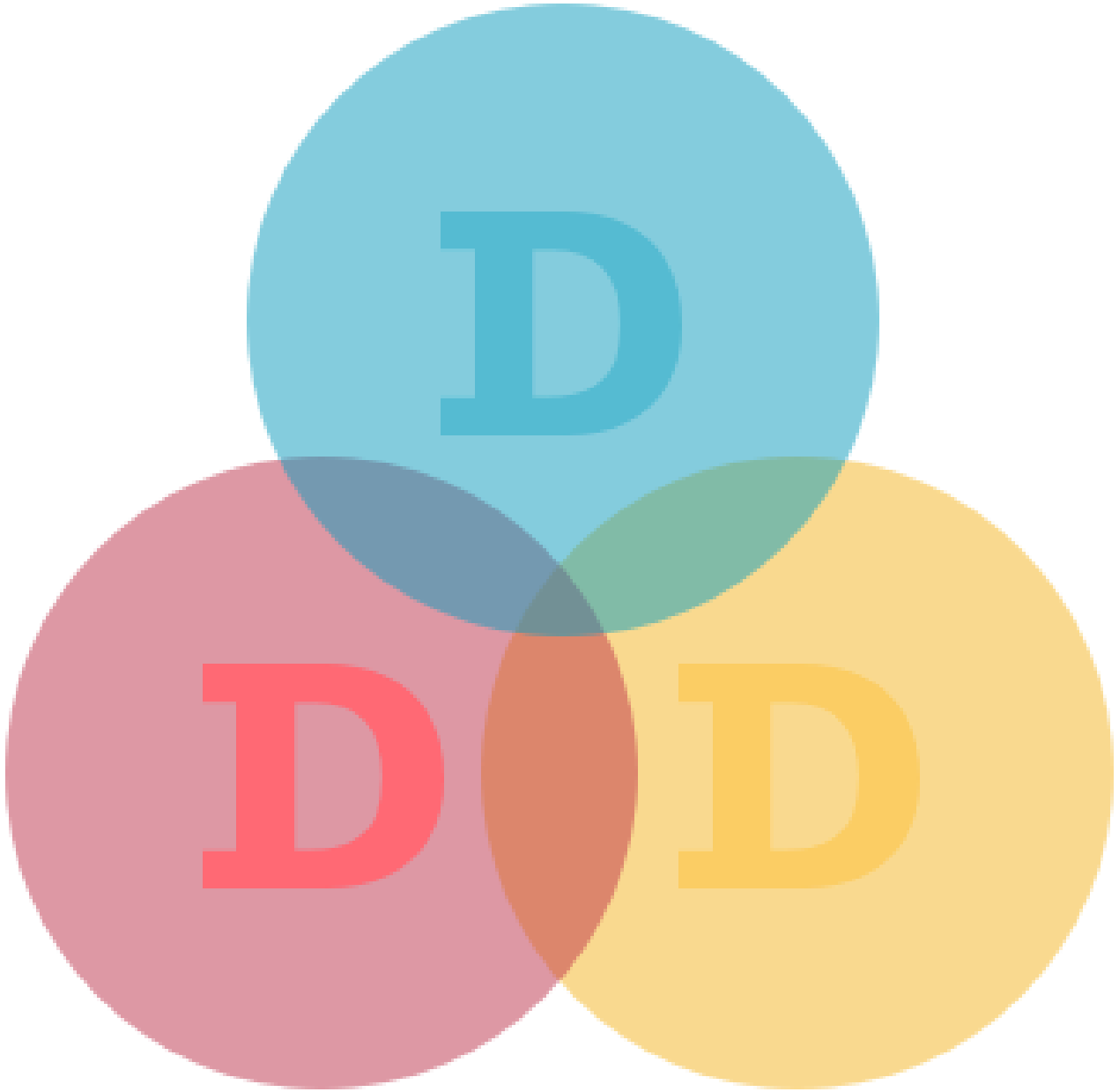

D
D
D

Cos’è il Domain Driven Design

Cos'è Domain Driven Design

Probabilmente la difficoltà sta nel definire cosa sia effettivamente «Domain Driven Design».

La definizione ufficiale è «it is a way of thinking and a set of priorities, aimed at accelerating software projects that have to deal with complicated domains» (è un modo di pensare e un insieme di priorità, finalizzato ad accelerare i progetti software che hanno a che fare con domini complicati)

Definizione che è in qualche modo sfuggente, ma in realtà possiamo dire tante cose su DDD, tutte parzialmente corrette, tutte probabilmente non soddisfacenti, ma che nel loro insieme possono darci un quadro più completo.

DDD è una metodologia ?

In realtà non è una vera e propria metodologia.

Definisce alcune pratiche di supporto allo sviluppo software, inserendosi all'interno di metodologie software collaudate quali i processi agili.

DDD è un insieme di pattern ?

In realtà è qualcosa di più.

DDD definisce un insieme di pattern astratti che normano e rendono metodiche le operazioni di realizzazione di un modello di dominio.

DDD è una tecnica di gestione della complessità?

Permette di fare a meno della complessità non necessaria.

Offre strumenti per una gestione consistente e scalabile della complessità intrinseca alla nostra applicazione.

DDD è una strategia di gestione progetti ?

Si tratta di tecniche derivate da progetti reali, con i vincoli classici che forzatamente ci costringono a compromessi.

DDD permette di compiere le scelte più sensate e di fare compromessi dove sensato, mantenendo l'integrità dove necessario.

DDD è un insieme di strumenti utili ?

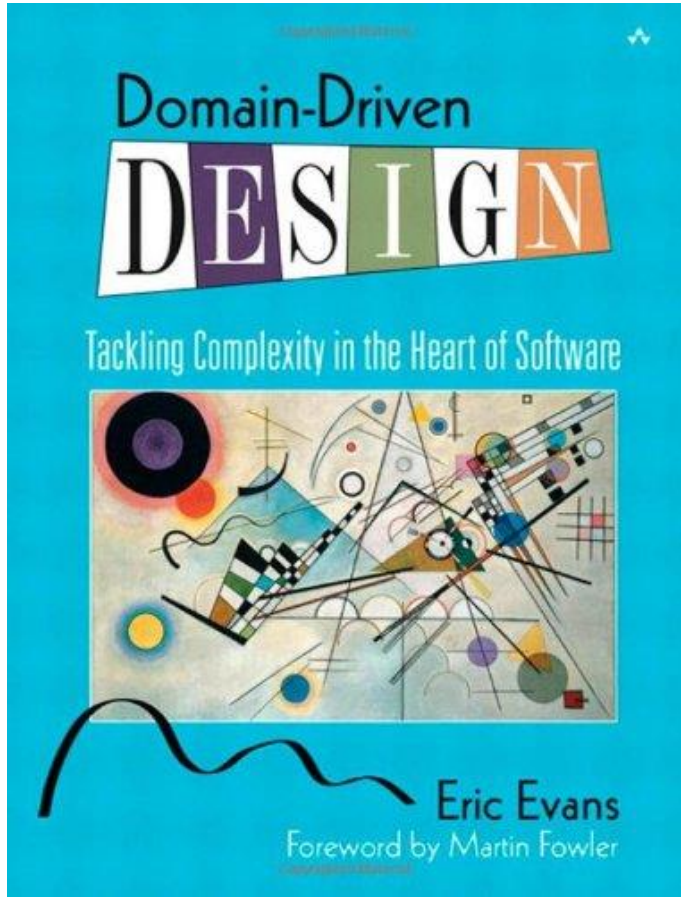
Da modi di pensare.

