Appunti di
Ingegneria del Software

A cura di:
Francesco Refolli
Matricola 865955

Anno Accademico 2022-2023

Part I

Architectural Patterns

Chapter 1

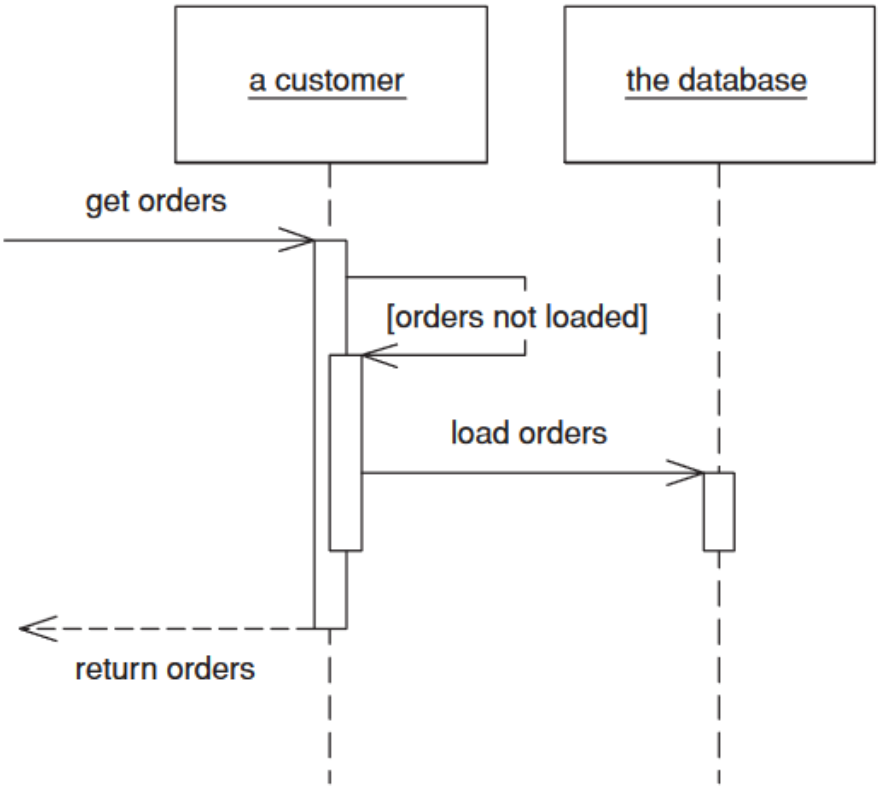
Lazy Load

1.1 Descrizione

"An object that doesn't contain all of the data you need but knows how to get it" - Martin Fowler

Quando una Classe deve mantenere memorizzati dei dati, e' utile che essi siano caricati tanto tardi quanto e' bassa la probabilita' che si ricorra ad essi. Ovvero si rimanda il caricamento dei dati alla prima occorrenza di utilizzo. Per ottenere questo e' necessario che essa possa ottenere o popolare i dati che gli spettano.

1.2 Vista UML



1.3 Vantaggi

Il vantaggio principale riguarda le prestazioni e in particolare la velocita' con cui una classe o istanza viene caricata in memoria. Se meno dati vengono precaricati (perche' al momento inutili) allora minore sara' il carico di lavoro dei calcolatori che la elaborano. Inoltre se l'istanza verra' eliminata dalla memoria senza che quei dati vengano acceduti, sara' possibile risparmiare anche sul tempo di eliminazione dei dati. Bisogna altresi' considerare che questo determina anche un uso piu' ottimizzato della memoria: per tutto il tempo in cui non sara' necessaria la risorsa A, ci sara' piu' spazio in memoria per utilizzare il resto delle risorse.

1.4 Svantaggi

Si supponga ora che la classe abbia come responsabilita' la manipolazione dei dati e il caricamento lazy degli stessi. Il Single Responsibility Principle prescrive che ogni classe abbia solo una responsabilita', in modo tale che abbia meno motivi possibili per diventare obsoleta. Se si vuole rispettare questo principio occorre esportare la responsabilita' di caricamento lazy ad una classe di supporto. Questo pero' genera necessariamente accoppiamento a seguito di questa nuova dipendenza. E' quindi essenziale effettuare una scelta ponderata che tenga conto delle priorita'.

