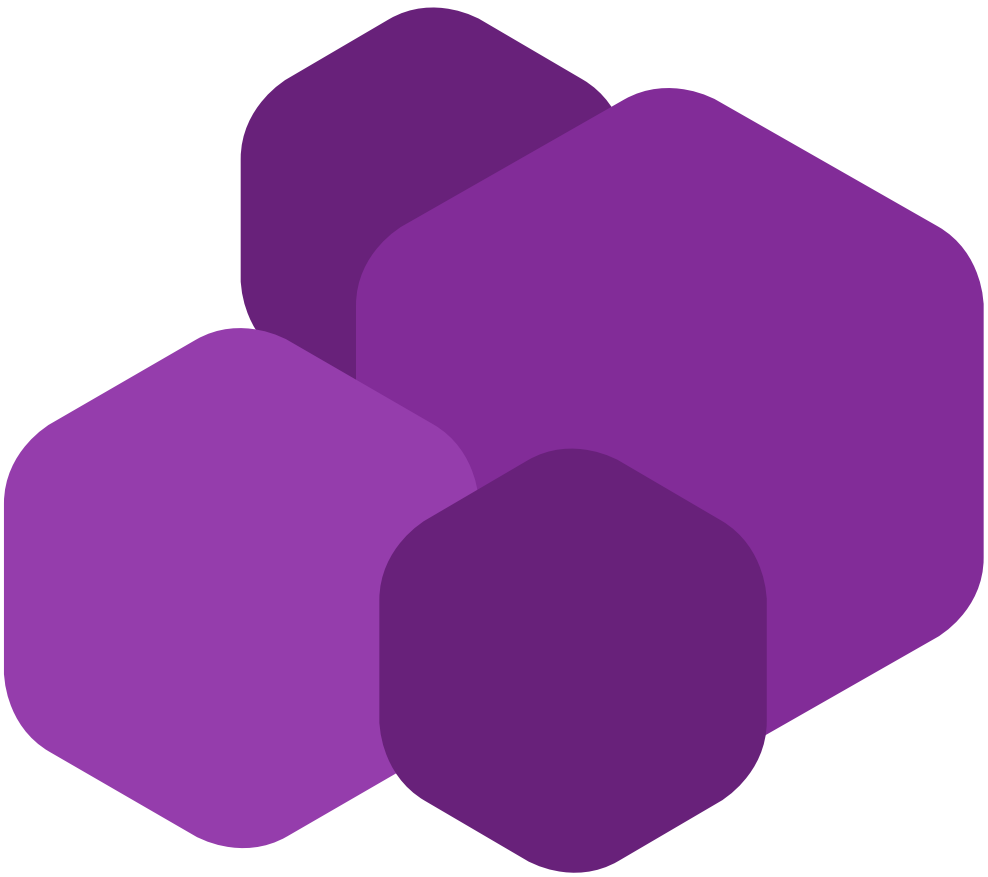
CODING PRACTICES

Yacoub Massad

Writing Honest Methods in C#

This tutorial describes approaches for making methods/functions more honest. A more honest method makes it easier for readers to understand what the method does by reading its signature, i.e., without reading its implementation.



Introduction

We developers spend lot of our time reading code. We read code so that we know how to change it to implement a new feature, fix a bug, etc. It is much better to work in a code base where code is easy to read and understand.

One thing that can make code readable is good naming. Giving variables, classes, and methods good names makes code easier to read.

But why is this true?

Consider the following method:

```
public static Discount CalculateDiscount(Cart cart, Customer customer)
{
  decimal discountRatio;

  if (customer.Status == CustomerStatus.Normal
    || customer.NumberOfTimesStatusUsed > 10)
  {
    discountRatio = 0;
  }
  else if (customer.Status == CustomerStatus.Silver)
  {
    discountRatio = 0.05m;
    customer.NumberOfTimesStatusUsed++;
  }
  else if (customer.Status == CustomerStatus.Gold)
  {
    discountRatio = 0.1m;
    customer.NumberOfTimesStatusUsed++;
  }
  else
  {
    throw new Exception("Invalid CustomerStatus");
  }

  var discount = discountRatio * Cart.GetTotalAmount(cart);

  return new Discount(discount);
}
```

Consider the `discountRatio` variable for example. What would we lose if we renamed it to "x"? Well, if we did so, the reader of the code will need to spend more time to understand what it represents.