Da approcci a particolari tipologie di problemi

Definisce tecniche specifiche da adottare in determinate fasi del progetto.

In definitiva cosa è DDD ?

In definitiva possiamo definire «Domain Driven Design» come «un approccio alla realizzazione del software pragmatico, consistente e scalabile» anche in presenza di complessità crescente del dominio applicativo.

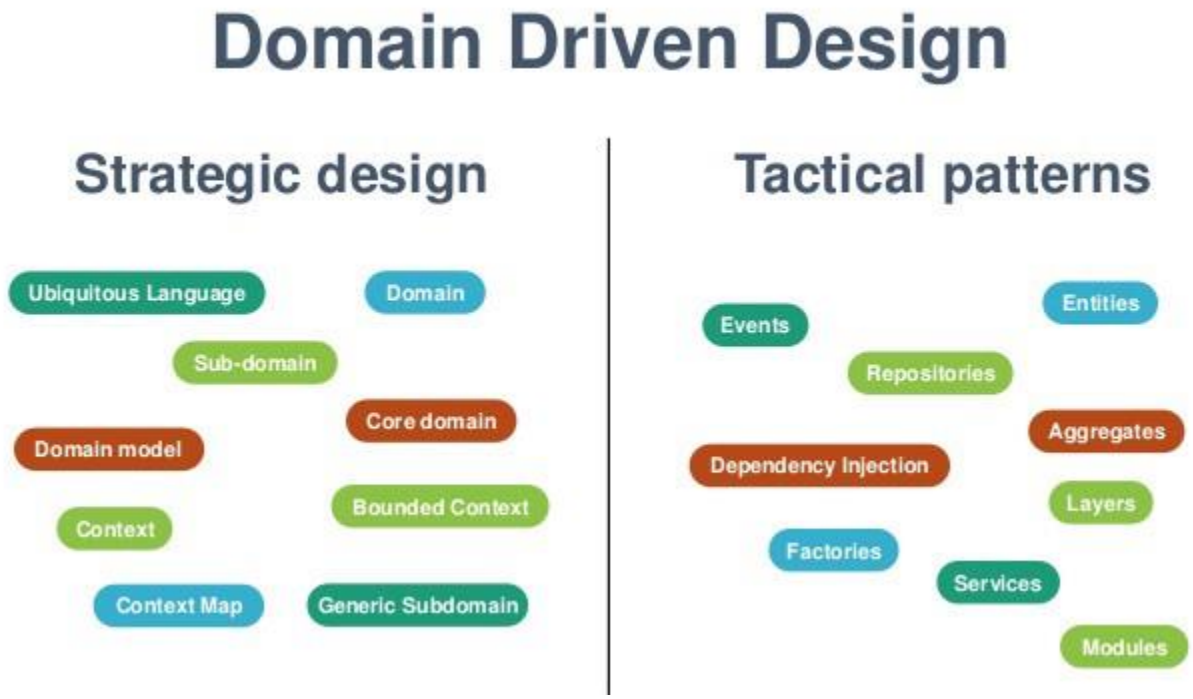


Domain Model

Il Domain Model Pattern

Da un punto di vista architetturale, DDD assume che la nostra applicazione sia realizzata sfruttando il «Domain Model Pattern».

DMP è un insieme di oggetti (generalmente localizzati in uno specifico layer della nostra applicazione) che rappresenti il modello del nostro dominio applicativo mediante una combinazione di dati e comportamenti caratteristico della «business logic» della nostra applicazione.



Il Domain Model Pattern

In applicazioni particolarmente semplici è possibile che il nostro modello di dominio abbia una corrispondenza quasi 1 a 1 con le tabelle del nostro database.

In scenari più complessi si ricorre a strategie di mapping più complesse tra il data layer e il domain model, o a specifici tool per l'object-relational mapping (ORM) che garantiscano la necessaria flessiblità.

All'interno del domain model vigono le leggi dell'OOP: c'è una forte attenzione all'attribuzione delle responsabilità fra le varie classi e alla corrispondenza del modello con il dominio sottostante.

Il Domain Model Pattern

In definitiva è necessario che il domain model sia una componente specifica della nostra applicazione perché:

- **rappresenta una delle componenti a maggior valore aggiunto della nostra applicazione**
- **ha un aspettativa di vita indipendente dalla tecnologia circostante**
- **è un area in cui le modifiche sono frequenti in risposta a specifiche esigenze del business.**

La realizzazione di un buon domain model è comunque un'operazione non a costo zero in quanto presuppone la presenza all'interno del team di sviluppatori-designer-analisti con buoni skills nel campo della OOP.

Anemic Domain Model

Anemic Domain Model

Anni e anni di **framework** per le varie piattaforme di sviluppo hanno distorto il ruolo del «**Domain Layer**» trasformandolo in quello che viene correntemente definito come «**Anemic Domain Model**», in cui le nostre classi di dominio:

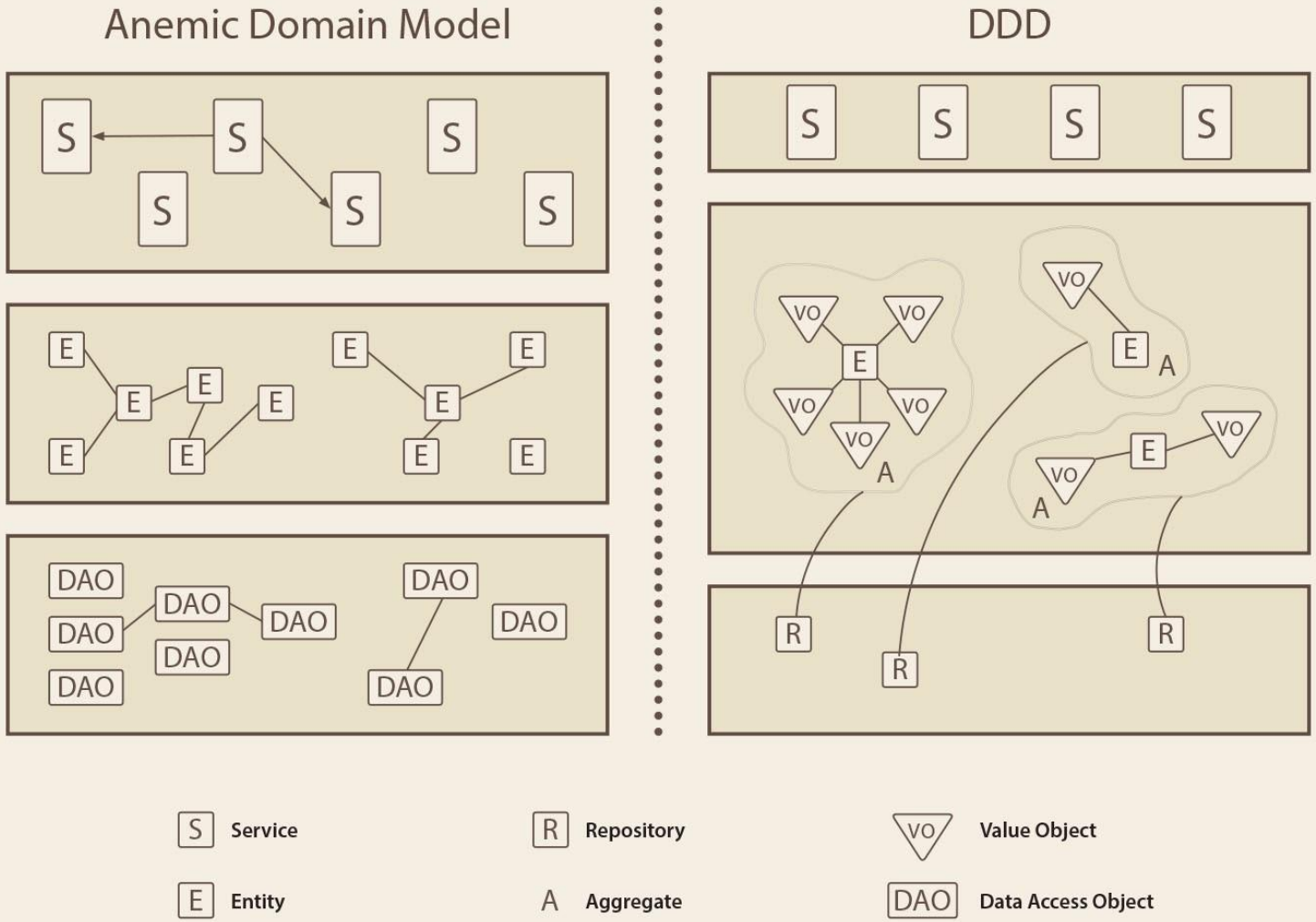
- sono la **proiezione** 1 a 1 delle tabelle sul nostro **DBMS**
- non espongono **metodi significativi** se non una collezione di getter e setter
- possono addirittura venire «generate» da tool di reverse engineering.

