Dokumentacja projektu zaliczeniowego z przedmiotu Design Patterns

Projekt oraz implementacja programowego mechanizmu load-balancing na warstwie mapowania O-R. Technologia .Net

Karolina Pieszczek, Anna Wójcik, Sylwia Zoń, Alicja Brajner

semestr zimowy 2019/2020

sebalance

ORM - Object-Relational Mapping

ORM (Mapowanie obiektowo-relacyjne) jest techniką programowania używaną do konwertowania danych pomiędzy relacyjnymi bazami danych a programem napisanym w języku obiektowym. Pozwala na bezpośrednią interakcję z bazą danych używając obiektów stworzonych w języku obiektowym.
Mapowanie obiektowo-relacyjne korzysta z tzw. “Data Layer” (w większości przypadków jest to biblioteka napisana w języku obiektowym). Warstwa ta jest tłumaczem pomiędzy językiem obiektowym a bazą danych. Posiada kod, który odpowiedzialny jest za tworzenie kwerend oraz zapewnia zbiór funkcji pozwalających na interakcję z bazą danych.

NHibernate

NHibernate jest rozwiązaniem ORM dla platformy .NET. Dostarcza środowisko do mapowania obiektowo-relacyjnego dla tradycyjnego, relacyjnego modelu bazy danych. Jego podstawową funkcją jest mapowanie z klas platformy .NET do tabeli baz danych oraz od typów danych CLR do typów danych SQL.

Load Balancing

Coraz częściej jeden serwer nie potrafi udźwignąć obsługi dużej liczby zapytań przychodzących od użytkowników sieci. Dlatego korzysta się teraz z techniki **równoważenia obciążenia** (ang. **load balancing**). Polega ona na rozpraszaniu obciążenia pomiędzy wiele procesorów, komputerów, dysków, połączeń sieciowych lub innych zasobów.
Rolę kluczową gra tutaj **Load Balancer.** To on rozdziela zapytania przychodzące od użytkowników do rzeczywistych serwerów. Niezależnie od tego, jaki serwer w klastrze zostanie wybrany, użytkownik powinien uzyskać identyczne odpowiedzi.

Trójwarstwowa architektura równoważenia obciążenia składa się z :

1. Load Balancer ,
2. klaster serwerów rzeczywistych, na których uruchomione zostają usługi pracujące w sieci,
3. pamięć masowa i bazy danych, które są współdzielone przez serwery pracujące w klastrze, umożliwiające udostępnianie identycznych danych oraz usług.

Działanie

1. Wiele użytkowników wysyła wiele zapytań w tym samym czasie.
2. Trafiają one do Load Balancera.
3. Load Balancer analizuje obciążenia na poszczególnych dostępnych serwerach.
4. Dokonuje optymalnego wyboru .
5. Zapytanie użytkownika zostaje wysłane do wybranego serwera .

Zalety

- Maksymalizacja wydajności
- Brak problemu w przypadku awarii - możliwość natychmiastowego przekierowania danego procesu na inny, działający serwer
- Komfort pracy dla administratora

Wykorzystywane wzorce projektowe

Obserwator

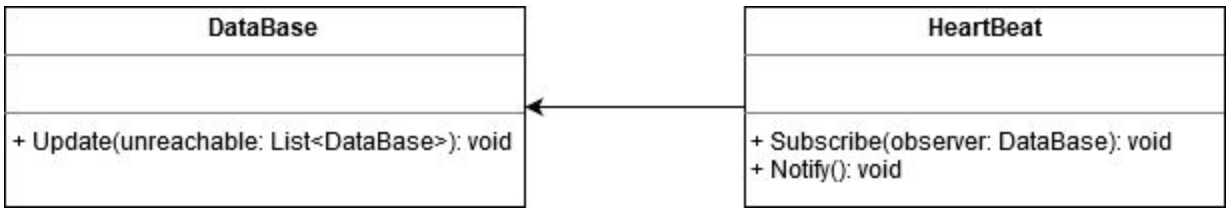
W ogólnym przypadku jest to wzorzec, który nasłuchuje wystąpienia jakiegoś zdarzenia (przykładowo zmiana stanu). Jeśli ono wystąpi zostaną poinformowane wszystkie podłączone obiekty. Dzięki temu możemy połączyć ze sobą obiekty w relacji jeden do wielu – to znaczy obiekt obserwowany z grupą obiektów go obserwujących.

