Taccuino:	Agile		
Creato:	08/10/2019 10.53	**Aggiornato:**	03/01/2020 10.35
Autore:	Domenico Fico		

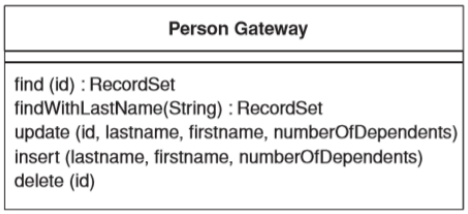
DataSource Architectural Patterns

Ci sono quattro tipo di soluzioni il cui obiettivo è collegare cose che in genere sono diverse;

- Table Data Gateway;
- Row Data Gateway;
- Active Record;
- Data Mapper;

L'obiettivo comune è far si che la logica possa comunicare con il db, separando l'accesso al db dalla logica.
Il problema di includere le query nella parte di logica è che la comprensione del codice è accessibile solo ad un fullstack engeneer.
Separando le due parti invece si avrà una parte di codice java molto semplice comprensibile a tutti e una parte di SQL che può essere rivista anche da un database developer.

Table Data Gateway
"E' un oggetto che agisce come un gateway verso una tabella di una base di dati.
Una sua istanza gestisce tutte le righe di una tabella"
Un TDG racchiude tutto il SQL per accedere ed interagire con una singola tabella. Le classi esterne chiameranno i suoi metodi per interagire con il database.
Come funziona TDG?
Ha una semplice interfaccia che consiste in diversi metodi *FIND* per ottenere i dati dal db e altri metodi per effettuare operazioni del tipo Update, Insert e Delete.
Effettua un mapping con la parte SQL partendo dall'input del metodo.
E' stateless.
Come tipo di ritorno ha una semplice struttura dati come una mappa.
La maggior parte delle volte, si hanno diversi oggetti TDG mappati a diverse tabelle del db, ma nei casi più semplici è anche possibile avere un unico oggetto TDG che riesce a gestire tutti i metodi di tutte le tabelle, in grado così di accedere all'intero db.
Può anche essere presente un metodo findRow per accedere ad una singola riga.
Quando usare TDG?
Quando abbiamo bisogno di fornire alla logica un unico punto d'accesso al db.
Il TDG è il pattern più semplice per fornire questo tipo di interfaccia.
Riesce a rendere trasparente l'uso delle *stored procedures* nascondendo la struttura delle tabelle sottostanti.
E' anche conosciuto come *DAO - Data Access Object*.




```
class PersonGateway {
    public IDataReader FindAll() {
        String sql = "select * from person";
        return new OleDbCommand(sql, DB.Connection).ExecuteReader();
    }
    public IDataReader FindWithLastName(String lastName) {
        String sql = "SELECT * FROM person WHERE lastname = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter("lastname", lastName));
        return comm.ExecuteReader();
    }
    public Object[] FindRow(long key) {
        String sql = "SELECT * FROM person WHERE id = ?";
        /*  Costruzione risultato  */
        return result;
    }
}
```

Row Data Gateway

"E' un oggetto che agisce come un gateway, ma per una solo riga del db non per l'intera tabella."

E' un'alternativa al TDG.

L'oggetto restituito dal RDG è esattamente l'oggetto presente nella base di dati ma può essere manipolato tramite i metodi propri del linguaggio di programmazione. Peggiora la complessità dei test a causa dei continui accessi alla base di dati.

Come funziona RDG?

Ogni riga mappata tramite il RDG viene usata come un oggetto e ogni sua colonna diventa un campo.

Dove mettere le operazioni di find?

E' possibile usare diversi metodi statici di find, ma facendo così si perde la possibilità di applicare il polimorfismo per adattare diverse data source. Il secondo metodo, anche quello più usato, è quello di abbinare una classe Finder e una classe Gateway per ogni tabella.

Quando usare RDG?

Non è una buona scelta se il mio dominio è modellato ad oggetti infatti può essere usato un *Data Mapper*.

Una sua estensione è chiamata *Active Record*.

Nell'implementazione la grande differenza è che c'è bisogno di avere una variabile privata per ogni colonna della riga della tabella.

