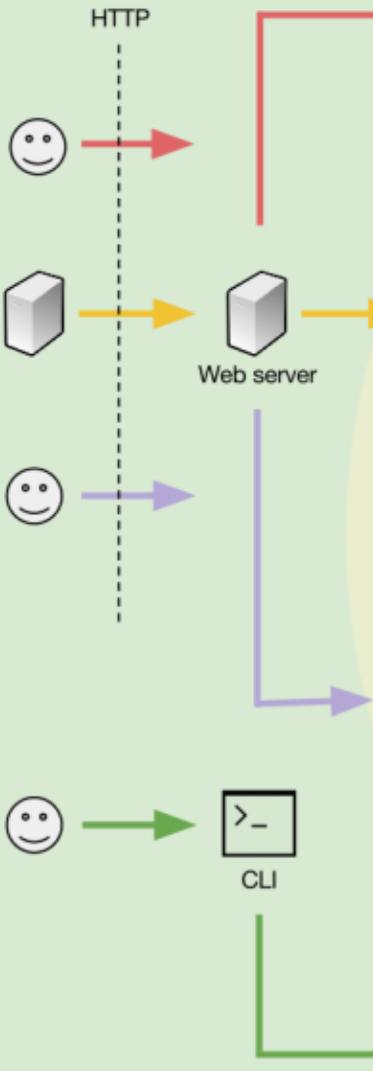
Understand all of this,
but use only what you need

