**How to create a fluent interface in C#**

I had everything ready to do a video tutorial for building a fluent interface, when I had some (still-unsolved) technical problems this weekend. So, in the interest of finally releasing this information, I’m just going to do it as a blog post.

**What is a fluent interface?**

Have you ever seen a cool-looking series of method calls that almost reads like natural language?

|  |  |
| --- | --- |
| 1 | DeleteRowsFrom(“Account”).Where(“Level”).IsNotEqualTo(“Admin”).Where(“Balance”).IsEqualTo(0).RunNow(); |

That’s called a fluent interface, and it has some practical advantages, other than just looking nice.

**Why create a fluent interface?**

I always like to start with the “why”, to make sure the final solution actually solves a problem.

Fluent interfaces have two main things they try to solve.

First, they make the code more readable, and easy to understand. This makes it easier to make changes later on.

Second, they can be used to force the programmer to perform certain steps before they perform others. This can be a problem with some object oriented programming. An object can have a method that uses data it expects to have been set in another method. However, it doesn’t force the programmer to call the first method – it’s just something that all the developers are supposed to know.

So, when you get a new programmer on the team, you have a problem. If you go back to make a change to the program you wrote three years ago, and you’ve forgotten all these silent requirements, you have a problem.

With a properly-written fluent interface, you can enforce “rules of grammar”, and make it so certain methods cannot be called, until all the required setup methods have been called.

**Steps to build a fluent interface**

There are three steps I use to create a fluent interface. They are:

1. Define all possible combinations of the natural language syntax
2. Create the interfaces that enforce the grammar rules
3. Build the class, implementing the interfaces

If all that doesn’t make sense, right now, don’t panic. I’m going to go through it slowly, and give lots of code samples.

**Step 1 – Define the natural language syntax**

All you need to do here is write down the possible combinations you want to use in your fluent interface.

For this example, I’m going to show how to build an object for deleting rows from a SQL database.

The reason to use a fluent interface for this is to ensure the developer includes at least one WHERE condition, or explicitly states that they want to delete all the rows in the table.

If you’ve been writing database code for any length of time, you’ve probably heard of at least one developer who wrote a delete query, and forgot to include a WHERE clause, causing all the rows to accidentally be deleted. You may even be that developer (most of us have done that, at some point).

For this sample, we’re not going to build a complete delete query object. There’s a huge amount of code I’d need to write for that. We’ll just cover the basics, so you get an idea of how a fluent interface would work in a “real-world” example.

|  |  |
| --- | --- |
| 1  2  3 | DeleteRowsFrom(table).AllRows();  DeleteRowsFrom(table).Where(column).IsEqualTo(value).RunNow();  DeleteRowsFrom(table).Where(column1).IsNotEqualTo(value1).Where(column2).IsEqualTo(value2).RunNow(); |

This covers the basic deletes I want to be able to perform. A delete of all rows in the table, and a delete with one or more WHERE conditions that check if a column is either equal to, or not equal to, a passed value.

Now we need to break apart the syntax. I use the acronym ICE to identify the different types of methods. That stands for:

* I – Initiating/Instantiating
* C – Chaining/Continuing
* E – Executing/Ending

**Initiating/Instantiating methods**

These are the methods we use to start the call. Here, we only have one, DeleteRowsFrom(table).

**Chaining/Continuing methods**

These are the methods we call in the middle of the statement, and that let us call another method after them. Here, we have Where(column), IsEqualTo(value), and IsNotEqualTo(value). When any of these methods are called, you can keep on calling other methods. That’s known as “method chaining”.

**Executing/Ending methods**

These are the methods that finally do some action, and end our statement –  AllRows() and RunNow().

Now that we know all the parts, and how we want to use them, we can define the grammar our fluent interface will use.

**Step 2 – Create interfaces to enforce grammar**

Before going into how to add grammar to our fluent interface, I’m going to tell you why you need to do this. And in order to do that, I need to show you a bit about method chaining.

Method chaining is just calling one method immediately after, and in the same line of code, you’ve called another method.

|  |  |
| --- | --- |
| 1  2  3  4 | string value = input.ToString();  value = value.ToLower();  value = value.Trim();  value = value.Substring(1, 5); |

|  |  |
| --- | --- |
| 1 | string value = input.ToString().ToLower().Trim().Substring(1, 5); |

They both do the same thing.

