

The Iterator Pattern

Graeme Geldenhuys

2008-11-20

As the name suggests, an Iterator is something that allows you to sequentially access all the items in a collection. This patterns article will cover how to implement them for Delphi and Free Pascal collection classes and highlight some differences compared to other languages like Java.

What is an Iterator?

Implementing a design pattern normally takes a reasonable amount of time and effort. It is not something that takes seconds to implement. The nice thing about the Iterator, is that it is one of the easier patterns to implement - almost as quick and easy as the Singleton. So in this article I want to introduce to you a pattern that everybody could use straight away. It hardly uses any extra code, it normally reduces your lines of code. That also means a few less lines to debug in your application!

As described in the Design Patterns book by the GoFⁱ, an Iterator is something that allows you to sequentially access all items in a collection. In its simplest form, it allows you to test if there are more items in the collection and then access the next item. So lets jump right into the implementation. We could define a very simple Iterator interface as follows:

```
ITBIterator = Interface(IInterface)
['{B2A449B4-5D0A-4F14-AC11-CA055EDA3ED7}']
  function HasNext: Boolean;
  function Next: String;
end;
```

Here we have two simple functions. `HasNext()` that returns `True` if there are more items in the collection or `False` otherwise. The function `Next()` returns the next item in the collection. At the moment this Iterator only returns a `String` type in the `Next()` method, but we will deal with that later when we implement a few more Iterator interfaces.

So how would code look like that uses such an Iterator. Before I show you that, let me show you the traditional way that developers traverse through a collection, so you we can compare the difference.

```
for i := 0 to MyStrings.Count-1 do
  ShowMessage(MyStrings.Items[i]);
```

That is fairly standard code traversing through a `TStrings` collection and I'm sure we have all seen code like that before. We directly access the collection and it's storage structure. So lets get to our Iterator example and see how it compares. Ignoring for the moment of how we get an instance of this Iterator, which we will deal with later, the usage will look as follows.

```
while MyIterator.HasNext do
  ShowMessage(MyIterator.Next);
```

By now developers would normally say: "What is the big deal?". My answer is simple. Yes you can continue writing iteration code like we did in the first example, but you will quickly notice some problems. You would have to slightly modify that code if you decided to traverse over a `TTreeView` or a `TObjectList` or `TCollections` etc... Things get even more complex when you decide to traverse over all cell items in a `TStringGrid` or list items in a `TListView`. The point is that even though many different collection classes give you a way to traverse over their items, the actual iteration code is slightly different for each collection class. The iteration code never stays consistent because the internal storage structure of each collection class is slightly different to the next. So your iteration code is tightly tied to the collection class you are using.

This is where good object oriented programming and design patterns come into play. The golden rule is to always design your software so it is flexible to future changes. So lets list a few of the benefits of using Iterators:

- They give the developer a consistent interface to traverse over a collection.
- The iteration code stays independent of the collection type you are using.
- It is very easy to have multiple Iterators traversing over the same collection.
- Different Iterator implementations allow you to be more flexible about what items in the collection you are accessing, without affecting the iteration code. For example: Your Iterator may only give you access to a subset / filtered view of the items in the collection.

Iterator Interfaces and Implementation

As I mentioned earlier, the Iterator interface we defined in the beginning of this article only returns the `String` type. This is fine for a `TStrings` collection, but what about something like the `TList` collection that contains `TObject` items. Using the Java language this is easily handled, because in Java almost everything is an `Object`. So Java Iterators can simply return a `Object` for any collection type and it would be handled correctly. While using Object Pascal we are not quite so lucky. The solution is simple though. To overcome this issue, we have to implement different Iterator interfaces for

