I COME HERE NOT TO BURY DELPHI, BUT TO PRAISE IT

I Come Here Not to Bury Delphi, But to Praise It

By Patrick Martin

Overload, 27(153):10-14, October 2019

What helps a programming language gain traction? Patrick Martin remembers why he used to use Delphi.

But first what is Delphi?

It is a riddle, wrapped in a mystery, inside an enigma.

And why write about it?

It's not a controversial statement that Delphi is not what it once was in its heyday [OracleAtDelphi05]. Nevertheless, I think it's worth reviewing what might have formed part of the secret sauce that was part of its success back then. The current version now supports building 64-bit projects across the 3 main desktop platforms, to select one area of evolution of the product.

Furthermore the original aspects that were key for me are not only still there, but better than before.

Non goals and own goals

There are many things I will not be discussing in this article. For example, there is always Much To Discuss when it comes to choice of programming language – I am told – but I will be attempting to steer clear of controversy of that nature.

Comparing laundry lists of features between languages in some kind of checklist knockout tournament is certainly not the aim here.

Instead, I want to recall – or if you will, eulogise – a rich seam of features of the tool that for me made Delphi the game changer that it was then...

Back when I used it full time, these techniques were what made it so productive and what's more fun to work with, and I humbly submit that there are few tools that come close to touching it even now.

Remember the 90s were wild, man
This was the time of the rise of the Component based model – people could pay (remember paying for software?) a nugatory amount for a component that would emulate, say some portion of the Excel spreadsheet editor and embed it into their software [Wikipedia-4]. In Delphi I could study the built-in in components, or follow the tutorials and write my own if needs be, or figure out how to achieve my aims using the existing functionality.

First, a quick review

Delphi is a commercial product for developing software [Embarcadero-1], [Wikipedia-1], with a proprietary IDE and version of Object Pascal [Wikipedia-2] that integrates tightly with the solution [Embarcadero-3]. There is even a free version you can download from [Embarcadero-2], and if you can puzzle your way past the registration djinns, you can have it installed and up and running in a few minutes.

Table 1 shows a heavily abridged table of releases with my comments for some key milestones.

Year	Release	Supports development for	Notable enhancement
1993	Delphi 1.0	Win16	From out of nowhere, handling a GPF [1]
1996	Borland Delphi 2	Win32	First win32 compiler
1998	Inprise Delphi 4	Win32	Last version allowing 16-bit development
2003	Borland Delphi 8	Win32	.NET
2005	Borland Delphi 2005	Win32	
2011	Embarcadero Delphi XE2	Win32, Win64	First version producing 64-bit binaries
2012	Embarcadero Delphi XE3	Win32, Win64	Last version with .NET support
2018	Embarcadero Delphi 10.3 Rio	Win32, Win64	Current day

Table 1

For prior art: there is even an ACCU article [Fagg98] article, and if you want a much funnier, arguably less slightly less technical summary of the early days, try this on for size [Stob12].

Here are seven bullet points I've chosen to give a flavour of the system:

- **Fast**
  Compile times were always in the vanguard, currently there are quotes of many thousands of lines per second.

  The time from a standing start of just source to a fully linked native executable that was ready to go was also very, very short. In the days of 'spinning rust' drives, this is a feature that really mattered – there is a nice little review here [Hague09].

- **Strongly typed** (mainly [2])
  `for` loops could only be Ordinal types. I got over the shock of not being able to increment a `double` type very quickly and never looked back.

  You could (and should) declare enums and sub-range types. It would then be a compilation *and runtime* error to assign incorrect values to these types, if you chose to enable the strict compilation mode, which you almost always should.

```
type
// everyone likes cards
  Suit = (Club, Diamond, Heart, Spade);
// small things that it's just embarrassing
// to get wrong
  SmallNumber = -128..127;
  SomeCaps = 'A'..'Z';
  Month = 0..11;
```

- **Run time type information at all times**
  One could always identify the type of an object at runtime and it was built into the language – with a little more effort one could browse all the types in the programs' type system. This will come in useful for building up complex objects, as we will see.

