Design By Contract

by Sascha Frick

When we write new software, we strive to reuse existing code in the form of classes and components. Delphi's VCL contains a rich set of ready-to-use classes and many third-party vendors offer additional components for instant reuse.

While it is possible with Delphi to write procedural programs, I believe you are far better off using Delphi's object oriented features as much as possible. With object orientation, a class is either reused by means of inheritance or by means of aggregation. It has become clear over the years that the latter, being a black box approach, is the preferred way of reuse most of the time: using a class as part of the implementation details of another class to implement specific behaviour is far easier to achieve than proper use of inheritance. After all, all the hype about components and reuse stems from the idea of easy to use pluggable software components to build new systems by simply putting existing software pieces together. Delphi itself uses this sort of reuse most of the time. When you place a `TButton` on a form you are not subclassing a new `TButton` class but are reusing the existing `TButton` class as part of your form.

While reusing classes by aggregation is easier than by using inheritance, it is not a trivial matter! For almost all components except the most simple, we have to know a great deal about the component itself in order to be able to use it effectively and correctly. Usually, a good starting point for understanding how to use a component is its documentation. But, many times even very good documentation does not tell you everything you need to know and you end up digging into the source code just to find out how exactly things have to be done in order to get the desired results.

In other words: the component you wish to use demands that you (its client) meet certain obligations in order for it to produce the desired results. Now, wouldn't it be nice, if a component stipulated your obligations for its use in a clear and testable form, so you could immediately see what your job is in order for the component to work as designed? That's the basic idea behind design by contract: the client of a component and the component itself form a contract that states their mutual responsibilities.

This design technique was introduced by Bertrand Meyer and is a key feature of his programming language Eiffel, that supports design by contract very elegantly. 'Eiffel?', you may ask, 'I don't use Eiffel!'. Don't despair, we will see shortly how we can benefit from using the approach with our favourite development tool. Fortunately, with the advent of version 3, the designers of Delphi added support for assertions.

Assertions lie at the heart of design by contract. An assertion is a `Boolean` expression that is never supposed to become `False`. If an assertion does evaluate to `False`, the assertion is said to have failed which is considered a *programmer error* worth raising an exception for. Since all contracts are stipulated in the form of one or more assertions, a failed assertion is a *breach of contract*. Usually, assertions are only used during testing and debugging and their evaluation is disabled in the final product. We will see in a moment that Delphi offers excellent support for assertions ready for use!

So, how exactly does Delphi support assertions? Starting with version 3, there is a procedure called `Assert` with the following signature:

```
procedure Assert(expr : Boolean
  [; const msg: string]);
```

The help file states: *'Use* `Assert` *as a debugging tool to test that conditions assumed to be true are never violated. [...]* `Assert` *takes a boolean expression and an optional message string as parameters. If the boolean test fails,* `Assert` *raises an* `EAssertionFailed` *exception.'*

You can see that the use of `Assert` fits nicely into our previous definition of what an assertion is: a `Boolean` expression that is expected to always be `True`. We also stated that a failed assertion should raise an exception. And that is exactly what happens whenever `Assert` is called with a expression that evaluates to `False`: we get the special exception `EAssertion-Failed`.

The second parameter to `Assert` is an optional message string for the exception. If you omit it, a default message is used for that purpose. The message is displayed along with the filename (including the complete pathname) and the line number on which the `Assert` failed. Delphi uses some compiler magic to make the `Assert` procedure work as intended, but we will not bother to go into details here.

We discussed earlier that Assertions are usually only used during testing and debugging and their evaluation should be disabled in the final product. As it turns out, with Delphi you can do exactly that. There is a compiler switch that allows you to turn the generation of `Assert` code on or off: `$ASSERTIONS ON/OFF` (long form) or `$C+/-` (short form). Note that this switch works at the complete source file level. Therefore, it is not possible to turn assertions on or off for only a *section* of code in a file. It is all or nothing!

Alternatively you can enable or disable the creation of assertion code for an entire project. Simply go to the Project Options dialog. On the `Compiler` tab there is the `Debugging` section where you can turn assertions on or off conveniently.