Currently, as the reader reads the method and sees the statements that assign values to the `discountRatio` variable (lines 7, 11, and 16), she/he will immediately realize that the purpose of the code (lines 4-22) is to calculate the discount ratio.

Keeping the variable name as "x", will only leave the reader wondering what it means. Maybe the fact that it is assigned the values of 0, 0.05, or 0.1 will make him/her guess that this is the discount ratio. To be sure, the reader has to reach line 24 where the x variable is used. Because this variable is multiplied by the total amount in order to calculate the discount, the reader can infer that x is actually the discount ratio.

What about method names? How can they be helpful?

When the reader is reading code that invokes a method, a good method name will enable the reader to understand this code.

For example, look at the `GetTotalAmount` method in line 24 in the last example.

Reading the name of the method, the reader can understand immediately that the code is trying to get the total amount for the items in the cart. This allows the reader to continue understanding this line; that the discount is calculated by multiplying the discount ratio with the total amount.

The reader does this without the need to go to the implementation of the `GetTotalAmount` method to see how it is implemented. The name itself gave enough information about what this method does.

This ability to give the reader enough confidence about what a method does **without having to read its implementation** (by reading its signature alone) is very much desirable.

Note: *the signature of a method includes its name, its parameters, and its return type.*

Although naming is important, it is not always the best way to give the reader such confidence. Consider for example the `CalculateDiscount` method. When the reader sees this method called, for example as in this line:

```
var discount = CalculateDiscount(cart, customer);
```

..he/she will understand the purpose of this line; i.e., calculating the discount that the customer gets for the items in the cart. However, there are some missing details that the reader will not be able to know unless he goes and reads the implementation.

Can you guess what are these missing details?

If you go back and take another look at the implementation of the `CalculateDiscount` method, you will notice that if the customer has `Silver` or `Gold` status, a property on the `Customer` object named `NumberOfTimesStatusUsed` will be incremented. This property is used to keep track of how many times the customer has used Silver or Gold status. Customers with Silver or Gold status can use such status for discount for a maximum of ten times.

We can say that the `CalculateDiscount` method is dishonest. It does things that are not declared in its signature.

Of course, we can rename the method to `CalculateDiscountAndUpdateNumberOfTimesStatusUsed`. This will fix the problem. In theory, we can depend on naming alone to make methods honest. We simply state what the method does in its method name, including any hidden details.

In practice, however, we cannot rely only on method names for the following reasons:

1. Methods are easily renamed. A developer might change a method name for any reason, potentially making the method dishonest.

2. In order to be completely honest, a method name would need to declare all the things that it does. This would make some method names very long.

3. A method can call other methods, which themselves call other methods, etc. For the top-level method to be honest, its name should declare all the things that all the methods it calls (directly or indirectly) do. How long would the top-level method name become?

Before going into other techniques for making methods more honest, let's first talk about what methods do and which parts of that they need to declare, in order for us to consider them honest.

The concept of method honesty is an informal one. Having said that, I think the following is the list of things a method should declare in its signature in order to say it is honest (let's call this list the honesty list):

1. Any changes to global state. For example, this could be mutating a global variable that keeps track of the total number of documents the application has processed so far.

2. Any change to the state external to the application. For example, this could be adding a record to the database. Or modifying a file in the file system.

3. Any mutation to method parameters. The `CalculateDiscount` method is an example of a method that mutates one of its parameters.

4. How the return value is generated. This can be divided into two:
   4-a. Any effect on the return value of the method that depends on global or external state. For example, this includes reading from a global variable, reading from the file system, or reading the current time using the system timer.

   4-b. Everything but the effects described in 4.a. This is equal to:
      i. The effect of the input parameters on the return value
      ii. The knowledge that the method contains.