Wzorzec składa się z dwóch podstawowych interfejsów:

- **Observable**– który zawiera trzy metody:
    - attach(Observer) – dodanie nowego obserwatora, który może być zainteresowany zmianami w obiekcie implementującym ten interfejs
    - detach(Observer) – usunięcie istniejącego już obserwatora
    - nofity() – powiadomienie wszystkich obserwatorów o zmianie stanu
- **Observer:**
    - update() – metoda, która jest wywoływana, kiedy nastąpiła jakaś zmiana

W przygotowywanej bibliotece będzie on pełnił funkcję nasłuchiwania czy wykonanie poleceń na poszczególnych bazach danych powiodło się zgodnie z założeniami. Jeśli któreś z zapytań zostanie odrzucone przez bazę lub jego realizacja nie powiedzie się dany serwer zostanie wyłączony z użytku lub naprawiony.

Przykładowy schemat działania wzorca w kontekście biblioteki **Sebalance:**



Klasyczne podejście do tego wzorca oraz implementacji nie jest niestety wolne od wad. Prymitywna i generyczna wersja pierwotna jest przede wszystkim obarczona problemem braku konkretnej informacji o tym, jakie zdarzenie zostało zaobserwowane, czyli co konkretnie zmieniło się w danym obiekcie. Można sobie jednak z tym poradzić stosując modyfikację wcześniej wspomnianych metod attach oraz update, które zastosowane zostaną w bibliotece. W

pierwszym przypadku podczas deklaracji można dodać dodatkowy atrybut, który informuje na którą grupę wydarzeń reagować ma dany obserwator. W deklaracji metody update() z kolei można zdefiniować, że przyjmowane dane dotyczyć mają wyłącznie konkretnych zachowań. Między innymi rezultaty odpowiedzi na wysyłane heartbeaty.

W obliczu tej wady (bardzo łatwej do rozwiązania) ogromna zaleta, którą jest brak konieczności aktywnego sprawdzania czy obiekt się zmienił działa na plus podczas praktycznego stosowania tego wzorca w projektach.

Fasada

Dzięki temu wzorcowi strukturalnemu otrzymujemy możliwość uproszczenia i ujednolicenia skomplikowanego i złożonego systemu. Tak udostępniony czytelny interfejs programistyczny ułatwia użycie zaimplementowanego systemu.

Klient, czyli dowolny kod korzystający ze złożonego systemu odwołuje się do fasady, która stoi przed nim i wykonuje operacje, zamiast do zawiłej implementacji poszczególnych elementów złożonego systemu.

Takie podejście pozwala na wprowadzenie warstw w projekcie, co zdecydowanie ułatwia niezależny rozwój klienta oraz systemu, a także poprawia czytelność kodu osobom korzystającym z zaimplementowanej struktury. Dodatkowo twórca udostępnionego systemu może zdecydować czy klient ma dostęp do metod znajdujących się „za” fasadą.

Wybór tego wzorca do projektu jest dość oczywisty, pomaga on bowiem w tworzeniu przejrzystej i spójnej biblioteki oraz poprawia czytelność dla klienta, co jest bardzo ważnym aspektem funkcjonalności i jakości systemu.

Szczegóły działania tego wzorca w kontekście biblioteki **Sebalance** zaobserwować będzie można dopiero po zaimplementowaniu funkcjonalności, które znajdą się za fasadą.

Strategia

Na starcie należy zaznaczyć, że wzorzec ten jest zgodny z zasadą otwarte-zamknięte (Open-closed principle), co oznacza, że możemy dodawać nowy kod nie zmieniając dotychczasowego. To jest właśnie powód, przez który Strategia jest tak powszechnie stosowana. Jest ona bowiem wygodną alternatywą dla problemu nieczytelnych instrukcji warunkowych wypełnionych ogromną ilością kodu. Problem rozwiązywany jest następująco: wszystkie bloki kodu w instrukcjach warunkowych dzielą pewną wspólną abstrakcyjną przestrzeń - interfejs, który później implementowany jest zgodnie z konkretnymi potrzebami poszczególnych przypadków.