- **Straightforward dependency management**
  The language has files called units – basically modules, which supported interface and implementation sections for exported symbols and internal only code.

  Circular dependencies were a compile time error, it's worth taking a second to let that sink in.

  This required the developer to structure their program as a Directed Acyclic Graph, which strongly encouraged a way of organising one's code in such a way that one really only had to inspect the interface section of a new dependency unit, and then choose whether to make it a dependency in the implementation or not.

  Rinse and repeat for the rest of the program.

  In addition, the order of initialisation and finalisation of the units was straightforward and robust (even if spelled incorrectly – see later ☺).

- **Extensible RTL and Visual class libraries exploiting the strengths of the language**
  Object Pascal supports class properties (read/write, write-only, read-only) as a first class feature. Objects on the stack are simply not allowed – I suspect eliminating this capability freed Delphi from having to deal with a large class of issues related to dynamic runtime use of code. Coupled with the ability to use the RTTI, these work together to support configurability of classes from properties.

- **Source-based component model**
  It's worth bearing in mind that the era when Delphi was conceived was the era of the rise of the component-based software model [Wikipedia-4]. For example, people could pay (remember paying for software?) a nugatory amount for a component that would emulate Excel and embed it into their software.

  In the very first release of Delphi was a thorough guide to writing components, proselytising for the style of authoring components. This was really high quality work and – true story – we kept a copy of the Delphi 1 guide chapters that didn't survive to later releases around to consult, as it remained relevant.

- **You could go deep if you wanted**
  For those minded to use them, the features of a language for the 'hard core programmer' were also there:
  - full access to the API of binaries of other languages at link time
  - a range of selectable calling conventions (see [Wikipedia-3] for details of these)
  - capability to hand-craft dynamic loading of code modules
  - all the usual crazy casting stuff some programmers like to do (rarely needed in Delphi)
  - inline assembly.

Contention: Delphi was inherently very dynamic for its time

This is my central thesis.

In 1999, I could fire up the IDE, load the source of a visual form connected to, say a database and see and navigate records fetched from the database *live in the designer* . The development environment was quick and effective to work in, and I had access to the source for debugging and simply improving my mind by reading the code.

That feature just on its own, helped to teach me a lot about the engineering of a coherent architecture. And for those prepared to take the time to investigate, it had a cornucopia of treasures to uncover beyond the super user friendly surface.

Example: how the ability to read and *debug into* the source make a difference

Figure 1 is a simple UI app I created in a few clicks with no code. Setting one breakpoint and stepping in using one key combination I see where the application launches the main UI form and then enters the main Windows interactive message loop.