The reason you can put all that code on one line is because each method returns an object, in this case, a string. And string objects have methods you can call on them. So, you can keep calling methods. This is “method chaining”.

With our fluent interface, we go a little farther. Not only does each method (other than the Executing methods) return an object of the same type, it actually returns the exact same object. So, when you call one method, and it does whatever it needs to do with the object, it ends with a “return this;”, to pass the now-modified object back along the chain.

With the SQL delete class we’re building, we could have each method return the SQL delete object. If we did that, our class would look something like this:

**DeleteQueryWithoutGrammar.cs**



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | using System.Collections.Generic;    namespace BuildAFluentInterface  {      public class DeleteQueryWithoutGrammar      {          private readonly string \_tableName;          private readonly List<WhereCondition> \_whereConditions = new List<WhereCondition>();            private string \_currentWhereConditionColumn;            // Private constructor, to force object instantiation from the fluent method(s)          private DeleteQueryWithoutGrammar(string tableName)          {              \_tableName = tableName;          }            #region Initiating Method(s)            public static DeleteQueryWithoutGrammar DeleteRowsFrom(string tableName)          {              return new DeleteQueryWithoutGrammar(tableName);          }            #endregion            #region Chaining Method(s)            public DeleteQueryWithoutGrammar Where(string columnName)          {              \_currentWhereConditionColumn = columnName;                return this;          }            public DeleteQueryWithoutGrammar IsEqualTo(object value)          {              \_whereConditions.Add(new WhereCondition(\_currentWhereConditionColumn, WhereCondition.ComparisonMethod.EqualTo, value));                return this;          }            public DeleteQueryWithoutGrammar IsNotEqualTo(object value)          {              \_whereConditions.Add(new WhereCondition(\_currentWhereConditionColumn, WhereCondition.ComparisonMethod.NotEqualTo, value));                return this;          }            #endregion            #region Executing Method(s)            public void AllRows()          {              ExecuteThisQuery();          }            public void RunNow()          {              ExecuteThisQuery();          }            #endregion            private void ExecuteThisQuery()          {              // Code to build and execute the delete query          }      }  } |
|  |  |

Notice that every “chaining” method returns an instance of the DeleteQueryWithoutGrammar object. This is what lets you keep calling the other public methods of the class.

However, there’s a problem. The way this is currently written, you could call things out of order, or not call some methods that may be required.

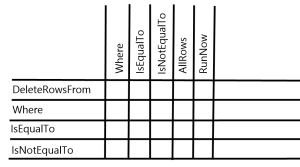
|  |  |
| --- | --- |
| 1 | DeleteQueryWithoutGrammar.DeleteRowsFrom("Account").IsNotEqualTo("Admin").IsEqualTo("Inactive").Where("Status").RunNow(); |

This is why we need to add the grammar rules. We do that with class interfaces.

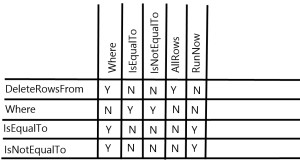
Before creating the interfaces, we need to define our rules of grammar – when the different methods are allowed to be called. So, with this class, we want to have rules like, you can only call IsEqualTo(value) immediately after a call to Where(column).

I do this by building a table like the one below.

In this table, there is a row for each Initiating and each Chaining method (on the left). There’s a column for each Chaining and Executing method (along the top).

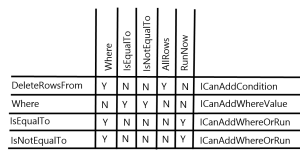
[](http://scottlilly.com/wp-content/uploads/2013/10/FluentGrammar1.png)

Now, we go through each row and mark which methods in the columns can be called after calling the method for the row.

[](http://scottlilly.com/wp-content/uploads/2013/10/FluentGrammar2.png)

On the first row, DeleteRowsFrom, we only want to allow the developer to call Where or AllRows.  On the second row, Where, we only want to allow calls to IsEqualTo or IsNotEqualTo next. For the two rows, IsEqualTo and IsNotEqualTo, we only want to allow the next method to be another Where or a call to RunNow.