Anemic Domain Model

In questo scenario, molto spesso la «**business logic**» è localizzata in classi che fungono da «**coordinatori di oggetti passivi**», il cui stato è modificato agendo sui setter (**con buona pace dell'incapsulamento e dell'integrità delle nostre classi**) in metodi «**non esattamente mantenibili**».

Il verificarsi delle tre condizioni precedenti equivale a dire «**la programmazione ad oggetti è morta, tornate alle vostre case**».

Anemic Domain Model



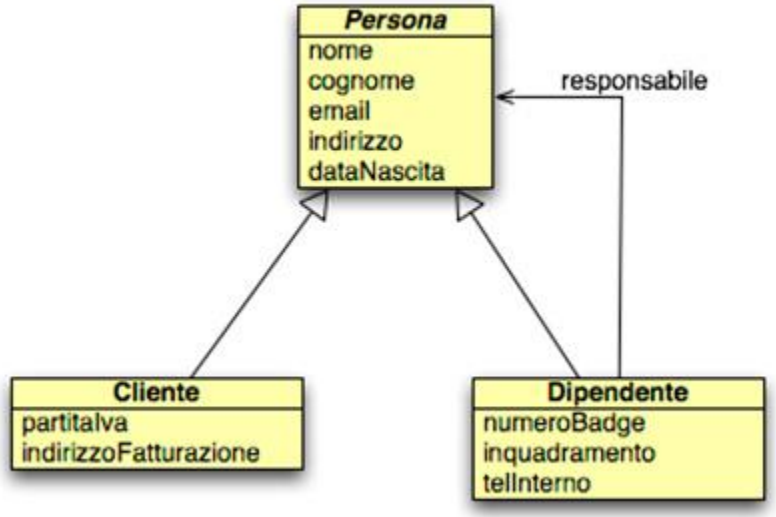
Ereditarietà e classi astratte

Ereditarietà e classi astratte

Spesso la presenza di gerarchie di classi non è di per se' una condizione sufficiente a poter definire il nostro modello un vero e proprio Domain Model.

Nella maggior parte dei casi si tratta solo di un modo per «risparmiare» nella definizione dei campi comuni, finendo per complicare inutilmente la realizzazione del mapping con ORM, senza aggiungere reale valore all'applicazione.

Una tipica applicazione dell'ereditarietà che permette di risparmiare la definizione di alcuni attributi nelle classi Cliente e Dipendente è:



Ereditarietà e classi astratte

In realtà, la caratteristica che giustifica l'uso dell'**ereditarietà** all'interno di un domain model **è l'effettiva presenza di comportamenti polimorfici**, ovvero azioni compiute dagli oggetti di dominio che necessitino di un'implementazione specifica per le diverse classi.

Se analizziamo le ultime applicazioni realizzate (senza DDD), ci accorgiamo che questo genere di operazioni non è implementato nelle classi di dominio, ma in una classe che si chiama ...**Controller** o ...**Manager** o (peggio) in quella classe da 50.000 righe di codice (definita ironicamente la classe di DIO) che fa praticamente **tutto quello che c'è da fare nel nostro sistema** (è uno sporco lavoro, ma qualcuno deve pur farlo) lasciando alle altre classi le briciole.

Ereditarietà e classi astratte

A ben guardare c'è qualcosa di perverso in tutto ciò: la programmazione a oggetti, come era stata concepita, prometteva la realizzazione di sistemi modulari e di facile manutenzione.

Purtroppo le implementazioni tipiche finiscono invece per essere sostanzialmente procedurali.

La complessità, i costi e la «pesantezza architetturale» sono gli stessi di sistemi di grandi ambizioni, ma le prestazioni e la manutenibilità spesso non sono adeguati alle aspettative.

Ereditarietà e classi astratte

Col tempo, abbiamo finito per **ridimensionare le aspettative**, imparando ad accontentarci, e reprimendo quella vocina che ci ricordava come erano carine le prime realizzazioni OOP fatte all'università.

DDD invece afferma esattamente il contrario: **la realizzazione di un buon modello di dominio è la chiave per realizzare buone applicazioni software, ma soprattutto applicazioni che «servano» e soprattutto OOP compliant**.

Perchè un modello?

Perchè un modello?

La realizzazione di un modello è importante in svariate discipline.

È necessario per semplificare e rendere efficaci determinate operazioni.

Il modello è uno strumento estremamente potente, funzionale a uno scopo, o meglio all'uso che verrà fatto della nostra applicazione.