all the collection types we want to use. These interfaces will basically differ only by the return type of the `Next()` method. The important thing to remember is that the usage of the iterators will still stay the same. Here is a partial list of some Iterator interfaces. The full source code, including other iterator interfaces, are available on the Toolbox Magazine cover DVD.

```
{ traversing TString collections }
ITBStringIterator = interface(IInterface)
[ '{B2A449B4-5D0A-4F14-AC11-CA055EDA3ED7}' ]
    function HasNext: Boolean;
    function Next: string;
end

{ traversing TList collections }
ITBIterator = interface(IInterface)
[ '{9C2BC10D-54C8-4B59-88B5-A564921CF0E3}' ]
    function HasNext: Boolean;
    function Next: TObject;
end;

{ traversing TString collections with Objects associated
to each item }
ITBStringAndObjectIterator = interface(ITBStringIterator)
[ '{287373DC-A90D-400E-BAEE-C85474C317A8}' ]
    function HasNextObject: Boolean;
    function NextObject: TObject;
end;

{ traversing InterfaceList collections }
ITBInterfaceIterator = interface(IInterface)
[ '{9B599C5B-4BBB-43F6-AF8E-09FEE9AE0E20}' ]
    function HasNext: Boolean;
    function Next: IInterface;
end;
```

The Iterator interfaces I am introducing here is designed as per the Java-style Iterators. This is important to mention, so as to help you understand the implementation. Java-style iterators do not have a `Current()` method. The cursor or index position of the iterator is between items in a collection, and not pointing directly at items in the collection. See Figure 1. One reason for this design is so that it requires less error checking in the iteration code. It also makes more sense when your iterator interfaces contains methods like `Add()`, `Remove()`, `SetItem()` etc. - modifying your collection, but always leaving your iterator in a stable state afterwards. The implementation I am showing here does not contain these modifier methods - this is purely to keep things simpler for this article. The entire Java-style API and implementation is based on the idea that the iterator remembers which item it skipped last. This is very important to remember! I believe it is a nice feature of Java's iterators and I am sure you will grow to love it. As a side note: Microsoft's .NET iterators, which are called Enumerators for some reason, use the opposite approach. In .NET the iterator cursor/index points directly at an item in the collection.

So to recap. The `Next()` function returns the next item in the collection and advances the iterator. The first call to `Next()` advances the iterator to the position between the first and second item, and returns the first item; the second call to `Next()` advances the iterator to the position between the second and third item, and returns the second item;

and so on.

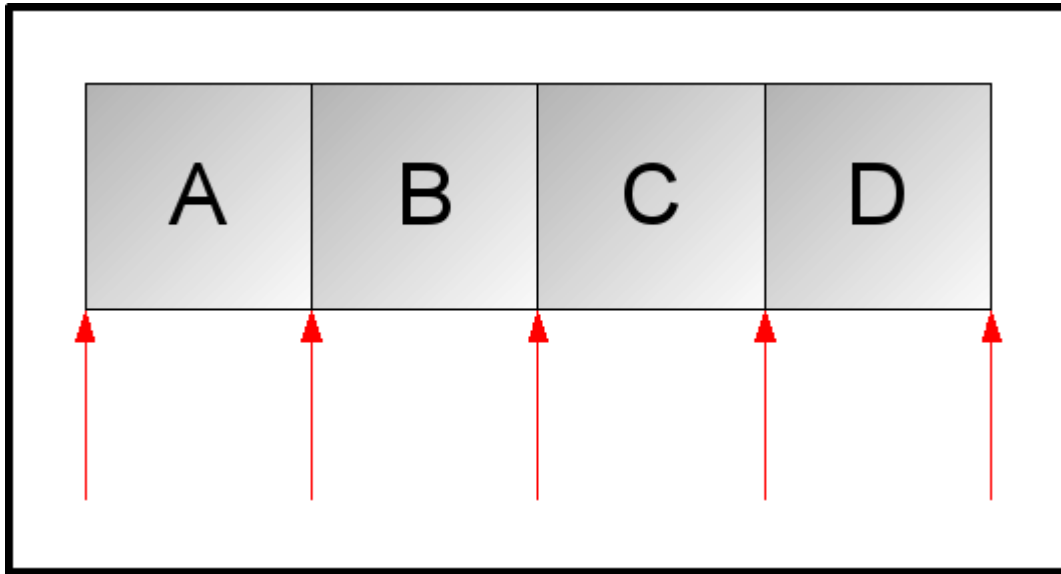


Figure 1: Java-style iterators point between items rather than directly at items.

