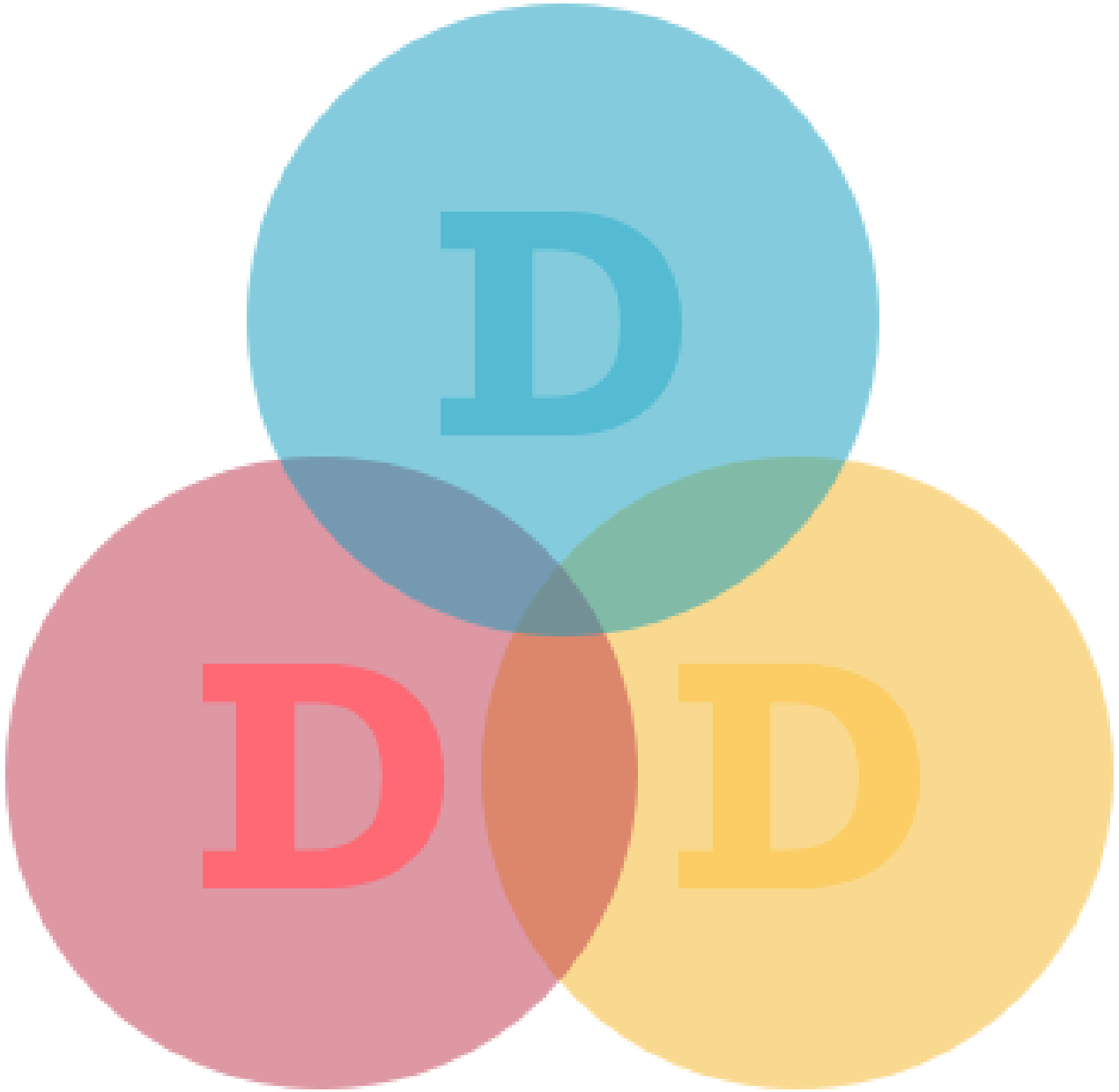

D
D
D

Primi passi nel Domain Model

Cos'è Domain Driven Design

Dopo aver introdotto i concetti generali del **Domain Driven Design**, cominciamo ad analizzare gli aspetti fondanti di questo «nuovo» approccio alla progettazione del software, guardando da vicino alcuni concetti relativi all'architettura e descrivendo qualche pattern.

Abbiamo introdotto alcuni elementi fondamentali di Domain Driven Design , ora vediamo come metterli in pratica in uno scenario di piccole dimensioni, affrontando i primi **pattern** caratteristici di DDD.

Domain Driven Design è un approccio alla progettazione del software che abbraccia diversi aspetti della questione, dal generale (nella parte denominata **Strategic Domain Driven Design**) al particolare.

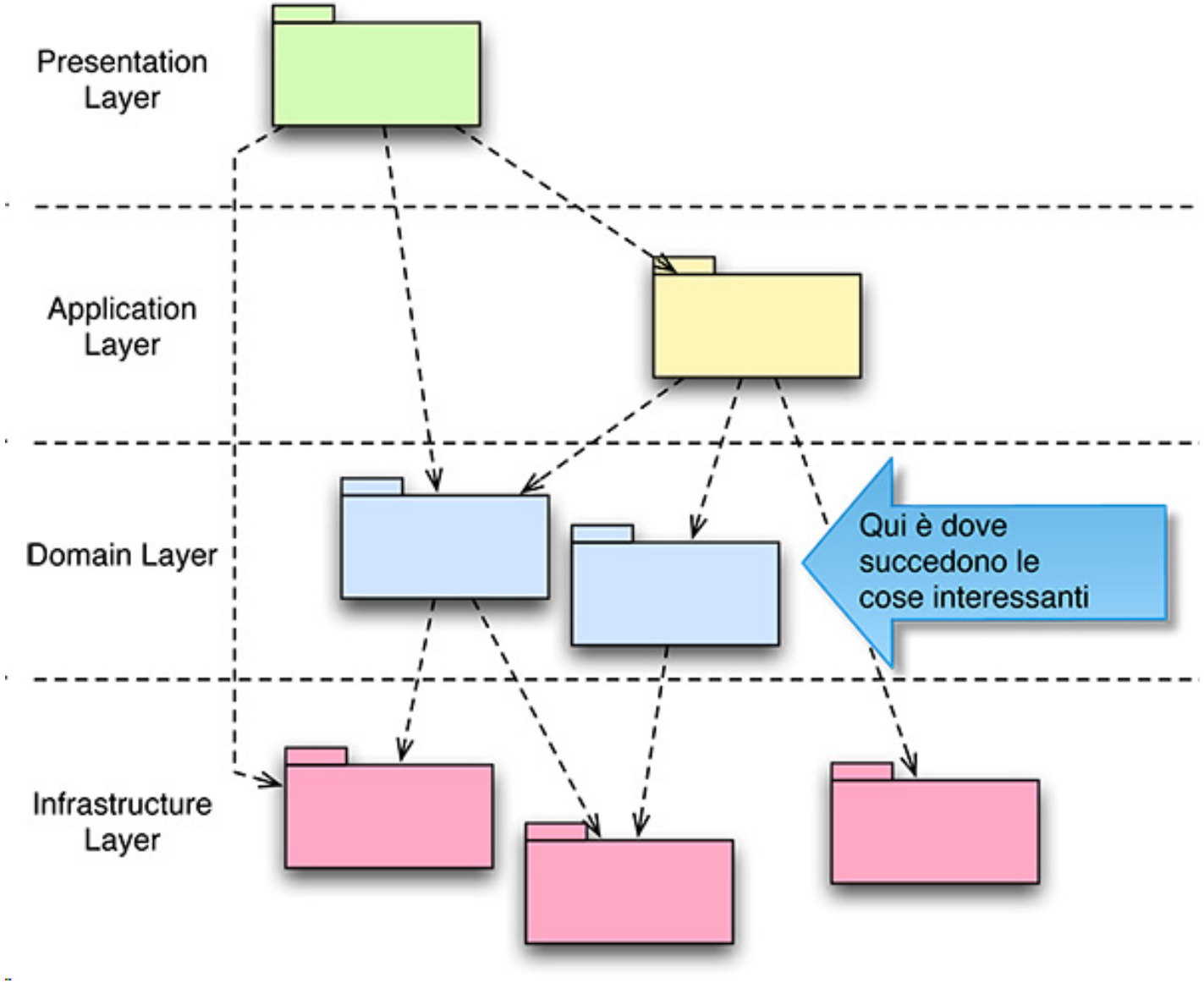
Partiamo dal «basso» ovvero assumendo di voler applicare i principi di DDD a una singola applicazione, tralasciando per ora le relazioni con il mondo esterno.