Normally, you switch assertions on or off for an entire project, especially when you are compiling the final release build. But during testing and debugging it is sometimes helpful to be able to enable or disable assertions for individual source files. Be aware, though, that the individual source file setting overrides the project settings. So it is always a good idea to remove any `{$C+}` (or `$ASSERTIONS ON`, if you prefer) from your source file, once you don't need it anymore.

Note that the possibility of simply turning off the creation of `Assert` code for the final product is a big advantage. It allows you to leave all the assertions in the source code in place for testing and documentation purposes, while enabling you to remove them from production code with a simple rebuild of your project. So, there is no excuse for not using assertions! It won't make your final product run slower, but it can help tremendously to run it more reliably!

Assertions: Pre-Conditions, Post-Conditions

With design by contract you use three sorts of assertions: pre-conditions, post-conditions and invariants. Both pre- and post-conditions operate on the method level. A *post-condition* describes the world after the execution of a method. Or, in other words, a post-condition states *what* results you may expect from a method without specifying *how* the operation is performed. This goes along nicely with the notion of separating interface from implementation. If, for example you have a method `CalculateSquareRoot` that (not surprisingly) calculates the square root for a number, the Post-Condition can be formulated as *Value = Result * Result*, where *Value* is the input parameter.

A *pre-condition* states how we expect the world to be, before performing a certain method. So in our square root example the pre-condition could be *Value >= 0*. This pre-condition expresses the fact that it is illegal to call the `CalculateSquareRoot` method with a negative value. Whoever calls this method with an illegal argument is breaking the contract. Therefore, it is the caller's responsibility to never call `CalculateSquareRoot` with a negative number. But whose responsibility is it to assert that this does not happen?

A naive first guess might be to put the burden on the caller, since it is his responsibility to adhere to the defined contract. But if every caller has to assert the pre-condition this leads to a duplication of code which is clearly not desirable. Therefore, it is best to assert the pre-condition in the method for which it is defined. Note that this doesn't mean the calling code must not check that it doesn't break the contract, it simply means that the contract is enforced by the method for which an assertion is defined.

Phew, that sounds rather abstract! So let's look at the square root example in Delphi. Please note that this first example is rather trivial and the post-condition especially is not very helpful in this scenario, therefore I've omitted it from the example. We will see a more realistic example later in this article. Listing 1 shows the code for a simple form shown in Figure 1. All that this masterpiece of modern programming does, is to calculate the square root from the value entered into the `TEdit` `edtValue`, when the user clicks the `Calculate` button, and showing the result in the second `TEdit` `edtResult`.

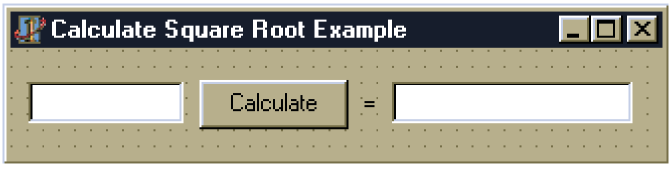
```
type
  TForm1 = class(TForm)
    edtValue: TEdit;
    btnCalc: TButton;
    edtResult: TEdit;
    lblEquals: TLabel;
    procedure btnCalcClick(Sender: TObject);
  private
    function CalcSquareRoot(AValue: Extended): Extended;
  public
  end;
var
  Form1: TForm1;

implementation

{$R *.DFM}
function TForm1.CalcSquareRoot(AValue: Extended): Extended;
begin
  Assert(AValue >= 0,'Pre-Condition failed: value must be >= 0');
  Result := Sqrt(AValue);
end;

procedure TForm1.btnCalcClick(Sender: TObject);
begin
  edtResult.Text := FloatToStr(CalcSquareRoot(StrToFloat(edtValue.Text)));
end;

end.
```

➤ *Listing 1*

On the first line of `CalcSquareRoot` we see the `Assert` for the pre-condition. On the second line the actual calculation is performed. Now let's look at the code that calls this method: In `btnCalcClick` we call `CalcSquareRoot` by passing the value of the `Text` property converted into a float. Notice that no code is included that insures the value we pass is a valid number and that it adheres to the contract defined by `CalcSquareRoot`.

