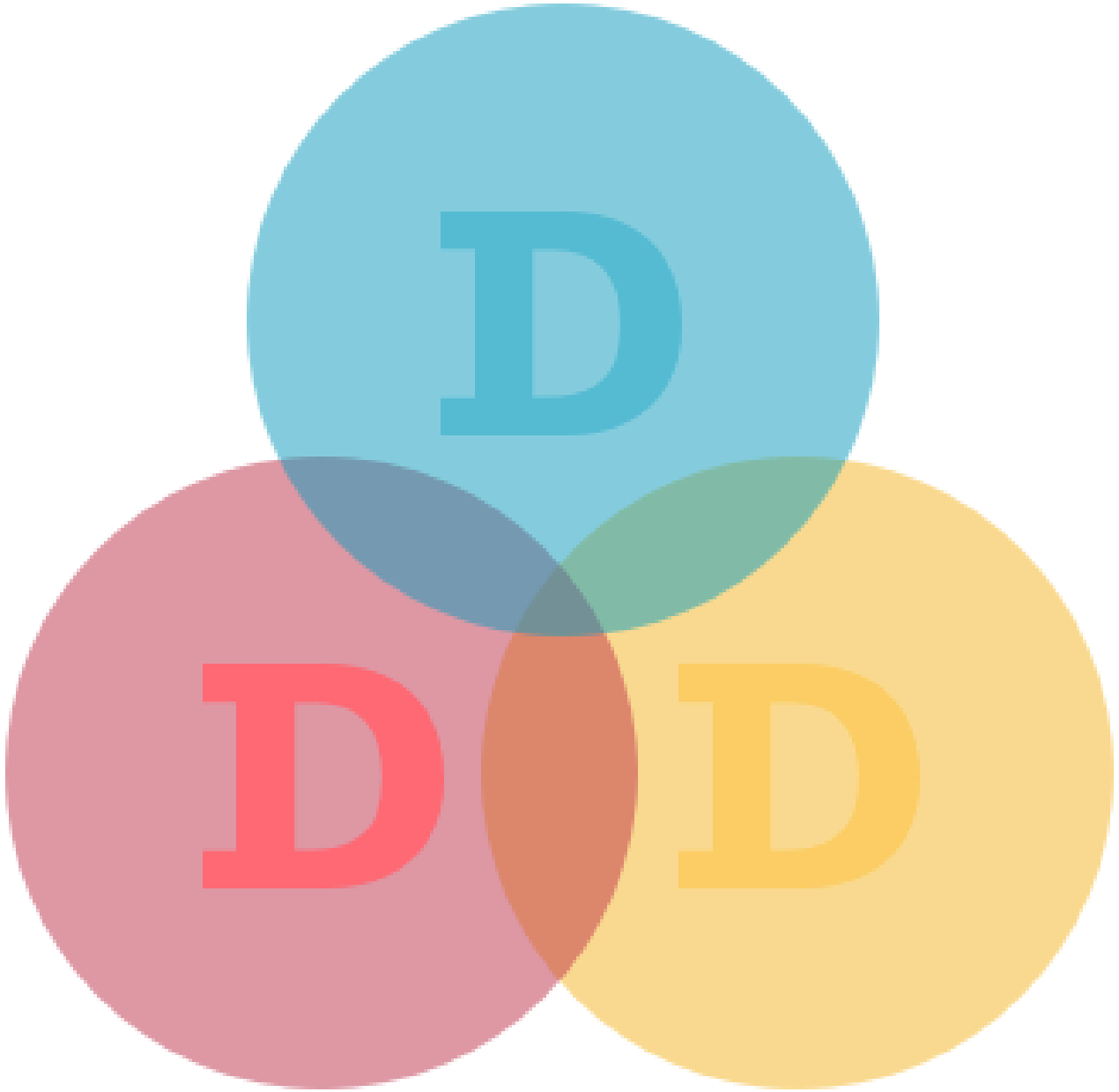

D
D
D

DDD Building blocks. Aggregates e repositories

Abbiamo visto i primi elementi fondamentali per la realizzazione di un domain model (Entities e Value Objects) e abbiamo raggiunto qualche importante risultato.

Il nostro domain model non è piatto, ma alcune famiglie di classi svolgono ruoli diversi che iniziano a delinearsi, assumendo una struttura e un'organizzazione caratteristiche.

Aggregati

Aggregati

Le classi del nostro Domain model non sono tutte uguali; alcune hanno un ruolo ben definito e possono svolgerlo in maniera indipendente mentre altre necessitano di altre classi per poter svolgere un compito significativo.

In un Domain Model, così come nella realtà sottostante, certi concetti tendono a raggrupparsi naturalmente per formare delle unità concettuali.

Un esempio classico di questa situazione, è fornito dalla coppia Ordine-VoceOrdine: un ordine necessita delle informazioni di dettaglio, e allo stesso tempo una voce ordine da sola ha un contenuto informativo insufficiente per svolgere un ruolo significativo.

Prese singolarmente le due classi non sono in grado di garantire l'integrità concettuale al nostro modello.