Primary/Driving Adapters

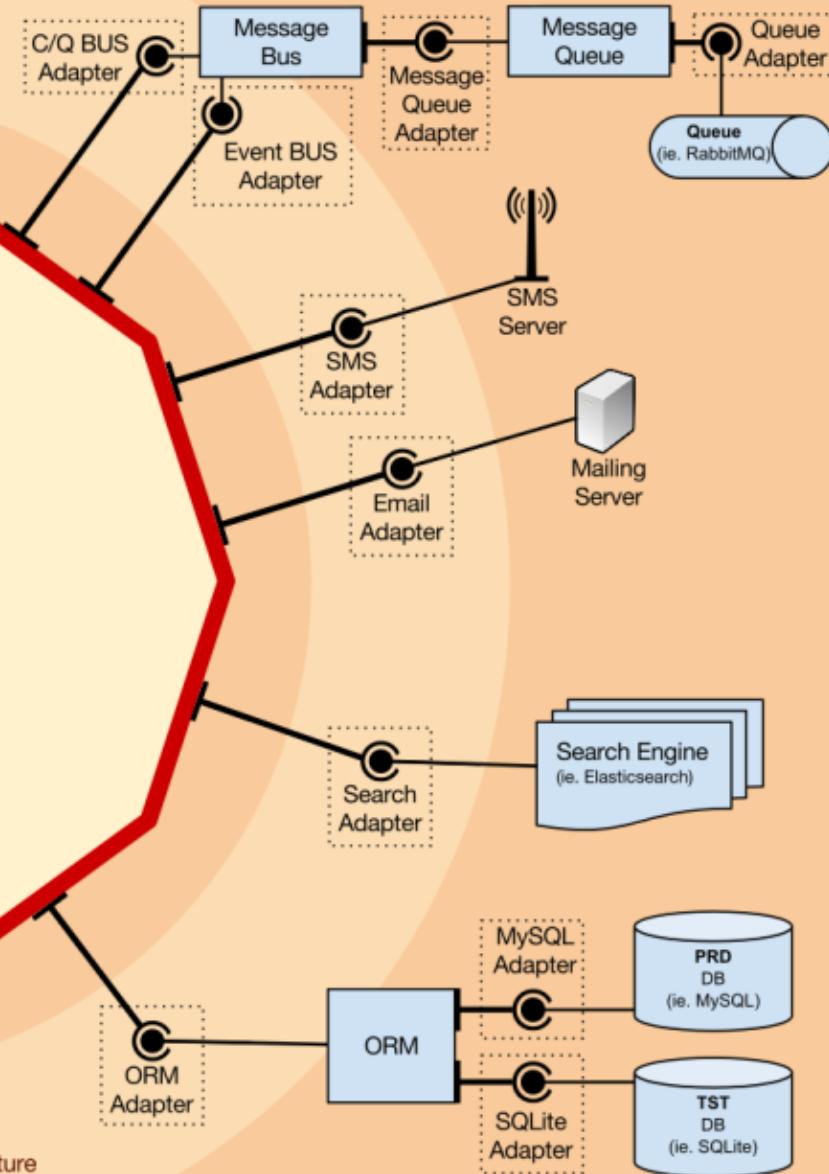


Understand all of this,
but use only what you need

Primary/Driving Adapters

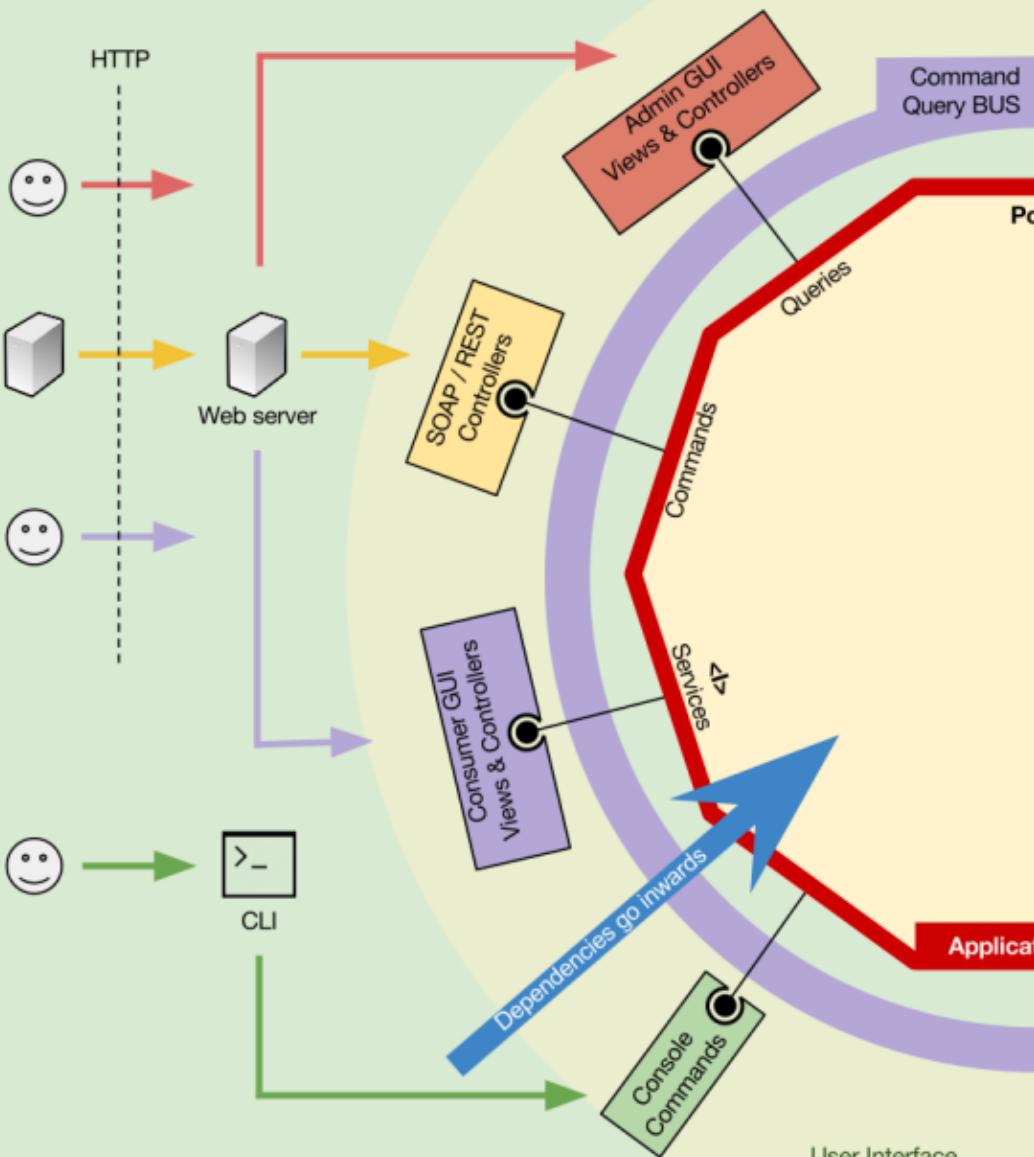


Secondary/Driven Adapters

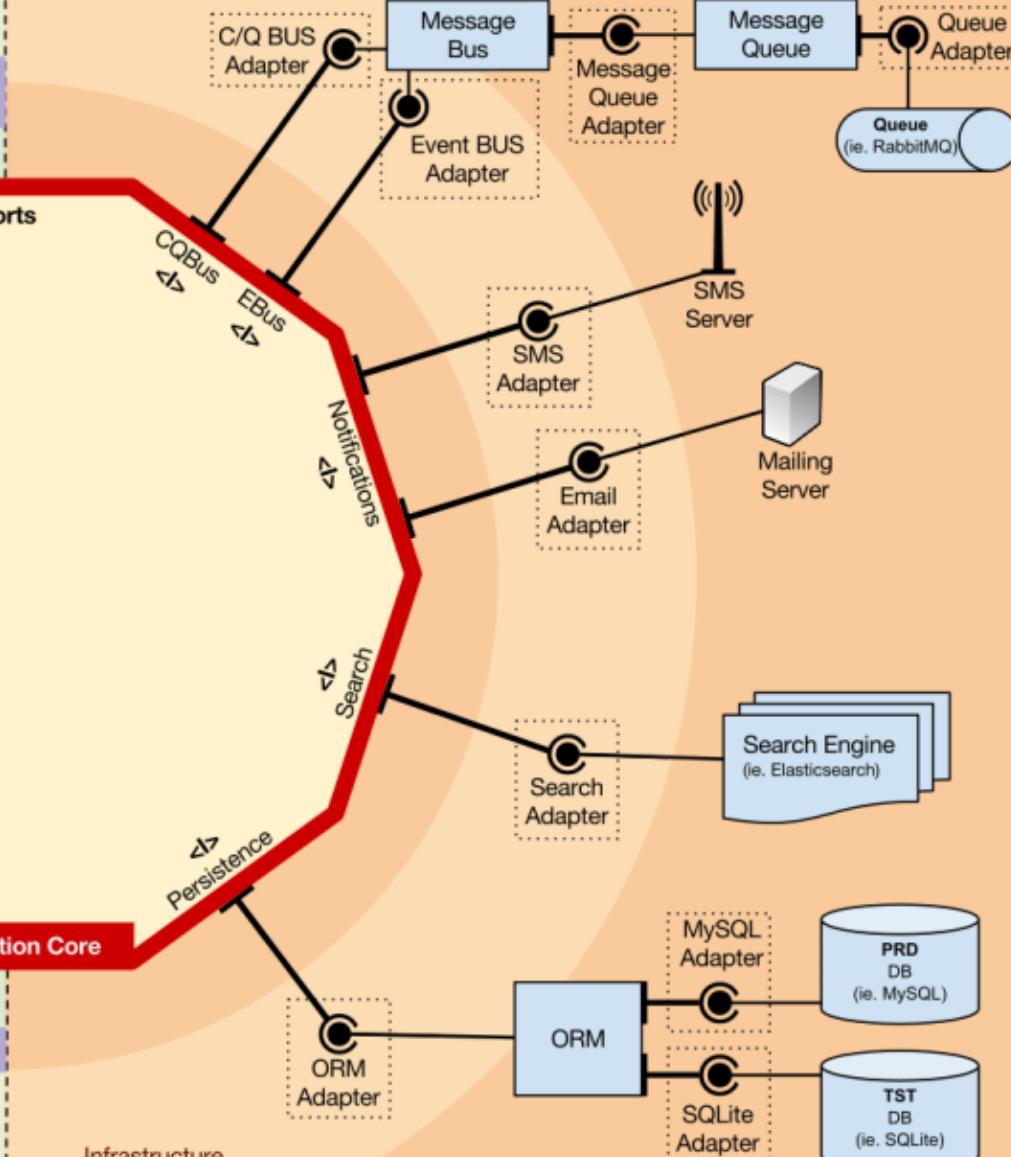


Understand all of this,
but use only what you need

Primary/Driving Adapters

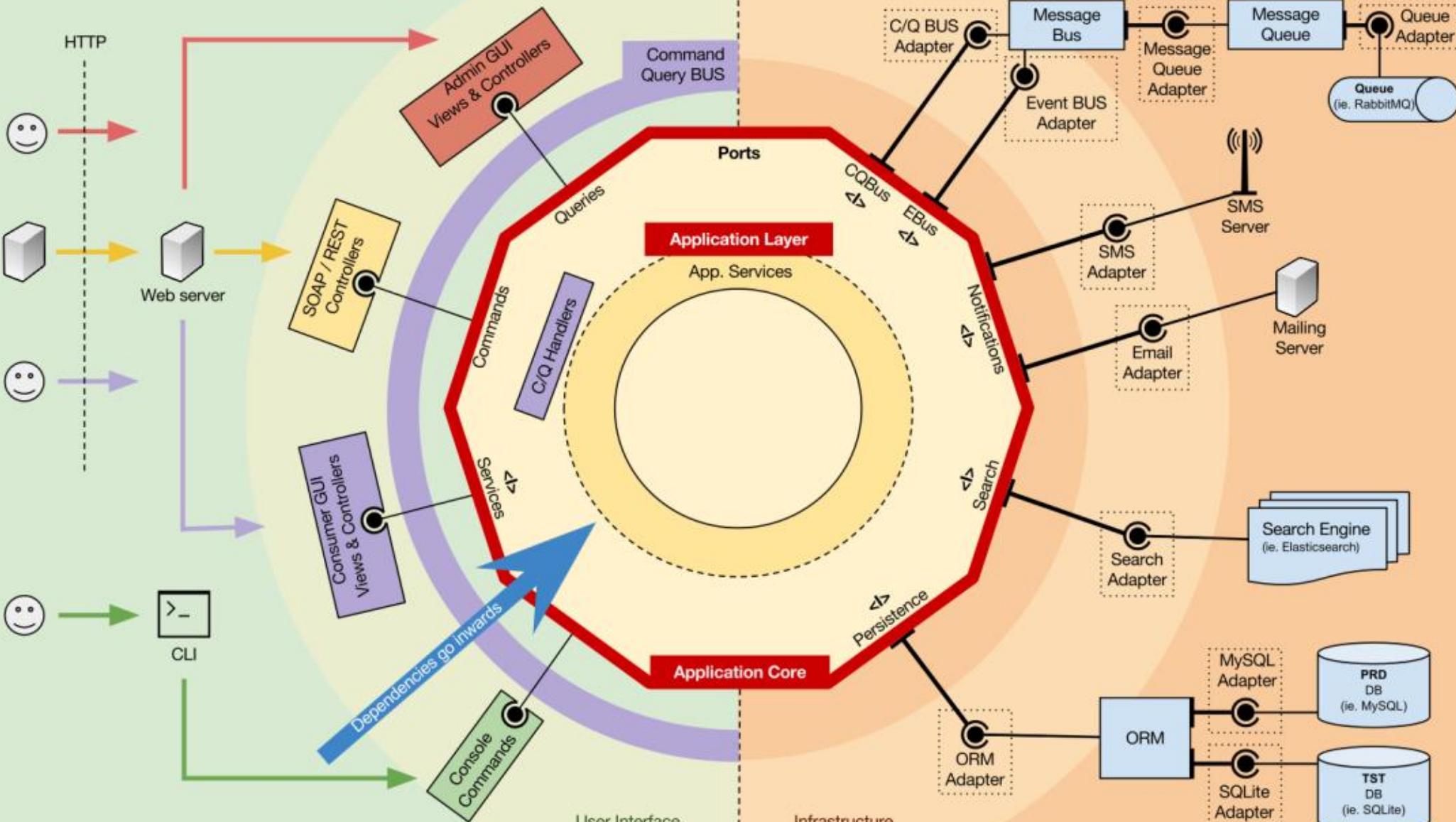


Secondary/Driven Adapters



Understand all of this,
but use only what you need

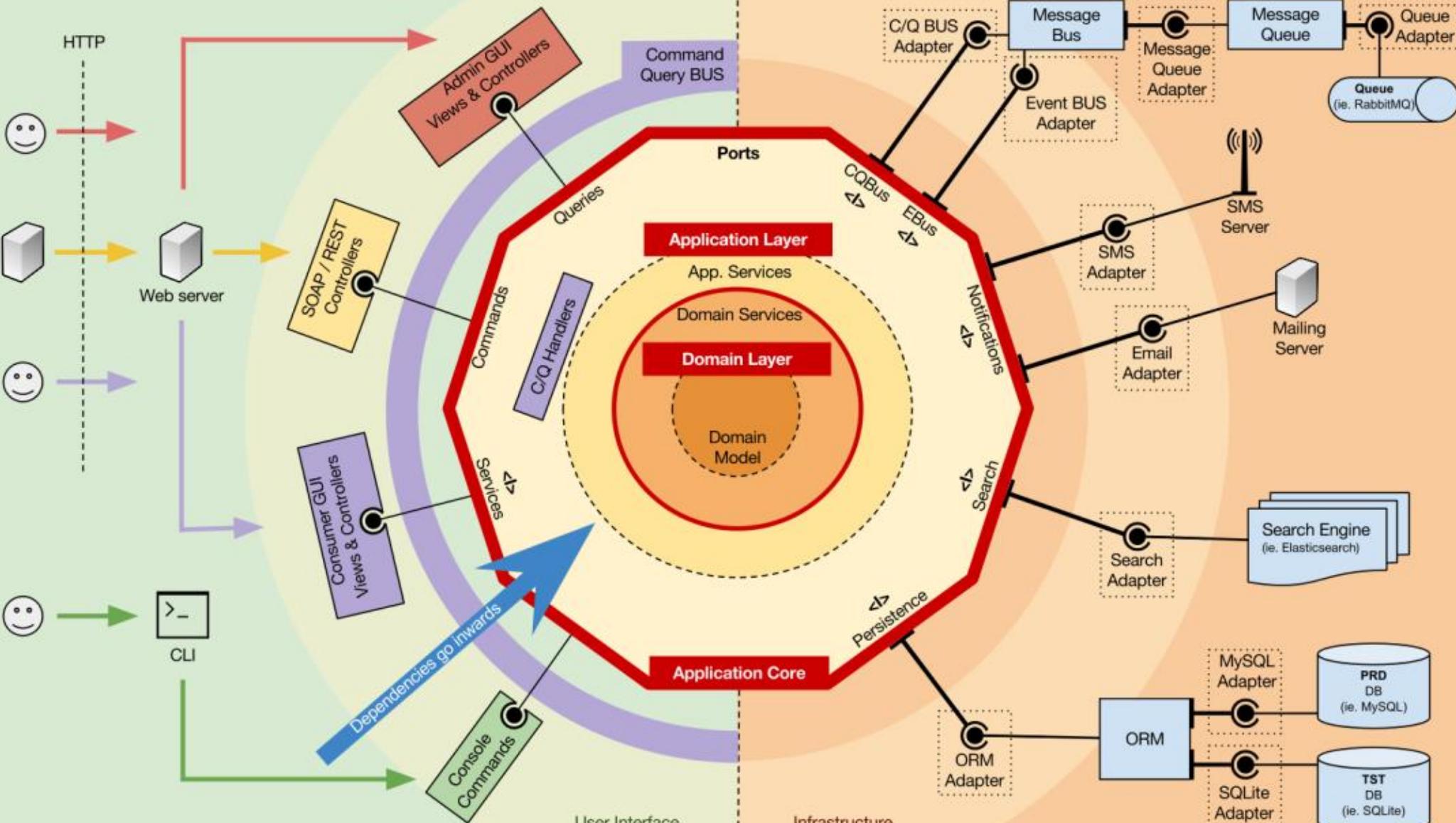
Primary/Driving Adapters



Secondary/Driven Adapters

Understand all of this,
but use only what you need

Primary/Driving Adapters

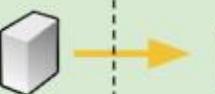


Secondary/Driven Adapters

Understand all of this,
but use only what you need

Primary/Driving Adapters

HTTP



Command
Query BUS

Commands
Queries
Services



Application Layer

App. Services

Domain Services

Domain Model

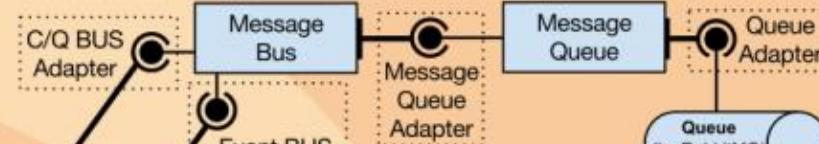
Component

Dependencies go inwards

User Interface

Infrastructure

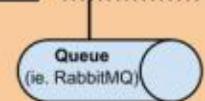
Secondary/Driven Adapters



Message
Bus

Message
Queue

Queue
Adapter



SMS
Server

Mailing
Server

Email
Adapter

Search
Engine

(e.g. Elasticsearch)