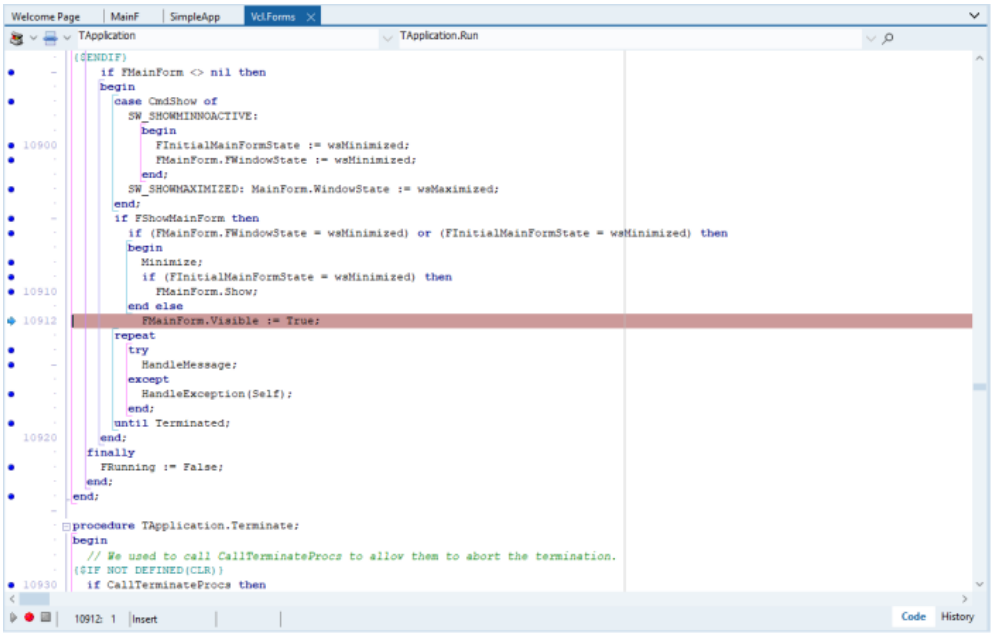


Figure 1

Delphi's streaming system and form design

Now the real killer app for the app development was the fully synchronised visual designer

Let's have a look at some actual code to plug together some hypothetical framework objects. Note, this process relies upon the concepts of

- properties
- the Delphi closure type (reference to method call on object instance)
- and RTTI to allow the RTL to work all the magic of wiring up the properties
- there is also a hint of a framework which defines the ownership from the line `TfrmClock.Create(Application);`

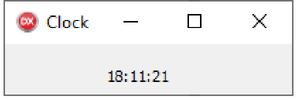
```
begin
  frmClock := TfrmClock.Create(Application);
  lblTime := TLabel.Create(frmClock)
  lblTime.Caption := '...'
  tmrTick := TTimer.Create(frmClock);
  tmrTick.onTimer = tmrTickTimer;
  frmClock.AddChild(tmrTick);
  // ...
end;
```

And let's have a look at some hypothetical DSL code to describe the moral equivalent of that code

```
object frmClock: TfrmClock
  Caption = 'Clock'
  object lblTime: TLabel
    Caption = '...'
  end
  object tmrTick: TTimer
    OnTimer = tmrTickTimer
  end
end
```


Full disclosure. Of course it's actual real DSL (edited slightly for space)! *The iDE would generate all of that for you.*

Now, with the mere addition of the following line to a class method called `tmrTickTimer` … we have a clock app!




```
  lblTime.Caption := TimeToStr(Now);
```

So, that’s assembling visual components visually sorted then.

Registry singletons done right (TM)

Listings 1–4 are an example illustrating how deterministic initialisation of modules would allow for very simple, yet very robust registration concepts, giving the following output:

```
Registry Adding: TSomeProcessor
Registry Adding: TAnotherProcessor
Program starting
Registration complete
Program exiting
Registry Removing: TAnotherProcessor
Registry Removing: TSomeProcessor
```

Note the initialisation follows the lexical ordering in the program unit *in this case* (but see later), and also that the de-init occurs perfectly in the inverse order.

```
program registration;
{$APPTYPE CONSOLE}
uses
  SysUtils,
  AnotherProcessor in 'depends\AnotherProcessor.pas',
  SomeProcessor in 'depends\SomeProcessor.pas',
  SomeRegistry in 'depends\SomeRegistry.pas';
begin
  try
    WriteLn('Program starting');
    WriteLn('Registration complete');
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
  Writeln('Program exiting');
end.
```

Listing 1

```
unit SomeRegistry;
interface
type
TSomeRegistry = class
public
  procedure RegisterClass(AClass: TClass);
  procedure DeregisterClass(AClass: TClass);
end;
function GetSomeRegistry: TSomeRegistry;
implementation
var
  mSomeRegistry : TSomeRegistry = nil;
// details omitted
initialization
  mSomeRegistry := TSomeRegistry.Create();
finalization
  mSomeRegistry.Free;
end.
```

Listing 2

```
unit SomeProcessor;
type TSomeProcessor = class
// details omitted
end;
initialization
   GetSomeRegistry.RegisterClass(TSomeProcessor);
   // register our class
```