Architettura di riferimento

Abbiamo visto che le tecniche di Domain Driven Design, per una singola applicazione, richiedono la presenza di un Domain Model, all'interno del business layer.

C'è qualche differenza rispetto ad una classica architettura a tre livelli, è bene quindi approfondire le responsabilità dei rispettivi layer.

Architettura di riferimento



Presentation Layer

Contiene tutte le componenti software relative al canale (o ai canali) di presentazione utilizzati e la corrispondente logica di visualizzazione.

In altre parole, questo strato conterrà la nostra UI così come la logica di navigazione e di aggiornamento dei campi mostrati a schermo.

Application Layer

E' lo strato che espone la «logica applicativa», generalmente strutturata a servizi.

Si tratta di un layer «leggero» che contiene logica essenzialmente di coordinamento, ma non la logica applicativa.

Si tratta di componenti generalmente stateless, salvo nei casi in cui siano responsabili di transazioni o operazioni «lunghe» e lo stato dell'operazione diventi una informazione chiave.

Domain Layer

E' lo strato che contiene la «logica di business» e lo stato delle componenti significative dell'applicazione.

In questo strato il modello racchiude l'essenza del dominio applicativo che la nostra applicazione deve supportare.

Infrastructure Layer

E' lo strato che contiene tutte le «componenti tecnologiche» necessarie a portare a termine i compiti dei vari strati.

Di fatto tutte le librerie o le interazioni con la persistenza sono localizzate in questo strato.

Infrastructure Layer

In **Domain Driven Design**, l'obiettivo è l'implementazione di un modello del dominio applicativo il più possibile utile.

Si tratta di un bersaglio in movimento, in quanto sia la nostra comprensione del sistema che le esigenze degli utenti sono in continua evoluzione.

Inoltre, in un **dominio complesso**, un modello che ne catturi l'essenza costituisce un valore ed è bene che **non sia legato in alcun modo a componenti tecnologiche** che potrebbero anticiparne l'**obsolescenza**.

Infrastructure Layer

Il nostro modello dovrà quindi essere

- **semplice** (nel senso di privo di ogni inutile ridondanza)
- **chiaro** (il ruolo di ogni componente dovrebbe essere autodescrittivo)
- **conforme** alla nostra **comprensione** del dominio
- **espresso** in termini di **Ubiquitous Language**

La struttura a layer separa molto bene gli effetti delle evoluzioni tecnologiche (che si concentreranno sul presentation o sull'infrastructure layer) da quelle legate al business, che avranno il loro cuore nel domain layer.