Search
Adapter

MySQL
Adapter

PRD
DB

(e.g. MySQL)

SQLite
Adapter

TST
DB

(e.g. SQLite)

ORM
Adapter

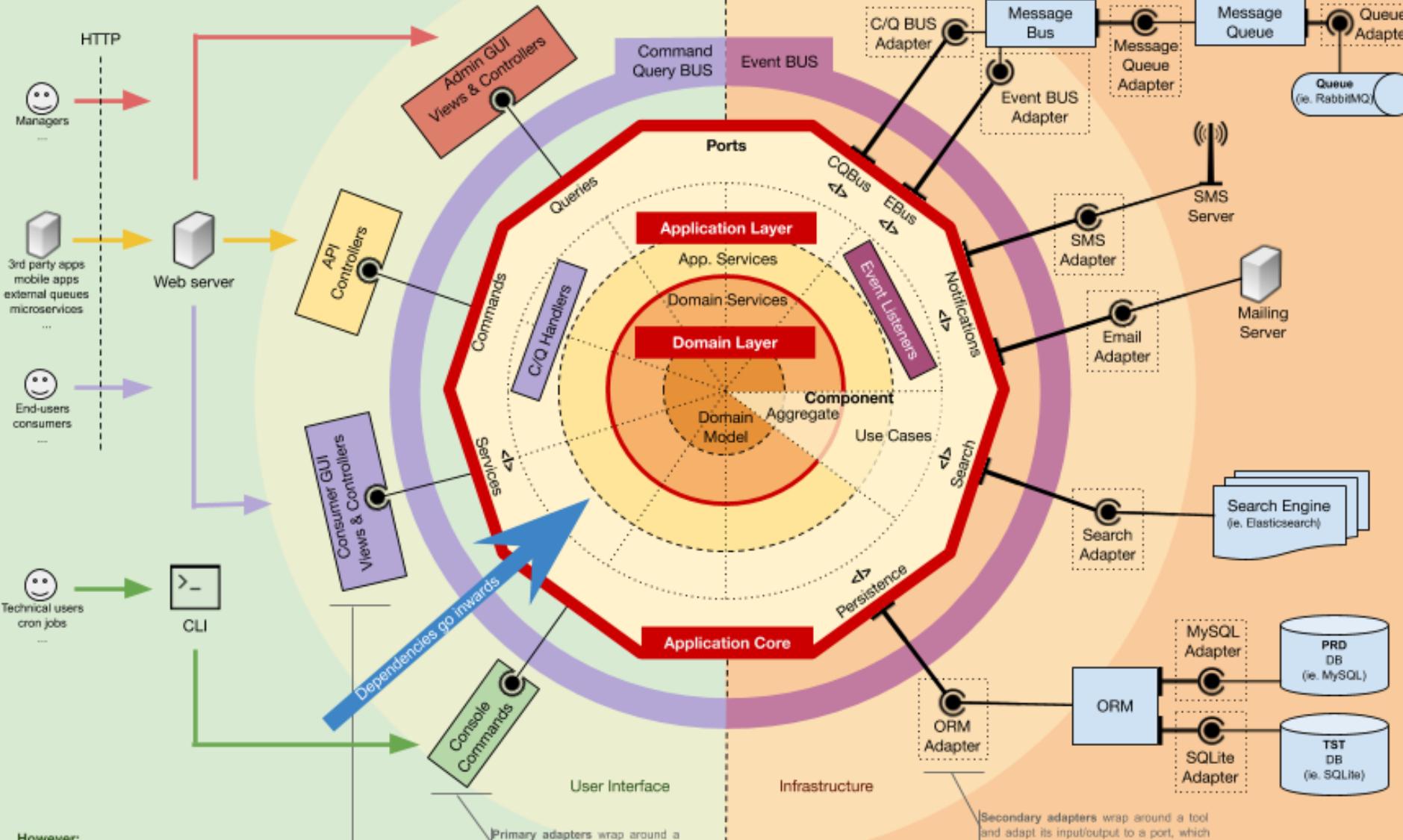
ORM

ORM

Primary/Driving Adapters

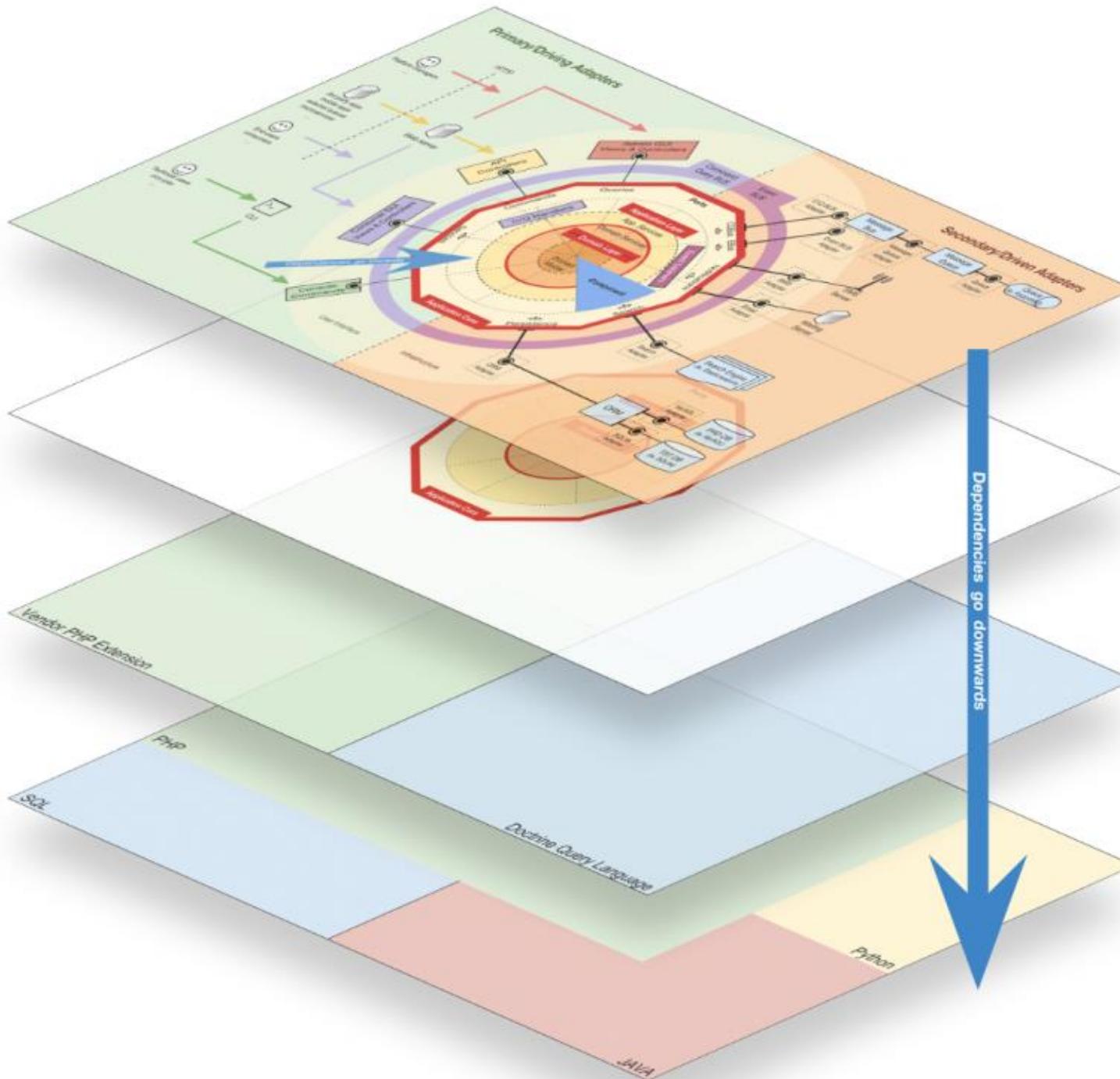
Explicit Architecture

Secondary/Driven Adapters



However:

- The map is not the territory.
- Plans are worthless, but planning is everything.
- Understand all of this, but use only what you need.
- The actual architecture is driven by the project requirements.