We need a class interface for each different set of rules. In this case, IsEqualTo and IsNotEqualTo both allow the same methods to be called after them, so they can use the same interface.

[](http://scottlilly.com/wp-content/uploads/2013/10/FluentGrammar3.png)

Now we know what grammar we want to allow, and we’re ready to create our class interfaces.

Here’s the code for the interfaces. Notice that the methods defined in the interface don’t return a DeleteQuery object. Instead, they return an object of the interface datatype we have in our grammar table. By doing this, along with changing the return datatypes on the methods, the developer will only be able to call the methods that are available to them, based on our grammar rules.

Can I add conditions?

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace BuildAFluentInterface.Interfaces  {      public interface ICanAddCondition      {          ICanAddWhereValue Where(string columnName);          void AllRows();      }  } |

**Can I add where or run**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace BuildAFluentInterface.Interfaces  {      public interface ICanAddWhereOrRun      {          ICanAddWhereValue Where(string columnName);          void RunNow();      }  } |

**I can add where values**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | namespace BuildAFluentInterface.Interfaces  {      public interface ICanAddWhereValue      {          ICanAddWhereOrRun IsEqualTo(object value);          ICanAddWhereOrRun IsNotEqualTo(object value);      }  } |

**Step 3 – Build the class that implements the interfaces**

We’re almost done!

All that’s left is to have the DeleteQuery class implement the interfaces, and have the methods return the correct datatypes. I normally create the interfaces first, then the class. However, for this post, I wanted to show how method chaining works.

Here is what the class looks like with the changes.

**Delete query with grammer**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | using System.Collections.Generic;  using BuildAFluentInterface.Interfaces;    namespace BuildAFluentInterface  {      public class DeleteQueryWithGrammar : ICanAddCondition, ICanAddWhereValue, ICanAddWhereOrRun      {          private readonly string \_tableName;          private readonly List<WhereCondition> \_whereConditions = new List<WhereCondition>();            private string \_currentWhereConditionColumn;            // Private constructor, to force object instantiation from the fluent method(s)          private DeleteQueryWithGrammar(string tableName)          {              \_tableName = tableName;          }            #region Initiating Method(s)            public static ICanAddCondition DeleteRowsFrom(string tableName)          {              return new DeleteQueryWithGrammar(tableName);          }            #endregion            #region Chaining Method(s)            public ICanAddWhereValue Where(string columnName)          {              \_currentWhereConditionColumn = columnName;                return this;          }            public ICanAddWhereOrRun IsEqualTo(object value)          {              \_whereConditions.Add(new WhereCondition(\_currentWhereConditionColumn, WhereCondition.ComparisonMethod.EqualTo, value));                return this;          }            public ICanAddWhereOrRun IsNotEqualTo(object value)          {              \_whereConditions.Add(new WhereCondition(\_currentWhereConditionColumn, WhereCondition.ComparisonMethod.NotEqualTo, value));                return this;          }            #endregion            #region Executing Method(s)            public void AllRows()          {              ExecuteThisQuery();          }            public void RunNow()          {              ExecuteThisQuery();          }            #endregion            private void ExecuteThisQuery()          {              // Code to build and execute the delete query          }      }  } |

Notice on line 6 that we state this class implements the interfaces we created. That lets us return this object, cast as an object the interface datatype. By doing this, the developer only has access to the methods listed in the interface.

So, on line 30, when the Where method does its “return this;”, the object is returned as an object whose datatype is ICanAddWhereValue. The only methods available for the developer to use in the method chain are the ones listed in ICanAddWhereValue – in this case the IsEqualTo(value) and IsNotEqualTo(value) methods.

That’s how to have the rules of grammar enforced. With it, we can be sure that the delete query has everything we need to build it properly, once an executing method is called.

**Summary**

It takes a bit of planning and work, so you probably don’t want to build a fluent interface for everything. I find them useful for code I put into a library for re-use, or to give to other programmers. With a fluent interface, IntelliSense handles many of the “how do I use this” questions, that no one ever bothers to look in the documentation for.

I’ve also put the source code up on GitHub at <https://gist.github.com/ScottLilly/6976156>.