whogloo

# Because it still compiles isn’t a strategy

Forty years of brilliance, baggage
and bloody good reasons

Julian Lyndon-Smith



A support group with slides

# Hands up if you maintain code older than a colleague

Keep your hand up if that colleague
has asked what “xz” means.

**Welcome. You are among survivors.**

# Meet Barry, Brenda and Dave

**Every long-lived system has a cast.**

## Barry

The frustrated business expert

**“There must be a better way.”**

Accidentally commissions a forty-year platform.

## Brenda

The person who builds it

**“Give me until Thursday.”**

Solves the actual problem. Acquires seventeen more.

## Dave

The person who knows how it works

**“Don’t touch that. I’ll explain later.”**

Still explaining it. Still hasn’t written it down.

Barry had an idea. Brenda built a system. Dave became a dependency.

# MIS will get back to you

## Corporate answer

“We can schedule that.”

After the steering committee.
And the budget committee.

## Operational answer

“We need to ship on Friday.”

The spreadsheet is smoking.
The warehouse is improvising.

A department discovers that waiting is also a technical risk.

# The niche came first

## What you could buy

Departmental computing
A more affordable machine
General ledger and payroll

## What you still needed

Your production rules
Your awkward stock model
Your particular customers

The machine arrived. Your business did not come in the box.

# Then accounting turned up

## Thursday’s useful program

Solve the niche problem
Get the operation moving
Make somebody’s life easier

## The next thirty years

“Can it invoice?”
“Can it do payroll?”
“Can it run the company?”

Forty years later, we are still running Brenda’s Thursday.

A little respect for the pioneers

# They set out to get the bloody job done

A useful program became an essential system.
An essential system became a company asset.

**Nobody ordered a forty-year platform.**

# The 4GL lowered the barrier

## Business knowledge

What an order means
When stock is usable
Which exceptions matter

## Executable business rules

Read a record
Apply a rule
Update and display

The architecture review was often “Does it work on Monday?”

Compatibility deserves applause

# My 1985 code still compiles

The business knowledge survived.
The investment survived.

**Some surrounding technology did not. Hello, VBX.**



**Compatibility has excellent manners**

# The compiler accepts it

That is not a character reference.

**Syntax passed. Architecture has not yet testified.**

# Still compiles. Still bloody awful.

**Forty years later, the compiler still has no taste.**

```
if false then .
```

Nothing happens.
A remarkably stable feature.

```
if false then .
  message "Order approved".
```

Approval runs anyway.
The indentation lied to you.

```
if error-status:error then
  message "Success".
else
  message "Success".
```

Both paths report success.
Management loves this one.

Backward compatibility preserves our mistakes with impressive loyalty.

# Old code. Newly written old code.

## Written in 1985

Still correct
Stable business behaviour
A deliberate maintenance choice

## Written last Tuesday

Same hidden dependencies
Same session assumptions
“Because that’s our standard”

A recent timestamp can conceal a very old decision.

# The early toolbox had hard edges

## The early environment

64K compiled-code limit
No procedure parameters pre-v6
No internal procedures pre-v6

## Library reuse came later

No persistent procedures pre-v7
Separate programs needed state
Reusable source needed a home

Your architecture had a memory allowance.

# Eight characters. Forty years of consequences.

## DOS: 8.3

8 characters for the base name
3 characters for the extension

custmnt.p  ordent.p  invprt.p

## The convention survived

Long filenames arrived.
Our standards stayed short.

The decoder ring stayed too.

The operating system got long filenames. Our standards didn’t get the memo.

# The .r file paid for folders too

## Names and folders add up

```
run cust_mnt.p.
/* 10 filename characters */

run /some/path/a.p.
run /some/path/b.p.
```

## Short calls through PROPATH

```
propath = propath
  + ",/some/path".

run a.p.
run b.p.
/* 3 filename characters */
```

Short names saved space. PROPATH removed repeated folder text.



**Two letters for tables. Four for fields.**

# Table names in witness protection

I have seen this standard
in at least four independent systems.

**The database knows what “xz” means. Dave knows too.**

# Short names had practical reasons

## The text editor

Fixed-width alignment
Limited visual assistance
A source listing you could scan

## The compiled-size budget

Identifiers and program references
Small compilation units
Space was worth saving

We optimised for the machine and the editor we actually had.

**The memory budget has left the building**

# We saved six bytes

Then spent thirty years
explaining what “xz” means.