Let's compile the code with assertions turned on for the project. (Note: if you have previously compiled the project successfully, you will have to *rebuild* the project, since changing compiler settings in the Project Options dialog will not force a recompile.) You can easily check whether your `Assert` code is active by simply looking at the respective line in the Delphi editor. The presence of the little diamond on the left side shows you whether code for your `Assert` statement has been created (see Figure 2). If you run the progam and enter non-negative numbers, (not surprisingly) all works fine. Since we don't do any checking with the entered text, the program

➤ *Figure 1*



is not very robust, though. Let's enter a negative number and see what happens. As soon as you click the `Calculate` button, an error message (as shown in Figure 3) pops up. This error message basically tells you that the contract for `CalcSquareRoot` has been violated by the caller. In our simple example it is easy to find the offending code to remedy the situation. In more involved programs you usually need the help of the debugger to find the offending caller by looking at the stack trace.

Before we fix our code to correctly use `CalcSquareRoot` according to the contract, let's first check what happens when we run a version of our program with assertions disabled. With positive values we still get the same results. But what happens when we enter some negative number? We still get an exception, but this time the message is very different (see Figure 4). There is no clue on what exactly went wrong and where it did happen. A user of your program would just not be very happy about this sort of error and sadly turns away to use his or her good old solar-powered minicalculator you gave them as a present!

Now let's improve our code to include the necessary checks to make sure we honour the contract. Listing 2 shows the corrected version. As you can see, there is quite some checking done. First of all, we ensure that only numbers are processed. We do this by trying to convert the string into a float in a `try..except` block. If this step was successful, we then perform the necessary check to adhere to the

```
function TForm1.CalcSquareRoot(AValue: Extended): Extended;
begin
  Assert(AValue >= 0,'Pre-Condition failed: value must be >= 0');
  Result := Sqrt(AValue);
end;
procedure TForm1.btnCalcClick(Sender: TObject);
var
  Value: Extended;
begin
  try
    Value := StrToFloat(edtValue.Text);
  except
    on EConvertError do begin
      ShowMessage('Please enter a numerical value');
      Exit;
    end;
  end;
  if Value < 0 then begin
    ShowMessage('Square Root for a negative number is not defined!');
  end else begin
    edtResult.Text := FloatToStr(CalcSquareRoot(Value));
  end;
end;
```

➤ *Above: Listing 2* ➤ *Below: Listing 3*

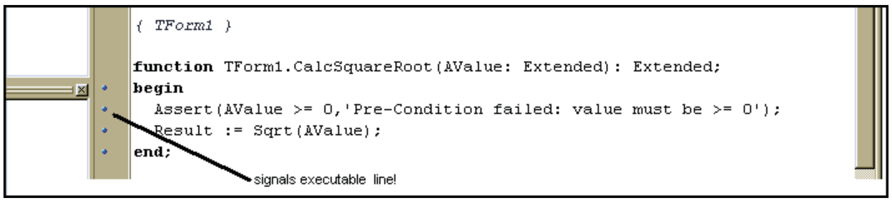
```
const
  S_PreFailed   = 'Pre-Condition failed: ';
  S_PostFailed  = 'Post-Condition failed: ';
  S_InvFailed   = 'Invariant-Condition failed: ';
```

contract defined by `CalcSquareRoot`. This guarantees that we can safely call the method. Remember what we stated earlier: it is the caller's burden to perform the necessary checks prior to calling the `CalcSquareRoot` method. The `CalcSquareRoot` method itself only asserts that the contract is honoured and throws an exception if a caller fails to comply.

As we mentioned earlier, failing to comply to the contract is considered a programming error that should be removed during testing and debugging! In other words: it should never happen that an assertion fails during execution of a released program. And that's why you can turn them off for your release build. Some people may now argue that the fact that something shouldn't happen doesn't actually mean it won't happen and that it is, therefore, better to build even your final release with assertions enabled. Personally, I don't think that this is true. Under the assumption that you consequently use assertions for critical code and that you do serious regressive unit testing (we all do that, don't we!), you should be able to reduce breaching code (ie code that breaks the contract) to zero. You can and should use assertions in your beta releases, but your final product must be reliable enough to run safely without assertions turned on!