Approfondimenti sul Domain Model Pattern

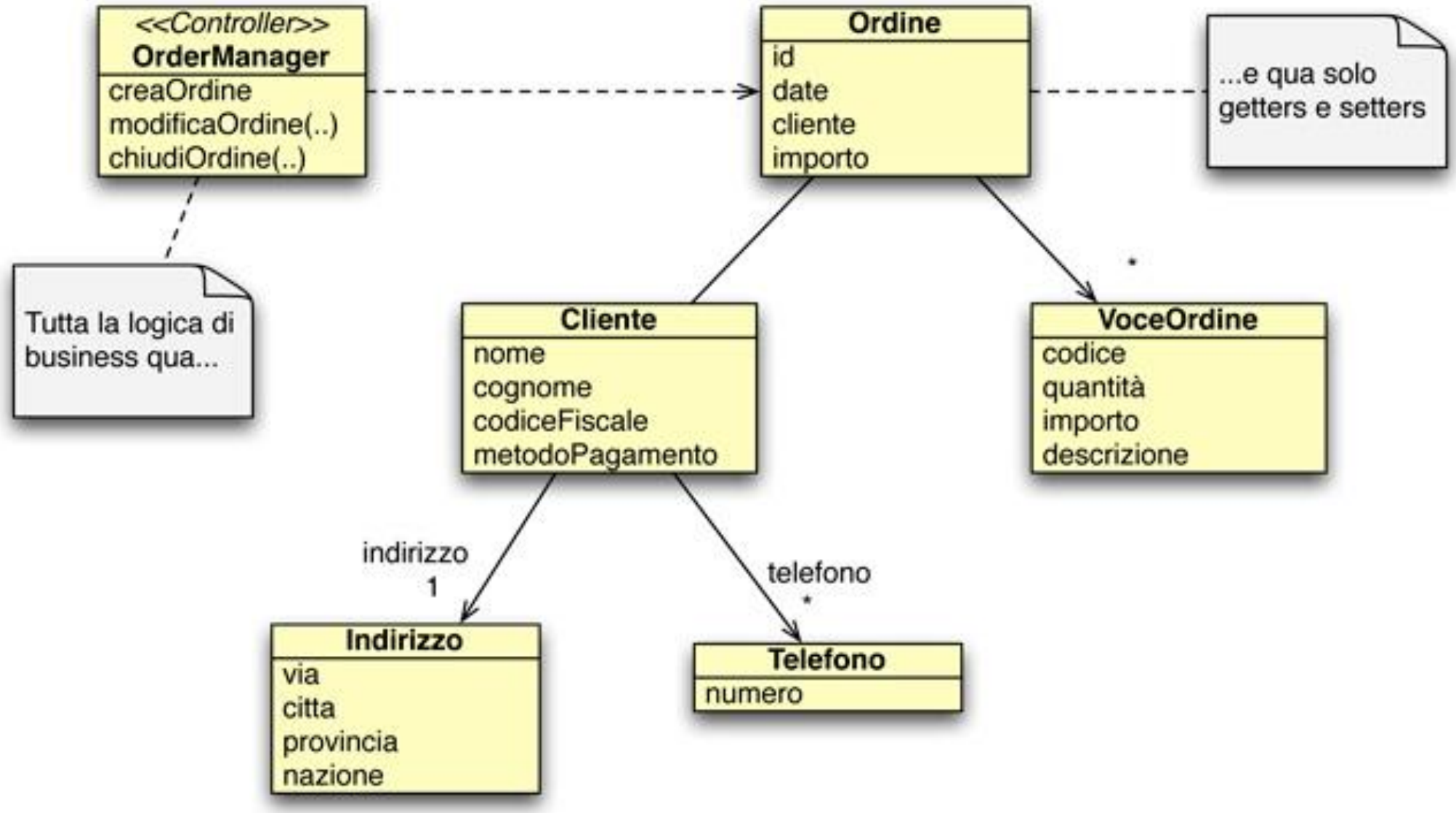
Il Domain Model Pattern

Abbiamo iniziato a esplorare il Domain Model Pattern.

Ciò che caratterizza questo pattern architetturale è che all'interno delle classi del nostro Domain Model troviamo sia i dati, cioè gli attributi, che il comportamento, ovvero i metodi di business.

In altre parole, non ci troveremo con classi (variamente denominate, in genere *Controller o *Manager) che manipolano i nostri oggetti di dominio, come avviene nell'Anemic Domain Model o implementando pattern alternativi quali il Transaction Script.

Esempio di Anemic Domain Model



Il Domain Model Pattern

Uno schema come quella mostrato precedentemente ha il vantaggio di identificare chiaramente il ruolo di coordinamento (la classe OrderManager) ma localizza gran parte della complessità in un unico punto (sempre la classe OrderManager).

In questo scenario abbiamo un accoppiamento forte tra la classe OrderManager e le altre entità del modello: OrderManager deve sapere come sono fatte queste classi per poterle modificare.

Il modello risulta leggibile in prima stesura (spesso corrisponde a un documento di analisi), ma la sua evoluzione risulta difficoltosa.

Il Domain Model Pattern

In un contesto reale, la complessità non si distribuisce in questo modo: domini di elevata complessità sono il risultato dell'interazione di molte componenti di media o bassa complessità, secondo relazioni più o meno strutturate.

Ciò che appare inestricabile dall'alto, spesso espone una sua raffinata semplicità se osservato da vicino.

Raggiungere questa «raffinata semplicità» è esattamente il nostro scopo.

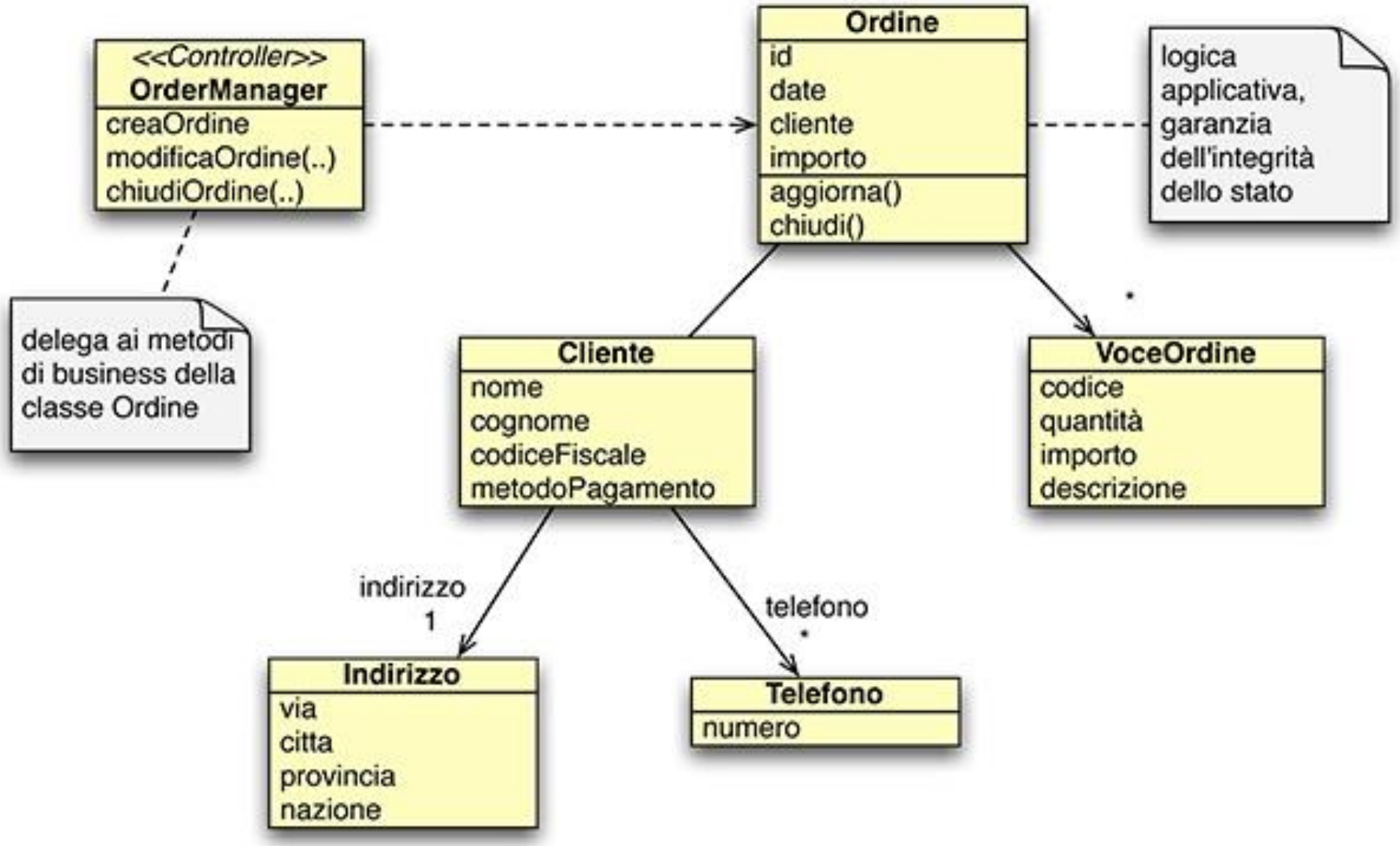
I Domain Model si presta meglio di altri pattern architetturali a rappresentare domini complessi, perché scompone la complessità fra i vari oggetti che lo compongono, delineando ruoli e responsabilità.