Listing 3

```
unit AnotherProcessor
type TAnotherProcessor = class
// details omitted
end;
implementation
initialization
  GetSomeRegistry.RegisterClass
  (TAnotherProcessor);
  // register our class
```

Listing 4

Add this `uses` directive into `SomeProcessor` , adding a source level dependency to `AnotherProcessor` from the `SomeProcessor` *implementation* (Listing 5).

```
unit SomeProcessor
...
interface
uses
  AnotherProcessor, // <- indicate we need this
  SomeRegistry;
...
```

Listing 5

The output is:

```
Registry Adding: TAnotherProcessor
Registry Adding: TSomeProcessor
Program starting
Registration complete
Program exiting
Registry Removing: TSomeProcessor
Registry Removing: TAnotherProcessor
```

Note this happens when updating the implementation of single unit, not the program code, which remains blissfully agnostic of the changes. In this way we have been able to clearly and unambiguously capture a program dependency that was previously not knowable from inspecting the source.

There are corollaries

- RAII *per se* is out, although your classes must of course still behave sensibly
  - this may have been noticed – properties need to have workable defaults (or default behaviour that makes sense)
  - once you have committed to a property based system for configuring objects, what constructors could you possibly write? Instead of solving that hard problem, the component is plugged into the framework
- no automatic destruction of class instances
  - destruction is explicit in Delphi's Object Pascal and – key point – with the Delphi component framework the object deletion would be handled for you correctly
  - coupled with the streaming system's ability to 'automagically' find and instantiate the right classes when streaming in a definition, you spend a lot less time worrying about 'ownership' – because (a) it's done for you, and (b) if you wanted to do it yourself, you may well get it wrong or find yourself fighting the existing framework every step of the way
- classes are exclusively references types – so no objects on the stack, à la c++
  - this may feel like an intolerable constraint, but it happens to fits in well with the concept of dynamic extensibility ->

  The code to construct an object can be supplied, even updated on the fly, because *all objects are the same size* – they are the size of a pointer!

  - delivering essentially, a 'plugin' system that is capable of plugging in classes and their type metadata *on the fly*
  - by the way: the IDE does this every time you rebuild a component package you are working on in the IDE
- Exceptions can only throw objects, and also, given the singly rooted hierarchy we can always walk our way to the actual instance type if you have imported its interface
  - conveniently an extensible object designer system can simply roll back the stack from the offending starting point
  - *what is more* if the failure is a property setter invoked by the IDE, then the IDE system can simply reject arbitrary failed attempts to set a property *without additional a priori knowledge of the internals of the components that are interacting*

Example of design-time and run-time exception handling

So, let's see an example of the exception handling strategy in action. Here is the behaviour when I attempt an operation in the IDE that cannot be fulfilled (Figure 2) and how that component is implemented (Figure 3). Note ACCU has a free add-in in the IDE, via the component package which can be plugged in via the IDE's extensibility.