Invariants

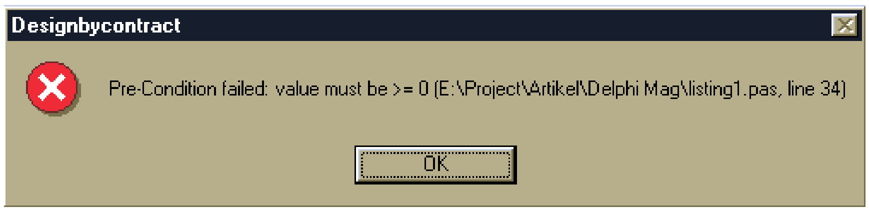
There is one sort of Assertion that we have not explained so far: the so-called invariant. Simply put, invariants are assertions that are *always* true for all objects of the class that defines them. 'Always' just means that whenever an operation on an object can be performed, the invariant must be true. In practice this means that invariants become part of the



```
{ TForm1 }

function TForm1.CalcSquareRoot(AValue: Extended): Extended;
begin
  Assert(AValue >= 0,'Pre-Condition failed: value must be >= 0');
  Result := Sqrt(AValue);
end;
```

➤ *Above: Figure 2*

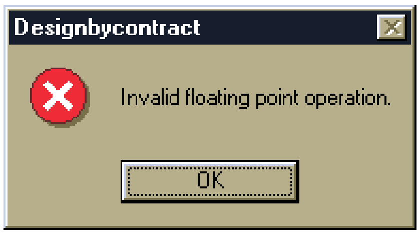
➤ *Below: Figure 3*



Designbycontract

Pre-Condition failed: value must be >= 0 (E:\Project\Artikel\Delphi Mag\listing1.pas, line 34)

➤ *Figure 4*



Designbycontract

Invalid floating point operation.

pre- and post-conditions of all public methods of the class. Unfortunately this sort of assertion cannot be dealt with automatically in Delphi. If you want to use invariants, you have to manually ensure that they are checked in each public method as part of the contract of the respective methods. You can alleviate the problem a little by providing a special method that you can call from the body of your methods. You should name the method consistently, eg `AssertInvariants`. It is also a good idea to declare the method in the `protected` section of your class definition, so subclasses may use it in their own code to ensure consistent behavior with your class. You may even want to declare the method `virtual`, in order for subclasses to be able to refine the invariants. We will look at the use of assertions in the context of inheritance and the inflicted challenges that come with it in a moment. It is worth noting that using a special `AssertInvariants` method provides better consistency and reduces the duplication of `Assert` code in the class. There is one drawback with this approach though. Even if you build your code without assertions enabled, the `AssertInvariants` method still gets called, even though the `Assert` procedure in the body of the method itself is never executed. I have found this slight overhead not to be a problem for any normal application. Things may be different for very performance critical code and you might want to adopt a different strategy under these circumstances.

Assertions In Action

Let's look at Listing 4, which presents a more complete and slightly more complex example of using assertions in various situations. The class `TSimpleAccount` describes a very basic `Account` object with very little useful