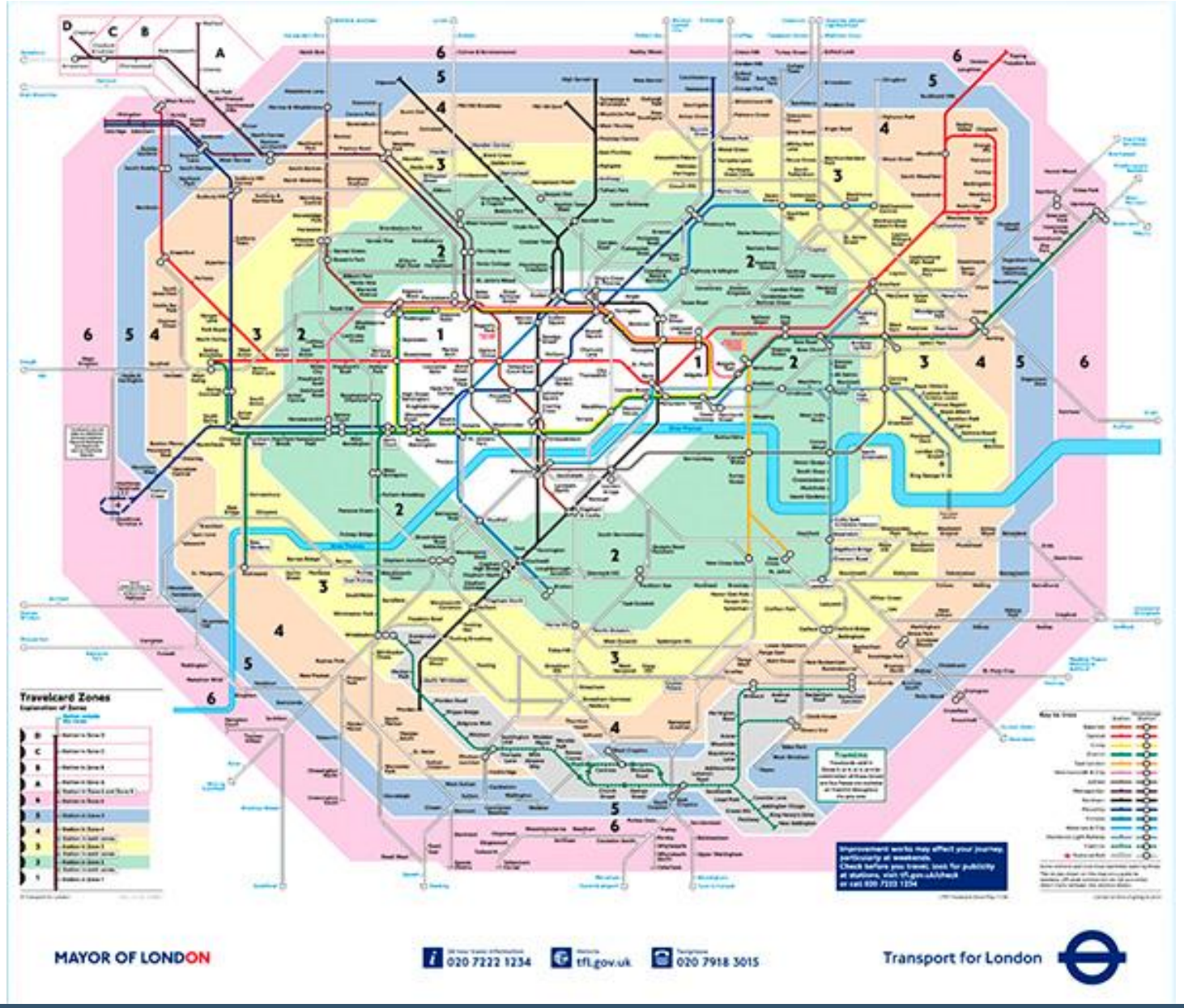
Lo scopo della nostra applicazione

Il punto centrale delle operazioni di **modellazione** è rappresentato dall'individuazione di uno scopo.

La stessa **entità concreta** può essere rappresentata in molti modi diversi, l'utilità di un modello rispetto a un altro è funzione dello scopo.

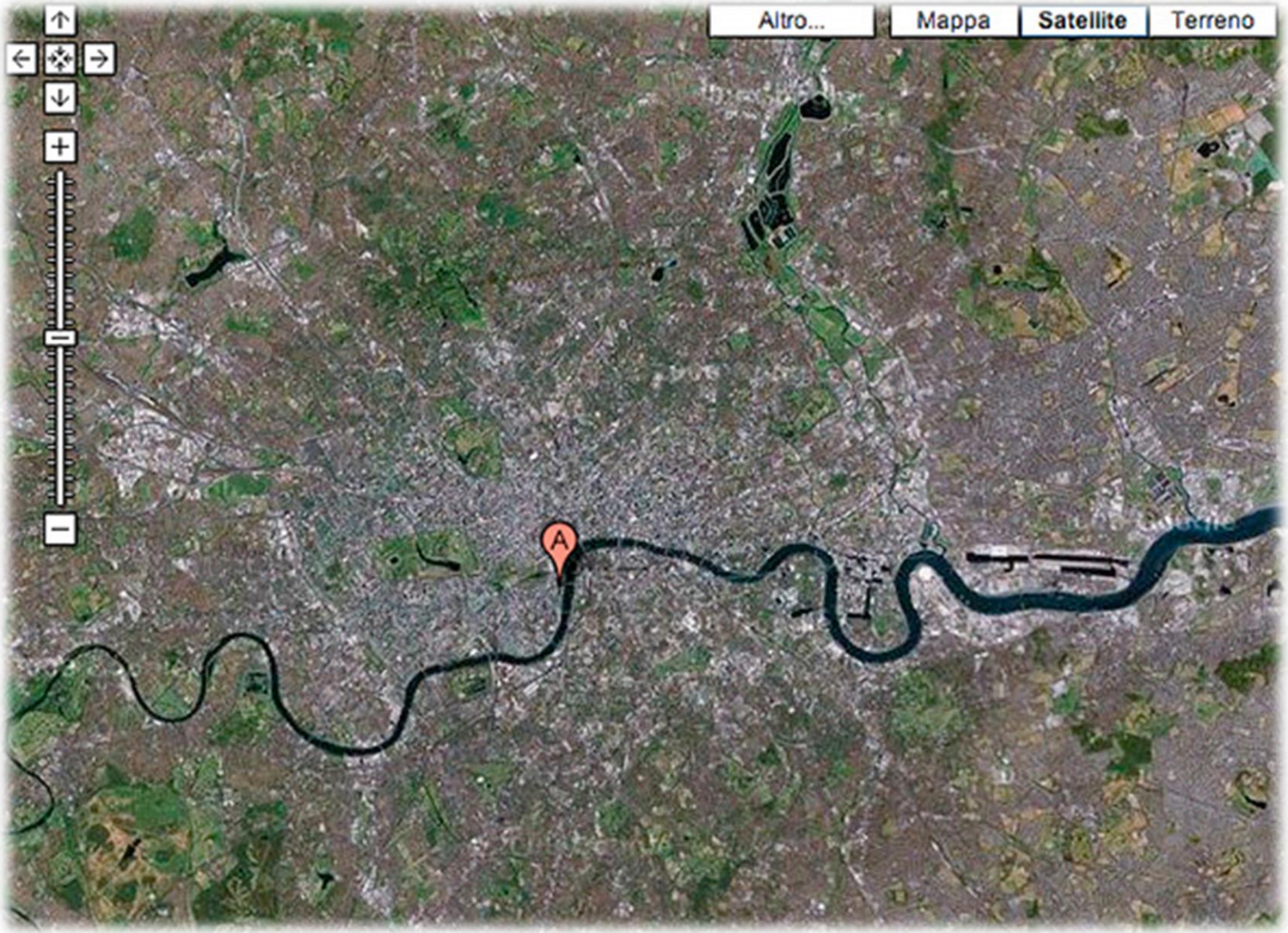
Prendiamo ad esempio le successive rappresentazioni dello stesso dominio: la mappa della città di Londra.