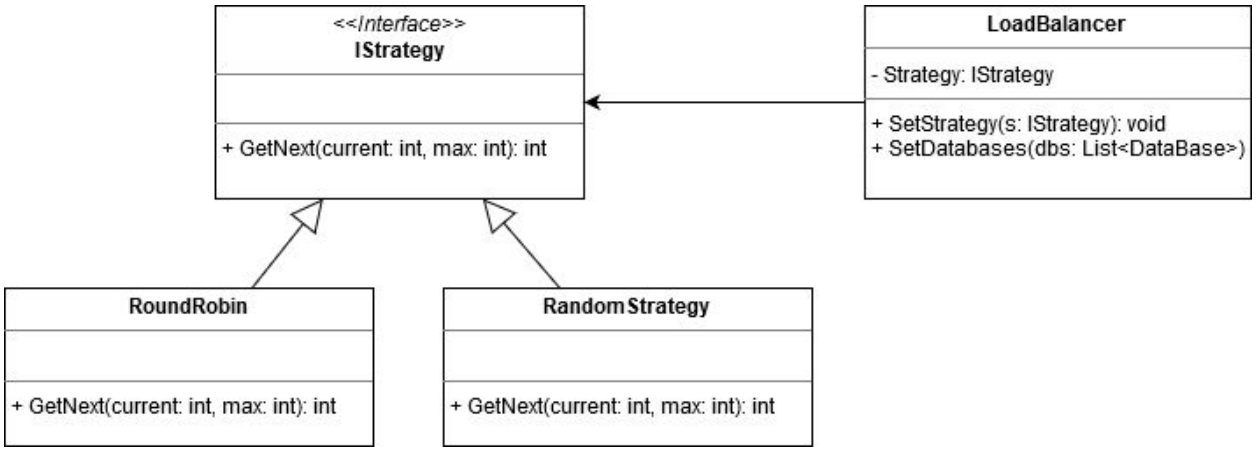
Do najważniejszych elementów implementacji wzorca należą:

- klasa **Context** – przechowuje pole do którego przypisuje się aktualnie wybrany obiekt strategii, dzięki temu można zmieniać je w trakcie działania algorytmu
- interfejs **AbstractStrategy** – posiada zdefiniowane metody potrzebne do wszystkich algorytmów
- klasy **ConcreteStretegy** – implementuje **AbstractStrategy** oraz precyzuje konkretne algorytmy i scenariusze działania

W bibliotece **Sebalance** Strategia świetnie spisze się podczas obsługi błędów w momencie, w którym jedna z jednostek/serwerów z bazą danych nie odrzuci któreś zapytanie. W tej sytuacji istnieje wiele kroków, które mogą być podjęte podczas rozwiązywania tego problemu: usunięcie tego elementu z sieci serwerów oraz praca na zmniejszonym o jeden zbiorze, zgłoszenie problemu do administratora lub zresetowanie jednostki oraz przywrócenie jej do działania przy użyciu Unit of Work.

To tylko przykładowe rozwiązania problemu. W zależności od potrzeb każdy klient korzystający z biblioteki powinien mieć możliwość zdefiniowania własnego scenariusza działania w takiej sytuacji, a wzorzec Strategia bardzo ułatwi mu dodawanie spersonalizowanych funkcjonalności.

Dodatkowo wzorzec ten zostanie zastosowany podczas wyboru sposobu przechodzenia po poszczególnych jednostkach z kopiami bazy danych przy wykonywaniu poleceń wymagających pobrania informacji. W tym przypadku można korzystać z licznych algorytmów, na przykład z poniżej opisanego Round Robin lub także przy wyborze odpowiedniego serwera do wykonania zapytania można kierować się informacją, na którym z nich mamy najmniejsze obciążenie lub czas oczekiwania na wykonanie. Dzięki temu, że do rozwiązania tego problemu użyjemy Strategii również sam użytkownik będzie miał możliwość rozszerzenie w łatwy sposób biblioteki do swoich potrzeb, jeśli uzna, że zna algorytm, który lepiej poradzi sobie w jego aplikacji.