To explain this point, I am using the following example:

```
public static int DoSomething(int input)
{
  return input + 1;
}
```

In this method, the output is equal to the input parameter, plus one. The effect of the input parameter on the return value is that it participates in the addition operation that eventually produces the output. The "+ 1" part is the knowledge this method contains.

This method does not mutate anything, and it does not depend on anything external to generate its return value. Therefore, only 4.b is relevant to the honesty of this method.

To make this method honest, we can simply rename it like this:

```
public static int AddOne(int input)
{
  return input + 1;
}
```

Now, we can easily argue that the effect of the input parameter on the return value is clear from the signature (4.b.i); i.e., it is the value that 1 is going to be added to. We can also easily argue that the

knowledge that the method contains is also clear (4.b.ii); i.e., adding one (to the input).

Because this method mutates nothing, and its return value does not depend on any global or external state, we can consider this method to be **pure**.

For more information about purity, see the Functional Programming for C# Developers article.

I will first talk about honesty for pure methods, and will then discuss honesty related to impure methods.

Making pure methods more honest

Let's assume for a moment that when we read code and see a method invocation, we can immediately tell if the method is pure or not without needing to go and read the method implementation.

For example, imagine that there is a feature in Visual Studio that colors the method invocation with a specific color (say green) to indicate to the reader that this method is pure. Now, when you read code and see a method invocation in green, you know the method is pure, and therefore you don't need to worry about points 1, 2, 3, and 4.a from the honesty list. Although you do need to keep 4.b (the effect of the input parameters on the return value, and the knowledge of the method) in mind.

Here are some techniques that can help make pure methods honest:

Make the return type of the method declare all the possible outputs of the method.

Consider the following method signature:

```
public Customer FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

This method takes in an array of customers, and search criteria, and it returns a matching customer object.

But what happens when no customer object in the array matches the specified criteria?

Does the method return null? Or does it throw an exception? Or worse, could it return a customer that has some `Id` property set to 0?

To answer these questions, we have to look at the implementation of the method.

The problem with this method is that the return type of the method does not declare all the possible outputs. It just declares that it returns a "Customer".

Here is a better signature:

```
public Maybe<Customer> FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

The `Maybe` type is a type that represents an optional value; it either has a value or it does not. This is similar to the nullable type feature in C# that is available for value types.

For more information about the `Maybe` type, read the Designing Data Objects in C# and F# article.

Now, we can answer the above questions without the need to read the implementation of the method. We know now that in case no customer matched the criteria, the method will return `Maybe<Customer>` that does not have a value.

It is still possible that the method above throws an exception when the customer is not found, or even return a `Maybe<Customer>` that contains a customer with some Id property set to 0.

If this happens, then the method signature is lying.

The problem is no longer that the method signature does not answer some questions. Now, the problem is that the method signature provides deceitful answers. This is much worse.

This means that developers working on a codebase are required to have some minimum level of discipline as to not make such mistakes. A developer who reads a signature of the method should safely assume that it does not deceive her/him. The best thing to have is a signature that answers all the questions correctly.

However, a signature that leaves some questions unanswered is better than one that gives deceitful answers.

I don't think there is another solution to this problem. Consider for example that the method above is implemented in such a way that it returns some `Customer` object with random data without even searching the list of customers. What prevents a developer from doing so?

By the way, C# 8 is expected to have nullable support for reference types, this would allow us to write the method signature like this:

```
public Customer? FindCustomer(
  ImmutableArray<Customer> customer,
  string criteria)
```

Make the method parameter types declare all possible input values.

Consider for example that you find the following invocation of the `FindCustomer` method:

```
var customer = FindCustomer(customers, "TotalPurchases > 1000$");
```

From the signature of the method, you know that the second parameter is the search criteria. From the argument passed in this specific invocation, you know that the search criteria supports filtering based on total purchases of the customer.

But what other ways of filtering does the search criteria support?

For example, can we search by the number of purchases in a specific month? Does the search criteria support the greater than or equal to operator (>=)? Can we have multiple filtering conditions? Also, does the method throw an exception if the search criteria is not supported? Or does it silently return a `Maybe<Customer>` with no value?