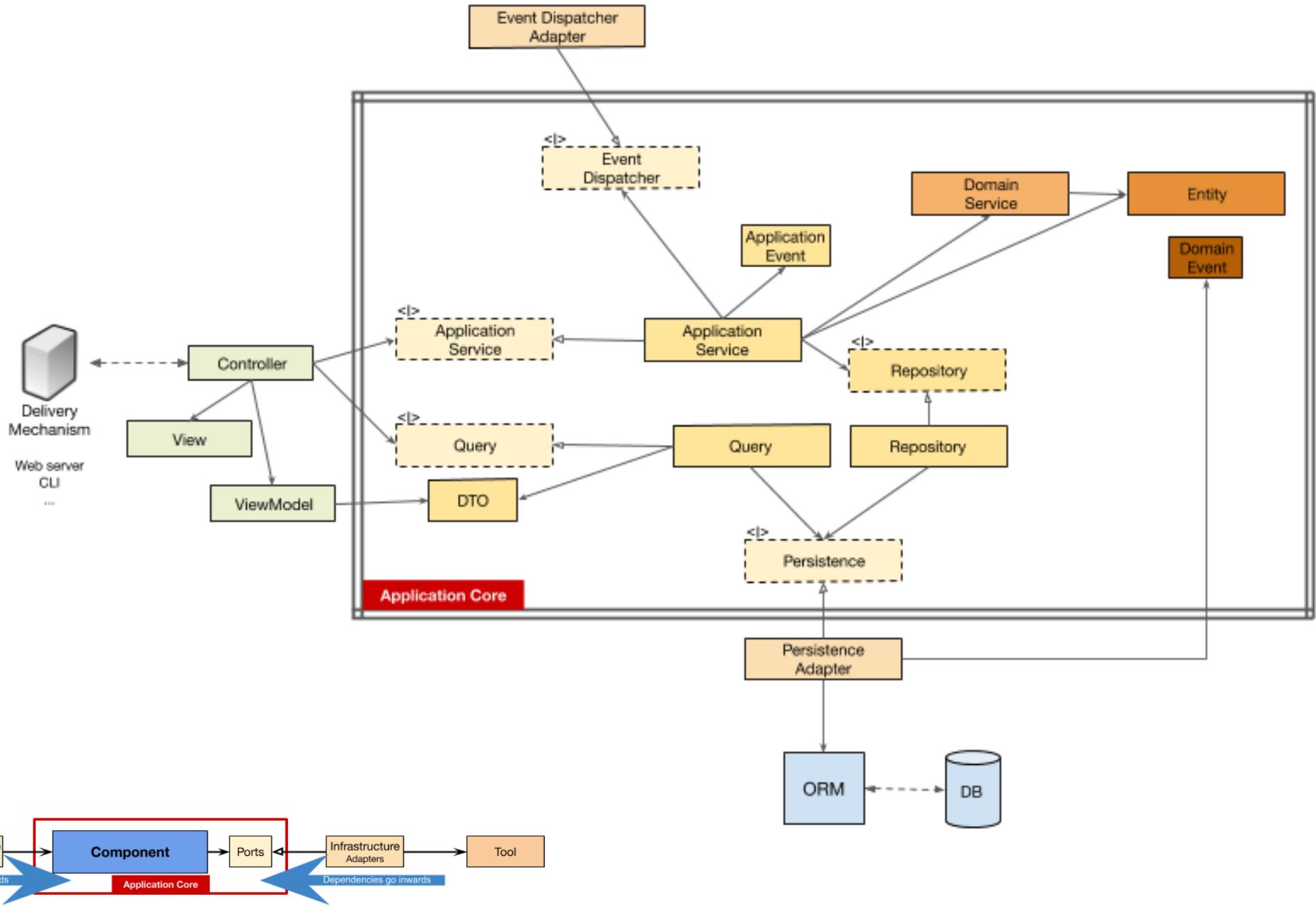


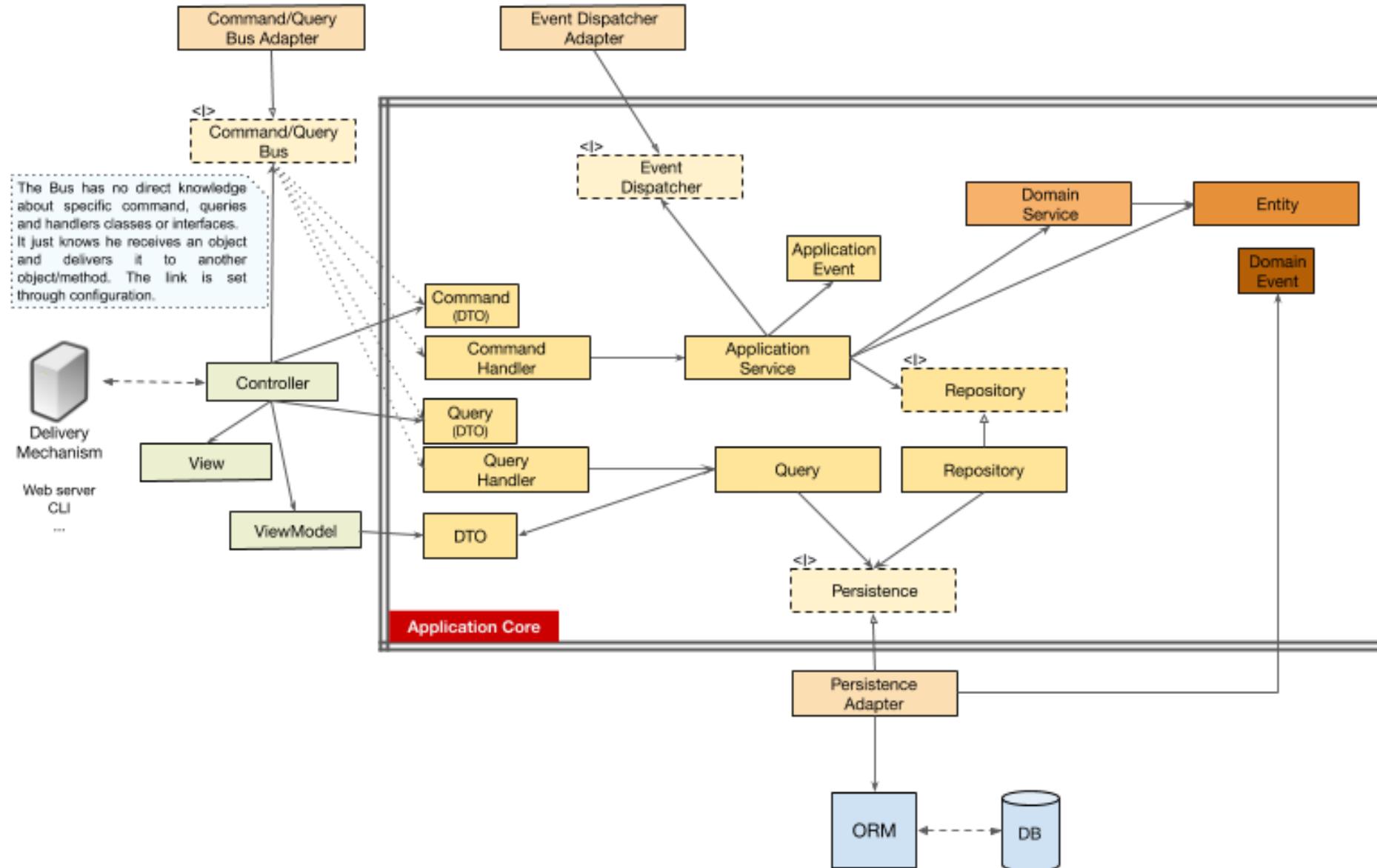
Application Core, UI and adapters

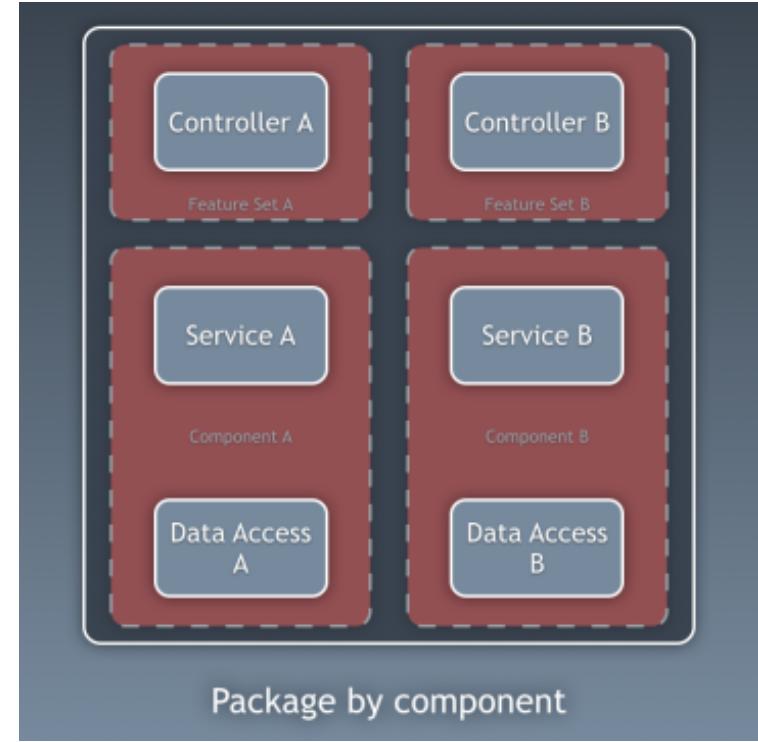
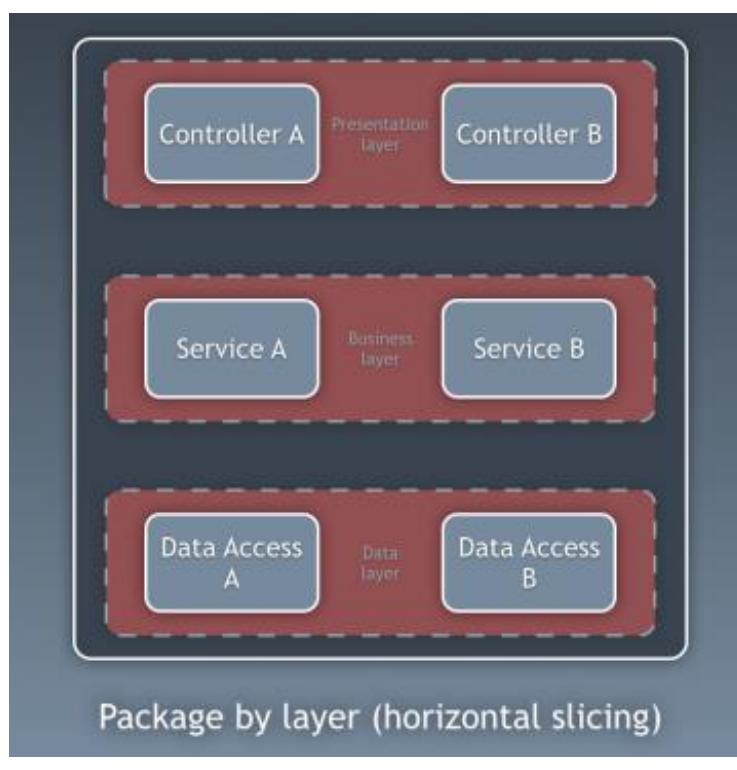
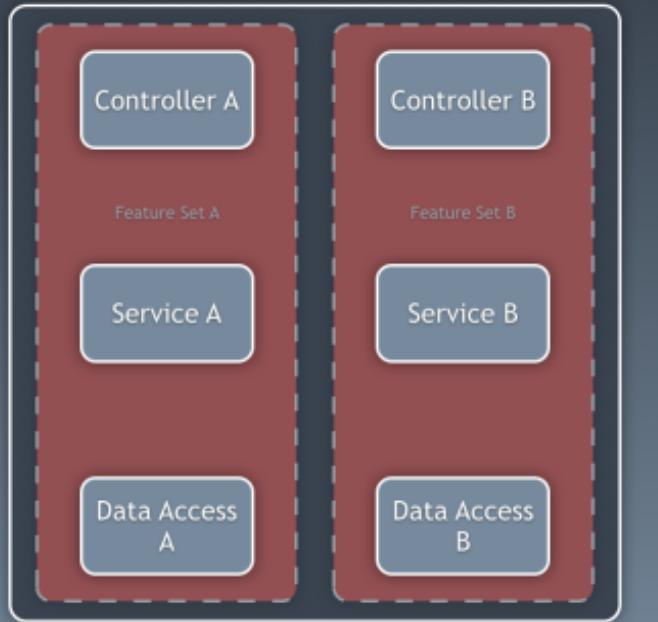
Shared kernel