Unit of work

Wzorzec ten służy do zebrania realizowanych operacji i wysłania ich do bazy jako jednej transakcji. Wzorzec ten najpierw zbiera wszystkie operacje, które chcielibyśmy wykonać, następnie składa je w jedną transakcję i wysyła w takiej formie do bazy danych.
Dużym atutem tego wzorca jest to, że dodaje nam on w programie dodatkową warstwę abstrakcji, która pozwala oddzielić przechowywane dane od logiki aplikacji. Dodatkowo podnosi jakość kodu ze względu na jego łatwość utrzymania oraz rozbudowywania.
Wzorzec Unit of Work okaże się być bardzo przydatny w bibliotece Sebalance podczas, gdy któraś z instancji bazy danych z jakiegoś powodu nie będzie mogła obsłużyć pewnego zapytania (np. zostanie odcięte od niej zasilanie lub wystąpi jakiś inny błąd techniczny). W tym momencie wzorzec rozpocznie monitorowanie zapytań modyfikujących zbiór danych realizowanych na działających bazach aż do momentu, kiedy uszkodzona instancja nie zostanie przywrócona do pełnej sprawności. Wówczas na podstawie zebranych operacji zostanie wysłana jedna transakcja do bazy, dzięki czemu niewielkim kosztem będziemy mogli kontynuować pracę na pełnym zbiorze danych, bez konieczności resetowania i budowania od nowa całej bazy danych.
Wszystkie operacje Insert oraz Update wykonywane na bazach danych są zapisywane przez Unit of Work w liście, dodatkowo każda z instancji bazy danych posiada swój licznik zapytań, które przeszła pomyślnie. Zarówno licznik, jak i lista zapisanych operacji jest resetowana co heartbeat, pod warunkiem, że każda z baz na niego odpowiedziała. Jeśli któraś z nich nie odpowiedziała na niego lub jakieś inne wcześniej wysłane zapytanie, lista nie zostanie usunięta, ani licznik zresetowany. W tej sytuacji konieczne będzie monitorowanie zapytań aż do momentu ponownego uruchomienia bazy lub osiągnięcia limitu heartbeatów bez odpowiedzi (50). Jeśli uszkodzona instancja powróci do pracy, zostanie pobrany numer ostatniego udanego zapytania, Unit of Work zbierze brakujące zapytania w jedną transakcję i zostanie ona wysłana do bazy, w której brakuje danych, dzięki czemu będzie ona gotowa do dalszej pracy. Po tej operacji będzie można zresetować liczniki oraz listę z operacjami.
Wzorzec zwykle składa się z interfejsu zawierającego metodę **Complete** (odpowiedzialną za zapisywanie) oraz obiekt konkretnej instancji bazy, na których może zostać wykonane jedynie pobranie zapytań do zapisania. Z kolei klasa UnitOfWork implementująca wcześniej opisany interfejs implementuje metodę Complete wskazując miejsce, gdzie dane mają być przechowywane.
Zapytania będą zapisywane w formie mapy - id zapytania oraz jego treść. Mapa będzie czyszczona po każdym heartbicie - wtedy też wyzeruje się id.

Biblioteka Sebalace

Ogólnie

Jest to biblioteka realizująca load balancing zaimplementowana w technologii .net.
Zawiera mechanizm, który jest warstwą pośredniczącą pomiędzy serwerem, a bazą danych. Pozwala na obsługę do 10 baz danych równolegle i stosuje się do zasad load balancingu. Wykorzystuje technologię hibernate do konwersji danych z baz danych na obiekty i udostępnia funkcje do działań na tych obiektach. W zależności od rezultatu wykonywanej operacji zwraca użytkownikowi stosowny komunikat.

Podział zasobów zależnie od operacji

1. Dodanie nowego rekordu do bazy danych - na każdym z zasobów
2. Usunięcie rekordu z bazy danych - na każdym z zasobów
3. Aktualizacja rekordu w bazie danych - na każdym z zasobów
4. Wyświetlanie danych z bazy danych - na wybranym zasobie

Do wyboru zasobu na którym zostanie wykonana operacja wykorzystuje algorytm Round Robin. Dla każdej dodanej bazy musi zostać utworzony obiekt SessionFactory który umożliwia tworzenie sesji odpowiedzialnych za łączenie się z bazą danych. Z jego pomocą powstaje instancja ISession, niezbędna do wykonywania operacji na konkretnej bazie. Obiekt BazaDanych będzie przechowywał wszelkie informacje związane z konkretną bazą jak i jej SessionFactory. Wszystkie bazy będą zebrane w Liście.