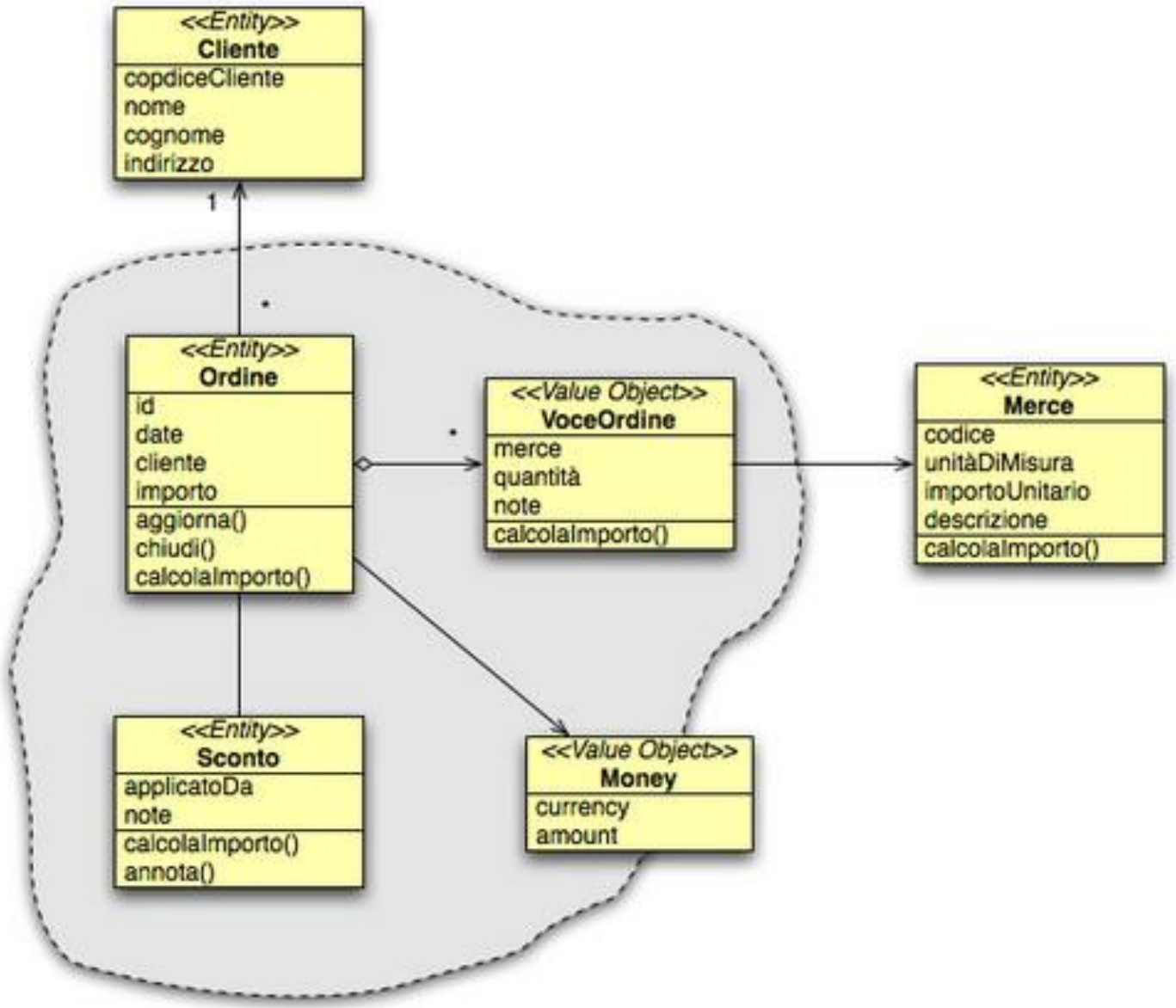
Aggregati

Da questo punto di vista, ci troviamo un una situazione abbastanza tipica: le singole voci ordine possono essere aggiornate singolarmente (ad esempio aumentando la quantità di una specifica voce) ma facendo in modo che questi cambiamenti si riflettano nel totale referenziato dalla classe Ordine.

Si tratta di una situazione tutt'altro che infrequente, e che può facilmente degenerare in qualcosa di intricato e difficilmente gestibile.

In Domain Driven Design, questa situazione rappresenta un caso tipico di **Aggregate**, ossia un insieme di classi del nostro dominio naturalmente raggruppate a formare un insieme coeso e consistente dal punto di vista funzionale.

Esempio: Ordine e VoceOrdine fanno parte dello stesso aggregato



Aggregati

Nell'esempio abbiamo supposto di trattare Sconto come un Entity, ma non si tratta necessariamente di una buona idea ☺.

Nell'esempio precedente, il nostro aggregato è rappresentato da Ordine, VoceOrdine e Sconto.

Money è un Value Object con caratteristiche tali da favorirne il riuso e la condivisione, quindi è in qualche modo «a cavallo» della linea di confine.

In questo caso si tratta fondamentalmente di una «classe di sistema» un po' come potrebbero esserlo Date o String, solo che è caratteristica del nostro dominio.

Dal punto di vista operativo, gli Aggregati rappresentano gruppi di classi che viaggiano naturalmente insieme da tre punti di vista.

Come individuare gli aggregati

- **Coordinamento transazionale**: le modifiche apportate a classi differenti all'interno dello stesso aggregato corrispondono alla stessa transazione.

- **Cancellazione**: la cancellazione di un elemento all'interno dell'aggregato implica logicamente la cancellazione di altri. Nel nostro esempio non ha senso cancellare la singola VoceOrdine senza toccare l'Ordine (il totale ed altre informazioni risulterebbero inconsistenti); viceversa la cancellazione di un Ordine implica logicamente la cancellazione delle VoceOrdine che ne fanno parte, indipendentemente dalla nostra scelta di implementarle come Entities o come Value Object. La cancellazione di un ordine non comporta però la cancellazione del Cliente, o della descrizione della merce dal catalogo.

- **Trasporto**: in caso di architetture distribuite, determinate strutture dati necessitano di essere trasportate (dal client al server, tra componenti equivalenti di un cluster, etc.); un aggregato rappresenta un'unità di trasporto sensata, che in genere ottimizza il traffico (porto tutto e solo ciò che mi serve per portare a termine il mio compito).

Aggregati

Il modo più semplice per definire i primi confini dell'aggregato è solitamente quello di verificare un confine sensato per la propagazione della cancellazione.

Il concetto è valido a prescindere da cosa si intenda per «cancellazione» all'interno del nostro dominio applicativo (semplice aggiunta di un marcatore o rimozione fisica del dato).