1.5 Quando Usarlo

Questa strategia e' utile quando i dati non sono immediatamente necessari per l'elaborazione e quando per recuperarli servono molteplici operazioni rimandabili. Quando una operazione deve essere fatta e' utile caricare tutti i dati che essa possa caricare per evitare di doverla ripetere. E' quindi consigliato evitare di inserire Lazy Load multipli a meno che non si tratti di dati eterogenei e ulteriormente rimandabili. Quando si vuole gestire dall'interno una inizializzazione onerosa e complessa e' di norma possibile applicare senza troppi problemi questo pattern per articolare questa fase senza sbilanciare il carico dell'applicativo.

Chapter 2

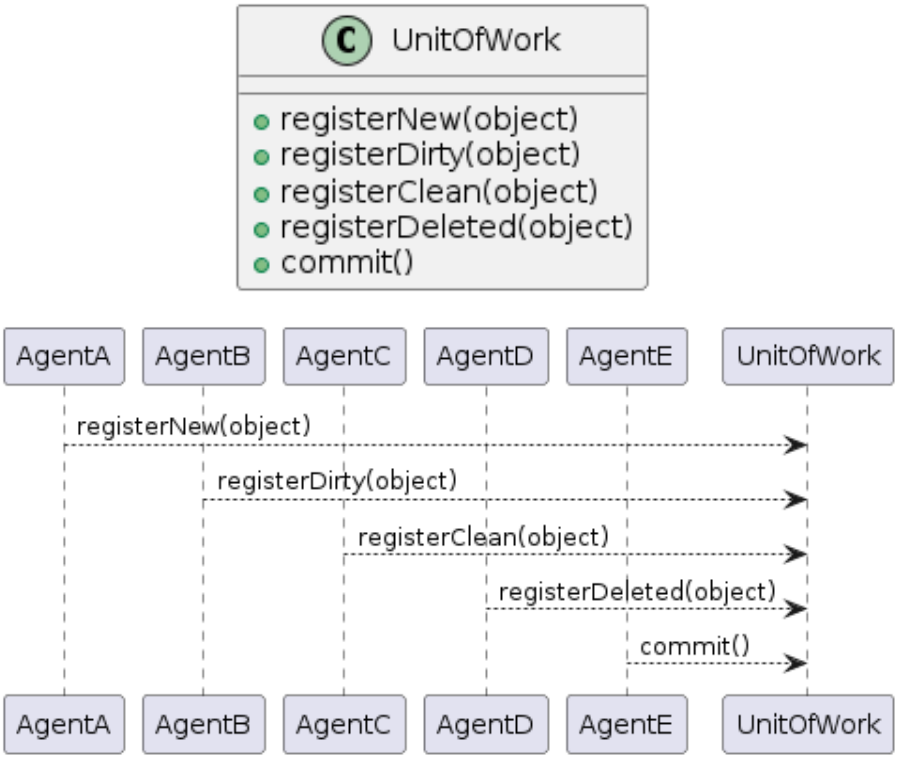
Unit of Work

2.1 Descrizione

"Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems" - Martin Fowler

Quando si hanno dei dati che devono venire progressivamente aggiornati in piu' parti di un applicativo ed e' importante tenere traccia di tutte le modifiche, assicurarsi che avvengano ciascuna correttamente senza conflitti dettati dalla concorrenza, e' possibile delegare le operazioni ad un unico soggetto che si occupi di gestire l'aggiornamento del sistema in modo sicuro ed efficiente.

2.2 Vista UML



2.3 Vantaggi

Centralizzare il meccanismo di aggiornamento dei dati permette di sfruttare in modo piu' efficiente le operazioni onerose, come le interazioni con il Database. Invece di avere tante piccole chiamate di scrittura su Database si puo' averne poche, o una sola, piu' pesanti. Apparentemente potrebbe sembrare una peggior soluzione, tuttavia molti, se non tutti, Database consentono cambiamenti in batch piu' veloci rispetto a singole operazioni.

I Database consentono l'uso concorrente delle loro API, tuttavia con questo pattern e' possibile controllare attivamente la concorrenza per risolvere o evitare i conflitti direttamente nell'applicativo. Questo consente di non dipendere dalla stabilita' del modulo che si occupa di concorrenza del Database scelto e favorisce l'intercambiabilita' dello stesso.