**Our succession plan is currently a séance.**

# Shared variables: the available contract

## Caller.p

```
define new shared variable
  amount as decimal no-undo.

amount = 100.
run child.p.
```

## Child.p

```
define shared variable
  amount as decimal no-undo.

amount = amount * 0.9.
```

“Leave the data lying around. Someone will find it.”

# What does this call need?

## Implicit context

```
run calculate-price.p.

/* Which inputs?
   Which outputs?
   Who created the state? */
```

## Visible contract

```
run calculate-price.p (
  input quantity,
  input unitPrice,
  output totalPrice).
```

The first contract is sometimes “Ask Dave.”

# Includes filled the library gap

## What you wanted

Common behaviour
Common definitions
One source to maintain

## What you could do

Insert source into the caller
Use its buffers and variables
Compile the combined program

Source reuse arrived before a comfortable library model.

# The include knows where you live

## Caller

```
define variable total
  as decimal no-undo.

for each orderLine no-lock:
  {add-line.i}
end.
```

## add-line.i

```
total = total
  + orderLine.qty
  * orderLine.unitPrice.

/* Caller supplies the world. */
```

One source copy. A dependency in every caller.

# Persistent procedures changed the options

## A retained instance

Procedure variables stay alive
Internal entry points remain callable
A handle identifies the instance

## Reusable behaviour

Load once
Call more than once
Release deliberately

A library could finally stay for more than one drink.

# A “class” in a .p file

## base-price.p retains state and exposes a function

```
define variable baseAmount as decimal no-undo initial 100.

function Price returns decimal ():
  return baseAmount.
end function.
```

State plus callable behaviour. This should look familiar.

# The derived procedure calls SUPER()

## discount-price.p

```
define input parameter hBase
  as handle no-undo.
this-procedure:
  add-super-procedure(hBase).

function Price returns decimal ():
  return super() * 0.9.
end function.
```

## The caller assembles it

```
define variable hBase
  as handle no-undo.
define variable hDiscount
  as handle no-undo.
run base-price.p persistent
  set hBase.
run discount-price.p persistent
  set hDiscount (hBase).
```

The relationship is real. You are the relationship manager.

# The same calculation in classes

## BasePrice.cls

```
class BasePrice:
  define protected variable
    baseAmount as decimal
    no-undo initial 100.

  method public decimal Price():
    return baseAmount.
  end method.
end class.
```

## DiscountPrice.cls

```
class DiscountPrice
  inherits BasePrice:

  method public override
    decimal Price():
    return super:Price() * 0.9.
  end method.
end class.
```

Classes arrived in 10.1A. INHERITS declares the relationship.

# Same input. Same business result.

## Procedure handle

```
message dynamic-function(
  "Price" in hDiscount).

/* Expected value: 90 */

hDiscount:
  remove-super-procedure(hBase).
delete procedure hDiscount.
delete procedure hBase.
```

## Object reference

```
define variable priceObject
  as DiscountPrice no-undo.

priceObject = new DiscountPrice().
message priceObject:Price().

/* Expected value: 90 */

delete object priceObject.
```

You have been building objects. You also did the plumbing.

# Under the bonnet: what the .r reveals

## Methods

Function machinery
under the newer construct

State plus callable entries

## Read/write properties

Getter function
Setter function

Access expressed as a property

New language capabilities can reuse established execution machinery.

# Familiar machinery. Stronger contracts.

## Super procedures

Dynamic procedure relationships
Runtime lookup of entry points
Separate procedure state

## Classes

Declared inheritance and types
Visibility and interfaces
Inherited members

Calling it “syntax sugar” must not hide the extra guarantees.



**Continuity can be good engineering**

# Progress builds on its past too

One class per .cls file.
A familiar procedure compilation model.

**The foundations can stay. The capabilities can grow.**



**Architecture has entered the chat**

# We changed .p to .cls

Congratulations.
The spaghetti now has a constructor.

**A new file extension cannot draw your boundaries for you.**

# Three lines. A lot of platform.

## Integrated record access, locking and display

```
for each customer no-lock:
  display customer.name.
end.
```

No connection factory in sight. The business person gets on with it.

# A newcomer sees different questions

## Experienced ABL reader

“That reads customers.”

The buffer is familiar.
The environment is familiar.

## Developer from another stack

“Where is the connection?”
“Where did Customer come from?”
“What owns the transaction?”