The answers for these questions will probably be scattered all over the place inside the implementation of the `FindCustomer` method and the many methods that it probably calls.

To make things better, we can change the type of the second parameter (the criteria parameter) so that we cannot pass values for such parameter that are not supported.

Consider this updated signature:

```
public Maybe<Customer> FindCustomer(
  ImmutableArray<Customer> customer,
  CustomerSearchCriteria criteria)
```

Now with this change, any random string cannot be passed as the criteria. Only valid instances of the `CustomerSearchCriteria` class can be passed.

We should use the type system of .NET to make sure that the `CustomerSearchCriteria` class does not allow us to express search criteria that are not supported.

In a previous article, Designing Data Objects in C# and F#, I talked about making invalid states unrepresentable. What we need to do is make invalid search criteria unpresentable by the `CustomerSearchCriteria` class. Please read that article for more details.

Here I will show the definition of the `CustomerSearchCriteria` in F# as it makes it easier for the developer reading the code to quickly understand supported cases.

```
type TotalPurchasesConditionsOperator =
    | Equals
    | GreaterThan
    | SmallerThan

type TotalNumberOfPurchasesInPeriodOperator =
    | Equals
    | GreaterThan
    | SmallerThan
    | GreaterThanOrEquals
    | SmallerThanOrEquals

type CustomerSearchCriteria =
    | TotalPurchasesCondition of
        operator: TotalPurchasesConditionsOperator *
        amount: Money
    | TotalNumberOfPurchasesInPeriod of
        operator: TotalNumberOfPurchasesInPeriodOperator *
        startDate: DateTimeOffset *
        endDate: DateTimeOffset *
        number: int
    | Any of conditions: ImmutableArray<CustomerSearchCriteria>
    | All of conditions: ImmutableArray<CustomerSearchCriteria>
```

This code defines three F# discriminated unions:

- CustomerSearchCriteria (4 cases)
- TotalPurchasesConditionsOperator (3 cases)
- TotalNumberOfPurchasesInPeriodOperator (5 cases)

These types become part of the signature of the method because the method's second parameter is of type `CustomerSearchCriteria`. From these types, we can know that the method supports searching based on the customer's total purchases, and also on the number of purchases in a specific period.

For the total purchases condition, three operators are defined (equals, greater than, and less than). For the other condition type, five operators are supported.

The `CustomerSearchCriteria` type also shows that we can filter based on the total number of purchases in any period defined by starting and ending dates. Also, we can see from the definition that we can combine conditions using `Any`, or `All`. The method now is much more honest.

The reason I used F# here to define the types is that in C#, we would need much more code to express the valid states of `CustomerSearchCriteria`. See the article Designing Data Objects in C# and F# for more details.

Note that there could be a case where the criteria comes from the user as a string. I.e., the user types the search criteria in some text box. In this case, there should be a special method to parse such string and generate a corresponding `CustomerSearchCriteria` object like this:

```
public Maybe<CustomerSearchCriteria> Parse(string criteria)
```

This specialized method returns no value if the criteria does not represent a valid criterion, or a `CustomerSearchCriteria` object otherwise.

We cannot depend on the .NET type system here to make sure that the caller of the method passes valid search criteria. To make the `Parse` method signature more honest, we can include some documentation about the syntax of the criteria string as Xml Documentation Comments. See this reference for more details.

Make the method name describe what it does.

One question a reader might ask about the `FindCustomer` method is: What happens if there are two customers that match the criteria? Does the method return the first customer? Does it throw an exception?

We can answer this question using the method signature by updating the method name. Assuming that the method returns the first customer that matches the criteria, we can rename the method to `FindFirstCustomer`.

A method name can become long. In many cases, this is not an issue. It is better to read a method name that contains ten words and understand what it does, than to go to the implementation of the method and read ten lines of code (or even more if the method calls other methods whose names are not descriptive enough).

Please note that a method name does not need to declare things that the method parameters already declare.

For example, consider the `CalculateDiscount` method from the first example in this article. There is no need to call it `CalculateDiscountForItemsInCartForASpecificCustomer`. The `cart` and `customer` parameters already make it clear that such discount calculation is related to the cart and to a specific customer.