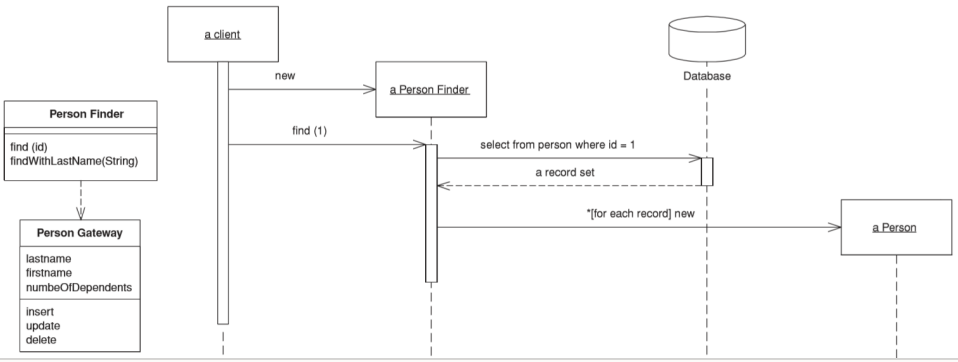
E in caso dei getter e i setter c'è un metodo per ogni campo.

Nota che il metodo find della classe finder ritorna un gateway.

Il gateway diventerà un campo (data) dell'oggetto reale.

Non è un buon modello in quanto duplica l'oggetto reale.

Ad esempio: i campi nome, cognome, etc, non saranno nell'oggetto Persona (che avrà solamente un campo PersonaGateway), ma nell'oggetto PersonaGateway.




```
class Person {
    private PersonGateway data;
    public Person(PersonGateway data) {
        this.data = data;
    }
    public int getNumberOfDependents() {
        return data.getNumberOfDependents();
    }
}
```

```
class PersonFinder {
    private final static String findStatementString = "SELECT id,
lastname, firstname, number_of_dependents " + " from people "
                                                + " WHERE id = ?";
    public PersonGateway find(Long id) {
        PersonGateway result = (PersonGateway) Registry.getPerson(id);
        if (result != null) return result;
        PreparedStatement findStatement = null;
        ResultSet rs = null;
        try {
            findStatement = DB.prepare(findStatementString);
            findStatement.setLong(1, id.longValue());
            rs = findStatement.executeQuery();
            rs.next();
            result = PersonGateway.load(rs);
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(findStatement, rs);
        }
    }
}
```

```
class PersonGateway {
    private String lastName;
    private String firstName;
    private int numberOfDependents;
     /*  Setter e Getter  */
}
```

Active Record

"Il problema di non mettere la business logic nel gateway è una pratica dei puristi, se non ci frega nulla di tutto ciò esiste l'ActiveRecord". [Ricca docet]

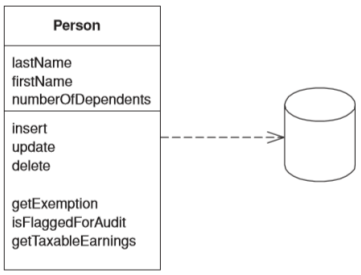
E' un oggetto che incapsula l'accesso al database e aggiunge la logica del dominio.

Come funziona l'AR?

Essenzialmente, l'AR è un domain model le cui classi corrispondono fedelmente alla struttura del database sottostante.

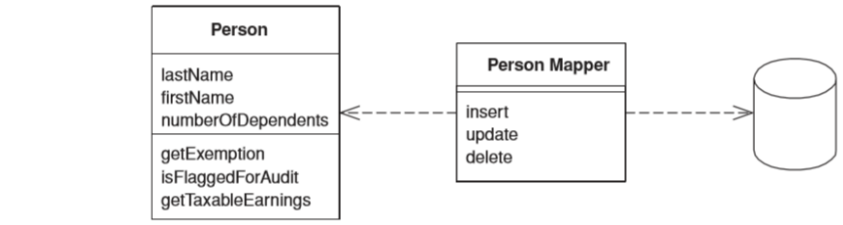
Una classe di AR tipicamente ha i metodi per: creare un'istanza a partire da una query, creare una nuova istanza che verrà successivamente caricata nel db, aggiornare un'istanza esistente con conseguente aggiornamento del db, vari metodi statici di find, setter e getter, e altri metodi di logica del dominio.