2.4 Svantaggi

Questo modulo avra' un accoppiamento necessariamente molto alto, in quanto verra' usato da tutto il sistema per aggiornare i dati. Naturalmente questa centralizzazione si porta con se' non pochi problemi. Si e' creato un Single Point of Failure. Il modulo avra' bisogno di particolare attenzione e manutenzione.

2.5 Quando Usarlo

Quando i dati sono distribuiti in tutto il sistema, vengono aggiornati con alta frequenza da piu' moduli o classi e' utile applicare questo pattern per migliorare le prestazioni.

Part II

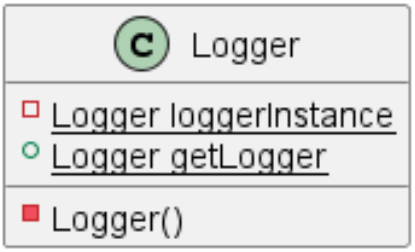
Architectural Patterns Applications

Chapter 3

Lazy Load

Fowler fornisce quattro implementazioni per questo pattern.

3.1 Lazy Inizialization



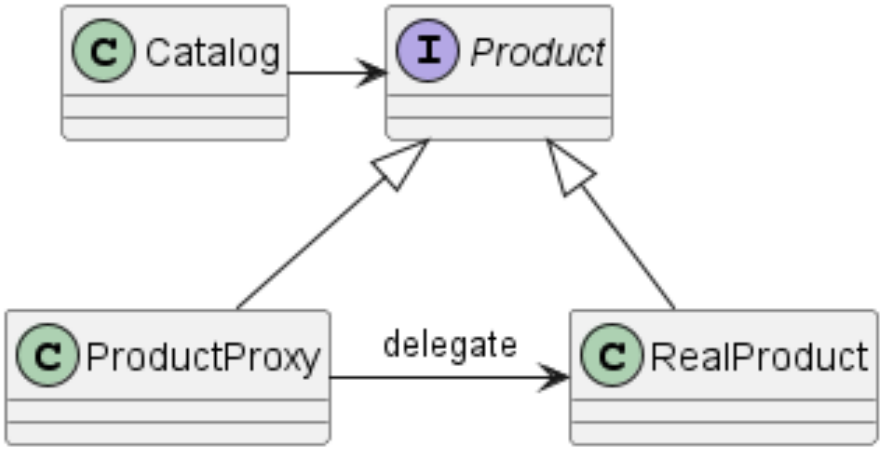

```
public class Logger {
    private static Logger loggerInstance = null;

    private Logger() {
        /* ... constructor ... */
    }

    public static Logger getLogger() {
        if (loggerInstance == null)
            loggerInstance = new Logger();
        return loggerInstance;
    }
}
```

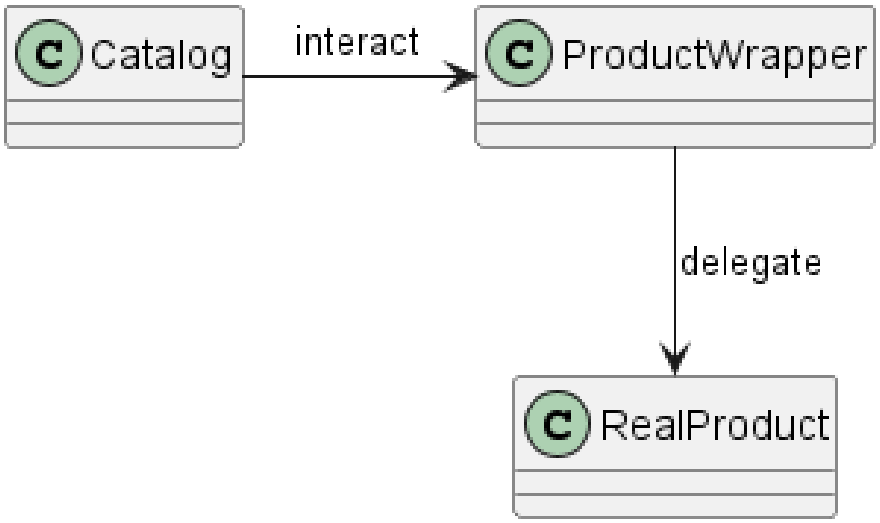
In questa implementazione si controlla la disponibilita' del dato ed eventualmente si istanzia. Quindi viene ritornato.

