ApiProtocols Example

Abstract

This example shows how contracts allow you to make the often implicit API protocol on a class explicit. API protocols are rules about what states an object goes through and when it is okay to call particular methods and properties. Clients are supposed to follow these rules, but often clients have to discover these rules by trial-and-error.

The example consists of a library exposing a class that has to be used in a certain way. A separate client application makes use of this class.

In this sample, you will learn how to write contracts that describe protocols on your classes and how such protocols are enforced on clients.

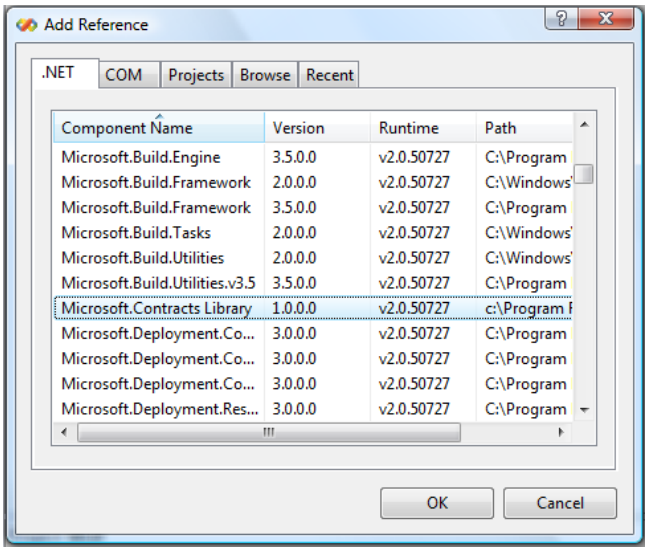
1 Adding the Contract Library Reference

If you are using Visual Studio 2008, or if you for some reason want to target a pre-v4 .NET runtime, then you need to:

- Change the target framework of the project.
- Manually add a reference to Microsoft.Contracts.dll

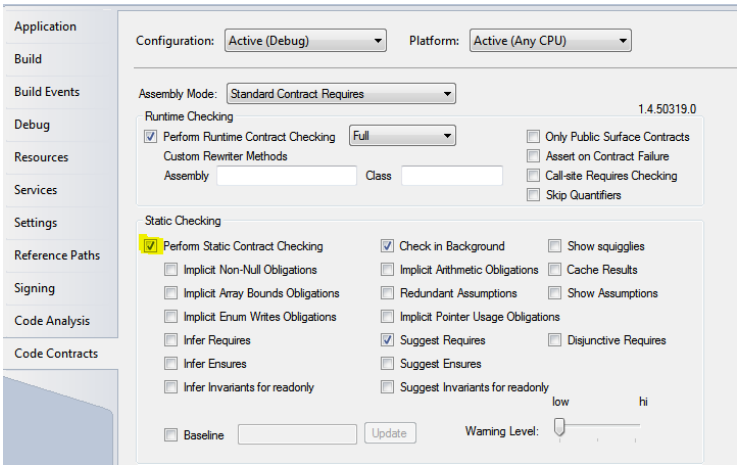
Otherwise, you may skip this section and go directly the next section!

To add the reference, open the ApiProtocols solution and right-click on References in the ApiProtocols project and select Add Reference. Find the Microsoft.Contracts library in the .NET tab as shown below and click OK.



2 A Simple Protocol: Nullable

Let's start by running the project with contract checking enabled. Go to the Properties of both projects Client and ApiProtocols, select the Code Contracts pane (at the bottom), and enable runtime and static checking by clicking on the appropriate boxes, as shown in this screenshot:



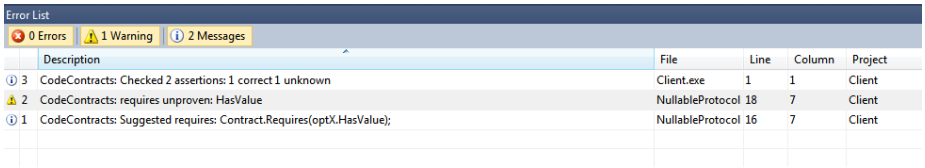
Then build the solution and run it (or hit F5). You should get an InvalidOperation exception in method Sum because we are trying to get the Value of an optional int which has no value.

You are probably familiar with nullable types in C#. Here, we are using nullable ints. A nullable int can hold an integer value or no value. The method

Value on nullables throws an exception if called on nullables that have no value. Thus this type has a very simple protocol:

> To call optX.Value, optX.HasValue must be true.

Stop the execution and look at the warning list (if no warnings are displayed, rebuild Client). You should see the static checker warn about the misuse of the nullable optX:



Double click on the warning stating that the requires HasValue is unproven. This should take you to the the spot we just hit in the debugger.