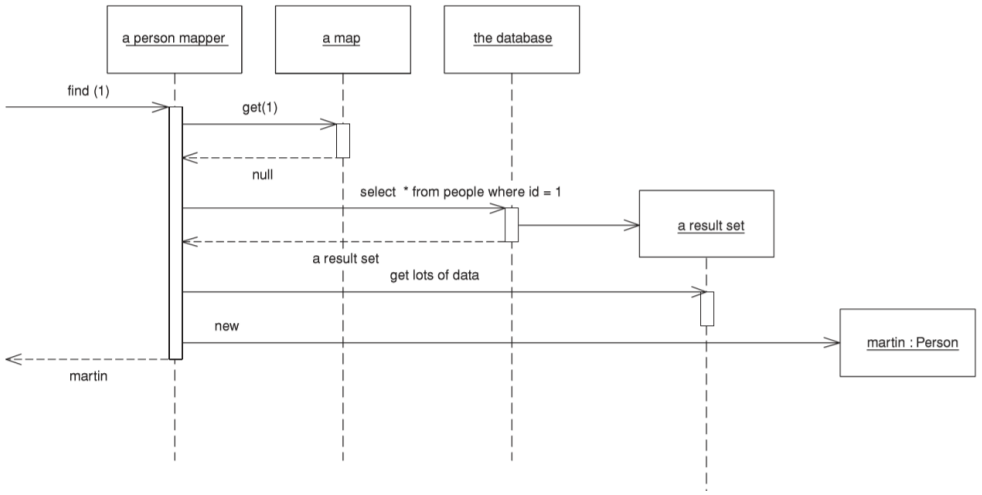
Questo pattern non prova minimamente a nascondere la presenza di una base di dati relazionale sottostante.
L'AR è molto simile al RDG, la principale differenza è che RDG contiene solamente l'accesso al db mentre l'AR la mischia con la logica.
Questa sua caratteristica di avvicinare così tanto la logica e l'accesso al db peggiora la leggibilità del codice rendendo difficile la manutenzione.
Quando usare AR?
L'AR è una buona scelta quando la domain logic non è molto complessa.
Il principale vantaggio dell'AR è la semplicità: è semplice da capire e da implementare. In più riduce il codice ridondante (vedi RDG).
Ma se la logica di dominio inizia a diventare più complessa e c'è il bisogno di implementare ereditarietà, relazioni, collezioni, etc, invece di continuare ad usare AR rendendo tutto disordinato e incomprensibile, una buona scelta è passare al Data Mapper.



Data Mapper
"*Un livello di mappatori che sposta i dati tra oggetti e un database mantenendoli indipendenti l'uno dall'altro e dal mappatore stesso.*"
La sua responsabilità è quella di trasferire dati tra la logica e il db e isolarli tra loro. Gli oggetti in memoria non hanno nemmeno bisogno di sapere che è presente un database.
Come funziona DM?
Il pattern si occupa principalmente di separare il dominio dal datasource.
Una semplice implementazione del DM mappa una tabella del db con un oggetto del dominio campo-a-campo.
Poiché è possibile che gli oggetti siano molto interconnessi tra loro, c'è bisogno di un modo per minimizzare le query al db, ad esempio usando un *Lazy Load*.
In base all'applicazione che si sta sviluppando può esserci la necessità di avere più DM, questo perché in una grande applicazione potrebbero esserci troppi metodi *FIND* per un singolo DM, di conseguenza ha molto senso splittare questi metodi in base alla classe. In questo modo si avranno tante piccole classi (che faciliteranno di molto la vita agli sviluppatori durante la manutenzione), organizzate in una gerarchia oppure fornendo ogni classe del dominio di un proprio DM. Aiutandosi con il *Metadata Mapping* è più semplice eseguire questo passaggio verso un sistema multi-DM. Inoltre sarebbe utile dotare ogni finder di un *Identity Map* per tenere traccia degli oggetti letti dal db.
Esempio: caso semplice in cui abbiamo le classi Person e Person Mapper.
Per caricare una persona dal db deve chiamare il metodo find sul Mapper. Il Mapper usa un *Identity Map* per verificare che sia già stato caricato, in caso contrario fa la richiesta al db.
L'intero livello del DM può essere facilmente sostituito anche per consentire ad un singolo oggetto del dominio di lavorare con un db differente.
Quando usare DM?

Quando voglio che lo schema del db e il domain model possano evolvere separatamente. Questo è possibile perché gli oggetti del dominio non hanno nessuna idea di come possa essere strutturato il db, tutto il lavoro è fatto dai mappers.
Il prezzo da pagare implementando il DM è quello di avere un livello extra nel progetto.
Nota:
Business Logic semplice -> Active Record
Business Logic più complessa -> Domain Model + Data Mapper