3.2 Virtual Proxy



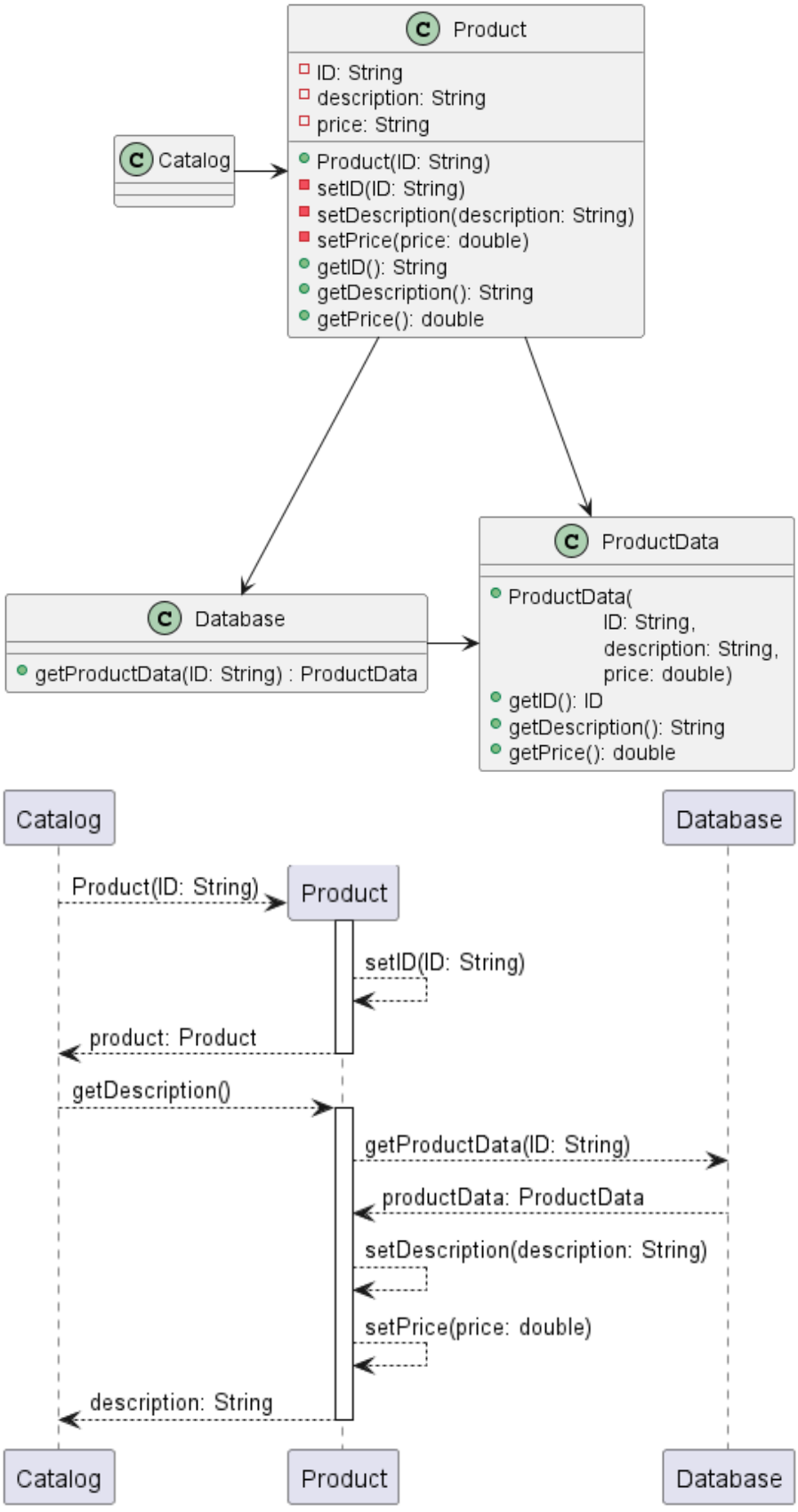
Si crea una astrazione con una interfaccia e un intermediario che delega al RealProduct gli obiettivi di Product e si occupa soltanto di fornire all'occorrenza i servizi del RealProduct al Catalog.

3.3 Value Holder



Si usa un ProductWrapper che assume tutte le responsabilita' che aveva il Product-Proxy La differenza con il Virtual Proxy e' che in questo caso il Catalog e' cosciente di stare interagendo con il ProductWrapper.