Another way to decrease the length of method names is to learn more about the domain. Sometimes, you can learn a new term that enables you to replace a few or several words in the method name, with one or two words.

Making impure methods more honest

Although pure methods are preferred over impure ones, without impure code, programs cannot do anything useful. We need impure code to read from and write to the console, the filesystem, the database, read the system timer, send a web request, generate a random number, etc.

Having said this, there are approaches to maximize pure code and minimize impure code. Mark Seemann has introduced one such approach which is called *Dependency Rejection*. You can read about it here: http://blog.ploeh.dk/2017/01/27/from-dependency-injection-to-dependency-rejection/

The basic idea of this approach is that we should design the core functionality of our applications to be completely pure.

When the program receives a request to do something, e.g. handle a Web API request, or handle a UI command, etc., the program does the following:

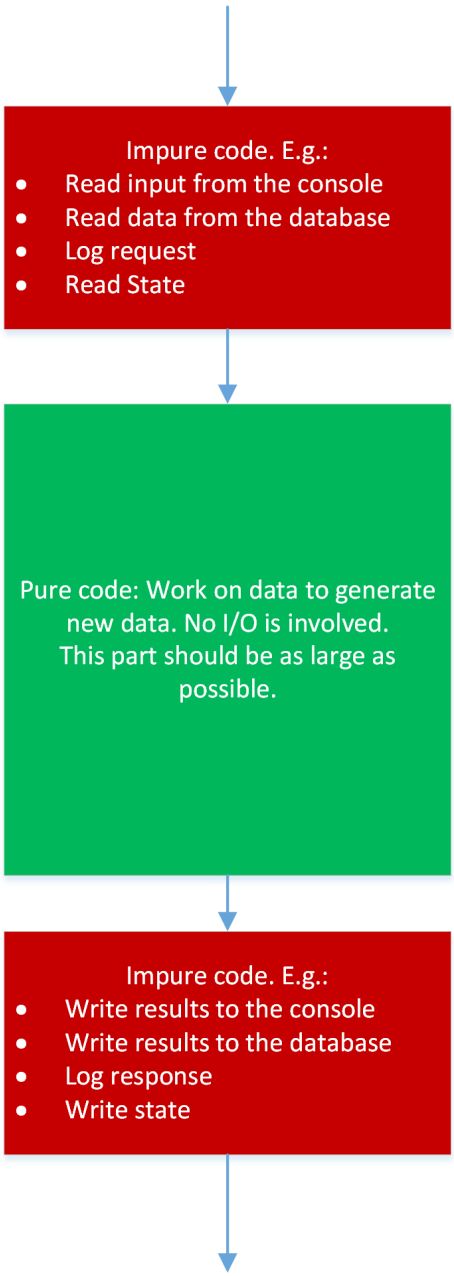


Figure 1 Dependency Rejection

1. It runs some impure code to prepare the data that will be given as input to the pure functionality. For example: read the list of orders for the customer whose id is specified in the Web API request from the database.

2. It runs the pure functionality over that input and collects the output of such pure functionality. For example: filter the orders to include only the ones whose total amount is larger than 1000$, group them by month, and then generate a report file (in memory) in PDF format from such data.

3. It runs some other impure code given the output from the pure functionality. For example: write to the log file that report XYZ has been requested for Customer X and return the report as a Web API response.

The adjacent figure represents this. Because the pure part is preceded and followed by impure parts, Mark Seemann calls this the impure/pure/impure sandwich.

The impure parts should be very simple. They should not contain much logic. All the business logic should be inside the pure part. If we can manage to have our applications designed like this, then we shouldn't care so much about the impure parts because they are very small. If the methods in the pure part are honest, then all is done.

However, there are cases where the program cannot be represented by the impure/pure/impure sandwich. In some programs, we need to run impure code frequently as we are processing the business logic.

For example, what if based on the orders a customer has placed in the last month (count, total amount, etc.), we need to talk to a different web service to retrieve some data required to generate the report?

Should we talk to all web services in the first impure part so that all data is ready for the pure part to select from? Or should we move such business logic (having different data in the report based on the orders) to the first impure part?