Distribution Patterns

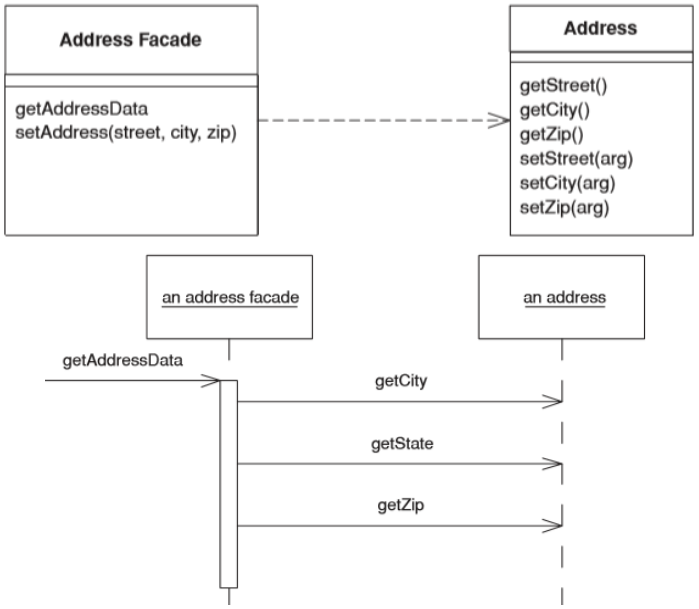
- Remote Facade;
- Data Transfer Object;

Remote Facade
"*Fornisce una facciata a grana grossa ad un oggetto a grana fine per migliorare l'efficienza su una rete*"
Semplifica l'accesso ad un livello che è molto più complesso, migliorandone l'efficienza.
Non solo per semplificare l'accesso, ma per ridurre le chiamate alla tabella e fare update in massa.
La RF non contiene nulla della logica del dominio.
Il compito principale è quello di tradurre metodi a grana grossa in quelli sottostanti.
Come funziona RF?
Tutta la logica è posta all'interno degli oggetti a grana fine che sono progettati per lavorare insieme. Avere a disposizione un oggetto Facciata separato che agisce come un'interfaccia remota consente di gestire gli accessi remoti in modo efficiente.
Semplifica l'accesso e crea metodi che fanno più cose contemporaneamente per ridurre il carico.

Quando viene invocato un getter della RF vengono generate molteplici chiamate ai getter dell'oggetto reale.
Tutti i getter e setter di un oggetto reale vengono rimpiazzati da un getter e un setter.
Può essere sia stateful che stateless.
La granularità della RF rappresenta un piccolo ostacolo durante la progettazione di un sistema. Si può scegliere se avere una grande RF con tanti metodi, oppure una RF per singolo caso d'uso o una via di mezzo tra le due. Dipende molto da che tipo di sistema si sta progettando.
E' progettata per rendere facile la vita agli utenti esterni non al sistema interno.
RF non ha la transaction script, semplifica l'accesso alla domain logic ma non contiene la domain logic

Quando usare RF?

Quando si ha bisogno di accedere ad un oggetto a grana fine, mantenendo i vantaggi d'uso di un'interfaccia a grana grossa (meno chiamate). Un uso comune di questo pattern è tra un presentation layer e un domain model, entrambi posti su processi differenti.
Non è mai usato con transaction script per che è coarser grained.



Data Transfer Object

"*Trasporta i dati tra processi con lo scopo di ridurre il numero di chiamate a metodi*"
E' un oggetto che racchiude tutti i dati da trasferire con la chiamata. Ha bisogno di essere serializzabile per viaggiare tramite una connessione. Di solito, lato server, è usato un assembler per costruire il DTO con tutti i dati da inviare.

Come funziona DTO?

"E' uno di quegli oggetti che le nostre madri ci vietano di scrivere"
Nel momento in cui un oggetto remoto fa richiesta di qualche dato, sta chiedendo un DTO adatto, ma di solito il DTO porterà molti più dati di quanto richiesto. A causa della latenza, che incrementa di molto il costo per chiamata, risulta essere molto meglio inviare molti dati in una richiesta piuttosto che fare molteplici richieste.
Gli oggetti del dominio che vengono trasmessi con DTO devono essere semplificati rispetto a quelli che sono effettivamente nel dominio. Infatti i campi che deve avere un oggetto DTO devono essere primitivi, stringhe o date,

oppure altri oggetti DTO. Questo perché devono essere serializzabili e comprensibile da entrambi i lati della connessione.
Oltre ai semplici metodi get e set il DTO è anche responsabile della serializzazione di se stesso in un qualche formato capace di viaggiare lungo una connessione. Il DTO non conosce il modo di connettersi all'oggetto corrispondente nel dominio perché, essendo posizionato in entrambi i lati della connessione, non è conveniente che sia dipendente dall'oggetto del dominio (le eventuali modifiche al dominio o al DTO dovrebbero essere indipendenti).
Come risultato avremo che il responsabile della creazione e modifica dei DTO sarà un oggetto esterno chiamato Assembler.
Figura
Quando usare DTO?
Quando abbiamo bisogno di trasferire diversi oggetti tra due processi in una singola chiamata ad un metodo.
Può agire come una sorgente comune di dati per vari componenti in layer diversi.