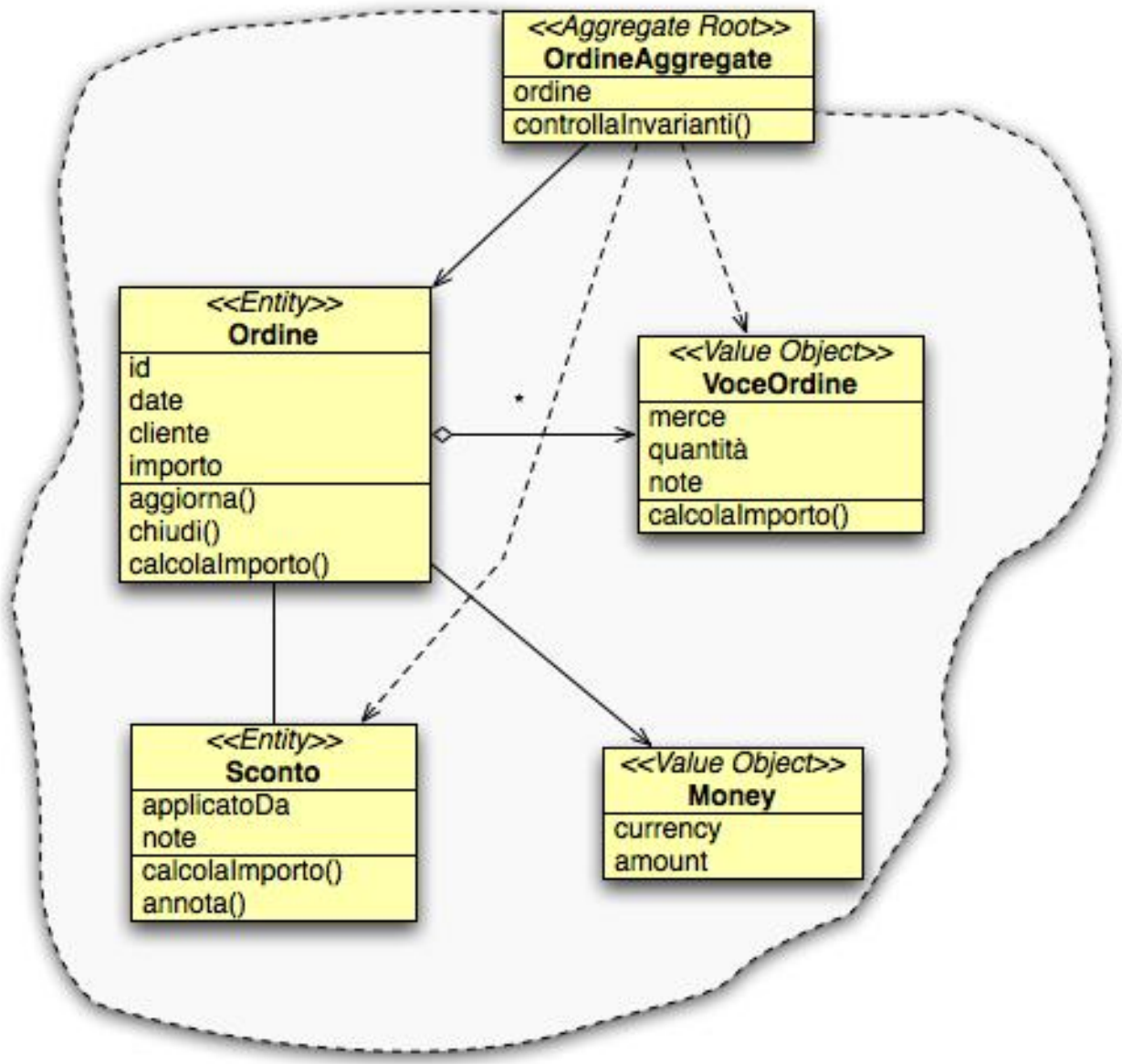
Gestione della consistenza

Il design dei nostri aggregati è responsabile della gestione della consistenza complessiva dell'aggregato.

In genere garantendo la validità di determinate invarianti caratteristiche (ovvero di condizioni che risultano sempre verificate) a seguito delle eventuali operazioni di trasformazione effettuate sulle classi dell'aggregato.

Nel nostro caso, una possibile invariante sarebbe che «in ogni momento l'importo dell'ordine deve essere uguale alla somma delle voci ordine, moltiplicata per la relativa quantità meno l'importo dello Sconto applicato».

Gestione della consistenza



Gestione della consistenza

Un aggregato è definito da un raggruppamento di classi significative del nostro Domain Model che costituiscono un insieme consistente nel suo complesso.

Gli Aggregate Boundaries rappresentano i confini all'interno dei quali l'Aggregate Root deve mantenere la consistenza dello stato complessivo.

La necessità di preservare la consistenza di ciò che avviene all'interno dei confini dell'aggregato porta a definire delle politiche di accesso strette verso i componenti al suo interno.

Gestione della consistenza - vincoli

- L'Aggregate Root è un entità responsabile del controllo degli invarianti.

- L'Aggregate Root ha un'identità valida globalmente; altre Entità all'interno dello stesso aggregato avranno un'identità ben definita solo all'interno dell'aggregato stesso (ci riferiamo al concetto di identità, non alla presenza di un Id unico per la persistenza).

- Non è possibile mantenere dall'esterno dell'aggregato un riferimento a un'entità interna all'aggregato. Ad esempio: Cliente non potrebbe referenziare direttamente VoceOrdine, ne' cambiare la quantità di una singola voce ordine senza la presenza di Ordine a gestire la coerenza del tutto (totale, ma anche verifica disponibilità, data di consegna, etc.). È possibile ovviamente ottenere un riferimento a un entità interna all'aggregato mediante una query o richiedendola all'aggregate root. Ma non possiamo modificarla, ne' attenderci che il suo stato interno rimanga consistente. Fondamentalmente si tratta di una classe in prestito.

- Questi vincoli non si applicano ai Value Objects, per le caratteristiche illustrate e discusse: trattandosi di oggetti immutabili, non è possibile modificarli, quindi nemmeno portare un aggregato che li referenzia in uno stato inconsistente.