Prog. languages extensions

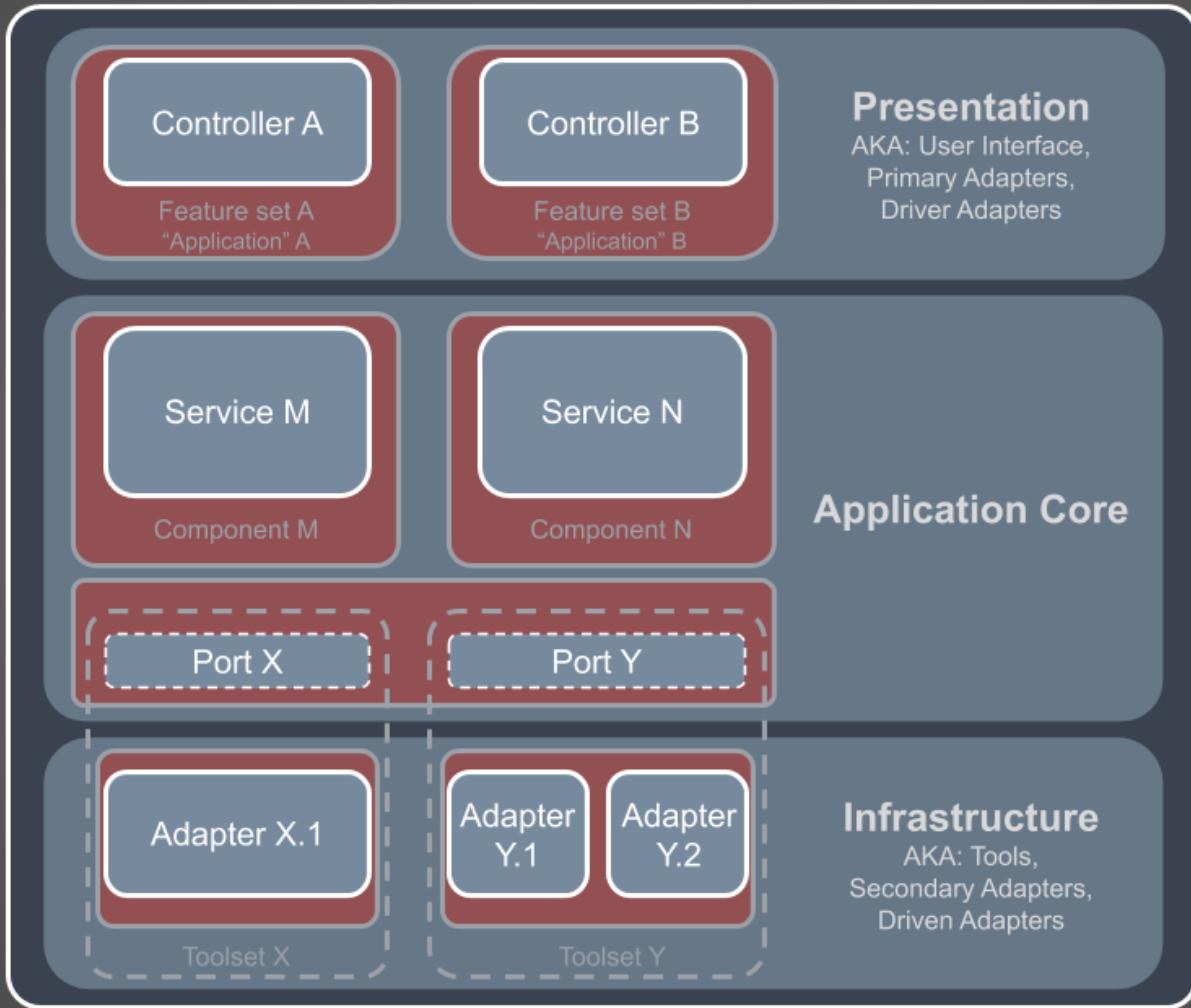
Prog. languages



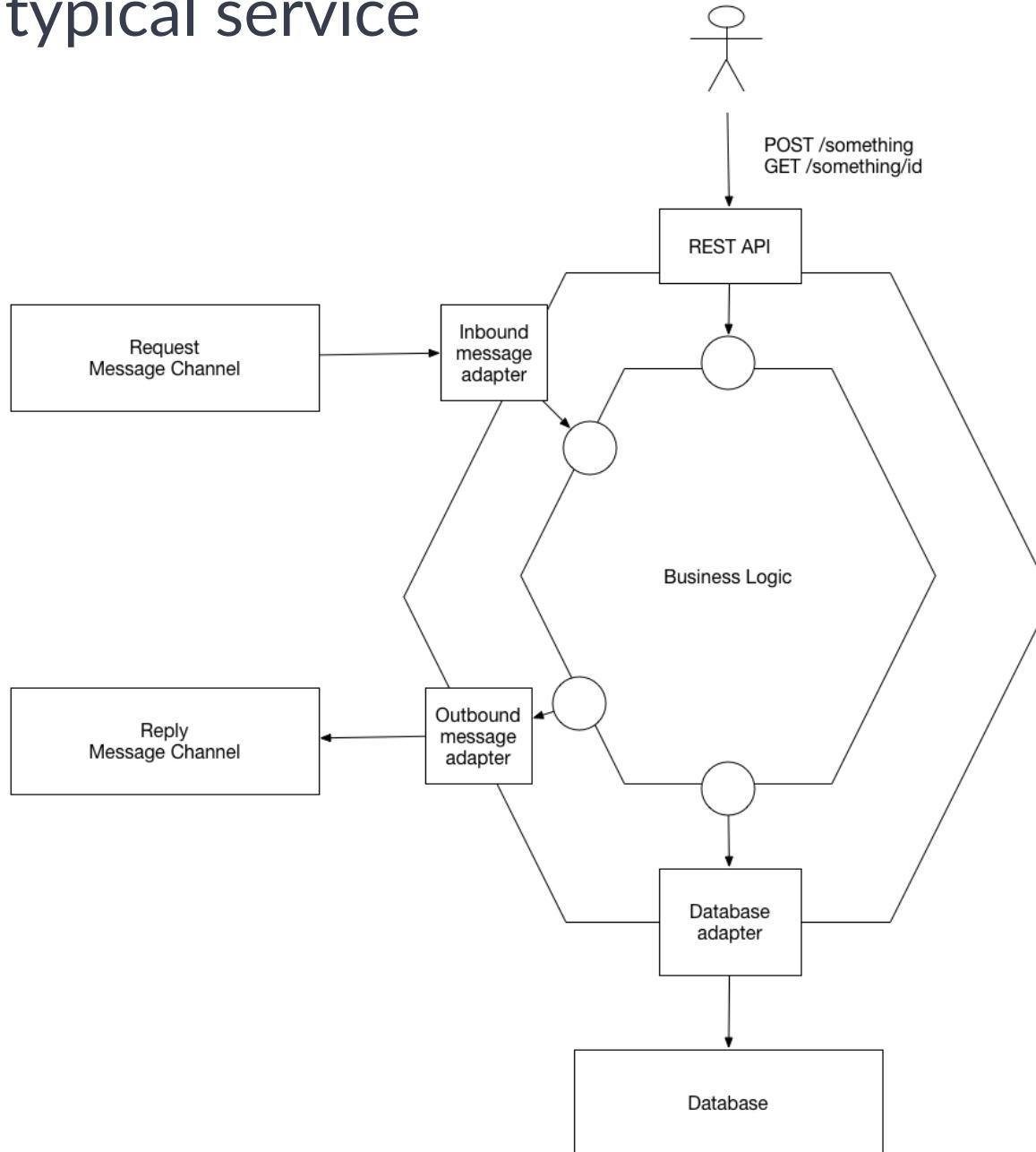




Package by Component



The structure of a typical service



Introduzione al Domain Logic Pattern

Tra i pattern per la modellazione della logica del dominio evidenziamo i seguenti tre modelli principali:

- **Transaction Script**
- **Domain Model**
- **Table Module**

L'approccio più semplice per memorizzare la logica del dominio è il «Transaction Script».