Londra: Rappresentazione 1



Londra: Rappresentazione 2

Londra: Rappresentazione 3



Il dominio dell'applicazione

Il dominio cui facciamo riferimento è esattamente lo stesso.

È lo scopo della nostra applicazione rendere più adatto un modello rispetto ad un altro.

Per un turista, la prima mappa è decisamente più utile: contiene tutte le informazioni necessarie a spostarsi in metropolitana da un punto ad un altro, sia pure tralasciando un sacco di dettagli.

Per un pilota di aereo che volesse sorvolare la città, la terza rappresentazione (quella con le immagini satellitari) sarebbe ideale, mentre la prima sarebbe sostanzialmente inutile.

Il dominio dell'applicazione

È interessante notare come la prima rappresentazione sia largamente imprecisa: le linee non sono realmente rette oppure ortogonali, le distanze non sono rappresentate fedelmente e così via.

Rispetto al vero, alcuni modelli risultano più accurati e precisi, eppure possono essere anche meno adatti all'uso.

Il nostro obiettivo sarà di individuare e costruire un modello che sia funzionale all'uso che viene fatto della nostra applicazione.

I confini del dominio

L'altro **elemento chiave** per la costruzione di un modello utile è la corretta individuazione dei confini del dominio applicativo.

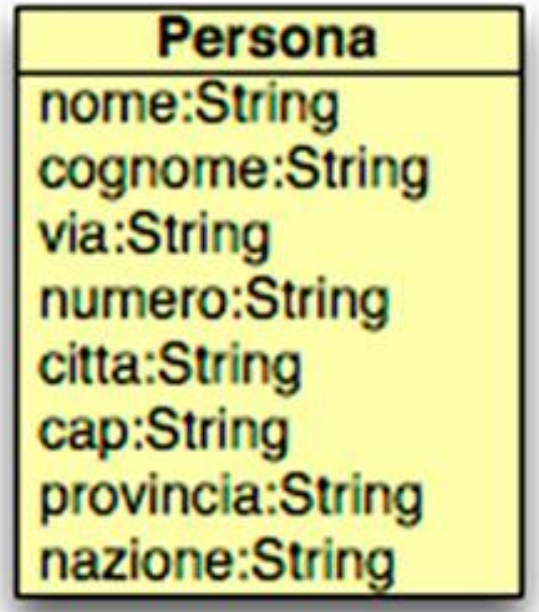
Nell'esempio precedente, abbiamo potuto tralasciare alcuni elementi di dettaglio, in quanto non funzionali allo scopo che ci prefiggevamo.

In generale, essere in grado di tracciare la linea che ci permette di **tenere fuori la complessità** è un'attività abbastanza cruciale durante le fasi di analisi.

I confini del dominio

Un esempio abbastanza classico è dato dalla gestione dell'indirizzo di un'anagrafica.

In un ottica «table driven» possiamo avere tutti i campi dell'indirizzo implementati come attributi della Persona.



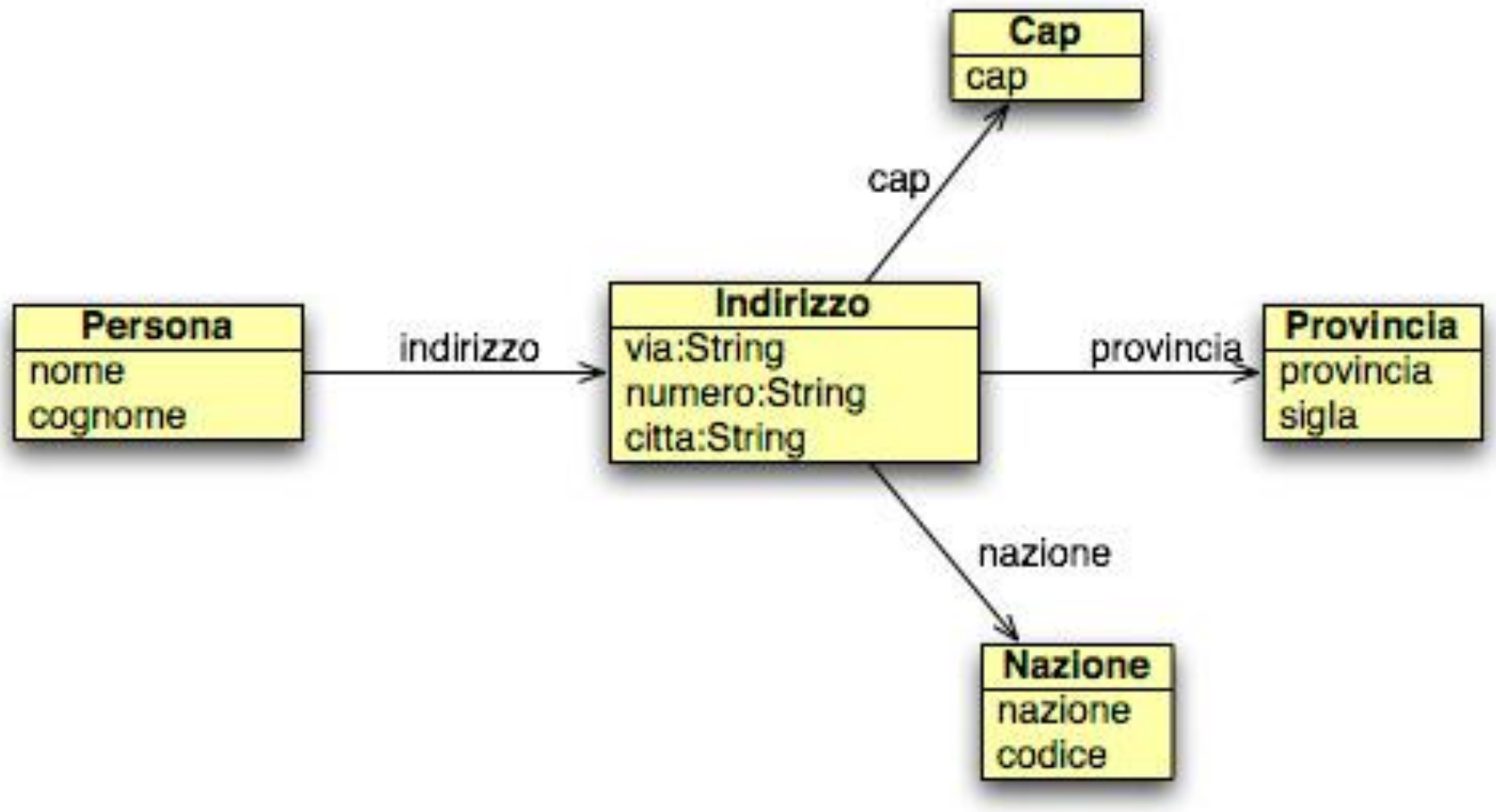
I confini del dominio

Tale rappresentazione ha chiaramente delle limitazioni: concettualmente un indirizzo e una persona non sono la stessa cosa, ma soprattutto questa scelta non permette a una persona di essere associata a più di un indirizzo.

Separando un po' le cose, magari nell'ottica di riuso di un componente grafico dedicato o di una tabella separata, oppure di alcune funzioni di validazione, possiamo separare l'indirizzo dalla classe che lo contiene:

I confini del dominio

Se invece vogliamo modellare l'Indirizzo in un ottica OOP più «estrema» allora potremo trovarci di fronte ad un modello più articolato, in cui diversi elementi del nostro modello sono trattati come classi.