Il Domain Model Pattern

Gli oggetti di un domain model sono «consapevoli» delle proprie caratteristiche ed espongono metodi rilevanti per lo svolgimento dei compiti chiave.

Ad esempio, in un'applicazione gestionale la classe Ordine potrà esporre il metodo chiudi(...) che innescherà una transizione di stato nell'Ordine stesso.

Il Domain Model Pattern



Il Domain Model Pattern

Che una classe sia **responsabile del proprio stato interno**, in effetti, non è una novità: si tratta del buon vecchio **incapsulamento** (messo a repentaglio dalla diabolica accoppiata getters+setters) che garantisce una migliore modularità della nostra applicazione e una maggiore possibilità di riuso.

All'Application layer è delegato il compito di chiamare l'oggetto di dominio responsabile di una determinata azione.

L'elemento forse più caratteristico (e per qualcuno, spiazzante) del Domain Model Pattern è che l'esecuzione di compiti anche relativamente semplici è spesso **distribuita** tra più oggetti.

Il Domain Model Pattern

Supponiamo di voler calcolare l'importo di un ordine, composto da più voci.

In generale si tratta di calcolare la somma di tutte le voci che compongono l'ordine.

$$\text{Totale} = (\text{prezzo}_1 * \text{qta}_1) + (\text{prezzo}_2 * \text{qta}_2) + \ldots$$

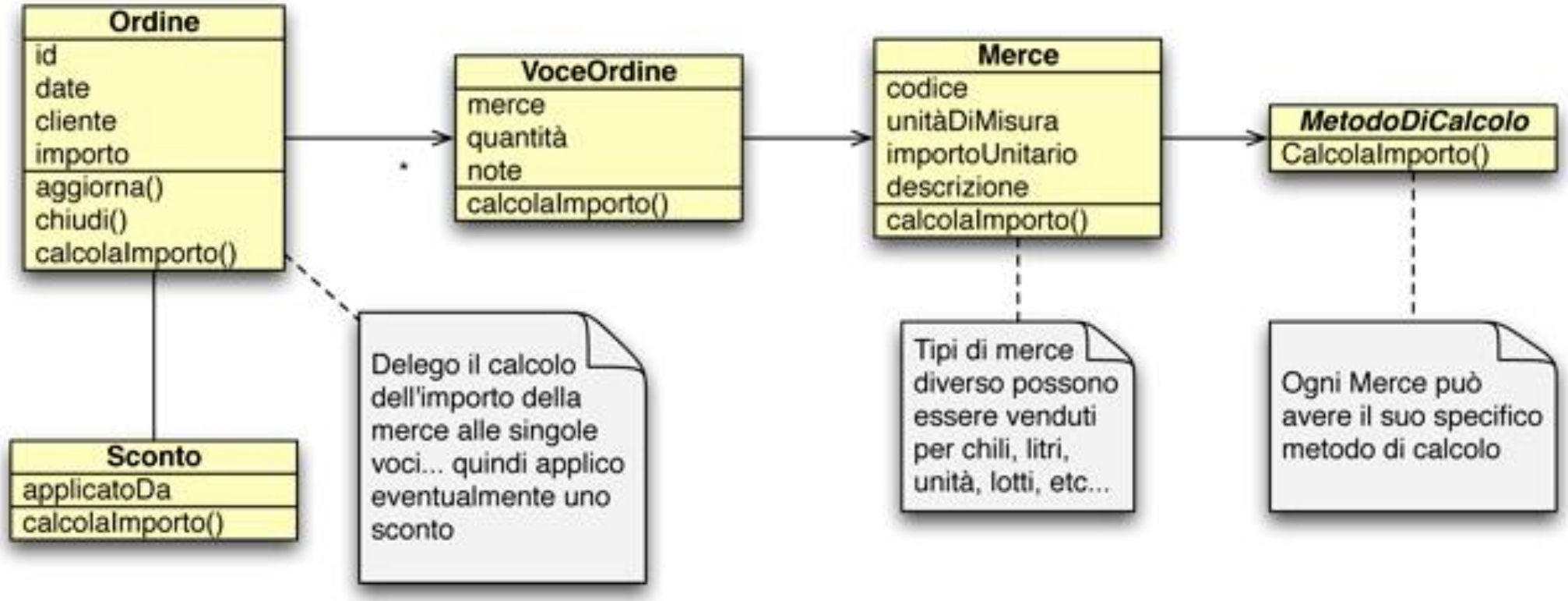
Questo, in un dominio ideale.

Il Domain Model Pattern

In un dominio reale, la complessità fa capolino, con qualche requisito «bastardo»:

- differenti categorie merceologiche sono espresse in differenti unità: pezzi, litri, metri, quintali, etc.
- il prezzo è funzione della quantità, ma non necessariamente in maniera lineare: comprando 100 potrei avere uno sconto. Comprando 1000 uno sconto maggiore.
- è possibile che siano presenti offerte speciali su determinati articoli.
- i commerciali si riservano di applicare una determinata percentuale di sconto «a prescindere».

Davanti a noi ci sono 2 strade: o una «piramide di if» oppure una scomposizione del problema in elementi più piccoli e singolarmente più semplici da gestire.



La responsabilità del calcolo dell'importo complessivo di un ordine viene ripartita su più componenti, ognuna con un ruoli ben preciso.

Il Domain Model Pattern

```
public class Ordine {
    Cliente cliente;
    Sconto sconto;
    List vociOrdine;
    Date data;
    public BigDecimal calcolaImporto() {
        BigDecimal importo = BigDecimal.ZERO;
        for (VoceOrdine voceOrdine :vociOrdine) {
            importo = importo.add(voceOrdine.calcolaImporto());
        }
        return importo;
    }
    public BigDecimal calcolaImportoScontato() {
        return sconto.applicaSconto(this.calcolaImporto());
    }

          }
```

Il Domain Model Pattern

```
public class VoceOrdine {
    private Quantity qta;
    private Merce merce;

            public BigDecimal calcolaImporto() {
        return merce.calcolaImporto(qta);
    }
}
```

```
public class Merce {
    String descrizione;
    MetodoDiCalcolo metodoDiCalcolo;

            public BigDecimal calcolaImporto(Quantity qta) {
        return metodoDiCalcolo.calcolaImporto(qta);
    }
}
```

Il Domain Model Pattern

```
public interface MetodoDiCalcolo {
    BigDecimal calcolaImporto(Quantity qta);
}
```

```
public interface Sconto {
    BigDecimal applicaSconto(BigDecimal importoNonScontato);
}
```

Interfacce che possono essere implementate... come ci pare! In altre parole abbiamo una implementazione del modello che oltre ad essere conforme alle specifiche «bastarde» si presenta come «tollerante al cambiamento».

