Design Patterns: Active Record

dr Szymon Murawski

March 25, 2020

Code for this exercise can be found at `https://github.com/Mishrakk/DBS2/tree/master/ActiveRecord`

1 Introduction

Databases provide persistence to our programs, meaning that we can save and load data in different instances of our program. It also adds another layer of complexity to our design and it's very easy to end up with a Big Ball Of Mud architecture. By definition:

> A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.
>
> The overall structure of the system may never have been well defined.
>
> If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.
>
> (Brian Foote and Joseph Yoder)

Thankfully there are design patterns that deal with properly structuring database access methods and connecting database objects to programming objects.

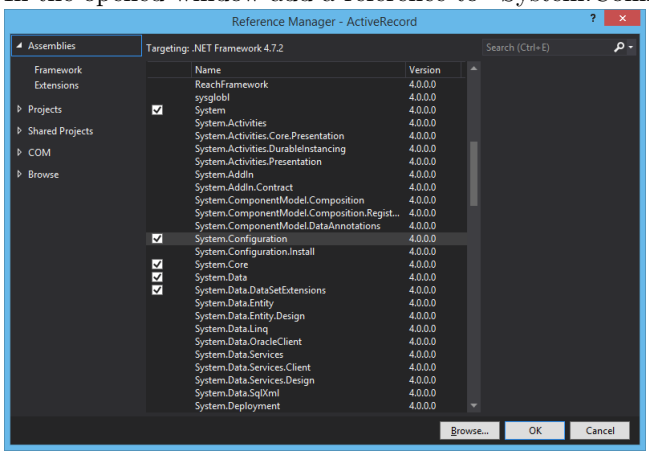
2 Configuration file

This is not a design pattern, but an important topic nevertheless. Since connection string may change depending on the computer you are trying to run your program from, keeping it directly in your code is a bad idea. It would require recompiling the whole solution every time it changes. Better to keep it

in configuration, where you can change it without the need of compilation. To do this do the following:

1. In Visual Studio click "Solution Explorer", then right click on "References" and select "Add reference"

2. In the opened window add a reference to "System.Configuration"



3. Open "app.config" and add the following lines under the `<configuration>` tag:

```
<connectionStrings>
  <add name="Rental" connectionString="Server=127.0.0.1;User
    Id=postgres;Password=pwd;Database=rental;" providerName="
    Npgsql" />
</connectionStrings>
```

4. Now in the code you can use the following line, to get the connection string from configuration file:

```
System.Configuration.ConfigurationManager.ConnectionStrings["
    Rental"].ToString();
```

3 Active record pattern

An active record is an object, that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data. An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.

The Active Record class typically has methods that do the following:

- Construct an instance of the Active Record from a SQL result set row

- Construct a new instance for later insertion into the table
- Static finder methods to wrap commonly used SQL queries and return Active Record objects
- Update the database and insert into it the data in the Active Record
- Get and set the fields
- Implement some pieces of business logic

For example let's consider a `copy` object in our DVD rental store program. In database a single copy has a corresponding row in `copies` table, while in C# we might have a class like the one below:

```
1  public class CopyRecord
2  {
3    public int ID { get; private set; }
4    public bool Available { get; private set; }
5    public int MovieId { get; private set; }
6    public CopyRecord(int id, bool available, int movieId)
7    {
8      ID = id;
9      Available = available;
10     MovieId = movieId;
11   }
12 }
```

Since we keep copies in database, we need a static function to load a specific copy, that would look something like this:

```
1  public static CopyRecord GetById(int id)
2  {
3    using (NpgsqlConnection conn = new NpgsqlConnection(
       CONNECTION_STRING))
4    {
5      conn.Open();
6      using (var command = new NpgsqlCommand("SELECT * FROM copies
       WHERE copy_id = @ID", conn))
7      {
8        command.Parameters.AddWithValue("@ID", id);
9
10       NpgsqlDataReader reader = command.ExecuteReader();
11       if (reader.HasRows)
12       {
13         reader.Read();
14         return new CopyRecord(id, (bool)reader["available"], (int
       )reader["movie_id"]);
15       }
16     }
17   }
18   return null;
19 }
```