3.4 Ghost



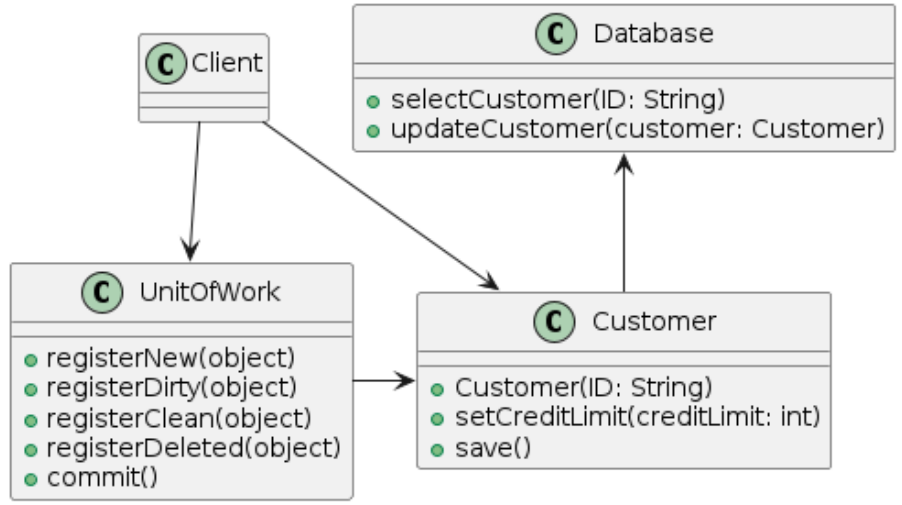
Il Ghost qui' e' la classe Product, la quale e' un oggetto che quando viene instanziato

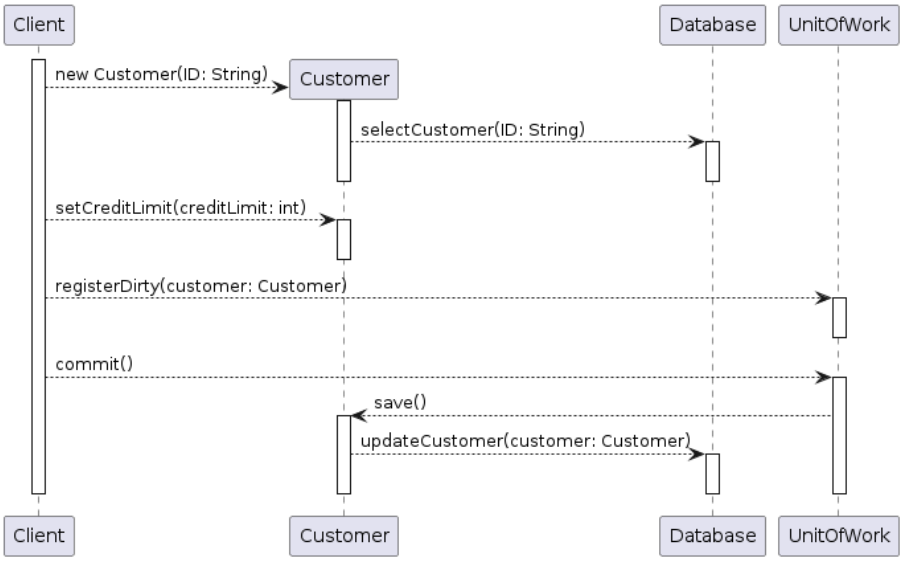
contiene solo parte delle sue informazioni (ID), ma quando richiesto carica il resto delle informazioni. Si noti che la classe ProductData e' immutabile, ed utilizzata solo per pattare informazioni tra il Database e il Product.