Ruoli e Responsabilità

Il succo di questo «gioco» sta nell'individuazione dei ruoli e delle responsabilità.

A chi spetta calcolare l'importo di un ordine ?

Chi ha le informazioni per dare la risposta: Ordine non le ha, chiede a VoceOrdine; VoceOrdine a sua volta delega a Merce, che a sua volta delega al proprio MetodoDiCalcolo che "sa" e quindi agisce.

Ordine inoltre delega a Sconto l'applicazione dello sconto sul totale.

Un modello in cui ruoli e responsabilità sono correttamente attribuiti alle classi risulta più facile da mantenere e da testare, ed è il miglior punto di partenza delle successive evoluzioni.

In ottica DDD è anche l'immagine precisa della nostra attuale comprensione del dominio.

Limiti del Domain Model Pattern

Il limite principale di tutto ciò è che richiede una notevole dimestichezza con i concetti e il «ragionamento» Object Oriented Programming.

Una struttura di questo genere sposta la complessità del calcolo dalle espressioni condizionali alle relazioni tra gli oggetti, riducendo le espressioni condizionali basate su if e switch, ma aumentando il numero delle classi coinvolte.

Non tutti i programmatori si trovano a proprio agio con codice strutturato in questo modo, a volte necessitano di una struttura che evidenzi maggiormente la «sequenza» delle operazioni, rispetto alle responsabilità, come avviene nel Transaction Script.

Limiti del Domain Model Pattern

Anche i programmatori con maggiore dimestichezza con i concetti OOP hanno opinioni divergenti su come implementarli in un Domain Model finendo per prodursi in interminabili discussioni alla macchina del caffè riguardo il miglior design pattern per catturare una determinata situazione del dominio.

Spesso, anche un Domain Model fatica a crescere oltre una determinata dimensione, perché l'implementazione è affidata a regole troppo astratte o generiche, che finiscono per creare esempi applicativi eleganti, ma con forti problemi di scalabilità al crescere della complessità del dominio, diventando intricati e di difficile gestione.

Qui entrano in gioco i primi pattern di Domain Driven Design.

Limiti del Domain Model Pattern

In realtà, l'ostacolo principale ad una corretta implementazione del Domain Model è dato dal sapiente gioco di specchi e lustrini offerto da alcuni framework che dichiarano di poter generare modelli di dominio a partire dal modello dei dati.

Si tratta di una strada assolutamente percorribile nel caso di dominii piuttosto semplici (e gli articoli su Grails lo testimoniano) ma non dobbiamo dimenticare che i tools automatizzano le cose che sono semplici da capire e che sono ripetibili, ma si trovano in difficoltà quando le cose sono ambigue o difficili da interpretare, finendo per risultare spesso d'intralcio in contesti più complessi.

Domain Driven Design Patterns

Domain Driven Design Patterns

DDD elenca una serie di pattern che nel loro insieme definiscono una **struttura coerente** per implementare correttamente un Domain Model.

I pattern non sono una invenzione di Domain Driven Design, ma semplicemente una raccolta.

Il valore è nell'insieme di pattern nel suo complesso più che in un pattern specifico.

Per chi è abituato a considerare i pattern tenendo come riferimento i pattern GoF, i pattern elencati da DDD possono risultare un po' spiazzanti: **il focus non è nel codice, ma nel ruolo che determinate classi vengono ad assumere nel contesto**.

In certi casi non esiste una vera e propria «reference implementation»: in prospettiva DDD spesso il ragionamento che sta dietro al codice finisce per essere più importante del codice stesso.

DDD riconosce e codifica l'esistenza di comportamenti ripetuti all'interno del Domain Model pattern che nel loro insieme definiscono una struttura coerente, e sono in grado di permettere la costruzione di modelli che non vengono messi in crisi dalla complessità, ma che hanno la capacità di accettare la sfida.

C'è però un limite che non vorremmo superare: è quello oltre il quale non siamo in grado di mantenere la coerenza del modello e dello Ubiquitous Language.

Entities

Entities

Le **entità** sono le componenti fondamentali del nostro dominio.

In un software di gestione ordini, queste saranno le classi corrispondenti ad Ordine, Cliente etc.

In un software bancario, utenti e conti correnti saranno con ogni probabilità modellati come entità.

OK, ma come posso definire il concetto di Entità all'interno del mio dominio applicativo?

Entities

Due sono gli elementi fondamentali per identificare le entità.

- **Le entità hanno una storia**: vengono create o inserite nel sistema, modificate, tracciate, si evolvono, subiscono trasformazioni nel corso del tempo, modificando il loro stato. Le trasformazioni sono parte integrante della loro vita; spesso la nostra applicazione non è altro che uno strumento di supporto alla gestione e al tracciamento delle trasformazioni.

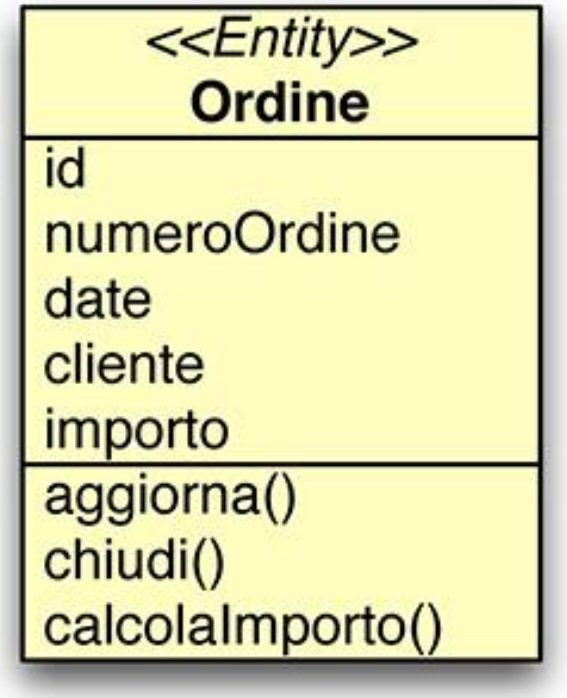
- **Le entità hanno una identità**: anche in presenza di attributi identici con altre istanza della stessa classe, esiste sempre una nozione di identità che permetta di distinguere questo «Mario Rossi» da quell'altro «Mario Rossi»

Le Entity sono le classi cardine attorno alle quali ruota il nostro dominio: l'intera applicazione può essere vista come una struttura che permette alle Entity di esistere e di trasformarsi.