We specified this contract for the Nullable type (we'll show you in a minute how to specify such contracts). The code here is trying to use a nullable argument optX without knowing if optX.HasValue is true. The checker complains about this. Note that the access on the second nullable value optY is correct, as it tests for HasValue first.

Fix the code so it won't bother us in future, e.g., mimicking the test around the access to optY.Value.

```
if (optX.HasValue) result += optX.Value;
```

3 A Class with a Protocol

Now let's take a look at class ClassWithProtocol. Objects of this type go through different phases. After construction, an object of this type must be initialized by calling Initialize before any other operation can be meaningfully performed. E.g., property Data returns the data passed to Initialize and thus should not be called prior to Initialize . Similarly, property ComputedData is only available after additionally calling Compute.

Such a protocol can be described by thinking of the object as being in one of three different states: NotReady, Initialized , Computed. The following table shows what operations are available in each state and how the object's state changes according to the operations:

State	Allowed Operations	New state(s)
-	Constructor	NotReady
NotReady	Initialize	Initialized
Initialized	Data, Compute	unchanged after Data, Computed (after Compute)
Computed	Data, ComputedData	unchanged

To save you some typing, we already wrote the contracts for this protocol. You can find them in the file ClassWithProtocolFinal. Feel free to copy-paste from there as we go through adding contracts in the rest of this sample.

4 Adding the State

There are many ways we could make the state of the object explicit, e.g., through multiple properties. The simplest way is to use an actual State property and an enum listing the states as we had in our table above.

Add the following code in ClassWithProtocol to keep track of what state the object is in:

```
public enum S {
  NotReady, Initialized , Computed
}

private S _state ;
public S State
{
  get
  {
    return _state ;
  }
}
```

Now it makes sense to update the state in each operation according to our table. Thus, in the constructor, we set the state to NotReady.

```
public ClassWithProtocol()
{
  _state = S.NotReady;
}
```

Similarly, in Initialize and Compute, we set the state to Initialized and Computed respectively.

```
public void Initialize (string data)
{
  this._data = data;
  _state = S. Initialized ;
}

public void Compute(string prefix)
{
  this._computedData = prefix + _data;
  _state = S.Computed;
}
```

Now, so far all we have done is make the state of our object explicit. That was the most important step, since now we can actually specify how to use this class.

E.g., we can now add pre-conditions to all method to specify what state(s) the object needs to be in so the operation makes sense.

Let's first do the two properties. According to our table, the Data property is accessible in all states but the NotReady state. We can easily specify this by the following contract:

```
public string Data
{
  get
  {
    Contract.Requires(State != S.NotReady);

    return _data;
  }
}
```

An important thing to note here is that we used the publicly visible property State and not our private backing field _state. It is important that contracts that callers must observe are visible to callers!

For the ComputedData property, the state must be equal to Computed. So we add the following:

```
public string ComputedData
{
  get
  {
    Contract.Requires(State == S.Computed);

    return _computedData;
  }
}
```

Now let's add the appropriate pre-conditions to the two remaining methods: Initialize requires NotReady, and Compute requires Initialized.

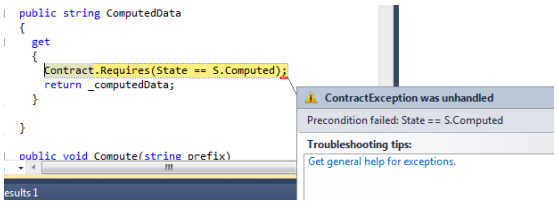
```
public void Initialize (string data)
{
  Contract.Requires(State == S.NotReady);

  this._data = data;
  _state = S.Initialized;
}

public void Compute(string prefix)
{
  Contract.Requires(State == S.Initialized);

  this._computedData = prefix + _data;
  _state = S.Computed;
}
```

At this point, we have enough contracts for runtime checking the protocol on clients. Hit F5 to compile and run. The execution should stop with the following message:



If you look at the call stack, you see that the execution is stopped at the point where we access the ComputedData property, but the state of our object is Initialized .

5 Making Static Checking Work

The specifications in the previous section are enough to catch errors *at runtime* made by the client. In order to catch errors in the implementation, we need contracts that specify how the methods change the state, in particular, what state is ensured by each method. These same contracts also allow the static checker to catch these errors at build time.