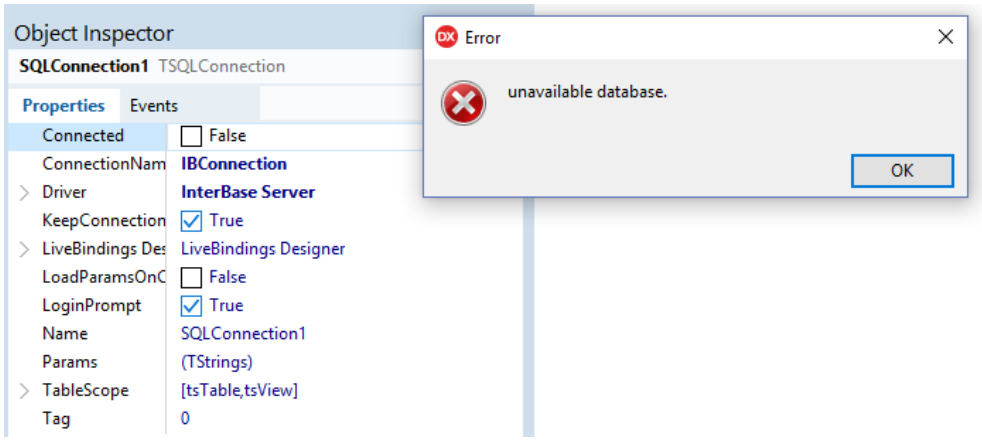


Figure 2

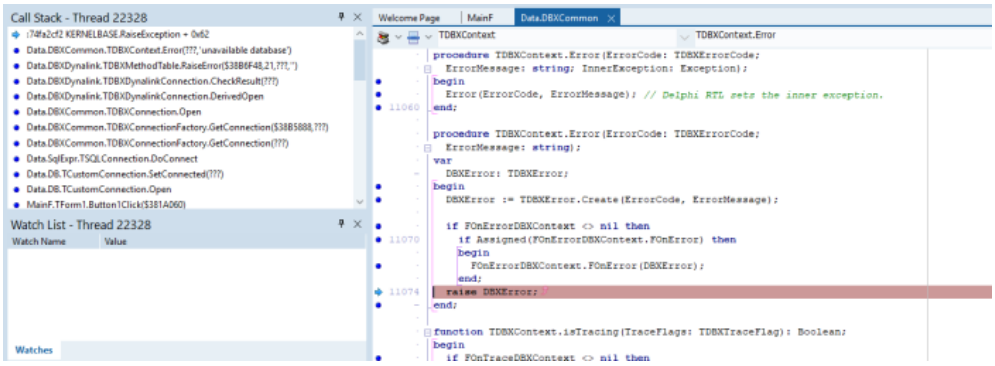


Figure 3

In order to investigate this I only had to add this code and hit ‘Debug’ – hence seeing the code by debugging the application, which will generate the same exception.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  SQLConnection1.Open;
end;
```

In the IDE, the property set on the object fails, and the user is notified.

In the application, the default exception handler installed for the application is invoked, as I elected to not install my own handler, or write an exception handling block.

That’s the power of a unified and usable approach to exception handling. Figure 4 shows how it looks in the app.

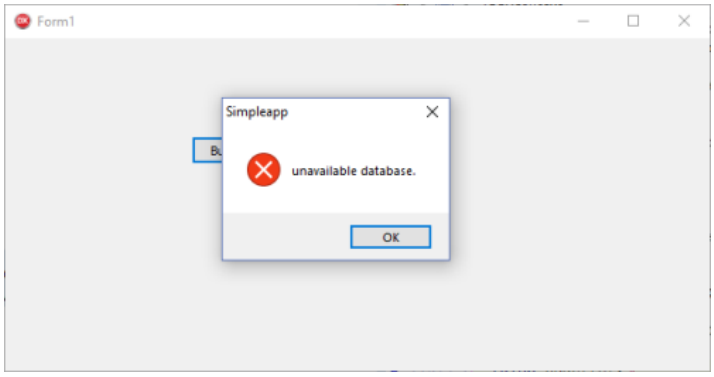


Figure 4

C’mon it can’t have been that perfect, can it?

Now I have to explain why Delphi is not enjoying the popularity it once did.

Web applications

Delphi was great for software that would be popped into the post, on a CD or floppy disks.

When the web based application revolution came, that became irrelevant for many new applications. I feel the offerings within the Delphi toolbox for web development didn’t seem to cut through on the feature set, and of course at the time, everything from the OS to the Development tool needed to be paid for.

Given the competition at the time was the LAMP stack Linux + Apache + MySql + PHP, it was clear how that would play out.

It was proprietary

So, a fact of life is: individual companies get in trouble, go off-beam etc. this can be a real concern. For example the 64-bit compiler took a long time to appear, and some companies like to take a very long view on their enterprise applications.

Cost concerns

It could end up looking pricey compared to free tools. Yet: ‘beware false economies’.

Quality issues

There were some releases that had surprisingly persistent niggles: the debugger in Delphi 4 could be a real pain, especially given the Delphi 3 debugger was an absolute pleasure to work with. This is the kind of thing that worries thoughtful programmers and managers with an eye to the future.

Another syndrome that I saw which was very sad, is that the marvellous extensible IDE was at risk from poorly programmed component packages. The IDE would be blamed for instability, when in fact, the code that it loaded into its core might well be the cause. With an improved architecture, that might have been mitigated, but perhaps not eliminated.

And finally: of course, native code is not to be 100% trusted, yet can only be run as 100% trusted.

Interfacing with code from other systems

Some might believe this was not possible, but in fact it was.

Of course the interaction with the Windows libraries was mainly via the win32 API, proving the point.

So, there was nothing preventing the user from making their own integrations, however these did require some expertise and effort to produce the translation units that could make use of the foreign function interfaces.

In fact, one of the long standing issues with Delphi for some people was that the translation units would not be updated quickly enough when new systems or features arrived in Windows. This resulted in the Delphi programmers either having to roll their own or wait for new Delphi releases.

In retrospect

So, in 2019, what conclusions can we draw?