Gestione della consistenza - Le conseguenze dei vincoli

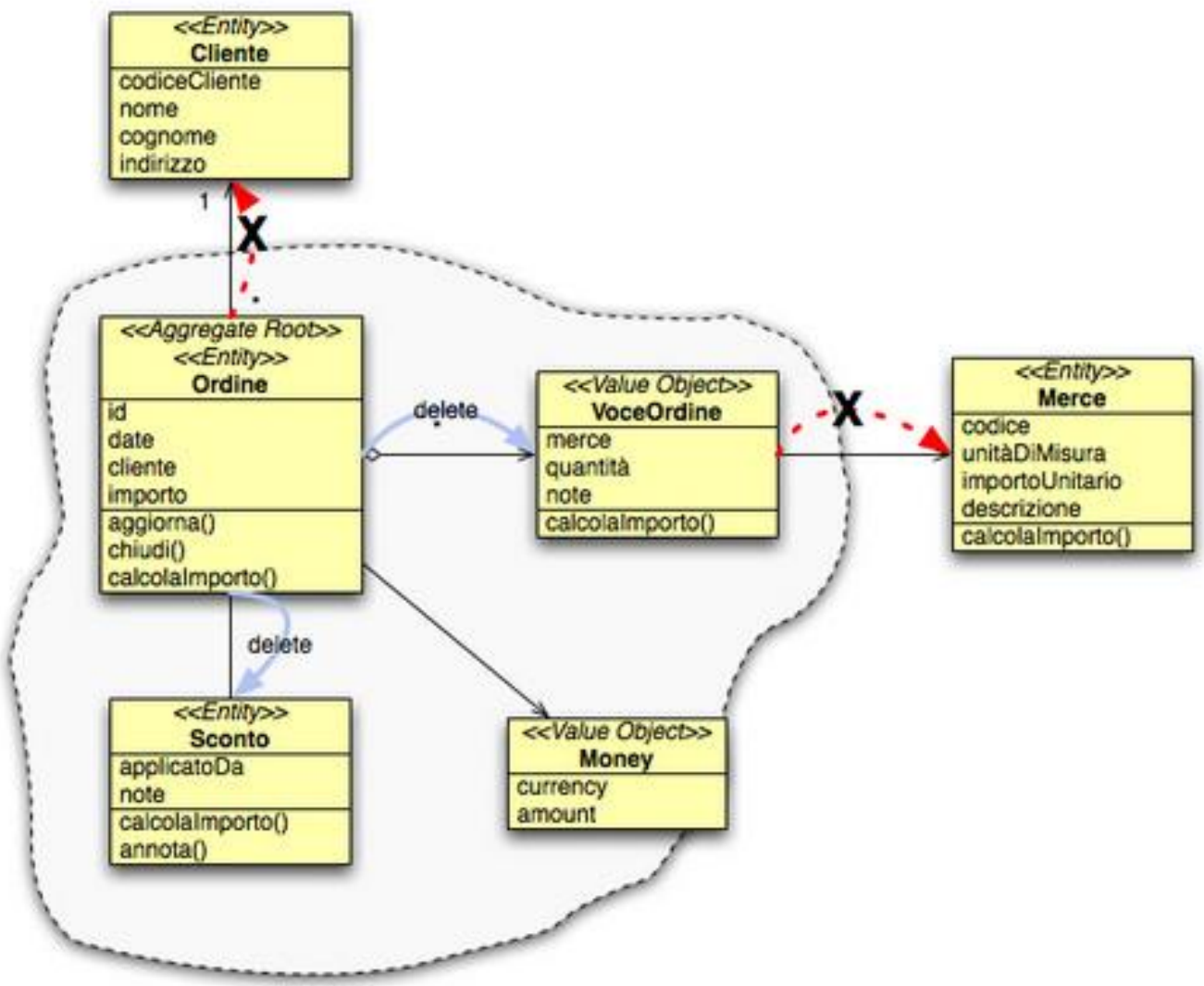
- **Gli unici oggetti che possono essere recuperati direttamente, mediante query sul database, sono le Aggregate Roots.**

- **All'interno dell'aggregato, possiamo referenziare le altre entità dell'aggregato (abbiamo garantito l'accesso in esclusiva tramite l'Aggregate Root, quindi siamo in grado di garantire la consistenza).**

- **Un'operazione di cancellazione (qualsiasi cosa significhi la cancellazione nella nostra applicazione) deve cancellare tutto ciò che è all'interno dei confini dell'aggregato in un colpo solo (vedasi figura su prossima slide).**

- **Quando un cambiamento è applicato a una qualsiasi delle entità del nostro aggregato, tutte le invarianti devono essere soddisfatte.**

Gestione della consistenza - Le conseguenze dei vincoli



Propagazione della cancellazione all'interno degli aggregati: la cancellazione della root scatena la cancellazione delle entità referenziate all'interno dell'aggregato.

Non è corretto propagare la cancellazione all'esterno dei confini dell'aggregato.

Per quanto riguarda i Value Object con una specifica rappresentazione su DB (VoceOrdine), è necessaria la cancellazione.

Per altri tipi di Value Object condivisi con l'esterno (Money) è sufficiente la dereferenziazione.

È interessante notare che i confini dell'aggregato sono validi a livello di istanza: ad esempio, un Ordine non può referenziare uno Sconto di un altro Ordine.

Aggregate Root

Aggregate Root

Generalmente, le operazioni di trasformazione dello stato passano quindi dall'aggregate root: ciò permette di propagare le variazioni di stato se necessario agli oggetti collegati e di coordinare la verifica delle invarianti.

In Entità con **transizioni di stato** di una certa complessità, spesso risulta conveniente mappare esplicitamente le transizioni di stato, utilizzando uno state diagram UML (identificando le operazioni che innescano tale transizione e verificando esplicitamente le invarianti) arrivando a implementare il tutto con uno **State Pattern** nei casi più complicati.

Aggregate Root

È possibile che determinate operazioni finiscano per interessare più aggregati: ad esempio il nostro sistema potrebbe tenere traccia del budget complessivo associato a uno specifico cliente, o variare la disponibilità della merce.

In questi scenari, l'approccio suggerito da DDD è quello di imporre transazioni all'interno dei confini del nostro aggregato, permettendo la propagazione al di fuori del nostro aggregato (aggiornando quindi il budget del nostro cliente o la disponibilità delle merci) in maniera asincrona o comunque disaccoppiata.

Aggregate Root

In molti casi i vincoli sono infatti rilassati: i dati devono ovviamente essere consistenti (altrimenti si potrebbe fare tutto a mano, senza usare il computer) ma non è necessario che tutti i dati siano istantaneamente consistenti nello stesso momento.

Usare gli aggregati in questo modo permette di limitare notevolmente l'accoppiamento fra le diverse entità caratteristiche del nostro Domain Model e di aprire opportunità interessanti in termini di architetture distribuite.