Opis algorytmu Round Robin

Round Robin jest to najprostszy algorytm szeregowania dla procesów w systemie operacyjnym, który przydziela każdemu procesowi odpowiednie przedziały czasowe, nie uwzględniając priorytetów. Każdy proces ma ten sam priorytet. Zaczyna od pierwszego procesu w kolejce, przydzielając każdemu procesowi po kolei odpowiednie zasoby.
Zaletą tego algorytmu jest jego prostota i łatwość implementacji.

Heartbeat

Heartbeat, w informatyce, jest to okresowy sygnał generowany przez sprzęt lub oprogramowanie. Zazwyczaj jest wysyłany między komputerami w regularnych odstępach. Jeśli punkt końcowy nie odbiera sygnału przez pewien czas zakładamy, że nie działa.

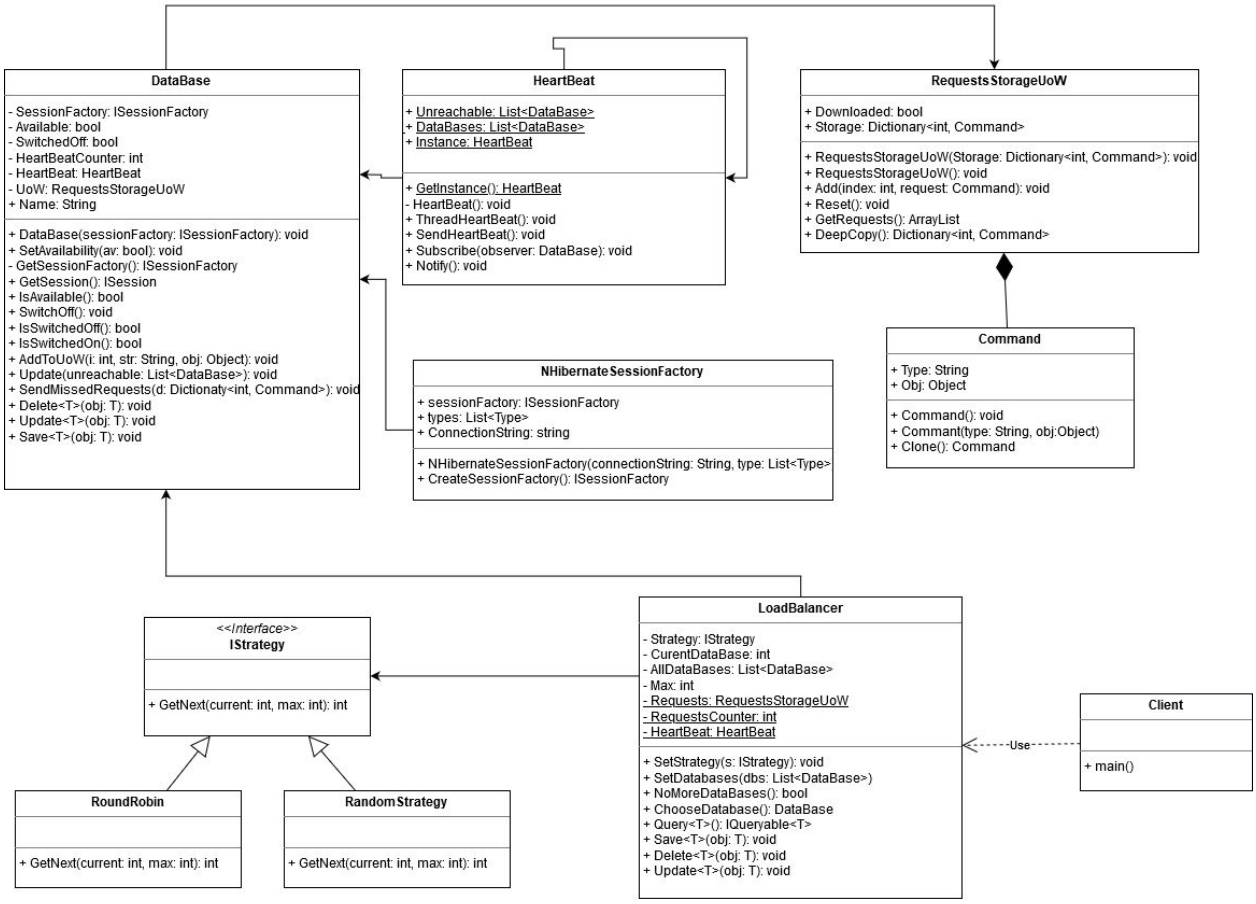
Biblioteka **Sebalance** wykorzystuje Heartbeat do kontrolowania czy wszystkie bazy danych, które obsługuje, działają.