I confini del dominio

Rispetto al modello con l'indirizzo separato, in quest'ultimo il data model può anche rimanere sostanzialmente invariato.

OK, abbiamo aggiunto un paio di attributi… ma non è questo il punto ☺.

La differente granularità può esserci utile se questa abilita dei comportamenti specifici da parte delle classi (validazione del CAP, gestione della visualizzazione della provincia, visualizzazione della bandiera della nazione, etc.).

I confini del dominio

Qual è il modello corretto ?

Non esiste, o meglio «...la domanda è mal posta».

La domanda corretta è:

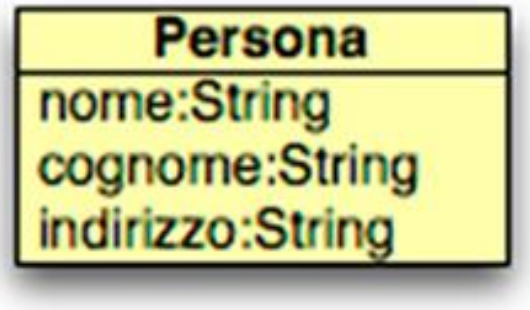
«qual è il modello più adatto al mio scopo ?»

Oppure:

«come sarà usata la mia applicazione ?»

I confini del dominio

Se lo scopo è solamente la stampa di un elenco indirizzi, e non la ricerca per aree geografiche, forse l'implementazione migliore è:



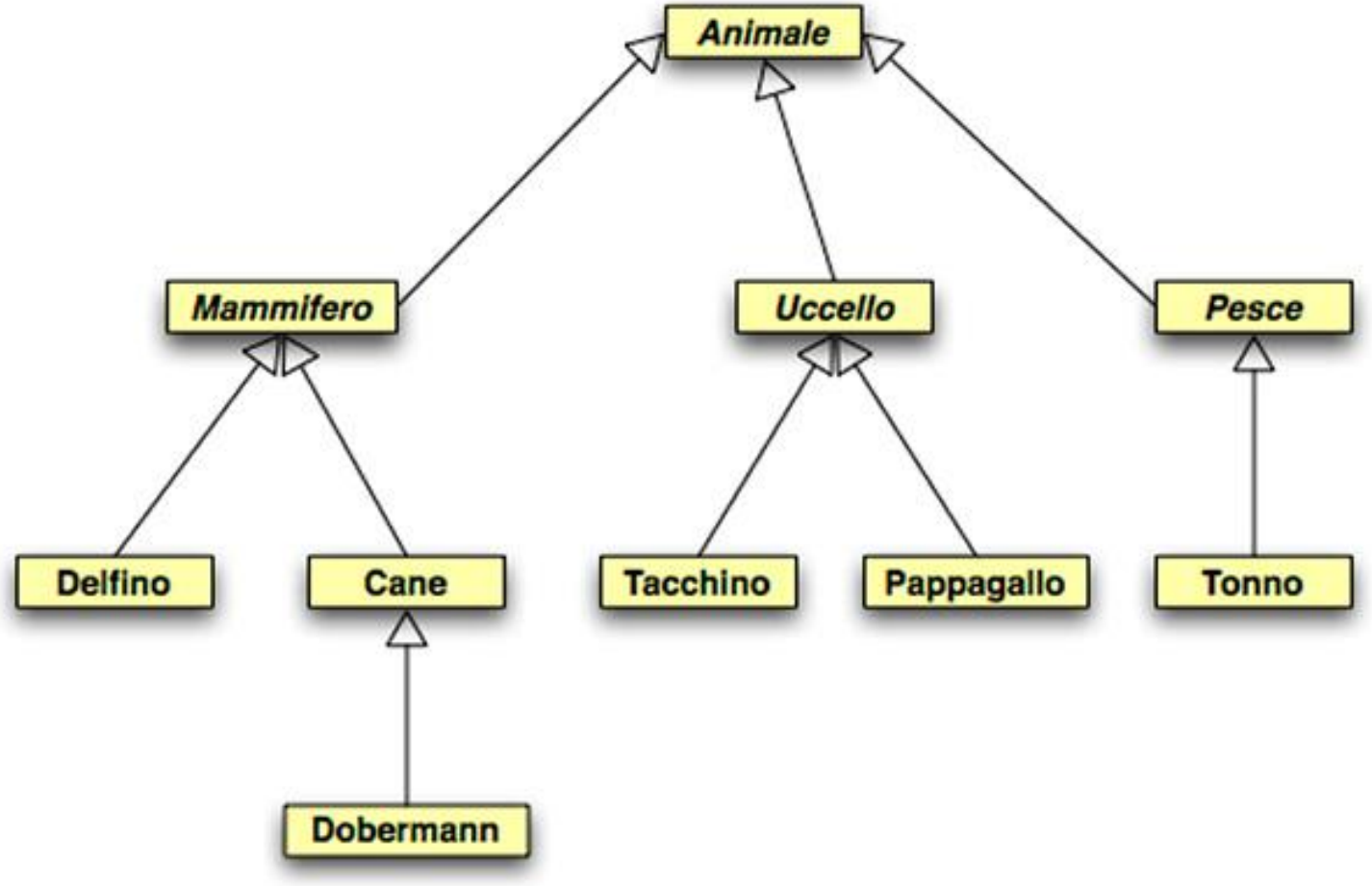
Apparentemente è blasfemo, ma può essere la soluzione più pratica da implementare.

Un'obiezione forte a una rappresentazione di questo genere è che è «concettualmente sbagliata».

Vero, ma non stiamo realizzando un «modello concettuale», bensì un «modello utile».

I confini del dominio

Un modello concettuale è spesso un retaggio degli esempi della OOP fatti su animali-mammiferi-uccelli-pesci (Una tassonomia, incautamente utilizzata per esemplificare le possibilità di OOP)



I confini del dominio

In realtà, problemi di mapping su un DB relazionale a parte, un modello di questo genere è corretto da un punto di vista concettuale, ma sostanzialmente inutile da un punto di vista applicativo.

La realtà che ci circonda è incredibilmente complessa, e per quanto possa essere forte la tentazione di catturarla brillantemente in una tassonomia a oggetti, il nostro compito è tenere fuori dalla nostra applicazione tutta la complessità che non è funzionale al nostro scopo, ossia all'uso che verrà fatto della nostra applicazione.