Entities

Torniamo all'esempio precedente: tra le classi che abbiamo identificato, un candidato ideale ad essere classificato come Entity è la classe Ordine.

Siamo interessati alla sua storia, alle evoluzioni che può avere nel nostro dominio (le transizioni di stato da aperto, ad evaso, a fatturato o ad annullato) ed alla sua identità, nel nostro dominio probabilmente caratterizzata dal numero d'ordine.



Il concetto di identità

A prima vista la presenza di una nozione di identità non sembra nulla di trascendentale: in genere le nostre classi di dominio avranno un'immagine sul DB e questa sarà necessariamente associata a una chiave unica di qualche genere.

Dobbiamo però fare attenzione a non mescolare il problema dell'univocità dei dati persistenti con il concetto di identità specifico della nostra applicazione.

Nel nostro esempio esiste già una nozione di identità caratteristica del dominio: il numero d'ordine è un concetto che esiste a prescindere dalla nostra applicazione, per motivi organizzativi e fiscali.

Tuttavia la situazione non è sempre così limpida: proviamo a fare qualche esempio, uscendo dal dominio della gestione ordini.

Il concetto di identità: Esempio 1

Nel dominio bancario, ogni conto è associato a un codice IBAN.

Il codice IBAN permette di identificare univocamente un conto.

Ma si tratta di un codice introdotto di recente.

Lo stesso conto può essere presente in archivio anche con la chiave precedente al codice IBAN.

Si tratta dello stesso conto ?

Il concetto di identità: Esempio 2

Voglio aprire un conto corrente.

Ho già un conto corrente in un'altra banca dello stesso gruppo.

La banca ha interesse a capire se io sono la stessa persona che ha già un altro conto.

A meno di non aver lasciato un imbarazzante scoperto, anche io, come utente, preferirei non dover inserire/comunicare tutti i miei dati un'altra volta.

Come può il sistema capire che sono sempre io e non un omonimo?

Il concetto di identità: Esempio 2

C'è il Codice Fiscale, tutti ce l'hanno ed è un codice di riferimento univoco.

Beh... non è proprio così: non è affatto detto che tutti ce l'abbiano (è vero per i giovani, non è detto per gli anziani... o meglio il CF c'è anche per loro, ma magari non lo sanno), e non è detto che sia univoco: c'è un certo numero di persone con il problema dell'omocodia ovvero più persone con lo stesso codice fiscale.

Inoltre il Codice Fiscale può cambiare nel corso del tempo (in caso di cambiamento di nome, cognome o sesso o risoluzione dell'omocodia).

Gli utenti in questa situazione, possono essere una percentuale trascurabile in un determinato contesto, ma possono essere un problema serio in un'applicazione che ha come requisito imposto dalla legge di tenere traccia di tutti gli utenti senza distinzione.

Il concetto di identità: Esempio 3

Complichiamo ulteriormente il quadro, voglio aprire il conto corrente e sono straniero.

Mi viene richiesto il Codice Fiscale… che non ho!

Decisamente poco user-friendly…

Il concetto di identità: Esempio 4

La banca è internazionale, e un unico software gestisce i correntisti di più nazioni, che ovviamente possono essere liberi di viaggiare e trasferirsi a loro piacimento.

Come posso identificare un utente che ha due conti in due nazioni diverse?

Il concetto di identità: Esempio 5

Voglio registrarmi su un'applicazione online, stile Facebook, per intenderci: in questo caso il controllo dell'identità si basa sull'unicità dell'indirizzo e-mail (anche se non è affatto detto che un account sia usato da una sola persona).

In altri casi è il nickname a garantire l'identità (della nostra entità, non la nostra...)

Il concetto di identità: Considerazioni sugli esempi

Si potrebbe andare avanti per ore.

Il punto è che il concetto di identità a cui sono associate le nostre entità non si riduce a distinguere due record sul database, ma è un concetto intrinsecamente legato al dominio applicativo: differenti contesti richiederanno differenti strategie e garanzie.

In alcuni casi possiamo permetterci di viaggiare leggeri, in altri casi è necessario affrontare in profondità le pieghe del concetto di identità all'interno degli scenari in cui la nostra applicazione andrà ad operare.

Trasformazioni di stato

Le nostre entità non sarebbero così interessanti per la nostra applicazione se non si evolvessero: **le Entity espongono gli operatori di trasformazione e sono responsabili dell'integrità del proprio stato interno**.

Vigono quindi le buone norme dell'incapsulamento: **proteggo lo stato interno**, offrendo all'esterno solo modificatori pubblici che garantiscano l'integrità dello stato.

In linguaggi come Java questo può essere un problema, perché cozza con le specifiche JavaBeans che prevedono la disponibilità di metodi getter e setter che «rompono» l'incapsulamento.

Trasformazioni di stato

Il punto è che questi metodi sono stati pensati per essere utilizzati dai framework, non dai programmatori.

Se tali metodi sono usati per gestire la persistenza o la visualizzazione di un oggetto su una pagina web, o la serializzazione-deserializzazione verso XML, non ha probabilmente senso accanircisi contro.

I problemi veri emergono quando noi sviluppatori andiamo ad alterare lo stato interno di un oggetto «a mani nude e senza protezione».

Il metodo equals()

In Java, come in altri linguaggi OOP, per controllare se due diverse istanze della stessa classe fanno riferimento alla stessa entità concreta (ad esempio perche' lo stesso dato viene acceduto da due processi concorrenti) ci affidiamo al metodo equals().

Questo metodo ci permette di specificare una relazione di uguaglianza (a patto che rispettiamo le specifiche e il contratto d'uso definiti da mamma Sun nel JavaDoc di java.lang.Object) che potrà essere utilizzata in più punti della nostra applicazione:

- nelle collezioni del Collection Framework per verificare la presenza di duplicati o per cercare/trovare un oggetto;
- dai nostri test per verificare che l'oggetto restituito da un metodo sia uguale a quello atteso, se utilizziamo i metodi assertEquals(...);
- da ORM, per riconoscere accessi concorrenti alle stesse entità da più parti della nostra applicazione;
- dalla logica business della nostra applicazione.

Il metodo equals()

Il problema è che non è detto che il significato attribuito al concetto di uguaglianza sia lo stesso in tutti i contesti.

La definizione del concetto di uguaglianza può inoltre porre dei vincoli implementativi (è il caso di Hibernate, ad esempio) che possono non risultare congrui con un concetto di uguaglianza definito altrove.

In determinate circostanze, potrà quindi essere necessario ricorrere a metodi per determinare una relazione di uguaglianza rilevante nel nostro contesto, con metodi specifici.

Value Object

Value Object

Non tutti gli oggetti caratteristici del nostro sistema devono per forza evolversi, non tutti necessitano di preservare la loro identità.