Both solutions have their problems.

Even in such programs, there are solutions to separate pure and impure code. Mark Seemann talks about one solution here: http://blog.ploeh.dk/2017/07/10/pure-interactions/

I am not going to talk about such a solution in this article. Instead, I am going to assume that we need to mix between impure and pure code.

Figure 2 Mixing pure and impure methods

In this figure, the green methods are pure, the red methods are impure, and the orange methods are "pure" methods that became impure because they call impure methods.

Let's explain this in more details:

- Green methods are completely pure. They contain pure logic.

- Red methods are very simple methods. They contain no business logic. They simply perform I/O, read global state, and/or update global state. An example of such methods is the `File.ReadAllBytes` method from the .NET framework. This method returns the binary content of a file, given its filename. One might argue that the `File.ReadAllBytes` method contains some logic. After all, such method and the methods it calls have to deal with a lot of concerns, like detecting if the file exists, authorizing the caller, handling the file system structure, etc. When I say that this method contains no logic, I mean it does not contain logic related to the problem the application is trying to solve. As far as the application is concerned, the `File.ReadAllBytes` "simply" reads the contents of the file.

- Orange methods contain logic as green methods do. But because they need to interact with the outside world as they are processing, they call impure methods. Going back to the example of contacting different web services based on customer orders, this logic would be in an orange method.

Let's talk about honesty of orange methods. How to make such methods honest?

The points I talked about for pure methods are still valid here. However, we need impure methods to tell us in their signatures about the *impurities* that they have.

Consider this example:

```
public XYZReport GenerateXYZReport(Customer customer, ImmutableArray<Order> orders)
```

This method generates some report (called the XYZ report) for some customer. The body of the method (not shown here) invokes some impure methods to collect some data.

Currently, the method signature does not declare that fact.

This can be partially fixed by the imaginary feature of Visual Studio that gives method invocations different colors based on whether the methods being invoked are pure, or not.

Even if you know by looking at the method invocation color that the method is impure, you don't know exactly what it does. Does it invoke a web service? Does it write to some file on the file system? Does it read the system timer? Does it update or read some global variable? Does it mutate any parameters?

What we can do is convert this method into a pure method.

Let's assume that the method above reads the current time using the system timer to include it in the report. Instead of making the method read the system timer itself, we can delegate such responsibility to the caller and make this method take the current time as a parameter like this:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  DateTime currentTime)
```

This is a simple application of the *Dependency Rejection* concept. The GenerateXYZReport method no longer depends on the system timer.

As explained before, this is not always possible.

Assume for example, that the report generation takes 30 minutes to complete. Different parts of the report will be generated at times that are minutes away from each other. Assume that it is important for business that each part includes the exact time in which it was generated. This means that during generation, we need to read the system timer at different times. We cannot simply take a simple value at the beginning of the generation process.

Consider this updated signature:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  Func<DateTime> readCurrentTime)
```

Now, the method takes a function as a parameter. In functional programming terms, this is called a higher order function. When the method needs to read the time, instead of invoking the `DateTime.Now` impure property getter, it invokes the `readCurrentTime` parameter to obtain the current time.

If the GenerateXYZReport function does not call any other impure methods (other than reading the system timer), then we can say that this function, in its current form, is a potentially-pure function.

It is potentially-pure because its purity depends on the purity of the `readCurrentTime` function passed to it. For example, if we call this method like this:

```
var result = GenerateXYZReport(customer, orders, () => DateTime.Now);
```

..then the function becomes impure because we passed a `readCurrentTime` function that reads the current system time via the impure `DateTime.Now` property getter.

However, we can also call it like this:

```
var result = GenerateXYZReport(customer, orders, () => new DateTime(2018, 6, 2));
```

Now, it is a pure function because the `readCurrentTime` function returns the same value every time.

If the method also communicates with some web services to obtain custom data based on orders, we can add another function parameter to the method like this:

```
public Report GenerateXYZReport(
  Customer customer,
  ImmutableArray<Order> orders,
  Func<DateTime> readCurrentTime,
  Func<Uri, SomeReportCustomData> obtainCustomDataFromWebService)
```

The `obtainCustomDataFromWebService` function would be used by the method to communicate with some web service to obtain some data.