So now that we know all about how iterators work, it is time we look at an actual implementation. We are going to implement the *Strings* iterator for a `TStrings` class. Because `TStrings` can also manage *Objects*, I thought we could implement both interfaces in a single class. Below is the interface for our `TTBStringsIterator` class.

```
TTBStringsIterator = class(TInterfacedObject,  
                           ITBStringIterator,  
                           ITBStringAndObjectIterator)  
private  
  FStrings: TStrings;  
  FCursor: Integer;  
  { ITBStringIterator and ITBStringAndObjectIterator }  
  function HasNext: Boolean;  
  function Next: string;  
  { ITBStringAndObjectIterator }  
  function HasNextObject: Boolean;  
  function NextObject: TObject;  
public  
  constructor CreateCustom(  
    const ASource: TStrings); virtual;  
end;
```

As you can see in the class declaration, we inherit from the `TInterfacedObject` class. This will handle our reference counting and automatically take care of freeing our Iterator when we are done with it. We also specified both Interfaces we want to implement. The `ITBStringIterator` and the `ITBStringAndObjectIterator`.

We created a custom constructor called `CreateCustom`, and pass in as a parameter the `TStrings` instance or collection we want to traverse. We store a reference to the `TStrings` instance as a private field variable called `FStrings`. We also have another private field variable called `FCursor`, which will keep track of where we are in our collection.

Both our Interfaces have two methods in common, the `HasNext()` and `Next()` functions. Because they are the same, we only need to declare them once in our class. For our class declaration to be complete, we still need to declare the two remaining functions of the `ITBStringAndObjectIterator` interface - `HasNextObject()` and `NextObject()`.

I would like to point out something in the class declaration. As you might have noticed, all the required Interface functions are declared *Private*. The reason for that is because we would never access those methods via an Object instance. We will only every access them via an Interface reference. So to enforce that rule and prevent temptation, I declared them *Private*.

Now for the implementation section of our Iterator class. As you will see from the following code, implementing an Iterator is very easy.

```
function TTBStringsIterator.HasNext: Boolean;
begin
  Result := False;
  if Assigned(FStrings) then
    if FCursor < FStrings.Count - 1 then
      Result := True;
end;

function TTBStringsIterator.Next: String;
begin
  Result := '';
  if HasNext then
    begin
      Inc(FCursor, 1);
      Result := FStrings.Strings[FCursor];
    end;
end;

function TTBStringsIterator.HasNextObject: Boolean;
begin
  Result := False;
  if Assigned(FStrings) then
    if FCursor < FStrings.Count - 1 then
      Result := FStrings.Objects[FCursor] <> nil;
end;

function TTBStringsIterator.NextObject: TObject;
begin
  Result := nil;
  if HasNextObject then
    Result := FStrings.Objects[FCursor];
end;

constructor TTBStringsIterator.CreateCustom(
  const ASource: TStrings);
begin
  inherited Create;
  FStrings := ASource;
  FCursor := -1;
end;
```

The constructor simply assigns our `TStrings` parameter to the `FStrings` field variable, and we set our cursor/index field variable `FCursor` to -1 to indicate that we are at

at the beginning of our collection, in front of the first item. Remember that the `Items` and `Objects` properties of a `TStrings` class is zero based, meaning the first item is at position 0. Also remember to look at Figure 1, to follow how the cursor jumps between items.

The `HasNext()` method returns a boolean value and defaults to `False`. If the current cursor position is smaller than the number of items in the collection, we return `True` to indicate that we have remaining items to traverse.

The `Next()` method returns an item from the collection. Because we are implementing an iterator for `TStrings`, the item we return is of type `String`. We default to an empty string as a small safety measure. We then call `HasNext()` to test if we have anything remaining to traverse. If the answer is `True`, we increment the cursor by one and then return the item we just jumped over.

The `HasNextObject()` implementation is very similar to the `HasNext()` method, but this time it does one extra check. It first checks that the current cursor position is smaller than the number of items in the collection. It then checks to see if there is an actual object stored in the `Objects` property. If all is well, it returns `True`.

The `NextObject()` implementation is also very similar to the `Next()` method, but it returns a `Object` instead of a `String`. It calls `HasNextObject()` to test if we have anything remaining to return. If the answer is `True`, it returns the next `Object` stored in the collection.

How to get an Iterator instance?