➤ *Listing 4*

```
unit listing4;
interface
uses
  classes;
type
  { forward declarations }
  TAccountLine = class;
  TSimpleAccount = class
  private
    FBalance: Currency;
    FBalanceCalculated: Boolean;
    FLines: TList;
    function CalculateBalance: Currency;
    function GetBalance: Currency;
    function GetAccountLine(Index: Integer): TAccountLine;
    function GetLineCount: Integer;
  protected
    function InternalGetAccountLine(Index: Integer):
      TAccountLine;
    procedure AssertInvariants; virtual;
  public
    constructor Create;
    destructor Destroy; override;
    function AddLine(const Text: string; Amount: Currency):
      TAccountLine;
    property Balance: Currency read GetBalance;
    property Lines[Index: Integer]: TAccountLine
      read GetAccountLine;
    property LineCount: Integer read GetLineCount;
  end;
  TAccountLine = class
  private
    FText: string;
    FAmount: Currency;
    procedure SetText(const Value: string);
  public
    constructor Create(const AText: string; AAmount:
      Currency);
    property Text: string read FText write SetText;
    property Amount: Currency read FAmount;
  end;
implementation
uses
  contnrs;
{ TSimpleAccount }
function TSimpleAccount.AddLine(const Text: string; Amount:
  Currency): TAccountLine;
  procedure AssertPre;
  begin
    AssertInvariants;
    Assert(Length(Text) > 0,S_PreFailed +
      'Length(Text) must be > 0');
    Assert(Amount<> 0,S_PreFailed + 'Amount must be <> 0');
  end;
  procedure AssertPost;
  begin
    AssertInvariants;
    Assert(Assigned(Result),
      'Result does not contain a valid AccountLine object');
  end;
begin
  AssertPre;
  Result := TAccountLine.Create(Text,Amount);
  FLines.Add(Result);
  AssertPost;
end;
procedure TSimpleAccount.AssertInvariants;
begin
  if FBalanceCalculated then begin
    Assert(FBalance = CalculateBalance,S_InvFailed +
      'Balance out of Sync');
  end;
end;
function TSimpleAccount.CalculateBalance: Currency;
var
  I: Integer;
begin
  Result := 0;
  for I := 0 to Pred(LineCount) do begin
    Result := Result + InternalGetAccountLine(I).Amount;
  end;
end;
constructor TSimpleAccount.Create;
begin
  FLines := TObjectList.Create(True);
end;
destructor TSimpleAccount.Destroy;
begin
  FLines.Free;
  inherited Destroy;
end;
function TSimpleAccount.GetAccountLine(Index: Integer):
  TAccountLine;
  procedure AssertPre;
  begin
    AssertInvariants;
    Assert((Index > -1) and (Index < LineCount),
      S_PreFailed + 'Index must be > -1 and < Count');
  end;
begin
  AssertPre;
  Result := InternalGetAccountLine(Index);
end;
function TSimpleAccount.GetBalance: Currency;
begin
  if not FBalanceCalculated then begin
    FBalance := CalculateBalance;
    FBalanceCalculated := True;
  end;
  Result := FBalance;
end;
function TSimpleAccount.GetLineCount: Integer;
begin
  Result := FLines.Count;
end;
function TSimpleAccount.InternalGetAccountLine(
  Index: Integer): TAccountLine;
begin
  Result := FLines[Index];
end;
{ TAccountLine }
constructor TAccountLine.Create(const AText: string;
  AAmount: Currency);
begin
  FText := AText;
  FAmount := AAmount;
end;
procedure TAccountLine.SetText(const Value: string);
begin
  FText := Value;
end;
end.
```

implementation. Its only purpose is to illustrate the use of Assertions in a more realistic scenario. A `TSimpleAccount` instance uses a list of `TAccountLine` objects that represent the individual lines that make up an account. Each `AccountLine` object consists of a `Text` string and an `Amount`. You can add a new `AccountLine` to an `Account` by using the `AddLine` method of `TSimpleAccount`. Let's look at the implementation of that method.

As you can see, the checks for the pre- and post-conditions are factored out in to local procedures named `Assert- Pre` and `AssertPost` respectively. Again, this introduces the slight overhead of additional method calls, but it improves the readability of the code tremendously. If I have a choice, I always favour readability and understandability of code over sheer speed of execution. Computers do get faster, on the other hand, I seem to get slower at comprehending with every day passing! So I do myself a favour and try to write readable and easily understandable code. The `AssertPre` first calls `Assert- Invariants` to make sure no invariant condition gets violated. Next, we check the input parameters and make sure, we have a non-empty variable and an amout value that is not 0.