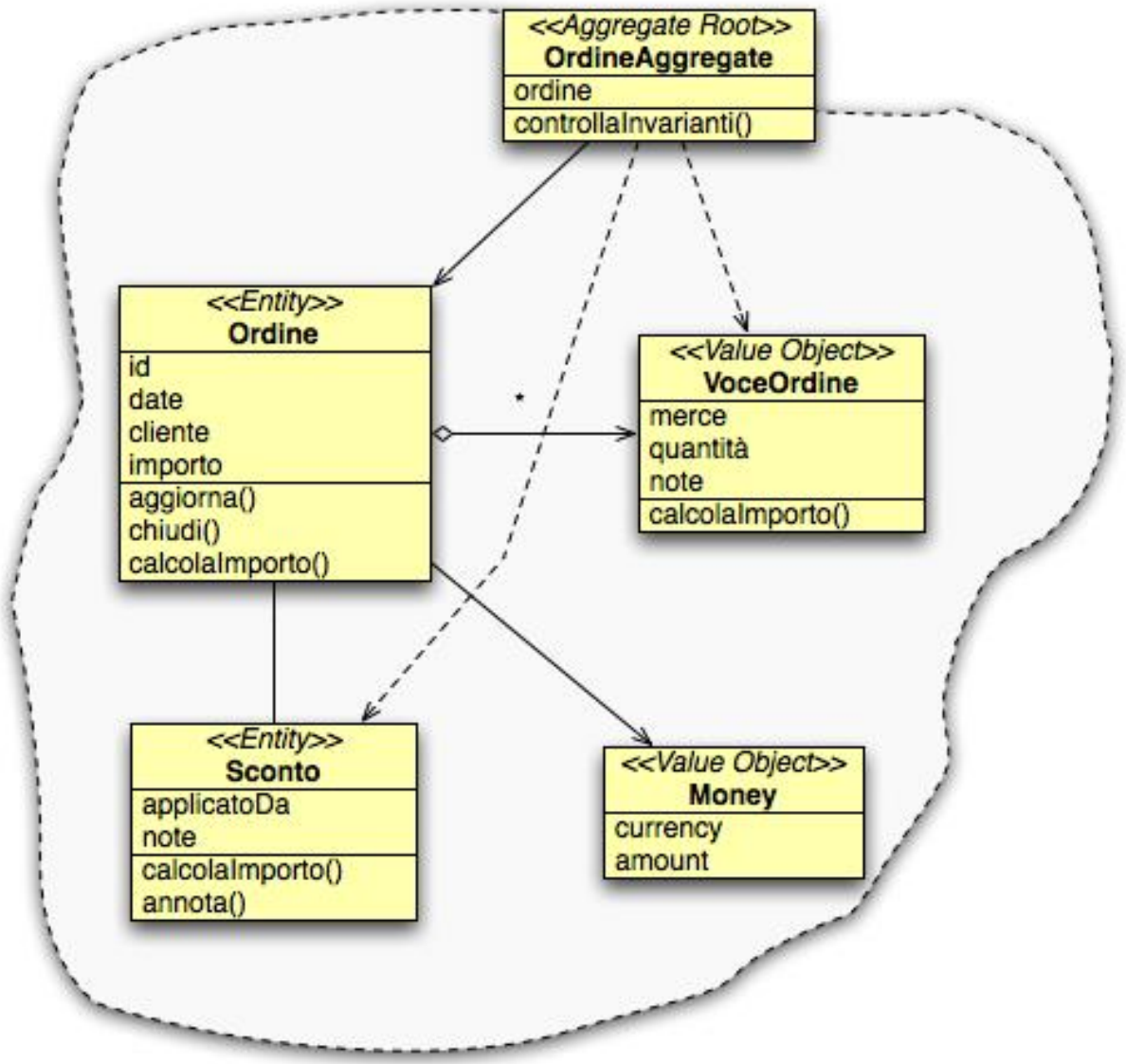
Aggregate Root Object

In alcuni casi, attribuire all'aggregate Root la responsabilità del controllo della consistenza complessiva può rappresentare un **eccesso** rispetto alla semplice verifica della consistenza degli attributi della **singola classe**.

Ciò diventa particolarmente evidente nei casi in cui buona parte delle operazioni di controllo della consistenza si traducono in coordinamento di più classi con un rischio di eccessivo accoppiamento.

In altre parole questo potrebbe essere un sintomo di una «**responsabilità in cerca di una classe**»: ciò può tradursi nella necessità di creare una nuova classe per il nostro aggregato (la naming convention per questa è «**aggregate root class**» + «**aggregate**»), che potremo chiamare OrdineAggregate.

Aggregate Root Object



Il risultato della creazione di una classe «OrdineAggregate» è essenzialmente una separazione delle responsabilità.

«Ordine» risulta responsabile della consistenza del proprio stato interno.

«OrdineAggregate» si occupa della correttezza dell'aggregato nel suo insieme, senza avere un'immagine sul supporto di persistenza.

Mappiamo il nostro dominio

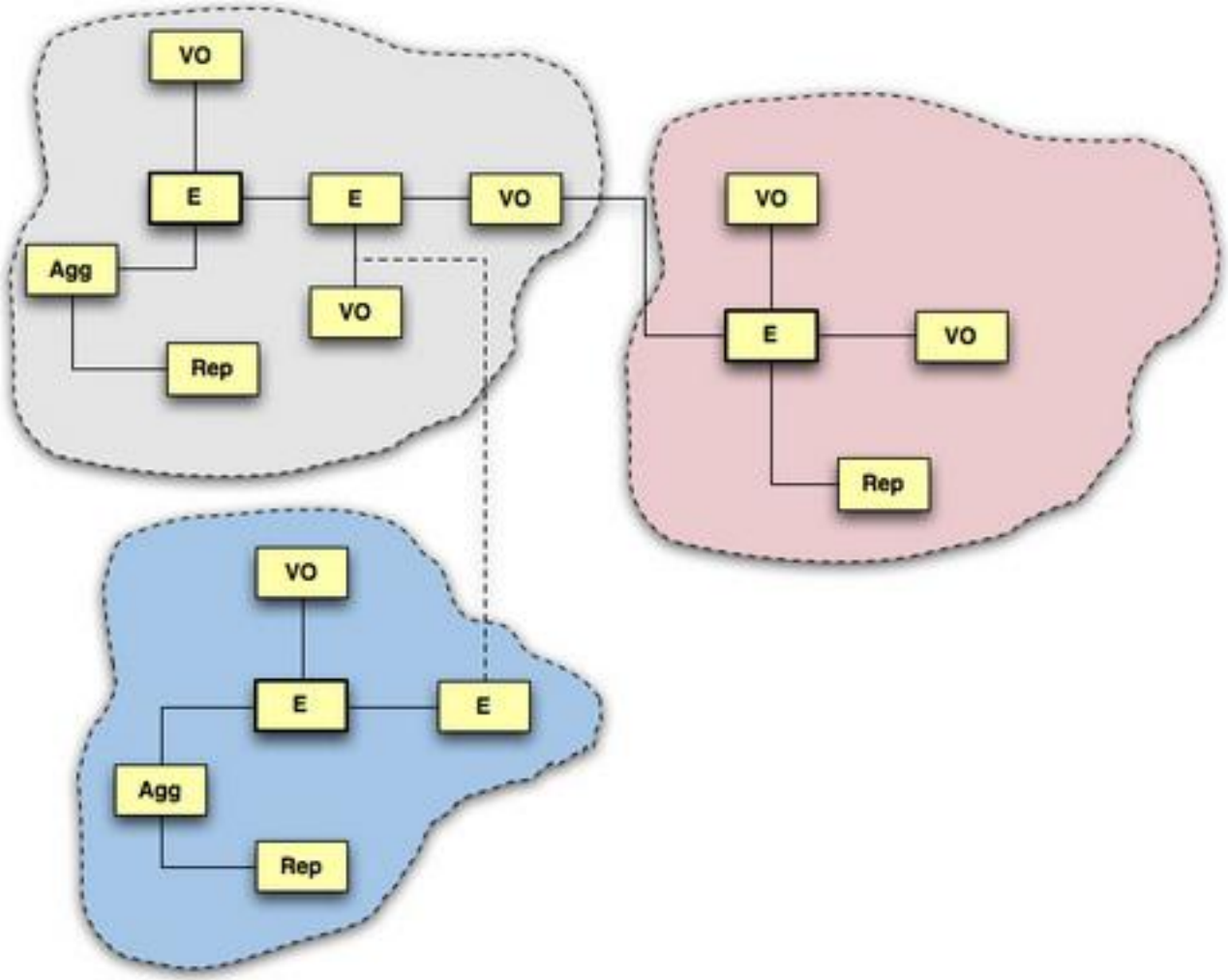
Una caratteristica interessante degli aggregati è che permettono di suddividere il nostro domain model in regioni non sovrapponibili, ciascuna delle quali sarà associata ad un certo numero di casi d'uso.