The last big mystery is how do we actually get hold of a iterator instance? We could create an instance as shown below:

```
var
  MyStringIterator: ITBStringIterator;
begin
  MyStringIterator :=
    TTBStringsIterator.Create(MyStringList);
  while MyStringIterator.HasNext do
    ...
end;
```

This would work, but it would totally defeat the point I am trying to make with Iterators. The following problems should be clearly visible:

- We are not saving on the amount of code lines required.
- By creating an explicit `TTBStringsIterator` instance, we also know too much about the `Iterator` instance and required types.
- We also have to include the unit in the `uses` clause that contains the `TTBStringsIterator` class.
- We know so much about the collection, we could just as well have traversed our collection using a `for` statement and `Integer` variable.

Now the nice thing in Java, is that all collection classes have built-in support for iterat-

ors. We can simply call a method and it returns the correct Iterator instance. Unfortunately we are not so lucky with Free Pascal or Delphi. There are no built-in support for Iterators in the collection classes, so we have to come up with another solution.

What we want to achieve is to obtain the correct Iterator implementation for a specific collection while staying ignorant of the types required. The solution is once again a simple one. We turn our attention to another design pattern to help us. What we just described is pretty much the definition of the *Factory Pattern* which I covered a few issues agoⁱⁱ. There are a few ways we can implement the solution using any one of the various factory patterns. I am going to use the *Factory Method* to implement an Iterator Factory class. Although my implementation is very simple, it's perfectly adequate for what we want. Our Iterator Factory will contain various methods to represent the different types of collection classes. We simply call the method of the class we are working with and it will return a Iterator Interface for that collection. Below is the interface section of our Iterator Factory:

```
TTBIteratorFactory = class(TObject)
  function Iterator(
    const ASource: TObject): ITBIterator;
  function StringIterator(
    const ASource: TObject): ITBStringIterator;
  function StringAndObjectIterator(
    const ASource: TObject): ITBStringAndObjectIterator;
  function InterfaceIterator(
    const ASource: TObject): ITBInterfaceIterator;
  function FilteredStringIterator(
    const ASource: TObject;
    const AFilter: string): ITBFilteredStringIterator;
end;
```

Please note that this is by no means a complete implementation. I kept it small to simplify it for this article. As we extend our code and develop new iterators we can add new methods to our Iterator Factory. That way the Iterator Factory can instantiate our new iterators. Below is the implementation for the `TTBIteratorFactory.Iterator` method which handles iterators for the `TList` and `TCollection` classes.

```
function TTBIteratorFactory.Iterator(
  const ASource: TObject): ITBIterator;
begin
  if ASource is TList then
    Result := TTBListIterator.CreateCustom(
      TList(ASource))
  else if ASource is TCollection then
    Result := TTBCollectionIterator.CreateCustom(
      TCollection(ASource))
  // Here we can extend it for TreeView support.
  // else if ASource is TTreeNode then
  //   Result := TTBTreeNodeIterator.CreateCustom(
  //     TTreeNode(ASource))
  else
    raise ENoIteratorImpl.CreateFmt(cNoIteratorImpl,
      [ASource.ClassName]);
end;
```

As you can see, we pass in the collection we are working with as a parameter to the

factory methods. We specified the `ASource` parameters as the generic base type `TObject`, so that any collection class could be passed in. Then we simply test to see if the `ASource` parameter is of any of the supported types. If we find a match, we create the correct iterator class instance and return it. If we did not find a match, that means this factory method doesn't support that collection type. We then simply raise an exception with a message notifying the user of the problem. To resolve the problem, the factory method needs to be extended to support that collection type, or a new factory method needs to be created for that collection.

In the code, you can also see a few commented lines. This is a possible place where this factory method can be extended to support the `TTreeView` nodes for instance.

We can now create and use an iterator instance using code as follows:

```
var
  itr: ITBStringIterator;
begin
  ...
  itr := gIteratorFactory.StringIterator(sl);
  while itr.HasNext do
    writeln(itr.Next);
  ...
end;
```

Now you might ask: "Where do we get the `gIteratorFactory` instance from?"