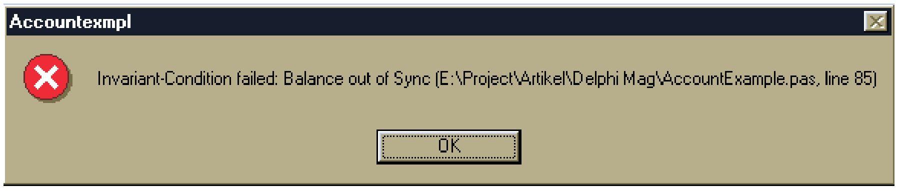
You may have noticed that I am using constant values for the message string to distinguish the different types of assertions. These constants are defined as shown in Listing 3. The actual method code is a no-brainer. We simply create a new `TAccountLine` object and add it to the internal list. Finally, we call `AssertPost`. Here we do two things. First, we again call `AssertInvariants` to make sure our call didn't change the object state in a way that violates any Invariant defined for instances of our class. Then we make sure that the `Result` we return to our caller actually contains an object reference. While this seems a silly thing to do in this simple scenario, I have included this test to protect my code from failing without notice through future 'enhancements'.

```
procedure TFrmAccountTest.AddNewLine(const Text: string; Amount: Currency);
var
  Line: TAccountLine;
begin
  Line := FAccount.AddLine(Text,Amount);
  ListBox1.Items.AddObject(Line.Text,Line);
end;
procedure TFrmAccountTest.FormCreate(Sender: TObject);
begin
  FAccount := TSimpleAccount.Create;
  {Let's add some Account Lines}
  AddNewLine('Test Entry',150);
  AddNewLine('Another Entry',-100);
  AddNewLine('Entry # 3',300);
  {now lets calculate the balance}
  edtBalance.Text := FloatToStr(FAccount.Balance);
  {Ok, lets add another Account Line}
  AddNewLine('Big badaboom',50);
end;
```

➤ *Listing 5*

Maybe someday I will come up with some clever sort of caching or reuse scheme that may reuse an exsiting TAccountLine object or create a new one if necessary. It is easy to write code that breaks the initial implementation and therefore fails to perform as intended. The simple check in AssertPost helps me spot this kind of trouble. (Note that the test is not foolproof. There is still the possibility that Result contains an illegal object reference. I leave it as an exercise to the reader, to implement a bullet-proof version of that code!) Now lets have a look at the AssertInvariants method. You will notice that the Assert is only performed when FBalanceCalculated is True. This has to do with the fact that we cache the Balance value, so we don't have to recalculate it every time we access the Balance property (see the implementation of GetBalance for details). If FBalanceCalculated is True our invariant must be checked to ensure our cached value is in synch with the actual total amount of our AccountLines. I added this invariant to make sure my class works as intended and that I do not introduce behaviour that breaks the contract.

Let's use our TSimpleAccount with a simple form that does nothing more than create a single SimpleAccount instance and then adds some AccountLines and displays them in a ListBox and the account balance in a TEdit. Listing 5 shows the relevant code. As you can see, after creating the Account object we add three AccountLines, then we calculate the balance and then we add another AccountLine. The actual code that adds the AccountLine both to the Account object and the ListBox is in the AddNewLine method. Now, if you run this code, you are in for a surprise. The adding of another AccountLine after we have calculated the balance results in an error as shown in Figure 5.


Accountexmpl

Invariant-Condition failed: Balance out of Sync (E:\Project\Artikel\Delphi Mag\AccountExample.pas, line 85)

➤ *Figure 5*

What went wrong? Obviously, our invariant failed. But where and why did this happen? A debug session reveals that the problem is in the AddLine method of TSimpleAccount. While the invariant holds true as part of the pre-condition, it fails during verification in the post-condition. So clearly our AddLine method is violating the contract. Well, we didn't do that much after all. We simply added another AccountLine. And that is the culprit. Since the GetBalanace method was called from one of our clients via our Balance property, the Balance value got calculated and cached. By adding another line this cached value becomes obsolete and needs to be recalculated, but we fail to tell our class to do so. Fortunately the violation of our invariant reveals the problem and we can now easily fix it by setting FBalanceCalculated to False after adding the new AccountLine object. Listing 6 shows the corrected method.