Note that the logic to determine which web service to use is still part of the GenerateXYZReport method. Only the impure part is extracted as a function parameter.

Although the GenerateXYZReport method is now potentially-pure, we know that in production, we are going to pass impure functions for the `readCurrentTime` and `obtainCustomDataFromWebService` parameters.

Why is this useful then?

It is useful because now the method is honest about the impurities that it contains.

We can now configure the imaginary Visual Studio feature to treat potentially-pure functions, as pure. If a method declares all impurities as function parameters, Visual Studio would be able to tell us with certainty that the method does not do anything impure apart from the things it is declaring via the function parameters.

What about methods that read or write to global state? And methods that mutate their parameters?

- For a method that reads global state, we should prefer to make it take the value of the state as an input parameter (like how we added the `currentTime` parameter). If this is not possible, e.g. the method needs to read state at many different times, we can make such method take a `Func<State>` (like how we added the `readCurrentTime` parameter).

- For a method that updates global state, we should prefer to make it return the new state value using the return type of the method. If the method already has a return type, we can use a tuple to return both the original return value and the new state value. Again, if for some reason we need to update the state multiple times during the method execution, we can make the method take some `Action<State>` parameter that we can call to set the new state.

- For a method that mutates any of its parameters, we should make the parameter types immutable and instead return any new values via the return type of the method.

The CalculateDiscount method I displayed in the beginning of this article has the following signature:

```
public static Discount CalculateDiscount(Cart cart, Customer customer)
```

The method mutated the `Customer` parameter to increase the number of times the customer `Status` has been used. We can change the method signature like this to avoid mutation:

```
public static (Discount discount, int numberOfTimesStatusUsed)
CalculateDiscount(Cart cart, Customer customer)
```

Now, it is the responsibility of the caller to somehow store the new value of `numberOfTimesStatusUsed` to where it needs to be stored. But now the method is more honest. It tells us that one output of the method is the new number of times the status was used.

For more information about immutability, see the Designing Data Objects in C# and F# article.
All my suggestions for making impure methods honest end up making the methods pure or potentially-pure. When this happens, the method signature becomes honest, but it also adds more work for the caller.

> For example, the method that calls the GenerateXYZReport method needs to pass values for both the readCurrentTime and the obtainCustomDataFromWebService parameters. But in order for the caller method itself to be potentially-pure, it has to take these as parameters itself. This would mean that all the methods that end up calling GenerateXYZReport (directly or indirectly) would need to take such function parameters if they wanted to be honest.

This does not scale well because if we change a lower-level method to take a new function parameter, all the methods at a higher level that invoke it directly or indirectly would need to be updated to take such parameters themselves.

How can we compose honest functions in a way that scales? I will talk about this in an upcoming article.

In this article, I also talked about an imaginary Visual Studio feature that helps the reader know which methods are pure and which are not. I will expand on this in an upcoming article.

Conclusion:

This article discussed honest methods/functions.

An honest function makes it easier to understand what it does from its signature, without the need to read its implementation. We can make methods more honest by making them declare all possible outputs and all possible inputs, by using names that describe what they do, and by making methods pure or potentially-pure.

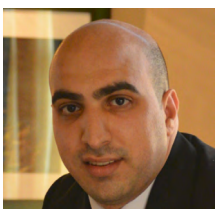
Potentially-pure methods are methods whose bodies are pure, but that invoke functions passed to them as parameters that could be pure or impure.

This tutorial ended on the note of explaining a problem related to making all impure methods in an application potentially-pure. We will see more about this problem and discuss a solution in an upcoming article.

Yacoub Massad

Author

Yacoub Massad is a software developer who works mainly with Microsoft technologies. Currently, he works at Zeva International where he uses C#, .NET, and other technologies to create eDiscovery solutions. He is interested in learning and writing about software design principles that aim at creating maintainable software. You can view his blog posts at criticalsoftwareblog.com.



Thanks to Damir Arh *for reviewing this article.*