Rapid Application Development with true visual design

Properties, Methods and Events allow complex UI to be defined in a very minimalist fashion, which is A Good Thing. There are many camps on this topic, but I hope I demonstrated above, the system supported fully visual development, all the way from specified in the designer to fully defined in code and all the waypoints between, and what is more: cleanly.

That was a strength – not all apps need to be coded the same way, or need the same level of complexity.

Strong typing can actually be fine for RAD

Caveat: with the right level of compiler and runtime co-operation.

Personally, I enjoyed debugging in Delphi, as it seemed that faults tended to be more reproducible and more easily reasoned through that some other languages.

Modules are awesome

When the programmer needs to employ a unit from another unit, they really only need to choose whether they want to add it to the interface or not – and there is a habit forming effect from the constant gentle reminder that it was preferable to factor your code well such that dependencies could be added in the implementation.

In some cases one could simply add a reference to a different module to a code file in order to modify/patch the programs’ behaviour – see prior example.

How many of these concerns sound familiar even today?

I suspect we can learn much from the design precepts of the previous glory days of Delphi and take some lessons forward for the next iteration of our tools. The observant reader will spot that I mention both compile time and run-time behaviour of a feature quite often. This is uppermost in my mind because although a hypothetical rapid development environment may have well tuned strictness and guarantees in the compiler or in the serialisation system, the true art is ensuring that there is the minimum ‘impedance mismatch’ between those two concepts.

There is little point in polishing those systems if I then end up spending my time fighting the edge cases when they interact. Typically that borderland is where the tool support is weakest, and also, it tends to be the most user visible portion of applications. My contention is that in making that area just easier to operate in, Delphi allowed developers to focus on the parts of application development that added the most value to the user.

References

Working code referred to in the article can be found at https://github.com/patrickmmartin/Brute (https://github.com/patrickmmartin/Brute)

[Embarcadero-1] https://www.embarcadero.com/products/delphi (https://www.embarcadero.com/products/delphi)

[Embarcadero-2] https://www.embarcadero.com/products/delphi/starter (https://www.embarcadero.com/products/delphi/starter)

[Embarcadero-3] http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Language_Overview (http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Language_Overview)

[Fagg98] Adrian Fagg (1998) ‘A Delphic Experience’ in *Overload* 29, available at https://accu.org/index.php/journals/565 (https://accu.org/index.php/journals/565)

[Fahrni18] Windows GPF, published 12 August 2018 at https://iam.fahrni.me/2018/08/12/1858/ (https://iam.fahrni.me/2018/08/12/1858/)

[Hague09] James Hague ‘A Personal History of Compilation Speed, Part 2’, available from https://prog21.dadgum.com/47.html (https://prog21.dadgum.com/47.html)