The rest of the code in Listing 4 shows some additional uses of assertions. You may have noticed that in some methods I use pre-conditions but no post-conditions while still in other methods there are no assertions at all. As for pre-conditions, you normally include them to ensure the Invariants or you check on the input parameters. There are rare cases where it is not worth checking the invariant. This happens when you have very few and very simple invariants and your method performs tasks that establish the desired object state, so the invariant can be met. But then again, be careful to not forget to check whether it is necessary to include the invariant check into your method's body when your invariants grow. You may find it better to always include a pre-condition that performs the necessary invariant test. If you have a method that doesn't alter the state of your object, you can usually omit the post-condition, unless you want to ensure the result does conform to specific rules, as we did in our AddLine method of TSimpleAccount.

➤ *Listing 6*

```
function TSimpleAccount.AddLine(const Text: string;
  Amount: Currency): TAccountLine;
  ...
begin
  AssertPre;
  Result := TAccountLine.Create(Text,Amount);
  FLines.Add(Result);
  // make sure Balance gets recalculated
  FBalanceCalculated := False;
  AssertPost;
end;
```

Assertions And Inheritance

So far, we have only used assertions in non-derived classes. So let's briefly touch on the subject of inheritance and the use of pre-conditions, post-conditions and invariants when subclasses are involved.

From a theoretical point of view, assertions play a very important role when it comes to subclassing. Due to the nature of polymorphism, it is possible for a subclass to override a method in way that is inconsistent with the baseclass's protocol. The strict use of assertions prevents this. Invariants and post-condition are valid for all subclasses. A subclass may tighten invariants or post-conditions, but it may not loosen them. On the other hand a subclass may not strengthen any pre-condition. As counter-intuitive as this may sound at first, it is very important to allow dynamic binding and the proper functioning of polymorphism. If a subclass does strengthen a pre-condition, this could lead to the failure of an operation on the subclass even though it would work for the superclass. This would break a fundamental principle of object orientation that states that you may safely use a more special class everywhere a more general class (ie a superclass) is permitted.

So much for the theory. Unfortunately it is very cumbersome to use assertions together with inheritance in Delphi, as in any other language that doesn't support assertions as part of the class declaration. In order to being able to successfully use assertions with subclassing, you need to establish strict conventions that govern the way assertions are implemented and used. And as with any convention, the whole scheme breaks down if only once you fail to comply.

Final Remarks On Using Assertions

I hope I was able to show you the benefits of using assertions in your code to establish simple contracts that you can use to thoroughly test your programs and add expressive power to your code. When people start working with assertions they often tend to overuse them at first. It doesn't make sense to check for every possible combination. A good assertion makes your code more understandable. Overusing assertions usually leads to duplicated code and ultimately is more obfuscating than helpful. It may even hinder future modifications! A simple test to see whether an assertion makes sense or not is this: ask yourself whether the code works, if the assertion fails. If it does, remove the assertion, because you don't need it. Also beware of duplicate code in assertions. It is just as bad in an assertion as it is in any other code.

Assertions And Subclassing: A Possible Solution

One problem when using assertions with subclassing is the possible duplication of `Assert` code. Since invariants and post-conditions may only be strengthened in subclasses, you must ensure that any derived class will honor all the assertions of all base classes. On the other hand, pre-conditions may be loosened by your subclass and must be able to prevent the evaluation of any pre-condition of your superclasses.

Clearly if you include the assertions in the method body you are in trouble for both pre-conditions and post-condtitions, because the verification and the execution of an inherited method form an indivisible unit. You can't execute one without the other. As we have seen with invariants, we can factor out the `Assert` code into its own method, so we can call it from within our class wherever need be and even override it to extend it in a subclass. You can use the same approach for pre- and post-conditions. As with the `AssertInvariants` method we used in our example your pre- and post-condition checks should reside in protected virtual methods. You should give all your `Assert` methods an expressive name and it might be a good idea to use some sort of prefix like `Assert` to clearly distinguish these methods from 'normal' code. You must also be careful not to inadvertently tighten a pre-condition or loosening a post-condition or invariant. Note also that this approach uses a lot of conventions. If you abandon it only once, you have given yourself away to the dark side of the force...

Sascha Frick is an independent coach and trainer for object orientation. He hears himself repeating the sentence 'All data should be declared private in a class' all too often during code reviews. In his all too sparse spare time he loves to watch good movies and eat Italian food that he loves cooking himself. He can be reached via email at saschaf@empros.ch