Si tratta di una scomposizione del nostro dominio in porzioni parzialmente indipendenti.

Esistono dei legami tra una regione e l'altra (altrimenti potrebbe trattarsi di applicazioni differenti), ma dovrebbero coinvolgere solo un numero limitato di casi d'uso.

In altre parole, possiamo «incrociare i dati» con i casi d'uso per verificare che le aree toccate siano consistenti e complete.

Mappiamo il nostro dominio



Una vista complessiva del Domain Model in cui gli aggregate boundaries definiscono delle regioni non sovrapposte.

Questa rappresentazione è utile per verificare che non vi siano presenti classi «orfane» o «condivise» e che i legami tra le classi che scavalcano i confini dell'aggregato non nascondano delle criticità.

In altre parole si tratta anche di una sorta di sanity-check sulla correttezza / completezza del nostro modello di dominio.

Relazione di Aggregazione

Nonostante la similitudine del nome e qualche affinità, l'Aggregate pattern di Domain Driven Design non ha una parentela stretta con la relazione di aggregazione definita da UML.

Nel caso di UML si tratta in effetti di una caratterizzazione di una relazione tra due classi ben definite, mentre nel nostro caso si tratta di un'organizzazione più complessa che coinvolge potenzialmente un maggior numero di classi, eventualmente aperta ad altri tipi di relazione (composizione, associazioni semplici, etc.)

Repositories

Repositories

Per quanto sia comodo poter ragionare facendo finta che il database non esista, a un certo punto è necessario fare i conti con la necessità di interagire con un supporto di persistenza, che renda i nostri dati permanenti.

Volendo essere capziosi, si potrebbe argomentare che la persistenza non è un vero e proprio requisito applicativo, ma una necessità contingente per star sicuri che lo stato dell'applicazione sia garantito anche in presenza di eventi quali cali di corrente o **out-of-memory**, ma la discussione non ci porterebbe molto lontano.

Se l'obiettivo è invece farvi odiare dal vostro DBA questi argomenti sono di provata efficacia.

Repositories

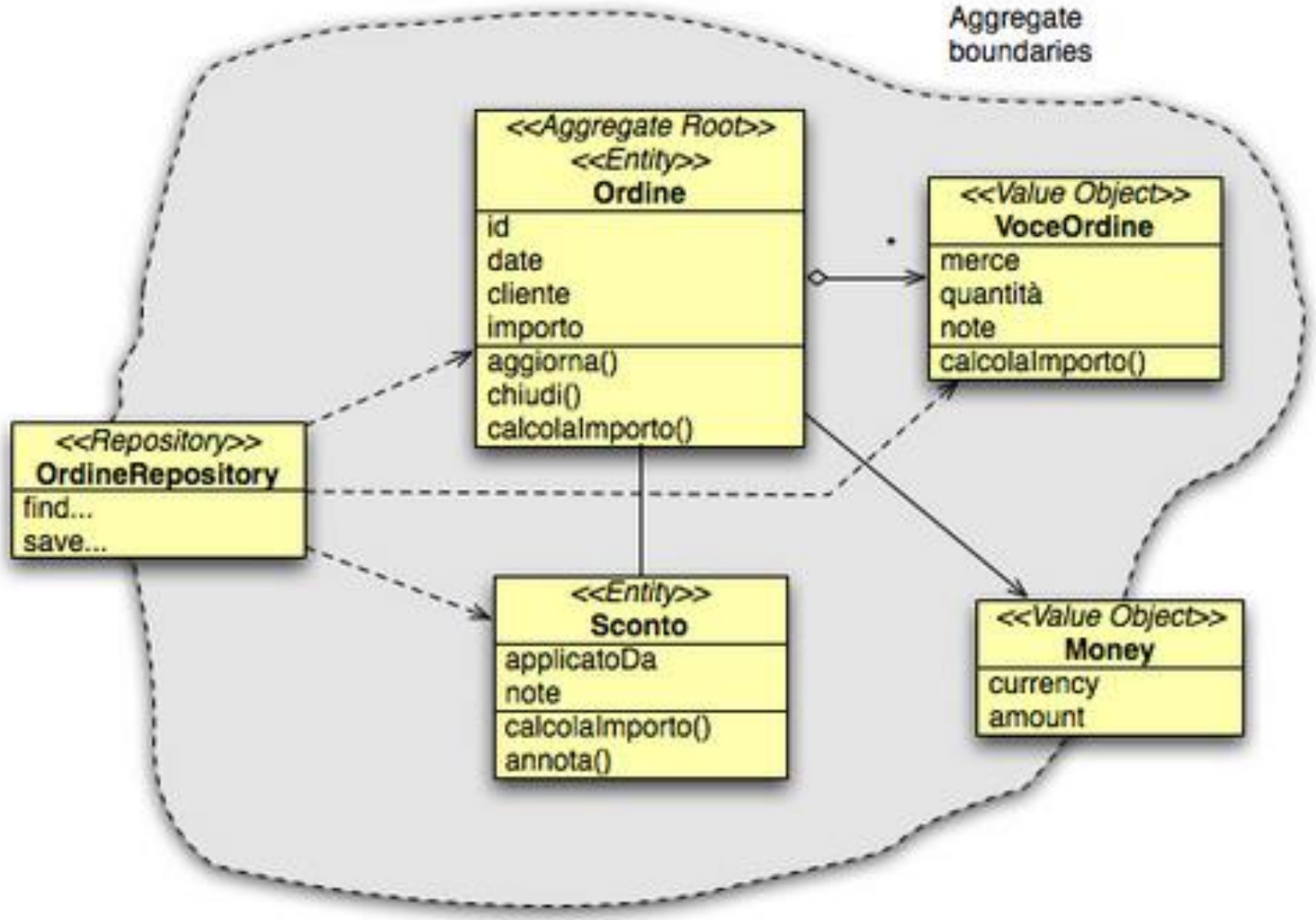
La definizione ufficiale di Repository è «un oggetto che permetta creare l'illusione di una collezione in memoria di tutti gli oggetti di un certo tipo».