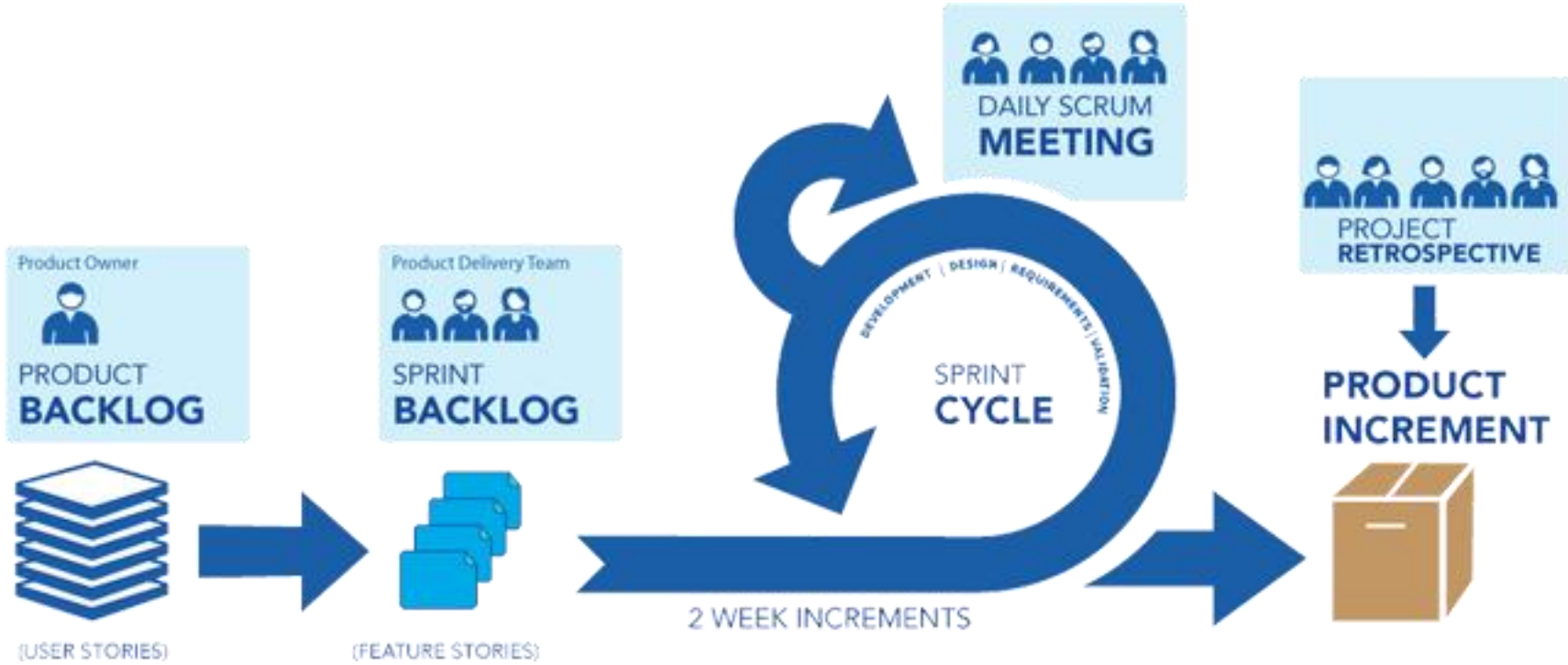
Abbiamo però un piccolo problema: il modo in cui la nostra applicazione viene usata non è una costante.

Le condizioni al contorno possono cambiare, come possono cambiare molte altre cose durante le fasi di sviluppo.

L'Ecosistema per il DDD

L'Ecosistema per il DDD

Per poter applicare con successo i principi di «Domain Driven Design» è necessario che siano verificate alcune condizioni, in genere coincidenti con l'ecosistema di un processo di sviluppo agile.



L'Ecosistema per il DDD

Il processo di sviluppo deve essere iterativo.

Le iterazioni frequenti permettono di raffinare il processo di apprendimento sulla base del feedback proveniente dagli esperti di dominio.

L'Ecosistema per il DDD

Gli esperti di dominio devono essere disponibili a supportare questo processo di apprendimento.

Deve esserci spazio per le «attività di refactoring» che permettano al software di evolversi sulla base delle informazioni acquisite nel corso delle iterazioni.

L'Ecosistema per il DDD

Deve esserci la possibilità di una «comunicazione efficiente» e la possibilità di «condividere efficacemente» informazioni tra tutti membri del team.

L'Ecosistema per il DDD

Troppo bello per essere vero? ..... forse, ma in determinati contesti queste condizioni esistono.

Sono semplicemente irrinunciabili per poter avere qualche speranza di successo.

Ciò è vero in tutti quei contesti in cui il dominio è complesso e dove il software deve risolvere problemi non banali.

In altre parole, sarà nostro compito cercare di fare in modo che queste condizioni siano presenti.

La comunicazione con l'esperto di dominio

La comunicazione con l'esperto di dominio

La realizzazione di un'applicazione software getta una sorta di ponte tra due mondi.

In partenza, l'analista e gli sviluppatori si trovano a essere padroni di un determinato paradigma (il linguaggio Java, UML, la programmazione a oggetti) e completamente a digiuno delle conoscenze specifiche del dominio sottostante all'applicazione.

Il nostro compito, sia come analisti che come sviluppatori è di «acquisire conoscenza» su uno specifico dominio e di trovarne una rappresentazione adeguata con gli strumenti che abbiamo a disposizione.

La comunicazione con l'esperto di dominio

Il tramite tra noi e la conoscenza del dominio è in genere rappresentato dagli esperti di dominio.

Queste persone - specialmente quelle veramente esperte - sono fondamentali per permetterci di costruire un modello coerente e utile della nostra applicazione.

È quindi importante che vi sia un canale aperto che permetta una comunicazione efficiente in entrambi i sensi.

Ubiquitous language

Un **prerequisito** abbastanza ovvio per cui possa avvenire una comunicazione di questo genere è «**parlare la stessa lingua**».

Non si tratta dell'italiano, così come non si tratta di UML o di un data model.

Si tratta di una lingua composta di termini il cui significato è **chiaro, condiviso e privo di ambiguità all'interno del dominio specifico**.

Soprattutto si tratta di termini adottati in maniera coerente, costante e uniforme sia nelle discussioni, che nella documentazione, che nel codice, da tutti i membri del team.

Ubiquitous language

Non si tratta di un vezzo, o semplicemente della necessità di mantenere un dizionario dei termini in uso all'interno del progetto.

Si tratta di uno strumento che rafforza e tutela l'integrità della visione del dominio da parte di tutti i partecipanti al progetto.

Si tratta dello strumento che permette al Domain Expert di convalidare o confutare il nostro grado di comprensione del dominio.

Si tratta anche di uno strumento di controllo: se il gruppo di sviluppo utilizza una terminologia non conforme alla descrizione del domain expert, o se introduce nuovi termini che non compaiono nelle conversazioni, avremo un campanello d'allarme che ci informa che l'implementazione sta divergendo rispetto al dominio originale, o che la nostra comprensione del dominio non è corretta.

Ubiquitous language

Ad esempio: pensiamo ad uno scenario in cui cerchiamo di tradurre esattamente un termine italiano in inglese ma purtroppo ci accorgiamo che non ne esista una traduzione specifica ma solo qualche termine che più gli si avvicina ma che ha un'accezione decisamente diversa.

Se il nostro linguaggio non permette di esprimere concetti chiave della nostra applicazione sarà necessario andare a esplorare questa discrepanza.

Una lingua rappresenta un dominio, scaturisce da un dominio, ma allo stesso tempo contribuisce a plasmarlo.

Nel nostro caso, l'«Ubiquitous Language» è il modello della nostra applicazione.

In generale tutti i membri del team dovrebbero essere in grado di descrivere il comportamento dell'applicazione utilizzando solo i termini caratteristici dello «ubiquitous language» eliminando qualsiasi ambiguità o incertezza.

Quale formalismo usare?

Quale formalismo usare?

In teoria lo strumento principe per la rappresentazione di un modello di un'applicazione ad oggetti dovrebbe essere **UML** (lo dice la sigla stessa: Unified Modeling Language).