Well, for that we can turn to yet another design pattern. We don't want to construct an Iterator Factory instance every time we need an Iterator. We also don't need multiple instances of the Iterator Factory. To solve both these problems we use the *Singleton* design pattern to handle the construction of the Iterator Factory when we need itⁱⁱⁱ. The Singleton design pattern will also take care of only ever creating one instance of the Iterator Factory. I have used a very simply but very effective Singleton implementation as shown below.


```

interface
...

  { Global iterator factory singleton }
  function gIteratorFactory: TTBIteratorFactory;

implementation
var
  uIteratorFactory: TTBIteratorFactory;

{ The lazy-man's singleton implementation. }
function gIteratorFactory: TTBIteratorFactory;
begin
  if not Assigned(uIteratorFactory) then
    uIteratorFactory := TTBIteratorFactory.Create;
  Result := uIteratorFactory;
end;

...

initialization
  uIteratorFactory := nil;

finalization
  uIteratorFactory.Free;

end.

```

We have a global or system wide function called `gIteratorFactory` which will return our Iterator Factory instance. We then have a variable called `uIteratorFactory` which is only visible in that unit. This variable will hold the instance of our Iterator Factory. When the user calls `gIteratorFactory` it checks to see if `uIteratorFactory` already has an instance assigned. If it hasn't, we create one and then return the instance. The finalization section will free our Iterator Factory instance when the application terminates.

That completes our Iterator pattern implementation!

So what's next?

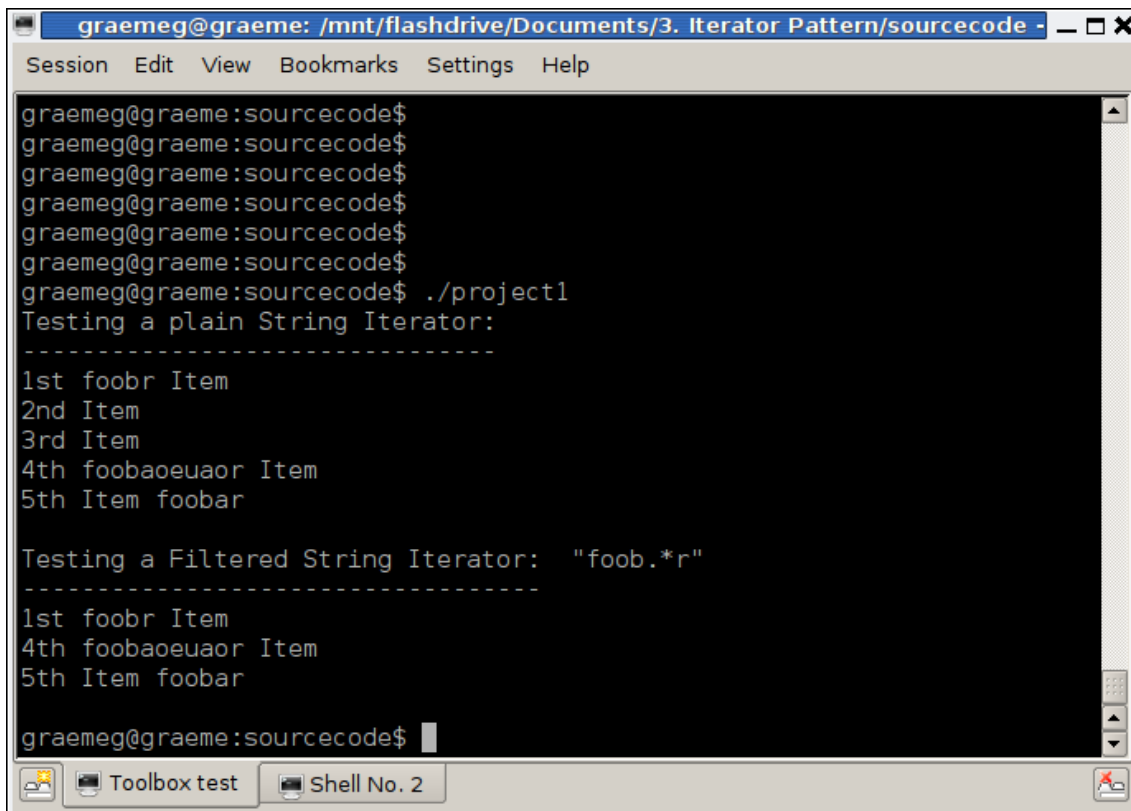
All the code I used so far in this article is included on the cover DVD. I also included a few more iterator implementations for other collection types, including a simple console application showing the code being used.