Chapter 4

Unit of Work

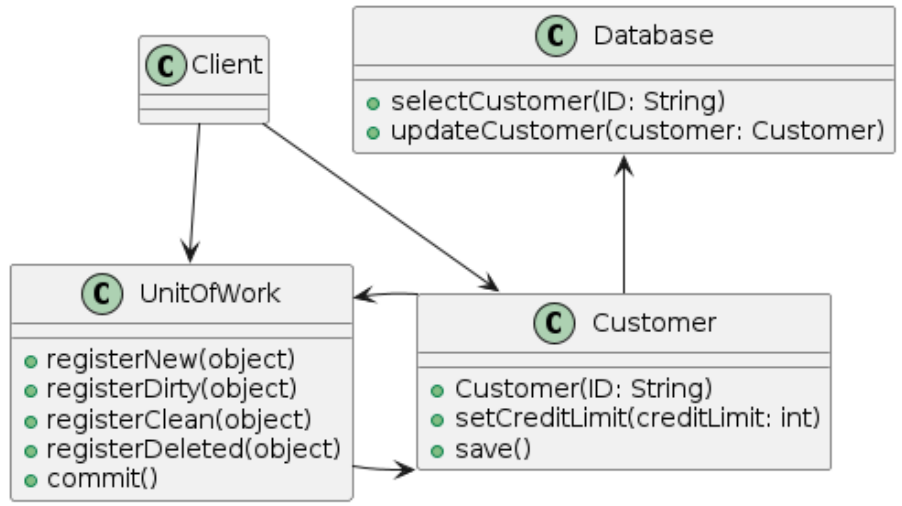
4.1 Caller Registration

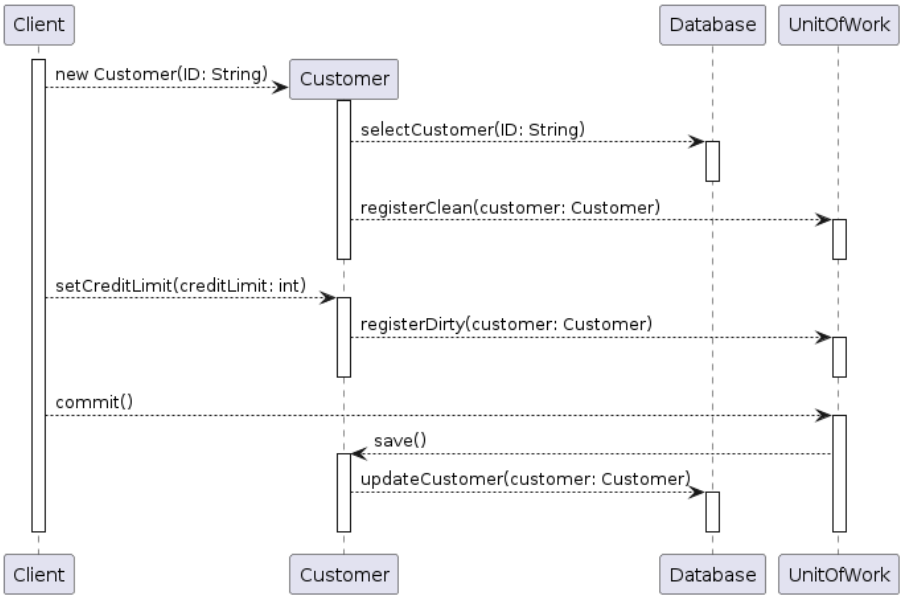




Qui il Client ha il compito di informare UnitOfWork che l'oggetto Customer ha subito un cambiamento. Il Client effettua il commit quando lo ritiene oppurtuno.