In altre parole vogliamo che la nostra modalità di accesso al supporto di persistenza, qualunque esso sia, sia il più possibile libera da riferimenti e dipendenze di tipo tecnologico.

A prima vista, la definizione coincide in larga parte con la tradizionale definizione di DAO e in effetti la matrice originaria non è molto differente.

Il nostro Repository espone i metodi di salvataggio e di ricerca per gli oggetti del nostro aggregato, esattamente come siamo soliti aspettarci da un DAO.

Repositories - Il nostro aggregato corredato di Repository



Collocazione del Repository

In Domain Driven Design, il Repository è parte integrante del Domain Layer, e questo può essere una piccola rivoluzione perché' siamo abituati a considerare le componenti di interfaccia con lo strato di persistenza come appartenenti al Persistence Layer.

Il motivo principale di questa scelta sta nel fatto che le operazioni di data retrieval sono parte integrante della nostra applicazione.

Privilegiare la realizzazione di un modello sul paradigma OOP rispetto al modello dei dati relazionale è una cosa, ma ignorare la presenza del supporto di persistenza, e le potenzialità offerte, specialmente nel caso di operazioni massive, è una scelta decisamente controproducente.

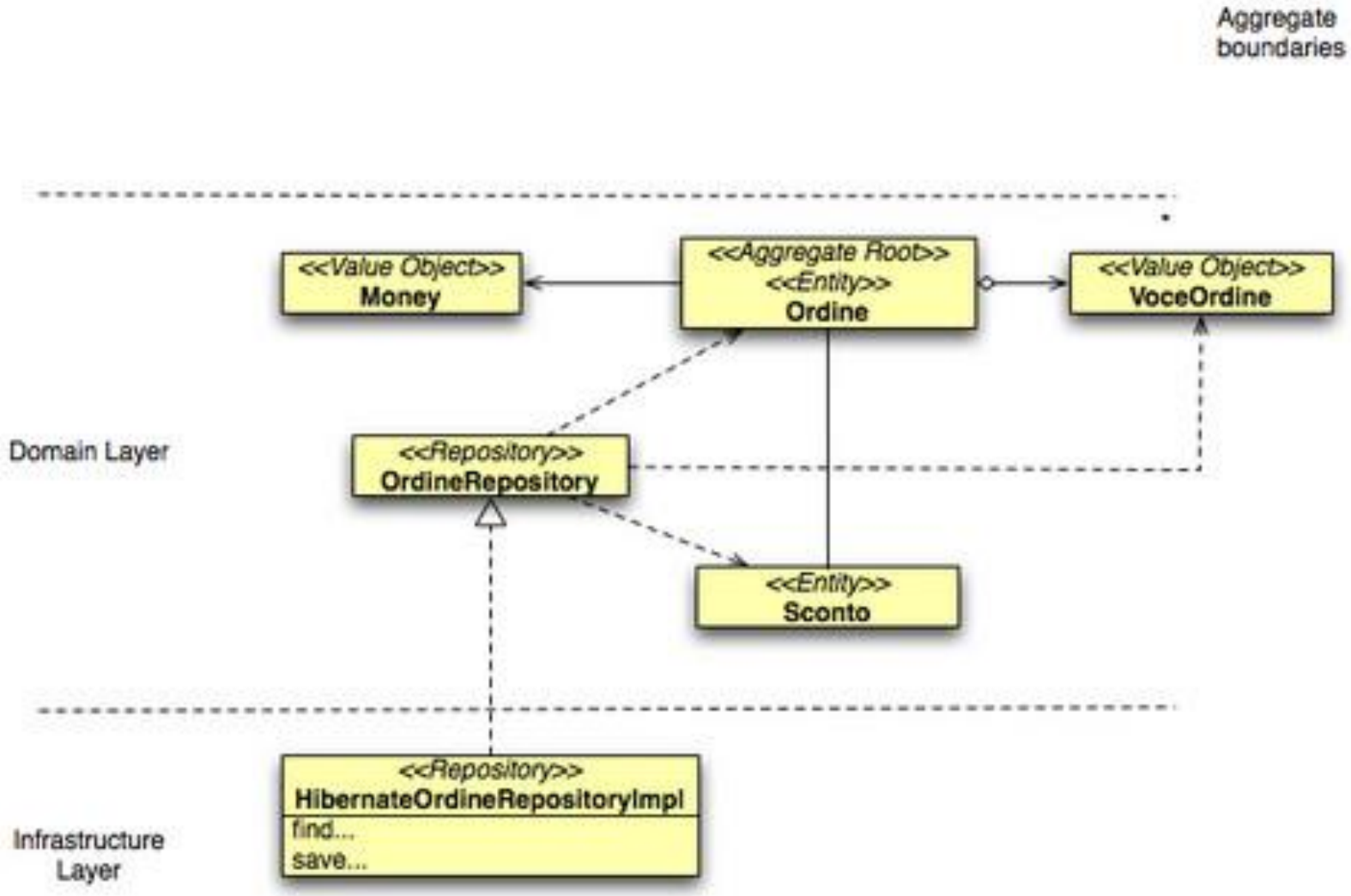
Collocazione del Repository

L'accesso ai dati è una fra le tante operazioni possibili all'interno del nostro Domain Model, sia pure con i vincoli che abbiamo espresso nelle precedenti lezioni.

L'implementazione del nostro repository può essere realizzata in vari modi, ma spesso non possiamo fare a meno di introdurre dipendenze tecnologiche significative (nel caso di Java: siano queste JDBC, JPA, Hibernate etc.)

Collocazione del Repository

La collocazione del repository rispetto all'architettura a strati di Domain Driven Design:



Collocazione del Repository

È interessante notare che al momento di separare interfaccia e implementazione, ci interessa solo che l'interfaccia del repository sia collocata nel Domain Layer, e che di conseguenza sia espressa in termini di oggetti di dominio e quindi dello Ubiquitous Language.

L'implementazione può, e spesso necessariamente deve, essere legata a determinate tecnologie: nel nostro caso abbiamo ipotizzato di appoggiarci a un ORM, ma l'elemento chiave è che la scelta implementativa è confinata nell'infrastructure layer.

Potrebbe anche trattarsi di un semplice file di testo anzichè del DB, ma il Domain Layer risulterebbe comunque consistente.