The iterators I have talked about and shown are quite basic. They always traverse the whole collection from the beginning to the end. Remember that you are not limited to only standard iterators that traverse a whole collection. Nothing stops you from extending the iterators to include all kinds of handy features. One such feature could be filtering! Say you wanted to traverse a `TStringList`, but only return a subset of items. This could be very handy in some cases. As an example, I created an `ITBFilteredStringIterator` interface in the accompanied article code. The `TTBFilteredStringsIterator` class included in the source code uses the *regular expressions*^{iv} unit called `regex.pp`, which is included with Free Pascal. This allows you to filter the returned items in a `StringList` collection using similar code to the fol-

lowing:

```
fitr := gIteratorFactory.FilteredStringIterator(  
    MyStringsCollection, 'foob.*r');  
while fitr.HasNext do  
    DoSomethingWithItem(fitr.Next);
```

Figure 2 shows the sample output of the demo program. We created a `StringList` with five string items in it. Then we created a standard string iterator and traversed the whole `StringList`. As items were returned, so we wrote them to the console window as output. We then created another iterator, but this time a filtered one, using the regular expression `'foob.*r'`. Again we traversed the same `StringList` and output the returned items to the console window. As you can see from the screenshot, the second iterator only had a subset of items returned from the `StringList` collection.



```
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$  
graemeg@graeme:sourcecode$ ./project1  
Testing a plain String Iterator:  
-----  
1st foobr Item  
2nd Item  
3rd Item  
4th foobaoueuar Item  
5th Item foobar  
  
Testing a Filtered String Iterator: "foob.*r"  
-----  
1st foobr Item  
4th foobaoueuar Item  
5th Item foobar  
  
graemeg@graeme:sourcecode$
```

Figure 2: Sample output of a String Iterator and Filter String Iterator against the same `StringList`.

Filtering is just one of many features we can add to Iterators. We can also extend the Iterator API so we can traverse a collection in a forwards direction and a backwards direction. Table 1 shows an extended Iterator API. This can easily be added to the sample iterators created in this article and add a lot more flexibility to your Iterators.

Hopefully I have shown you how handy the *Iterator Pattern* can be and how easy it is to implement. Not all design patterns are difficult to implement or complex in design. So now you can use the accompanied source code as a starting block and build you own custom iterators from there. Once you start using Iterators in your applications, you and your co-workers will never have to think twice about how to traverse a collection. Your code will be more consistent, easier to maintain and you should never need to access a

collection directly again.

| Function | Description |
|-----------------|--|
| Add(item) | Inserts a specified item into the collection. (optional modifier operation) |
| HasNext() | Returns true if the collection has more items when traversing the collection in the forward direction. |
| HasPrevious() | Returns true if the collection has more items when traversing the collection in the reverse direction. |
| Next() | Returns the next item in the collection. |
| Previous() | Returns the previous item in the collection. |
| Reset() | Jum to the beginning of the collection, setting the cursor/index before the first item in the collection. The iterator is now in the same state as if you just created it. |
| ToBack() | Jump to the end off the collection, setting the cursor/index after the last item in the collection. This needs to be called before you want to traverse the collection in the reverse order. |
| PeekNext() | Returns the next item without moving the iterator's cursor/index. |
| PeekPrevious() | Returns the previous item without moving the iterator's cursor/index. |
| Remove() | Removes from the collection the last item that was returned by the Next() or Previous() calls. (optional modifier operation) |
| SetItem(item) | Replaces the last item returned by the Next() or Previous() calls with the specified item. (optional modifier operation) |

Table 1: A more extensive Iterator API.

- i Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2, Seite 257.
- ii Toolbox 5'2008: Patterns in Pascal: bessere Programmpflege mit Simple Factories
- iii Singleton Pattern: http://en.wikipedia.org/wiki/Singleton_pattern
- iv Regular Expressions: http://en.wikipedia.org/wiki/Regular_expressions