Properties are assumed to be pure and thus don't change the state. We therefore don't need to specify a post state for them. According to our table, we add the following post-condition to the constructor:

```
Contract.Ensures(this.State == S.NotReady);
```

Similarly, we add the following to Initialize :

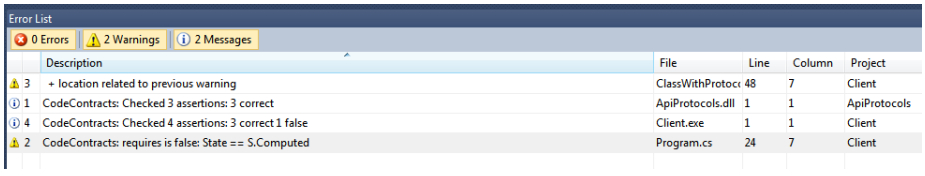
```
Contract.Ensures(this.State == S. Initialized );
```

and the following to Compute:

```
Contract.Ensures(this.State == S.Computed);
```

These contracts make sure that the implementation actually matches our table of state transitions. Suppose we forgot to update the _state variable in one of our methods. These specifications will catch that.

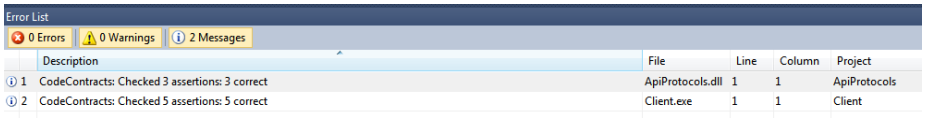
If you rebuild the solution now, you should get the following warning from the static contract checker:



Double-clicking on the warning takes you to the access of ComputedData that failed with the runtime check as expected. We can now fix the client code by calling Compute before accessing ComputedData.

```
c.Compute("whatever");
var data = c.ComputedData;
```

Now running the code produces no runtime error, and the static checker will produce the following output:



6 Taking Advantage of State Invariants

Now that we have our protocol setup, we can strengthen the post conditions on our properties. It would be nice to guarantee that the properties always return non-null strings. Let's start by turning on non-null checking by selecting the Properties on each project by selecting the implicit non-null checkbox.

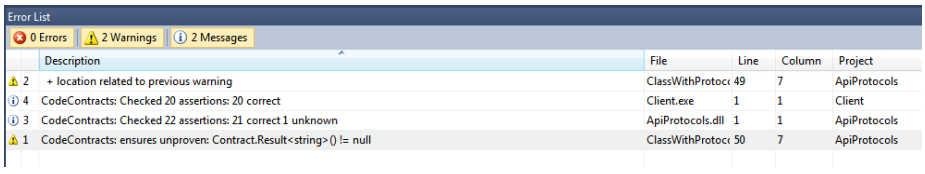
Building should produce a warning that data (the result of ComputedData) might be null in the Main method. Let's look at the ClassWithProtocol code again. If we require the argument to Compute to be non-null, then clearly, if we are in the Computed state, the field _computedData should be non-null. We add the following contracts to make this explicit. On Compute, let's add the following requires (recall that you can use the crn TAB TAB snippet):

```
Contract.Requires(prefix != null);
```

We also add the following ensures on the ComputedData getter (the cen TAB TAB snippet will do it):

```
Contract.Ensures(Contract.Result<string>() != null);
```

If you build now, you should get the following output:



The checker is now happy with the client code and can guarantee that the ComputedData value is non-null. However, it cannot yet prove the ensures of that getter in the implementation. The reason is that the invariant relating the _state and the non-nullness of _computedData is not evident to the checker. We can help it by specifying the following object invariant in ClassWithProtocol.

```
[ContractInvariantMethod]
protected void ObjectInvariant()
{
  Contract.Invariant(_state != S.Computed || _computedData != null);
}
```

The invariant specifies that either we are not in the Computed state, or field _computedData is non-null. Building now should return no more warnings, as the checker can make sure that every method actually satisfies this invariant.

As an exercise, you can try adding a similar invariant that specifies that _data is non-null in all states but the NotReady state.

7 Conditional Transitions

Not all protocols are as straight-forward as the one we have seen so far. Often, methods may have multiple outcomes. For example, suppose Compute had to access the file system and could only compute the proper value if a particular file was there. Thus, the method would not transition the state to Computed in all cases.

To examine this possibility, let's modify Compute so it returns a **bool** to indicate whether it succeeded in the computation. Otherwise, the state of the object remains Initialized , so the computation could be attempted again. To specify this, we modify the existing ensures contract to the following one:

```
public bool Compute(string prefix)
{
  Contract.Requires(prefix != null);
  Contract.Requires(State == S.Initialized);
  Contract.Ensures(Contract.Result<bool>() && State == S.Computed ||
                   !Contract.Result<bool>() && State == S.Initialized);
```

Don't forget to also return a boolean value. Since we haven't implemented the failing case, let's just return true. Building now should produce a warning in the client code that State == S.Computed might be false at the call to the ComputedData getter. In order for the client to satisfy the protocol, it must check the return value of Compute.

If we modify the client as follows:

```
if (c.Compute("whatever"))
{
  var data = c.ComputedData;

  Console.WriteLine(data.ToString());
}
```

the warning disappears. We have thus successfully refined the protocol of our class and automatically found client code that needed to be adjusted to our change.