Esso è essenzialmente una procedura inizialmente che prende l'input dalla presentazione, lo elabora con convalide e calcoli, memorizza i dati nel database e invoca qualsiasi esecuzione di altre operazioni da altri sistemi.

Infine risponde con altri dati alla presentazione effettuando eventuali calcoli aggiuntivi per organizzare e formattare la risposta.

L'idea è di realizzare un'unica procedura per ogni azione che un utente potrebbe compiere.

Possiamo paragonarlo a uno script per compiere un'azione o una transazione commerciale.

L'implementazione non deve essere una singola procedura inline di codice.

I «pezzi» della logica vengono separati in subroutine e queste subroutine possono essere condivise tra diversi Transaction Scripts.

Tuttavia, la forza trainante è ancora quella di una procedura per ogni azione.

Esempio: in un sistema di vendita al dettaglio potremmo avere Transaction Scripts

- **Per il checkout**
- **Per aggiungere qualcosa al carrello**
- **Per visualizzare lo stato della consegna e così via.**

Uno Transaction Scripts offre diversi vantaggi:

- È un semplice modello procedurale che la maggior parte degli sviluppatori comprende.
- Funziona bene con un semplice livello di origine dati utilizzando «Row Data Gateway» o «Table Data Gateway».

È ovvio come impostare i limiti della transazione: iniziare con l'apertura di una transazione e terminare con la sua chiusura.

È facile implementarlo usando strumenti che lavorano «dietro le quinte».

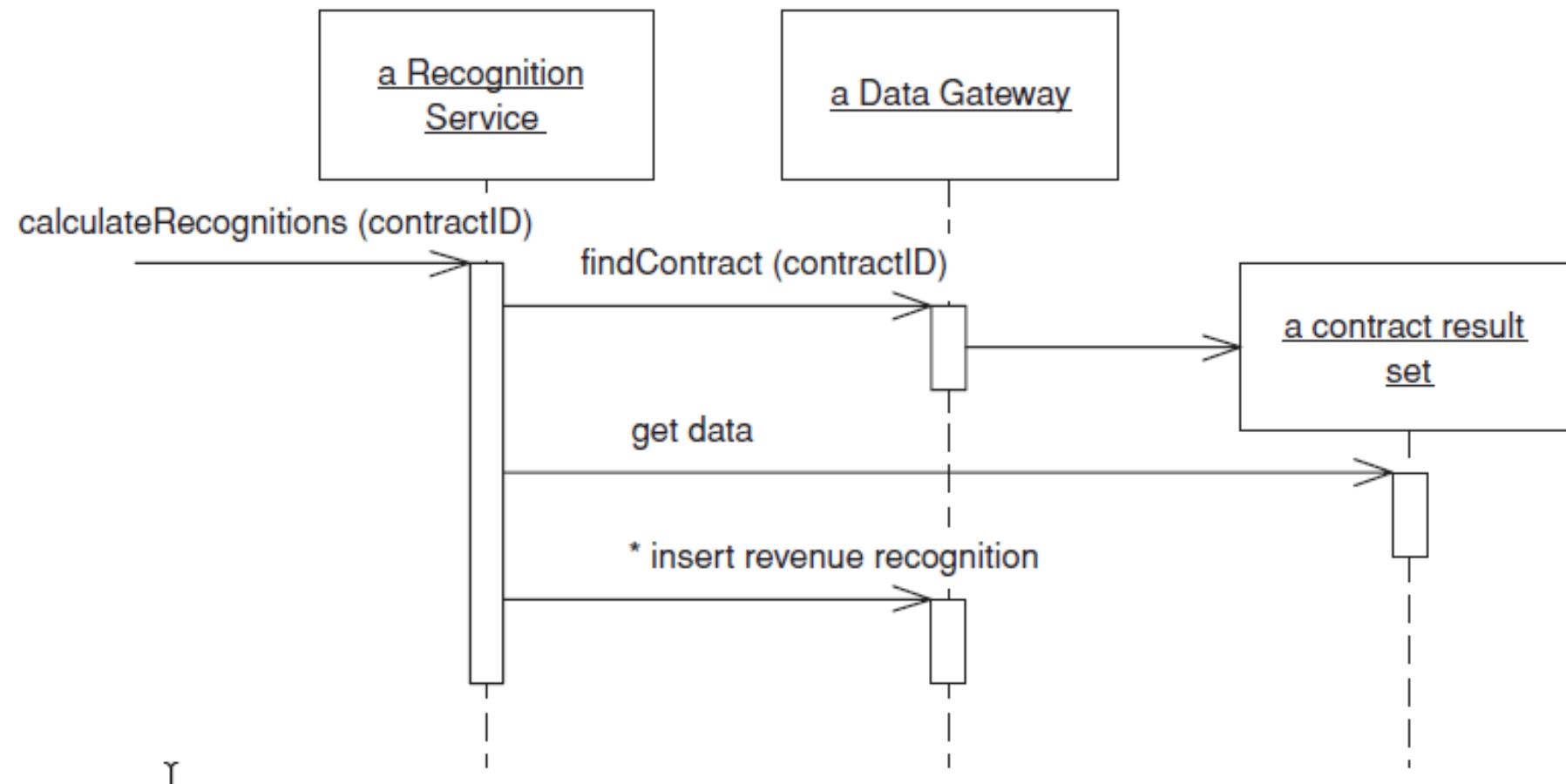
Purtroppo, vi sono molti svantaggi, che emergono con l'aumentare della complessità della logica del dominio.

Spesso tenderà a scrivere codice duplicato poiché diverse transazioni devono fare cose simili.

Alcune duplicazioni possono essere risolte scomponendo le subroutine comuni, ma anche così è difficile rimuoverle, ma soprattutto risulta più difficile individuarle.

L'applicazione risultante potrebbe diventare in breve tempo una rete piuttosto ingarbugliata di routine senza una struttura chiara e definita.

Transaction Script



Quando siamo in presenza di una logica complessa questo è il campo dove entrano prepotentemente in gioco gli oggetti e la OOP.

In questo scenario l'approccio migliore è quello di un Domain Model.

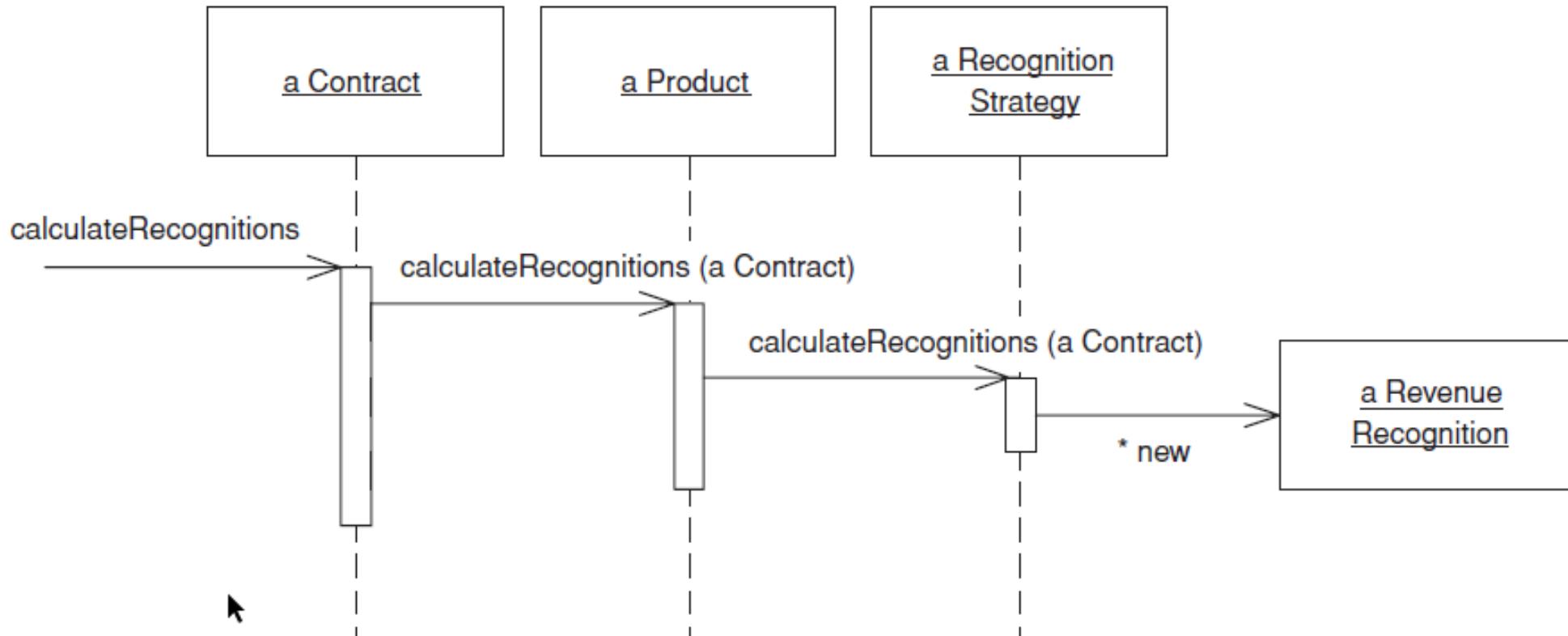
Con un Domain Model costruiamo un modello del nostro dominio che, almeno in prima approssimazione, è organizzato principalmente attorno ai «termini individuati nel dominio stesso» (inteso proprio come linguaggio utilizzato).

L'utilizzo di un Domain Model in opposizione a uno Transaction Script è l'essenza del cambiamento di paradigma ovvero il passaggio alla OOP pura.

Piuttosto che una routine con tutta la logica per un'azione dell'utente, ogni oggetto prende una parte della logica che è rilevante per esso.

Se non si è abituati a un Domain Model, l'inizio può essere molto frustrante (in pratica si passa da un oggetto all'altro cercando di scoprire dove si trova il «comportamento»).

Domain Model



In un **Transaction Script** il metodo implementato fa tutto il lavoro e gli oggetti sottostanti sono solo «Table Data Gateway» e tutto ciò che fanno è passare i dati allo script di transazione.