In realtà, per come sono strutturati i tool ORM, una separazione completa non è facilmente ottenibile al 100%, ma questo fa parte del livello di reversibilità che vogliamo garantire alle nostre scelte architetturali.

Collocazione del Repository

La separazione tra interfaccia e implementazione in questo caso ha un risvolto interessante.

Ci permette di testare il comportamento complessivo del nostro Domain Model in maniera indipendente dal supporto di memorizzazione utilizzato.

È infatti sufficiente realizzare un Mock che implementi il nostro repository rendendo il nostro Domain Model completamente autocontenuto ed indipendente dal supporto tecnologico.

Considerando che uno dei nostri punti di partenza era il desiderio di disaccoppiare l'obsolescenza del nostro modello da quella della tecnologia circostante, si tratta di un risultato da non sottovalutare.

Il corretto livello di astrazione

Nonostante la similitudine con il DAO pattern, esiste qualche vincolo ulteriore rispetto all'implementazione dei DAO che siamo abituati a maneggiare.

Entrambi svolgono un ruolo di separazione tra la rappresentazione nel codice del linguaggio OOP scelto e quella utilizzata per la persistenza.

Tuttavia il DAO spesso espone un'interfaccia fortemente data-oriented, o generica.

Il DAO può essere generato direttamente da qualche framework di persistenza.

Il corretto livello di astrazione

Non è infrequente che questo si traduca in codice del tipo:

```
public class OrderDao {
    public Order findById(Long id) {
        ...
    }
    public void save(Object order) {
        ...
    }
    public List findOrder(String parameters) {
        ...
    }
    ...
}
```

Soluzioni di questo genere hanno il pregio della genericità, e una certa apparente eleganza. Però hanno il problema di porre al di fuori del nostro Data Access Object la responsabilità di tradurre i parametri tipici del dominio nella loro rappresentazione a livello di dati.

Il corretto livello di astrazione

Ad esempio, se la nostra applicazione avesse le necessità di tracciare tutti gli ordini scaduti, il concetto di scaduto in cui

```
dataConsegna < oggi e statoOrdine == inAttesa
```

sarebbe un concetto chiave del nostro dominio.

Una query del tipo

```
SELECT * FROM orders WHERE orders.dataConsegna < getDate() AND orders.statoOrdine = 'inAttesa'
```

sarebbe indissolubilmente legata all'implementazione, mentre una chiamata da codice a

```
orderRepository.findOrdiniScaduti()
```

avrebbe il vantaggio di una maggior leggibilità, e la possibilità di un riuso legato alla semantica, anziché all'implementazione.

Il corretto livello di astrazione

In altre parole è necessario colmare questo gap tra significato logico di un determinato criterio di ricerca e strumenti disponibili nel nostro contesto per implementarlo.

In **Domain Driven Design** è importante poter disporre di costrutti il più vicino possibile al linguaggio naturale, per poter proficuamente collaborare con i nostri Domain Expert, e ciò porta alla necessità di definire metodi di ricerca «**concettuali**».

Non si tratta dell'unico strumento a disposizione per ottenere questo risultato ed in determinate circostanze il gioco può non «**valere la candela**», ma poter disporre di metodi che espongano il proprio significato facilita la crescita della complessità gestibile in progetti di larghe dimensioni.

Il corretto livello di astrazione

Inoltre, la presenza di un metodo di ricerca con una funzione ben precisa all'interno del dominio applicativo può essere corredata (ad esempio grazie al buon vecchio JavaDoc in Java) di una serie di informazioni al contorno, utili a valutare se il suddetto metodo possa essere utilizzabile anche in altri contesti (ad esempio potrebbe avvisarci di eventuali problemi di performance legati alla nostra query).

Aggregati e Repositories

Aggregati e Repositories

Un elemento interessante della scomposizione del dominio in Aggregati è che in modo abbastanza naturale ci troviamo a utilizzare un solo Repository per ogni aggregato.

Si tratta di una strada abbastanza sensata, in quanto permette di raggruppare in un unico punto tutte le chiamate al DB che coinvolgono le classi che fanno parte del nostro aggregato.

Tuttavia, questo approccio a volte finisce per cozzare contro alcune situazioni derivanti da codice generato da framework o da tool di reverse engineering.

Non è infrequente in effetti trovarsi ad avere a che fare con una moltitudine di DAO generati automaticamente, ognuno con i propri metodi CRUD (ovvero quelli facili da generare) secondo la regola "un'entità - un DAO".

La saggezza popolare in questo caso ci fornisce avvertimenti ambigui: "a caval donato non si guarda in bocca" ci farebbe propendere verso l'ipotesi di ringraziare per tanta abbondanza e usare i nostri DAO a testa bassa, mentre "non accettare caramelle dagli sconosciuti" dovrebbe metterci sull'avviso che non sempre ciò che è gratis è necessariamente buono.

Si tratta in definitiva di una valutazione costi-benefici che va effettuata caso per caso: sul testo di Domain Driven Design, una delle frasi più pragmatiche in effetti è «**Don't fight your frameworks**».

Però non dobbiamo dimenticarci che non tutto ciò che è gratis è necessariamente utile, anzi spesso si tratta di cose che non usiamo e che finiamo per ritrovarci tra i piedi.

DAO generati automaticamente risultano utili per la costruzione di «**CRUD applications**», mentre DDD è un approccio utile per contesti di maggior complessità.

In definitiva è possibile avere un ventaglio di soluzioni, di differente complessità:

- **il nostro Repository è un DAO (magari con qualche metodo in più);**
- **il nostro Repository chiama il DAO tecnologico occupandosi di nasconderne i dettagli al Domain Model, disaccoppiando modello logico e modello fisico dei dati;**
- **il nostro Repository svolge il ruolo di Facade per lo strato di persistenza, coordinando e smistando le chiamate su più DAO.**

La scelta di quale schema applicare è legata a quanto viene offerto dai nostri framework, dalla complessità del nostro dominio (quante interazioni coinvolgono più entità o più aggregati) e da quale livello di disaccoppiamento vogliamo raggiungere nei confronti della nostra piattaforma tecnologica.

Conclusioni

Conclusioni

Aggregati e Repositories sono forse gli elementi più importanti per l'applicazione dei principi di small-scale Domain Driven Design.

L'applicazione sistematica di questi pattern e dei principi di ragionamento che ci stanno dietro, porta a semplificazioni notevoli nella struttura del domain model offrendo un principio di costruzione consistente e scalabile: poche semplici regole permettono infatti di non trovarsi mai in situazioni intricate e di difficile risoluzione.