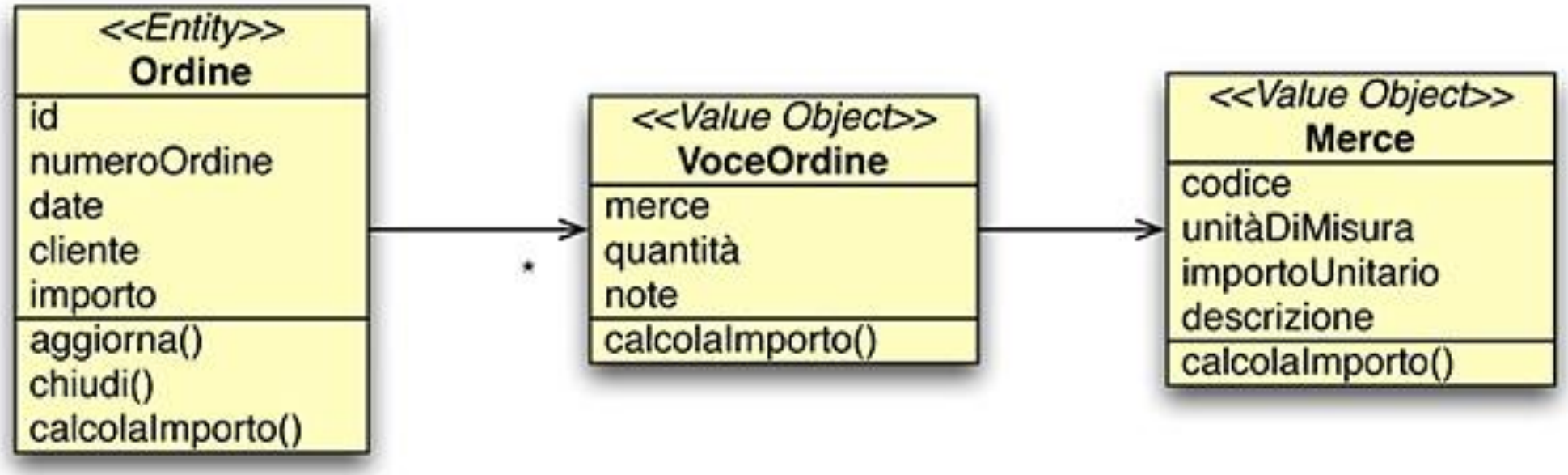
Anche nel mondo reale esistono oggetti per cui l'identità è importante, e altri per cui l'identità è ininfluente.

La nostra automobile è la nostra automobile con tanto di targa, e piccole ammaccature (noi sappiamo dove).

Value Object

Se acquistiamo un giornale all'edicola, poco importa se il giornalaio ci offra il primo o il secondo della pila.

All'interno del nostro dominio, gli oggetti per cui il tratto distintivo è il valore, sono denominati Value Object.



Value Object

La classificazione di Merce come Value Object è abbastanza naturale: specialmente per categorie merceologiche vendute a chili o a litri, si tratta di elementi spesso indistinguibili anche nel mondo reale.

La scelta di classificare VoceOrdine come Value Object oppure come Entity è per certi versi controversa: se scegliamo di definire lo stato d VoceOrdine al momento della creazione (definendo quantità e note come immutabili) allora possiamo trattare VoceOrdine come Value Object.

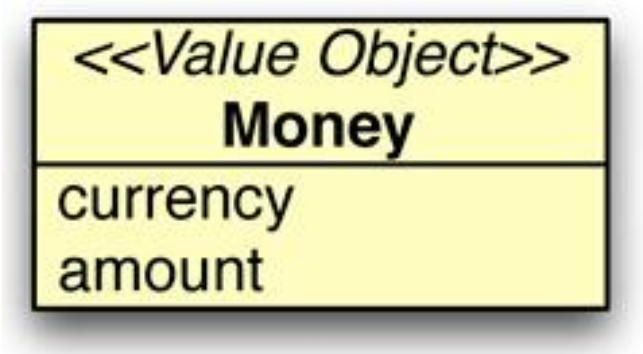
Se invece vogliamo permettere alla nostra applicazione di gestire modifiche di stato alle singole voci dell'ordine (con conseguente incremento della complessità in questa particolare area del dominio) allora siamo costretti a trasformare VoceOrdine in una Entity.

Value Object

In generale, ove possibile, è bene utilizzare Value Object (il modo migliore per gestire la complessità è averne il meno possibile).

Se un requisito ci forza ad indurre questa trasformazione, è bene verificarne la validità: si tratta di quelle modifiche apparentemente leggere, ma che invece finiscono per costare un sacco.

Un altro esempio di Value Object «concreto» tipico di numerosi domini potrebbe essere rappresentato dalla classe Money.



Value Object

In un contesto in cui il calcolo finanziario riveste una certa importanza, dove è possibile la presenza di valute diverse, è bene che importo e valuta siano attributi della stessa classe, favorendo una gestione centralizzata, e magari riusabile delle problematiche del calcolo multi-valuta o anche solo della corretta gestione della precisione.

Riuso ed eleganza del design a parte, il problema di fondo è che nel dominio finanziario si parla di importi, non di numeri.

La presenza di una classe che rappresenta un importo associato ad una valuta rende il modello del nostro dominio più vicino al dominio reale.

Il risultato di questa scelta sarà codice più leggibile, manutenibile e riusabile, ma è un «effetto di secondo ordine» dell'avere un modello aderente al dominio applicativo (ed espresso nello ubiquitous language).

Immutabilità

Un altro effetto interessante delle caratteristiche dei value objects è che essendo interessati solo al loro valore, non siamo interessati al loro stato.

O meglio: i Value Objects non contemplano l'idea di stato come attributo modificabile, sono cioè immutabili: una volta costruiti, il loro valore non può essere alterato; una possibile implementazione di Money che garantisca l'immutabilità potrebbe essere la seguente.

```
public class Money {
    private BigDecimal amount;
    private Currency currency;
    public Money (Currency currency, BigDecimal amount) {
        this.currency = currency;
        this.amount = amount;
    }
    public BigDecimal getAmount() {
        return this.amount;
    }
    public Currency getCurrency() {
        return this.currency;
    }
}
```

Immutabilità

Un solo costruttore ci obbliga a valorizzare entrambi gli attributi della classe.

Non sono presenti ne' modificatori ne' metodi setters per cui il nostro oggetto è immutabile.

Questa caratteristica non è priva di interesse: oggetti immutabili possono essere scambiati e condivisi all'interno della nostra applicazione, semplificandone la gestione, e in alcuni casi permettendo qualche risparmio in termini di occupazione della memoria

Il vantaggio principale è quello di poter tralasciare la gestione dello stato dei VO: in dominii complessi non è infrequente trovarsi in situazioni intricate con un gran numero di classi coinvolte.

