Design factors for database query DSLs in object-oriented languages

Pramod Kotipalli | Stanford CS343D | Fall 2020

Most applications that interface with a database are written with object-oriented programming (OOP) languages. Many applications use SQL databases to persist data. There exists an 'impedance mismatch' between the use of object-oriented systems that act on objects of non-scalar values and the storage of scalar values (e.g. strings and integers) organized in SQL tables. The ubiquity of OOP systems and SQL databases is hard to overcome; many developers must use both together to write effective and performant web applications. Numerous object-relational mapping (ORM) software libraries exist as libraries for popular languages such as Hibernate ORM for Java, Active Record for Ruby, and the Django ORM or SQLAlchemy for Python.

From the perspective of the programmer of a sophisticated backend API, three language 'front-end' design factors are key when selecting or implementing a query DSL for an OOP language (implementation details of ORMs will not be discussed):

1. Syntax readability
2. Type-checking
3. Extensibility

Syntax readability

In addition to a method-chaining (i.e. "fluent interface") interface common to many programming languages, some languages have taken to including data querying syntax natively, as in C# with Language Integrated Query (LINQ). This "query expression" syntax supports data-query-specific keywords like `from` and `select` as native language constructs.

```
class Person { public int Age; public String Name; }

Person[] people = new Person[] {
    new Person { Age = 16, Name = "Alex" },
    new Person { Age = 17, Name = "Jamie" },
    new Person { Age = 20, Name = "Savannah" } };
```

Code Fragment 1 - Initializing a `people` array in C#

```
IEnumerable<Person> minors = people.Where(p => p.Age < 18);
foreach (Person minor in minors) {
    Console.WriteLine(minor.Name);
}
```

Code Fragment 2 - LINQ through a *method chaining* or a *fluent interface* syntax

```
IEnumerable<Person> minors = from person in people
    where person.Age < 18
    select person;
foreach (Person minor in minors) {
    Console.WriteLine(minor.Name);
}
```

Code Fragment 3 - LINQ through *query expression* syntax

Note that both the method chaining and query expression approaches yield the same `IEnumerable` collection type: both approaches are semantically equivalent. However, some queries are more readable in one form over another. Some developers may even take the approach of using the query expression syntax for database queries and the fluent interface for in-memory collection manipulation. Therefore, supporting multiple query syntaxes is one important factor for when selecting the enclosing programming language.

Further, the support of syntactically-simple lambda expressions is crucial for readable queries. This is a feature that popular languages like Python do not support. An equivalent to Code Fragment 2 with Python lambdas would print as:

```
minors = Person.objects.filter(lambda p: p.Age < 18)
```

Code Fragment 4 - Python lambda syntax in an ORM query

The use of the `lambda` keyword over the `=>` in more sophisticated queries grows to be a tiresome syntax.

Type-checking

Design factors for database query DSLs in object-oriented languages

C#'s support of LINQ enables compile-time type checking of every query written in a program. Django's ORM is constrained to Python's dynamic type checking system leading to syntax like so for a similar C# query.

```
minors = Person.object.filter(age__lt=18)
for minor in minors:
   print(minor.Name)
```

Code Fragment 5 - A query for `minors` through a Python ORM

Note that Python-based ORMs must rely on method-chaining syntax, straying one level of abstraction away from the SQL-like syntax of C# query expressions. Also note the `age__lt=18` argument to the `filter` method; such methods (e.g. `lte` for `<=`, `gt`, `gte`, `in`, `contains`, etc.) are programmatically-generated by Django for each property of a database object. The limitations of Python's dynamic typing system means that Django must take arguments of the `filter` method, parse it (e.g. as `age` and `lt` for the `<` operator) and then evaluate its semantics before generating SQL, all adding additional overhead to the program's runtime without the benefits of compile-time type-checking.

As the complexity of object-oriented systems grows, developers tend to write less and less tests (if any to begin with) making the importance of compile-time type checking more important. Pre-deployment compilation or type-checking is an important factor for the selection of a programming language for an ORM, either for a programmer or an ORM implementor. Python's best approach to this problem is using an optional static type checker, mypy, which can check a Python program before it is deployed. However, mypy is not able to actually compile and optimize code as with statically-typed languages.

C# also supports statically-typed anonymous types enabling C# to represent intermediate query results in simple struct-like types without adding code bloat from an additional declaration.

Extensibility

C#'s support of extension methods is extremely useful for both readability and keeping code DRY. Take for example the expression of a time-difference object. To express a

time delta of 30 days, we can write `new TimeSpan(30, 0, 0, 0)`. An experienced programmer can read this as “30 days, 0 hours, 0 minutes, and 0 seconds” but even for such programmers using the `new TimeSpan` syntax can get quite tiresome. C# supports extension methods which form syntactic sugar for method chaining, a much more readable form:

```
public static class DateTimeExtensions
{
    public static TimeSpan Days(this int number)
    {
        return new TimeSpan(number, 0, 0, 0);
    }
}

...

30.Days(); // Equivalent to new Timespan(30, 0, 0, 0);
```

Code Fragment 6 - “Literal expressions” in C#

This form of language augmentability supports the creation of DSLs for domains further than that of generic database queries. We can now support method chaining that makes sense for a particular domain, such as for an API managing a store: `user.GetOrders().Where(order => order.DaysOnShelf < 3.Days());`. If we need to access such orders multiple times, we can simply declare a reusable extension method that encodes this lambda:

```
public static class OrderExtensions
{
    public static IEnumerable<Order> OnShelfForShortTime(
        this IEnumerable<Order> orders)
    {
        return orders.Where(order =>
            order.DaysOnShelf < 3.Days());
    }
}

...

user.GetOrders().OnShelfForShortTime(); // Equivalent form
```

Code Fragment 7 - “Literal expressions” in C#

With extensibility, query syntax is easier to read and lends itself to DRY implementations of large systems. While other ORMs support such augmentations, they are usually implemented through proxy objects or by utilizing the dynamically-typed nature of languages.

Final thoughts

While this essay may seem like a document exalting the expressiveness and simplicity of LINQ, I try to show that some core language features of C# are very useful for interacting with data through ORMs and creating DSLs for data queries. The support of native language query expressions and extension methods provide for readability and extensibility. The statically-typed nature of C# provides the programmer with many assurances during development. Many ORMs and data-interfacing query languages in OOP contexts can benefit from supporting C#-like language features.