[OracleAtDelphi05] ‘10 Years of Deplphi’, published on 8 February 2005 at https://blog.therealoracleatdelphi.com/2005/02/10-years-of-delphi_8.html (https://blog.therealoracleatdelphi.com/2005/02/10-years-of-delphi_8.html)

[Stob12] Verity Stob ‘The Sons of Khan and the Pascal Spring’, *The Register* , 16 January 2012, available at: https://www.theregister.co.uk/2012/01/16/verity_stob_sons_of_khan_2011/ (https://www.theregister.co.uk/2012/01/16/verity_stob_sons_of_khan_2011/)

[Wikipedia-1] ‘Delphi (IDE)’: https://en.wikipedia.org/wiki/Delphi_(IDE) (https://en.wikipedia.org/wiki/Delphi_(IDE))

[Wikipedia-2] ‘Object Pascal’: https://en.wikipedia.org/wiki/Object_Pascal (https://en.wikipedia.org/wiki/Object_Pascal)

[Wikipedia-3] ‘Calling convention’: https://en.wikipedia.org/wiki/Calling_convention (https://en.wikipedia.org/wiki/Calling_convention)

[Wikipedia-4] ‘Component-based software engineering’: https://en.wikipedia.org/wiki/Component-based_software_engineering (https://en.wikipedia.org/wiki/Component-based_software_engineering)

This article was first published on Github: https://github.com/patrickmmartin/pith-writings/blob/master/et_tu_delphi/article.md (https://github.com/patrickmmartin/pith-writings/blob/master/et_tu_delphi/article.md) .

- What is a GPF? [Fahrni18]
- Although, there were some funky compiler features that allowed for late-bound function calls, mainly to support scripting OLE objects.

Patrick Martin Patrick’s github repo was classified using a machine learning gadget as belonging to a ‘noble corporate toiler’. He can’t top that.

FAQ:

About Us
Advertise with ACCU (/faq/advertise)
Conferences (/faq/conference)
Committee Members (/faq/short-committee)
Constitution (/faq/constitution)
Values (/faq/values)
Study Groups (/faq/study-groups-faq)
Mailing Lists (/faq/mailing-lists-faq)
Privacy Policy (/faq/privacy-policy)
Cookie Policy (/faq/cookie-policy)

MEMBERS ONLY:

Study Groups (/members/study-groups)
Mailing Lists (/members/mailing-lists)
Review a Book (/members/review-a-book)
ACCU Committee
Annual General Meetings (/members/agm)
Complaints Procedure (/members/complaints)
Membership Updates
Log Out (/loginout/log-out)

CONTACT:

General Information (mailto:info@accu.org?subject=[ACCU])
Membership (mailto:accumembership@accu.org?subject=[ACCU])
Local Groups (mailto:local-groups@accu.org?subject=[ACCU])
Advertising (mailto:ads@accu.org?subject=[ACCU])
Web Master (mailto:webmaster@accu.org?subject=[ACCU])
Web Editor (mailto:webeditor@accu.org?subject=[ACCU])
Conference (mailto:conference@accu.org?subject=[ACCU])

LINKS:

World of Code (https://blogs.accu.org)
Essential Books (https://github.com/accu-org/essential-books/wiki)
Twitter (https://www.twitter.com/accuorg)
Facebook (https://www.facebook.com/accuorg)
LinkedIn (https://www.linkedin.com/company/accu/)
GitHub (https://www.github.com/accu-org)
flickr (https://www.flickr.com/groups/accu-org)
YouTube (https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw)
RSS Feed (https://accu.org/index.xml)

BUTTONS FOR YOUR WEBSITE:

(/img/accu/button-logo-120x32.png)

(/img/accu/button-logo-225x60.png)

Advertisement

Template by Bootstrapious (https://bootstrapious.com/p/universal-business-e-commerce-template).

Ported to Hugo by DevCows (https://github.com/devcows/hugo-universal-theme).

Customized for ACCU by Jim and Bob.

Hosting provided by Bytemark (http://www.bytemark.co.uk).