Poter evitare di farsi carico di una parte di queste è una considerazione che introduce delle notevoli semplificazioni.

Immutabilità

Money è un esempio abbastanza evidente di concetto presente in un dominio applicativo che necessita di una controparte nel modello e si tratta di costrutto molto frequente.

Dal punto di vista della persistenza, Money può essere associato a una o due colonne di una tabella sul DB, ed in generale non avremo una tabella importi.

Tuttavia è bene chiarire che una semplificazione del tipo...

«Entity --> identità --> id» e «Value Object --> no identità --> no id»

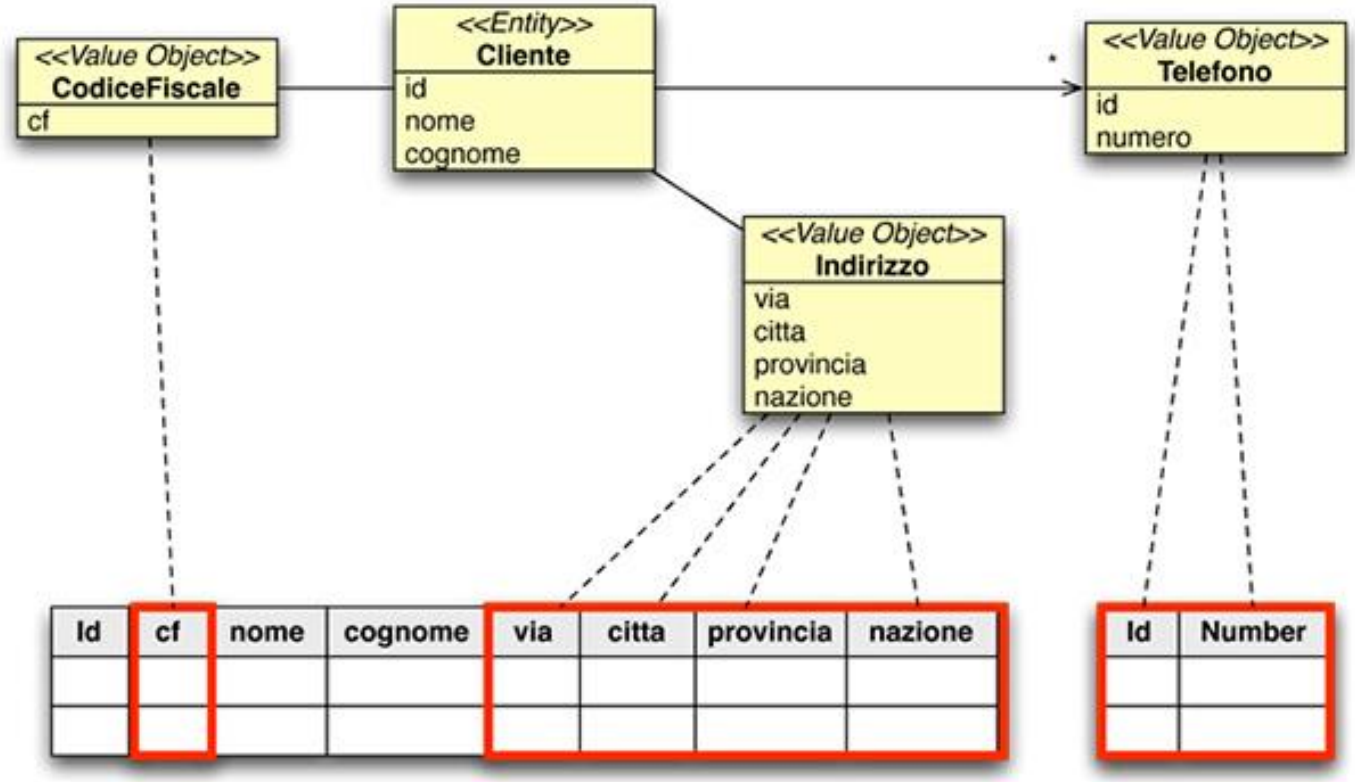
...non è corretta.

E' possibile avere value object con un singolo attributo, o più complessi ossia associati ad un singolo campo su DB oppure a un record su una tabella a parte, o addirittura dei Value Object corrispondenti a strutture più complesse.

Immutabilità

Di seguito un esempio di mapping tra value objects e tabelle del modello dei dati.

Un Value Object può corrispondere a un singolo campo, a più campi della stessa tabella o a una tabella separata; è anche possibile che corrisponda a una struttura più complessa.



Immutabilità

L'elemento chiave per una distinzione fra Entity e Value Object è la differente gestione del ciclo di vita.

Le modalità di gestione della persistenza non sono vincolanti dal punto di vista di Domain Driven Design.

Non sono, però, nemmeno un «dettaglio implementativo» da trattare con sufficienza.

Ciò perché il tipo di corrispondenza tra modello di dominio e modello dei dati è funzione del particolare contesto di applicazione e non può essere ricondotto ad uno schema predefinito.

Altre Strutture del DDD

Le factories

Le **factories** sono funzioni utilizzate per creare istanze di oggetti di dominio, che sono, per definizione, valide.

Le factories function possono eseguire una logica per costruire istanze valide e tale logica potrebbe essere diversa per ogni factory.

È per questo che ci aspetteremmo di avere più metodi factory in una classe di value object, sebbene questo non sia un requisito.

Le factories aiutano anche a rendere le cose implicite più esplicite usando nomi appropriati.

Domain Services

Nel codice possiamo dipendere da un servizio esterno, ma sappiamo che domain model non dovrebbero avere dipendenze esterne, quindi come risolviamo questo problema?

Possiamo usare un modello chiamato **Domain Service**.

In DDD, i servizi di dominio possono eseguire diversi tipi di attività,

Poiché non vogliamo dipendere da nulla al di fuori del nostro modello di dominio, non dovremmo inserire alcun dettaglio di implementazione all'interno del modello di dominio.

Ciò significa che l'unica cosa che avremo all'interno del progetto di dominio è l'interfaccia del servizio di dominio

Conclusioni

Conclusioni

Il primo punto di applicazione dei principi di Domain Driven Design è il modello della nostra applicazione.

L'implementazione di un **Domain Model** è un'avventura complessa, che richiede una buona dimestichezza con i concetti OOP, una spinta all'apprendimento della complessità del dominio e una certa disciplina nell'implementazione dei principi.

I primi pattern di DDD che abbiamo incontrato (**Entity** e **Value Object**) ci mostrano come il domain model non sia piatto, ma costituito da famiglie di classi con ruoli diversi.

Successivamente vedremo come queste si possano raggruppare tra loro formando delle organizzazioni coerenti e consistenti, che ci permetteranno di governare la complessità intrinseca nel dominio.