Al contrario, nel **Domain Model** più oggetti interagiscono, ciascuno inoltrando parte del comportamento a un altro finché un oggetto strategico non crea i risultati.

Il valore di un **Domain Model** sta nel fatto che una volta che si è abituati al modo di ragionare vi sono molte tecniche che permettono di gestire logiche sempre più complesse in modo ben organizzato.

Nel Domain Model mano a mano che definiamo algoritmi di calcolo aggiuntivi possiamo «tradurli» in codice aggiungendo nuovi oggetti della strategia di riconoscimento delle casistiche.

Con un Transaction Script invece andremmo ad aggiungere condizioni alla logica condizionale dello script.

Una volta che la finalmente la nostra mente verrà «deformata» dalla logica della OOP (quella vera) e dagli oggetti, scopriremo di preferire l'adozione di un Domain Model anche in casi abbastanza semplici.

I costi di implementazione di un Domain Model derivano dalla complessità del suo utilizzo e del livello dell'origine dati.

E' necessario tempo per digerire il Pattern affinché chi non conosce i modelli «pieni» di oggetti si abituino a un modello di dominio avanzato.

Tuttavia adottare questo Pattern ci aiuta a modellare meglio la struttura dei nostri oggetti secondo un paradigma chiaro e ben strutturato.

Ricordiamoci che però, una volta effettuato il passaggio, dobbiamo ancora occuparci della mappatura del database.

Più è complesso il modello di dominio tanto più complessa è la mappatura di un database relazionale (di solito si utilizza un Pattern Data Mapper); in questo approccio la mappatura può risultare più «costosa» rispetto ad altri pattern, ma se ne guadagna «dopo» in salute 😊.

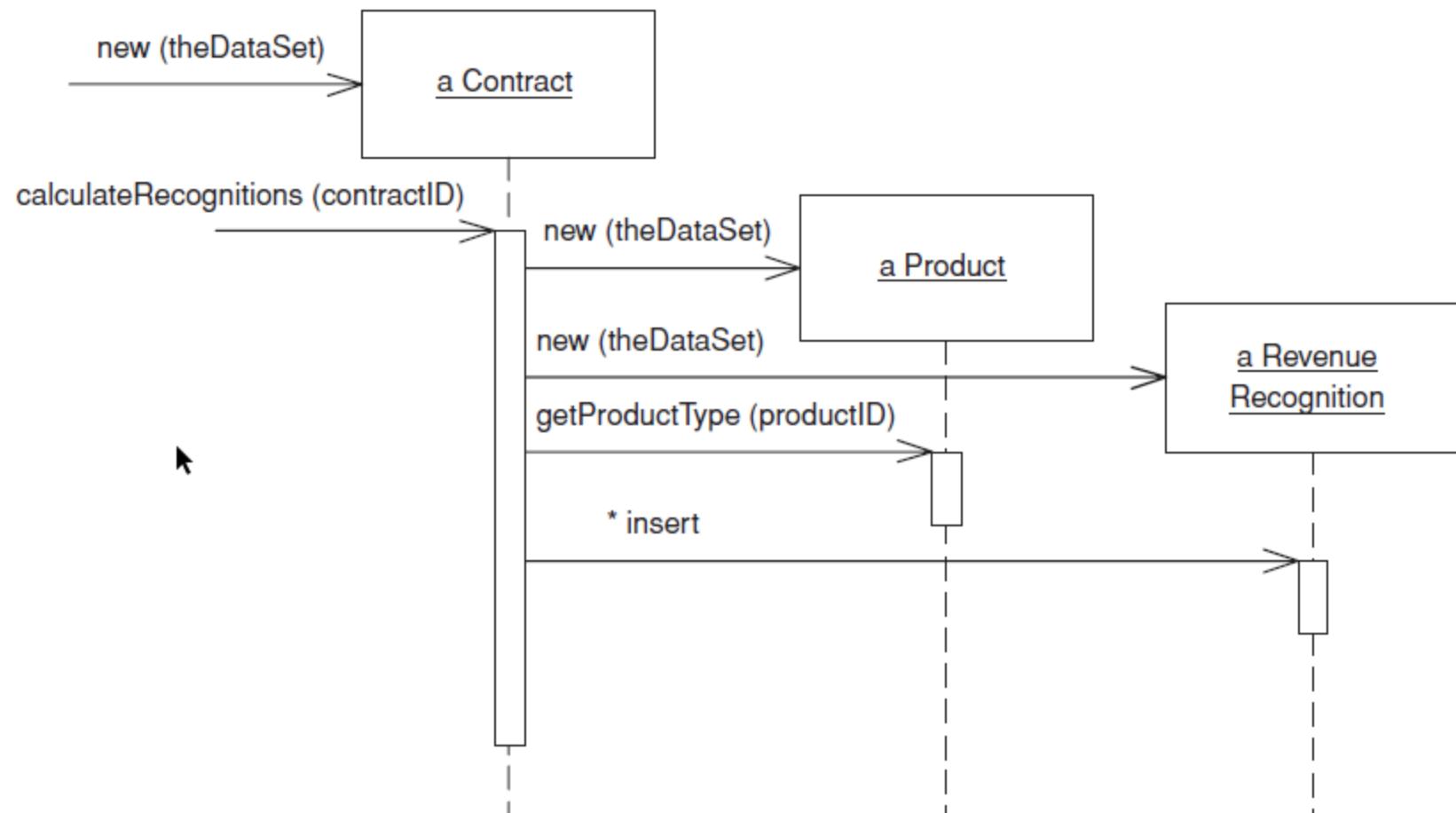
C'è una terza scelta per strutturare la logica del dominio, ovvero il Table Module.

A prima vista il Table Module sembra simile a un Domain Model poiché entrambi prevedono classi simili.

La differenza fondamentale è che un Domain Model prevede un'istanza per ogni oggetto nel database mentre un Table Module una sola istanza.

Il Table Module è progettato per funzionare con un set di record.

Table Module



Pertanto, il client di un oggetto Table Module invierà prima delle query al database per costruire un Record Set e creerà un oggetto a cui passerà il Record Set come argomento.

Il client può quindi invocare operazioni sull'oggetto per eseguire varie operazioni e se vuole fare qualcosa per un oggetto individuale, deve trasmettere un ID.

Un Table Module è per molti versi una via di mezzo tra uno Transaction Script e un Domain Model.

L'organizzazione della logica del dominio attorno alle tabelle piuttosto che alle procedure semplici fornisce più struttura e semplifica la ricerca e la rimozione dei duplicati.

Tuttavia è più rigido per cui non è possibile utilizzare certe tecniche di programmazione di cui ci si avvale su di Domain Model al fine di ottenere una struttura più elegante della logica, come ereditarietà, strategie e altri pattern OO.

Il più grande vantaggio di un Table Module è come si inserisce nel resto dell'architettura.

Molti ambienti GUI sono progettati per lavorare sui risultati di una query SQL organizzata in un set di record.

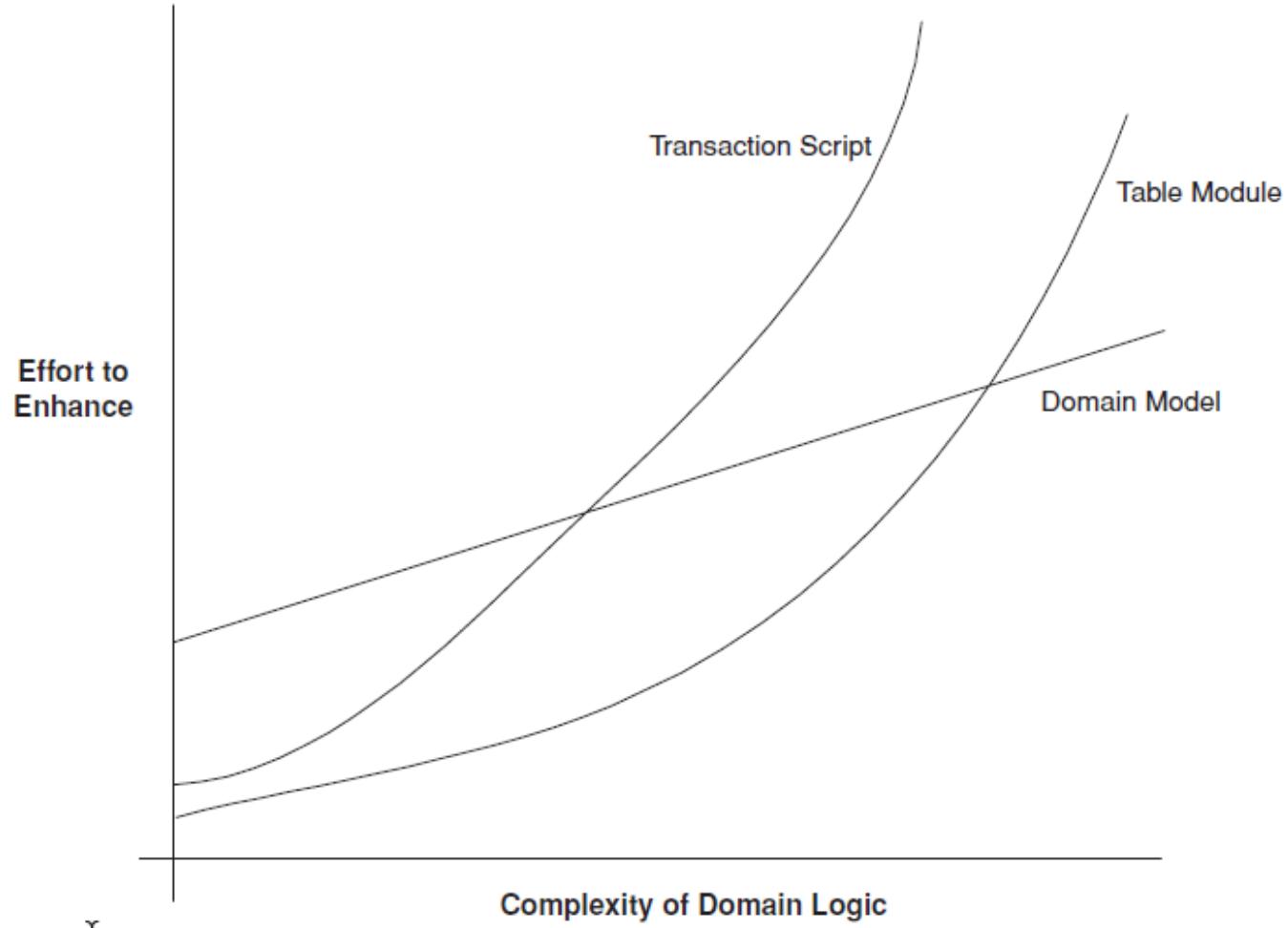
Poiché un Table Module funziona anche su un Record Set, è possibile eseguire facilmente una query, manipolare i risultati nel Table Module e passare i dati manipolati alla GUI per la visualizzazione.