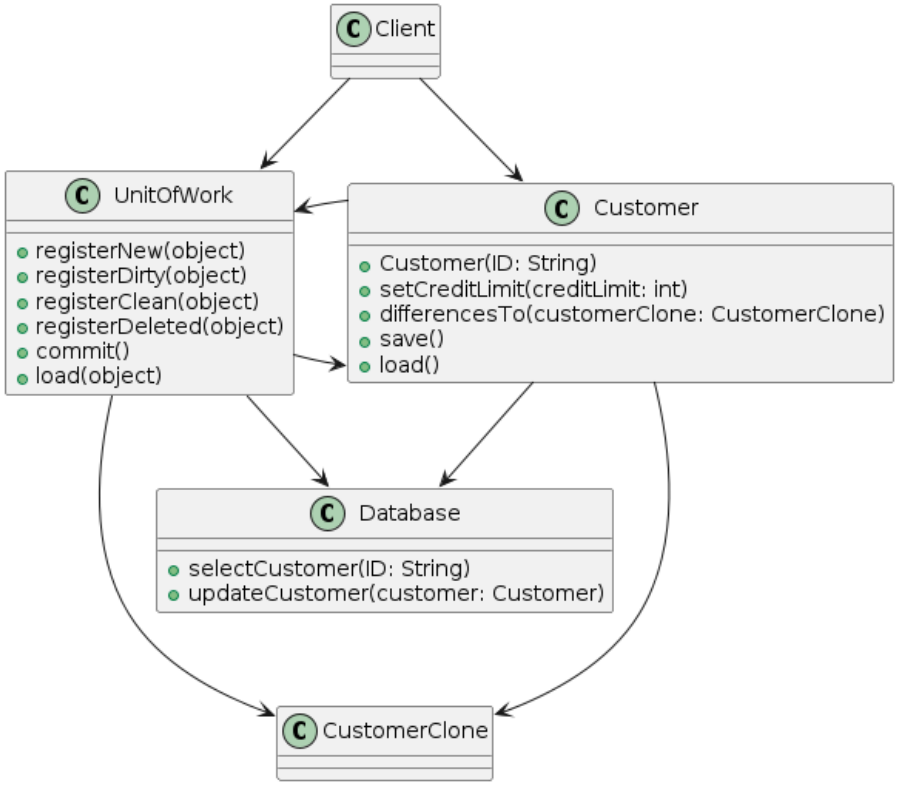
4.2 Object Registration

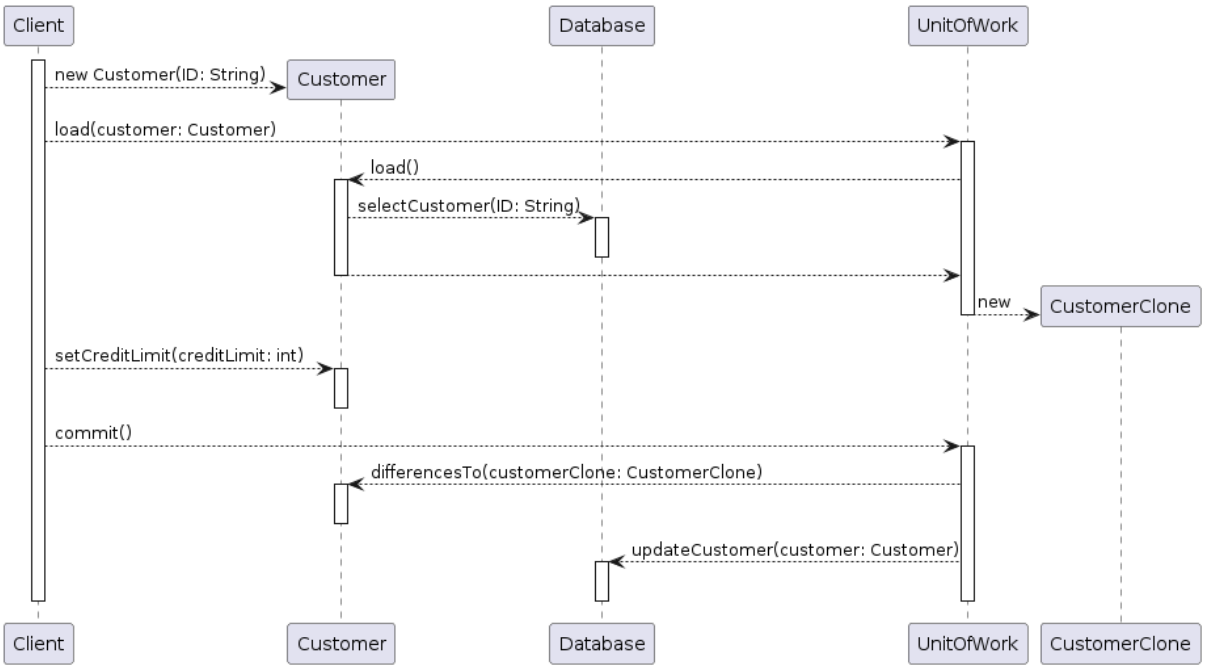




Qui invece e' l'oggetto Customer a registrare i propri cambiamenti al Il Client effettua il commit quando lo ritiene oppurtuno.

4.3 Controller





Qui UnitOfWork e' notificata solo al momento della creazione dell'oggetto e si occupera' successivamente di controllare se un oggetto presenta cambiamenti che necessitano di essere salvati. Si noti che qui UnitOfWork interagisce attivamente con il Database per aggiornare i dati. E' qui che essa ha l'opportunita' di gestire in modo ottimizzato le operazioni di scrittura. Il Client effettua il commit quando lo ritiene oppurtuno.