Similarly we need a save function, to transfer copy object from memory of the program to database:

```
1  public void Save()
2  {
3    using (NpgsqlConnection conn = new NpgsqlConnection(
       CONNECTION_STRING))
4    {
5      conn.Open();
6      // This is an UPSERT operation - if record doesn't exist in
       the database it is created, otherwise it is updated
7      using (var command = new NpgsqlCommand("INSERT INTO copies(
       copy_id, available, movie_id) " +
8        "VALUES (@ID, @available, @movieId) " +
9        "ON CONFLICT (copy_id) DO UPDATE " +
10       "SET available = @available, movie_id = @movieId", conn))
11     {
12       command.Parameters.AddWithValue("@ID", ID);
13       command.Parameters.AddWithValue("@available", Available);
14       command.Parameters.AddWithValue("@movieId", MovieId);
15       command.ExecuteNonQuery();
16     }
17   }
18 }
```

Observe, that `Save` function either adds a new record to the database (if it doesn't exist) or updates an existing one. In some implementations there are two separate operations for it, but in my opinion it just pushes the problem to the upper layer and might pollute business logic layer.

Since `CopyRecord` is the only class that "deals" with copies of movies, we need to include in it some business logic: functions that change the availability status, removal of a copy, renting a copy, returning it etc. Of course if we change something in an object we need to also Save it to the database

Very rarely a Record based on a single table would be sufficient. A single movie in our store can have multiple copies, so if we were to create a MoveRecord class it should have a list of all the copies.

```
1  class MovieRecord
2  {
3    private static readonly string CONNECTION_STRING =
4      System.Configuration.ConfigurationManager.ConnectionStrings["
       Rental"].ToString();
5    public int ID { get; private set; }
6    public string Title { get; private set; }
7    public int Year { get; private set; }
8    public double Price { get; private set; }
9    public List<CopyRecord> Copies { get; private set; }
10 }
```

Now of course we will need to handle the list of copies in functions that access the database for MovieRecord. For example Save function would look something like this:

```
1  public void Save()
2  {
3    using (NpgsqlConnection conn = new NpgsqlConnection(
       CONNECTION_STRING))
4    {
5      conn.Open();
6      // This is an UPSERT operation - if record doesn't exist in
       the database it is created, otherwise it is updated
7      using (var command = new NpgsqlCommand("INSERT INTO movies(
       movie_id, title, year, price) " +
8        "VALUES (@ID, @title, @year, @price) " +
9        "ON CONFLICT (movie_id) DO UPDATE " +
10       "SET title = @title, year = @year, price = @price", conn))
11     {
12       command.Parameters.AddWithValue("@ID", ID);
13       command.Parameters.AddWithValue("@title", Title);
14       command.Parameters.AddWithValue("@year", Year);
15       command.Parameters.AddWithValue("@price", Price);
16       command.ExecuteNonQuery();
17     }
18     Copies?.ForEach(obj => obj.Save());
19   }
20 }
```

When we save MovieRecord we also need to save all the copies in the list individually. Notice the "?" symbol on line 18 - `Copies` might be an empty list, so we need to protect ourselves from NullReferenceException.

4 Final remarks

Active Record Design Pattern is an easy patter that almost every programmer used at some point, often without knowing about this pattern existence! It allows for very fast coding, as you don't need to think a lot about architecture of your solution. For small and not overly complicated objects it is the best solution. Many frameworks are based on Active Record, in Ruby on Rails it is the default method.

In general however, Active Record is considered an anti-pattern as it has many problems. First of all it breaks the Single Responsibility Principle, as we introduce a class that does everything - business logic, database access, transactions. Because of it Active Records classes tend to exponentially grow in size with every new functionality added. This is especially visible as you have records that are in a relationship with other records. Another flaw with this pattern is that it is almost impossible to test, as you have a strong coupling between objects and database.

So, know that such a pattern exists, use it for prototyping or some other quick work, but in general try to avoid programming according to this design pattern.