È inoltre possibile utilizzare il Table Module sulla «risposta» alla GUI per ulteriori convalide e calcoli.

Quale scegliere tra i tre modelli ?

Non è una scelta facile e dipende molto dalla complessità della logica del dominio di riferimento.

Il grafico ci aiuta a rappresentare un confronto tra i tre Pattern.



Con una logica di dominio semplice, il Domain Model è meno interessante perché il costo per comprenderlo e la complessità dell'origine dati aggiungono un grande sforzo allo sviluppo che non verrà ripagato.

Tuttavia, con l'aumentare della complessità della logica di dominio, gli altri approcci tendono «a sbattere contro un muro» in cui l'aggiunta di più funzionalità diventa esponenzialmente più difficile.

Il problema, ovviamente, è comprendere dove si trova la «propria» applicazione sull'asse x.

La buona notizia è che possiamo affermare con certezza che dovremmo usare un Domain Model ogni volta che la complessità della logica di dominio è alta.

La cattiva notizia è che nessuno sa come misurare la complessità della logica di dominio.

In pratica, quindi, tutto ciò che si può fare è trovare delle persone esperte che possano fare una prima analisi dei requisiti e formulare un giudizio.

Vi sono alcuni fattori che alterano un pò le curve.

Un team che abbia familiarità con il modello di dominio ridurrà il costo iniziale anche se non lo abbasserà allo stesso punto di partenza degli altri a causa della complessità dell'origine dati.

Tuttavia, migliore è il team, più si deve essere inclini a utilizzare un modello di dominio.

Una volta presa la decisione questa non deve essere scolpita nella pietra, ma purtroppo è più difficile da cambiare.

Quindi vale la pena pensare in anticipo e decidere per quale pattern pendere.

Se si scopre di essere andato nella direzione sbagliata e si è ad esempio iniziato con Transaction Script, non bisogna esitare a eseguire il refactoring verso il Domain Model.

Se si è partiti con un Domain Model, passare a Transaction Script di solito è meno utile a meno che non si possa semplificare il livello di complessità dell'origine dati.

Questi tre modelli non sono scelte che si escludono a vicenda.

In effetti, è abbastanza comune utilizzare Transaction Script per parte della logica di dominio e Table Module o Domain Model per il resto.

Un approccio comune nella gestione della logica di dominio consiste nel dividere il livello in due.

Un «service Layer» viene posizionato su un Domain Model (o su un Table Module) sottostante.

Di solito il «service layer» viene usato solo con un Domain Model o un Table Module poiché un livello di dominio che utilizza solo Transaction Script non è abbastanza complesso da giustificare un livello di astrazione separato.

La logica di presentazione interagisce con il dominio esclusivamente attraverso il livello di servizio, che funge da API per l'applicazione.

Oltre a fornire un'API chiara, il livello di servizio è anche un buon punto in cui collocare strutture come il controllo e la sicurezza delle transazioni

Quando decidiamo di implementare un livello di servizio una decisione chiave è determinare quanto «comportamento» inserire in esso.

Vi sono diversi modi per implementare un «Service Layer».

Il caso minimo è quello di rendere il livello di servizio un «Facade» in modo che tutto il comportamento reale sia negli oggetti sottostanti e tutto ciò che il livello di servizio fa è il «passacarte» verso gli strati inferiori.

- In tal caso, il livello di servizio fornisce un'API più facile da usare perché è tipicamente orientata ai casi d'uso.**
- Ciò rende il servizio anche un punto conveniente per l'aggiunta di «wrapper transazionali e controlli di sicurezza».**

All'estremo opposto, la maggior parte della logica aziendale può essere collocata negli script di transazione all'interno del livello di servizio.

- Gli oggetti di dominio sottostanti sono molto semplici
- Se si tratta di un Domain Model sarà uno a uno con il database e sarà quindi possibile utilizzare un livello di origine dati più semplice come «Active Record».

A metà strada tra queste alternative c'è un mix più uniforme di comportamento: lo stile controller-entity.

Questo nome deriva da una pratica comune dove il punto è avere una logica specifica per una singola transazione o caso d'uso inserito nei transaction script (che sono comunemente indicati come controllers o services).

Si tratta di controller diversi dal controller dell' MVC (Model View Controller) o dell' «Application Controller» che incontreremo più avanti motivo per cui, per distinguerli, li chiameremo «use-case controller».

Il «comportamento» utilizzato in più di un caso d'uso si applica agli oggetti del dominio, chiamati entità.

Sebbene l'approccio «controller-entity» sia comune, gli «use-case controller», come qualsiasi Transaction Script tendono a incoraggiare il codice duplicato.

Il consiglio in generale è che, se si decide di utilizzare un logica a Dominio il Domain Model dovrebbe essere la scelta più giusta.

Ciò non significa che non bisogna mai avere services che contengano logica di business, ma che non è sempre necessario prevederli.

Gli oggetti di servizio procedurali a volte possono essere un modo molto utile per fattorizzare la logica, ma è bene tendere ad utilizzarli quando necessario piuttosto che come un layer architetturale predefinito.

Tendenzialmente, quindi, è preferibile avere un «Service Layer» quanto più «piccolo» possibile e soprattutto solo se vi è un effettivo bisogno.

Tuttavia molti bravi designer usano sempre un Service Layer unito a un bel po' di logica per cui, come architect, setiamoci comunque liberi di prende le nostre decisioni.

Service Layer

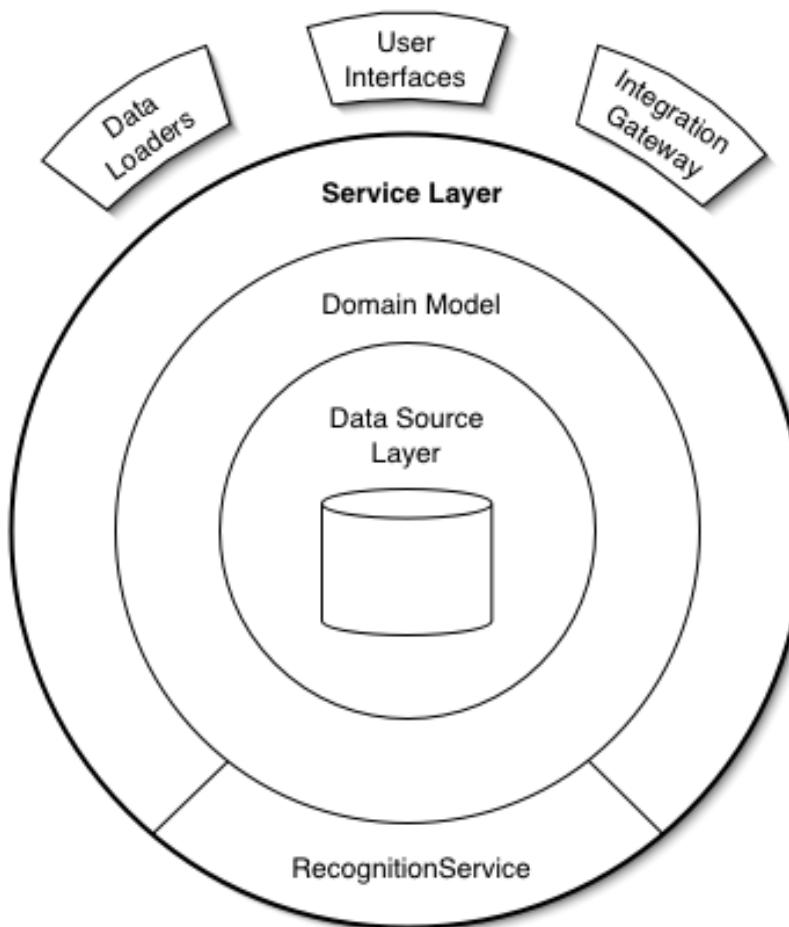
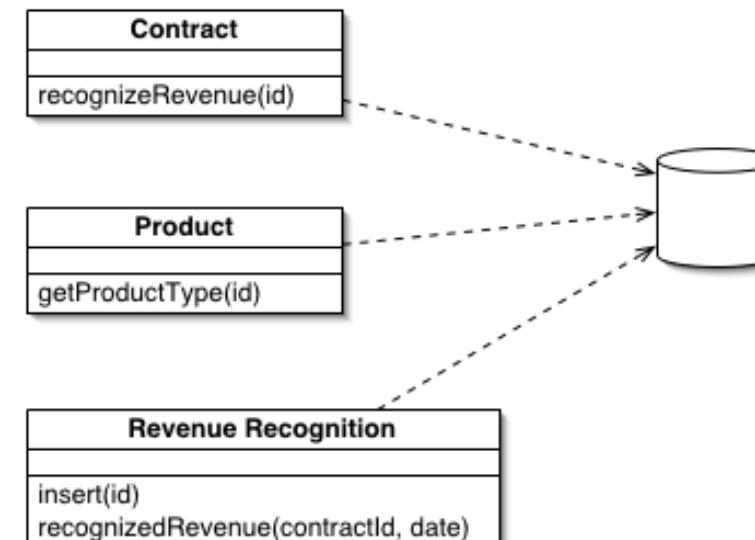


Table Module



Transaction Script

```
recognizedRevenue(contractNumber: long, asOf: Date) : Money  
calculateRevenueRecognitions(contractNumber long) : void
```

Domain Model