Familiarity is doing a substantial amount of work.

# The database used to come with the deal

## Integrated-platform mindset

“We are building in OpenEdge.”

Language and data model arrive together.

## Common modern expectation

“We have chosen our database.”

Which language fits our stack? Which libraries can we use?

“Can I use PostgreSQL?” is a reasonable first question.

# Business records and database sets

## An ABL record loop

```
for each customer
  where customer.country = "UK"
  exclusive-lock:
  customer.creditLimit =
    customer.creditLimit * 1.05.
end.
```

## A SQL set operation

```
update customer
set credit_limit =
    credit_limit * 1.05
where country = 'UK';

/* One expression of the set. */
```

Similar arithmetic. Different execution and transaction questions.

# An order & 1m+ rows deserve different questions

## One order

Validate its state
Apply its business rules
Commit its related changes

## A million qualifying rows

Filter close to the data
Consider bulk execution
Measure transfer and locking

FOR EACH is not guilty until measured.

# The defaults are doing things behind your back

## Undoable by default

Variables get rollback support.
Often useful? Sometimes.
Often needed? No.

```
define variable counter
  as integer no-undo.
```

VAR now defaults to NO-UNDO.

## SHARE-LOCK by default

Reads take SHARE-LOCK.
Other readers can share it.
Writers can get blocked.

```
find first customer
  no-lock.
```

Use deliberate locks for updates.

Even the loop counter gets a disaster recovery plan.

# Your .pf file has nostalgia

Documented OpenEdge 12.8 defaults. Measure before changing.

## Memory, circa “it’ll do”

-B: max(3000, 8 × users) blocks
3,000 × 8KiB ≈ 23.4MiB

-Bt: 255 temp-table buffers
Capacity depends on -tmpbsize

## Waiting and checking

-lkwtmo: 1,800 seconds
PASOE / WebSpeed: 10 seconds

-q: review for stable production
Fewer repeated code checks

We bought the RAM. Apparently we’re saving it for visitors.

The archaeological layer

# The constraint went away

The convention stayed.
Then we taught it to the next generation.

**“Because that’s our standard” is not a time machine.**

# The compiler brought a dictionary

**1,982**

## alphabetic keyword spellings

OpenEdge 12.8.8 compiler table, extracted by Julian

Other languages bring libraries. We also brought vocabulary.

# The same token can wear different clothes

## Spellings and abbreviations

BUTTONS / BUTTON
INFO / INFORMATION

Short forms share token codes.

## The learning cost

Recognise the alternatives
Read different generations of code
Separate syntax from convention

The compiler understands the shorthand. The new hire needs the decoder ring.

# WHY IS THE PAYROLL SYSTEM ANGRY?

## Uppercase keywords

```
FOR EACH customer NO-LOCK:
  DISPLAY customer.name.
END.

/* Keywords stand out in text. */
```

## Lowercase keywords

```
for each customer no-lock:
  display customer.name.
end.

/* Same intended behaviour. */
```

We modernised the editor. The examples kept shouting.

# Your compiler lived at work

## The historical barrier

An employer-funded licence
A development environment at work
Home access was not automatic

## The missed opportunities

An evening experiment
A student's first project
An outside contributor's test

"Hello, world" should not need a purchase order.

# Why the useful utility stays in-house

## Permission and incentives

Company or customer ownership
Commercial value in the framework
Someone must support it

## Extraction effort

Your schema
Your includes and shared state
Your session and version assumptions

“Just put it on GitHub” skips most of the job.

# The public projects deserve daylight

## Examples already exist

ABL support for VS Code
Public tools and libraries
Community-maintained collections

## The participation question

Can I install it?
Can I run its examples?
Can I test a contribution?

The scarce resource may be a second maintainer.



A possible ecosystem feedback loop

# Fewer people try it

Fewer independent libraries appear.
The next person has less to try.

**A small ecosystem makes every contributor more valuable.**

# What earns a place in the next version?

## Keep the value

Business knowledge
Correct, proven behaviour
Useful integrated capabilities

## Reconsider the baggage

Dependencies nobody can see
Standards nobody can explain
Workarounds in brand-new code

“It still works” starts the conversation. It does not finish it.

**OpenEdge didn’t just survive 40 years.**
**It carried everything with it.**

**The question is:**
**what should we keep carrying forward?**

Julian Lyndon-Smith