Tuttavia la disponibilità di una notazione non deve farci perdere di vista l'obiettivo delle attività di analisi, ovvero la **costruzione di un modello** (non la stampa di un documento ben impaginato) **efficiente del dominio applicativo**.

Tale modello è il risultato di un processo di apprendimento che può essere efficace solo se il maestro (il nostro Domain Expert) è in grado di comprendere, ed eventualmente correggere, i nostri semilavorati.

Se il formalismo non è comprensibile all'esperto di dominio, o, peggio, se è accessibile solo mediante un tool (non faccio nomi, ...ma avete capito), la situazione si **complica** inutilmente.

Quale formalismo usare?

È quindi necessaria una grande attenzione, per evitare di «tagliare fuori» i ruoli che non sono così avvezzi all'uso di UML (la presenza di omini e ovali ha statisticamente un effetto nefasto).

Possiamo limitarci a un sottoinsieme della notazione UML che possa essere condiviso con i domain expert.

Possiamo limitare, o addirittura bandire, l'uso di tool per la modellazione (hanno lo sgradevole side-effect di trasformare l'analisi da una discussione a un'attività «solista» di impaginazione e layout) e svolgere tutte le attività di analisi alla lavagna o su carta.

Possiamo impratichirci della notazione dei domain expert quando in determinati domini è già presente un «Domain Specific Language»; possiamo addirittura definirne una se lo riteniamo necessario.

Le strategie sono molteplici e vanno valutate caso per caso.

Quale formalismo usare?

Possiamo addirittura definirne una, se necessario per cui le strategie sono molteplici, e spesso da valutare caso per caso.

Non cambia il punto chiave: è necessario che la comunicazione con il Domain Expert sia la più aperta ed efficace possibile.

Perche' è un elemento cruciale per una corretta comprensione del dominio applicativo.

Qualsiasi ostacolo a una discussione aperta e bidirezionale va rimosso.


Il codice è il modello

Il codice è il modello

In un progetto con un tempo di vita non banale, la documentazione è fatalmente destinata all'obsolescenza.

Solo in processi estremamente strutturati (e di dubbia efficienza) abbiamo la totale certezza che la documentazione sia sempre sincronizzata con il codice.

Da questo punto di vista, DDD riecheggia l'approccio alla documentazione proprio delle metodologie agili: c'è un unico artifact di progetto che è sicuramente sincronizzato con il comportamento del sistema ed è il codice.

Il codice è il modello

La **documentazione** può essere necessaria a svolgere un ruolo specifico in una determinata fase del progetto, ma se consideriamo l'intera vita della nostra applicazione è chiaro che da un certo punto in poi, documentazione e codice fatalmente divergeranno.

Il punto non è «**fare a meno della documentazione**»: il livello di documentazione è un constraint di progetto che DDD non mette in discussione.

Il punto è **mantenere il codice costantemente allineato con la nostra comprensione del modello**, utilizzando la stessa terminologia (lo Ubiquitous Language) usata da analisti e domain expert e facendo evolvere il nostro codice alla luce delle scoperte che faremo man mano.

La nostra comprensione del dominio

La nostra comprensione del dominio

Da uno scenario di questo genere emerge anche un'altra informazione importante: la nostra comprensione del dominio applicativo è una variabile durante il ciclo di vita del progetto.

Inizialmente sarà decisamente bassa (un ottimo motivo per limitare il numero di scelte da compiere in questa fase) per crescere man mano che il progetto avanza.

Anche in presenza di requisiti stabili (caso comunque raro), il modello si evolve per l'aumentata comprensione del dominio da parte del team.

La nostra comprensione del dominio

Questo processo di apprendimento non è incrementale: i reali avanzamenti avvengono quando ci rendiamo conto che il modello esistente è sbagliato:

- Perchè non ha tenuto conto di un requisito non ancora completamente emerso,
- Perchè un concetto che ci sembrava oscuro all'inizio è diventato finalmente chiaro quando osservato da un'altra angolazione
- Perchè via via che l'implementazione avanza il quadro completo si fa più chiaro e così via.

A questo punto siamo di fronte ad un bivio: correggere anche le funzionalità già implementate alla luce delle nuove informazioni oppure «trovare il modo» per salvare capra e cavoli ?

Domain Driven Design è molto netta su questo punto: il codice deve essere la rappresentazione della nostra attuale comprensione del dominio.

La nostra comprensione del dominio

Una nuova scoperta deve riflettersi nel codice, per cui vogliamo andare a modificare anche il codice già esistente affinché il modello rifletta la nostra comprensione del dominio.

In caso contrario abbiamo introdotto un gap tra il codice e il modello, non documentato, che andrà a minare la nostra capacità di fare evolvere adeguatamente il sistema.

Si tratta in effetti di un concetto molto simile al «Technical Debt» ma con la particolarità di essere legato alla consistenza del modello più che a un difetto implementativo.

Esplorazione delle soluzioni

Una tecnica tipica di DDD, da applicare quando ci troviamo di fronte ad una di queste «rivelazioni sconvolgenti» è una sessione di «codice esplorativo».

- **Abbiamo scoperto che una determinata classe non ha senso ?**
- **Che in realtà questi attributi e queste responsabilità sono di pertinenza di un'altra entità del nostro sistema ?**
- **Magari di una che non avevamo ancora definito ?**
- **Che dobbiamo «tirare via» un'impalcatura che avevamo faticosamente (parola che avrebbe forse dovuto metterci sul chi vive...) costruito nelle iterazioni precedenti?**

Bene ..... mano al codice, facciamo le modifiche e vediamo che succede !

Esplorazione delle soluzioni

In generale le operazioni di «refactoring» possono essere complicate, ma spesso lo sono molto meno delle nostre previsioni: nella maggior parte dei casi stiamo togliendo complessità che si è rivelata non necessaria.

In ogni caso, per poter operare queste piccole «rivoluzioni» con tranquillità è necessario poter disporre di un dispositivo di sicurezza: i test automatizzati, che ci permettono di «ravanare» anche dentro le features già consolidate senza dover incrociare le dita, ma con la consapevolezza di sapere quello che stiamo facendo e le conseguenze che questo comporta.

La presenza di dati legati al modello precedente può rivelarsi un ostacolo all'applicazione del refactoring a funzionalità già in produzione, ma è bene verificare anche queste assunzioni per evitare che un ostacolo si trasformi in un alibi.

Del resto il refactoring può essere applicato anche sul DB, basta volerlo.

Prime Conclusioni

Conclusioni

Fino ad ora abbiamo essenzialmente studiato quali sono i principi guida di Domain Driven Design, ma abbiamo aperto anche una serie di problemi che dovranno essere approfonditi.

In particolare dovremo verificare come applicare praticamente i principi DDD nella realizzazione di un modello di una singola applicazione, e come arrivare a una implementazione.

Alcuni dei principi enunciati esposti sono molto rigorosi, e di difficile applicabilità: vedremo come questo rigore non è sempre necessario e come in realtà DDD definisca delle strategie estremamente pragmatiche.

Dovremo inoltre verificare come e quanto le strategie definite per un singolo dominio possano scalare al crescere della complessità e del numero di interazioni con altre applicazioni o altre porzioni della nostra applicazione.