Zgodnie z zasadą działania, co 5 sekundy jest wysyłane do każdej z baz danych zapytanie `SELECT null;`. W przypadku braku odpowiedzi daną bazę danych uznajemy za uszkodzoną.

Uszkodzona baza danych

W momencie wykrycia przy pomocy Heartbeat uszkodzonej bazy, kontynuujemy wysyłanie do niej zapytań do momentu, gdy nam odpowie. W momencie, gdy baza danych znów będzie sprawna bardzo prawdopodobne jest to, że będzie nieaktualna. Do zaktualizowania odzyskanej bazy danych jest użyty wzorzec unit of works, który zbiera informacje o tym, jakie operacje zostały wykonane w czasie, gdy baza była uszkodzona. Wywołujemy zaległe operacje na uprzednio uszkodzonej bazie i przywracamy ją do pracy.
W przypadku, gdy baza nie odpowie nam na 50 kolejnych heartbeatów, usuwamy bazę z listy dostępnych.

Diagram klas



Słownik klas i metod

1. DataBase

Klasa, której obiekty są odwzorowaniem wszystkich obsługiwanych instancji bazy danych, zawiera w sobie informacje konfiguracyjne.

- *DataBase* - konstruktor, który pobiera instancję klasy HeartBeat, zapisuje SessionFactory, tworzy nowy obiekt UnitOfWork oraz ustala pozostałe pola na wartości startowe
- *SetAvailability* - pozwala ustawić wartość pola Available na zadaną wartość w argumencie
- *GetSessionFactory* - zwraca SessionFactory obiektu
- *GetSession* - otwiera sesję dla danego SessionFactory
- *IsAvailable* - zwraca informację czy dana instancja bazy jest dostępna

- *SwitchOff* - pozwala ustawić wartość pola SwitchedOff na zadaną wartość w argumencie
- *IsSwitchedOff, IsSwitchedOn* - zwraca informację o tym czy baza jest wyłączona, baza zostaje wyłączona jeśli nie odpowiada na 50 HeartBeatów puszczanych co 5 sekund
- *AddToUoW* - metoda wywołuje metodę Add na polu UoW i dodaje zadane zapytanie
- *Update* - metoda wykorzystywana przez Obserwatora przyjmująca informację o zmianie stanu obiektu obserwowanego - HeartBeat oraz obsługująca to zdarzenie
- *SendMissedRequests* - wysłanie zapytań zapisanych w Unit of Work, gdy baza zacznie odpowiadać ponownie
- *Delete<T>,Update<T>, Save<T>* - wykonanie operacji Delete(object), Update(object), Save(object) (zapewnionych przez NHibernate) na danej instancji bazy danych

2. HeartBeat

Klasa możliwa do obserwowania przez subskrybujące ją instancje baz danych, jej zadaniem jest wysyłanie co 5 sekund HeartBeata, który sprawdza czy wszystkie bazy odpowiadają na zapytania.

- *GetInstance* - implementacja wzorca Singleton - zapewnia, że w obrębie projektu znajduje się tylko jedna instancja tej klasy
- *HeartBeat* - konstruktor, który inicjalizuje wartości początkowe pól obiektu
- *ThreadHeartBeat* - wątek obsługujący wysyłanie HeartBeatów w ustalonych odstępach czasu (5 sekund)
- *SendHeartBeat* - metoda wysyła do wszystkich instancji baz danych puste zapytanie i czeka na odpowiedź
- *Subscribe* - dodanie nowego subskrybenta (w tym przypadku obiektu DataBase) nasłuchującego
- *Notify* - wysłanie w powiadomieniu listy baz, które w danym beacie nie odpowiedziały na zapytanie

3. IStrategy

Interfejs wchodzący w skład wzorca Strategia, którego implementują konkretne klasy zapewniające sposób wyboru konkretnej bazy do wykonania zapytania.

- *GetNext* - wskazuje na indeks bazy, do której ma zostać wysłane zapytanie

4. RandomStrategy

Implementacja interfejsu IStrategy.

- G*etNext* - metoda zwraca losowy indeks z zakresu od 0 do maksymalnej wartości podanej jako argument

5. RoundRobin

Implementacja interfejsu IStrategy.

- *GetNext* - metoda zwraca kolejny dostępny indeks listy

6. RequestsStorageUoW

Klasa implementująca wzorzec Unit Of Work, w której obiektach zapisywane są zapytania, które nie zostały wykonane na danej instancji bazy. Wspiera aktualizowanie bazy danych, aby była zgodna z pozostałymi instancjami w momencie, kiedy któraś z nich została na jakiś czas odłączona

- *RequestsStorageUoW* - konstruktor inicjalizujący wartości początkowe pól
- *Add* - dodanie do listy nowego zapytania
- *Reset* - usuwanie zapisanych w liście zapytań
- *DeepCopy* - implementacja głębokiego kopiowania obiektów złożonych
- *GetRequests* - metoda zwraca listę zapytań zapisanych w obiekcie

7. Command

Klasa reprezentująca zapytanie do bazy danych

8. NHibernateSessionFactory

Klasa, która opakowuje SessionFactory dostarczone przez NHibernate i zajmuje się jego konfigurację. Umożliwia dodanie ConnectionString z poziomu programu klienta biblioteki bez konieczności dodawania ich w pliku konfiguracyjnym.

9. LoadBalancer

Fasada dla mechanizmu loadbalancer udostępniana użytkownikowi metody konfigurujące mechanizm oraz interakcji z bazami danych.

- *SetStrategy* - ustawia strategię na wybraną implementację
- *SetDatabases* - ustawia listę baz danych oraz pozostałe wartości pól, które ją określają
- *NoMoreDataBases* - sprawdza czy jakakolwiek baza jest jeszcze dostępna i odpowiada
- *ChooseDatabase* - wybór bazy danych dzięki strategii
- *Query<T>* - wysłanie Select do bazy danych
- *Save<T>* - wysłanie Insert do bazy danych
- *Delete<T>* - wysłanie Delete do bazy danych
- *Update<T>* - wysłanie Update do bazy danych

Konfiguracja

- plik konfiguracyjny wykorzystywany przez bibliotekę NHibernate

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-configuration  xmlns="urn:nhibernate-configuration-2.2" >
  <!--Session factory name -> sebalanceDP-->
  <session-factory>
    <property name="connection.provider">
    NHibernate.Connection.DriverConnectionProvider
  </property>
  <property name="connection.driver_class">
    NHibernate.Driver.MySqlDataDriver
  </property>

  <property name="dialect">
    NHibernate.Dialect.MySQL5Dialect
  </property>
    <!-- Assembly containing the embedded HBM mapping files -->
  </session-factory>
</hibernate-configuration>
```

- przykład dodawania baz danych z poziomu kodu użytkownika biblioteki

```
IStrategy s = new RandomStrategy();
var sessionFactory1 = new NHibernateSessionFactory("Server=104.211.12.200; Port=3306; Database = furnitureShop; Uid = sebalance; Password = sebalance; ",
        new List<Type> { typeof(Table) });
var sessionFactory2 = new NHibernateSessionFactory("Server=104.211.12.200; Port=3307; Database = furnitureShop; Uid = sebalance; Password = sebalance; ",
        new List<Type> { typeof(Table) });
LoadBalancer.SetDatabases(new List<DataBase>
{
    new DataBase(sessionFactory1.SessionFactory),
    new DataBase(sessionFactory2.SessionFactory)
});

LoadBalancer.SetStrategy(s);
```

- plik xml wspomagający mapowanie, wykorzystywany przez bibliotekę NHibernate

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" namespace="FurnitureShop" assembly="FurnitureShop">
  <class name="Table" table="Tables">
    <id name="TableId" column="TableId">
      <generator class="native" />
    </id>
    <property name="Material" column="Material" />
    <property name="Color" column="Color" />
  </class>
</hibernate-mapping>
```

Przykładowa klasa pokrywająca się z przykładowym plikiem xml

```
class Table
{
    public virtual int TableId { get; set; }
    public virtual string Material { get; set; }
    public virtual string Color { get; set; }
}
```

Pliki xml służące do mapowania obiektów pisane są już przez użytkownika aplikacji.