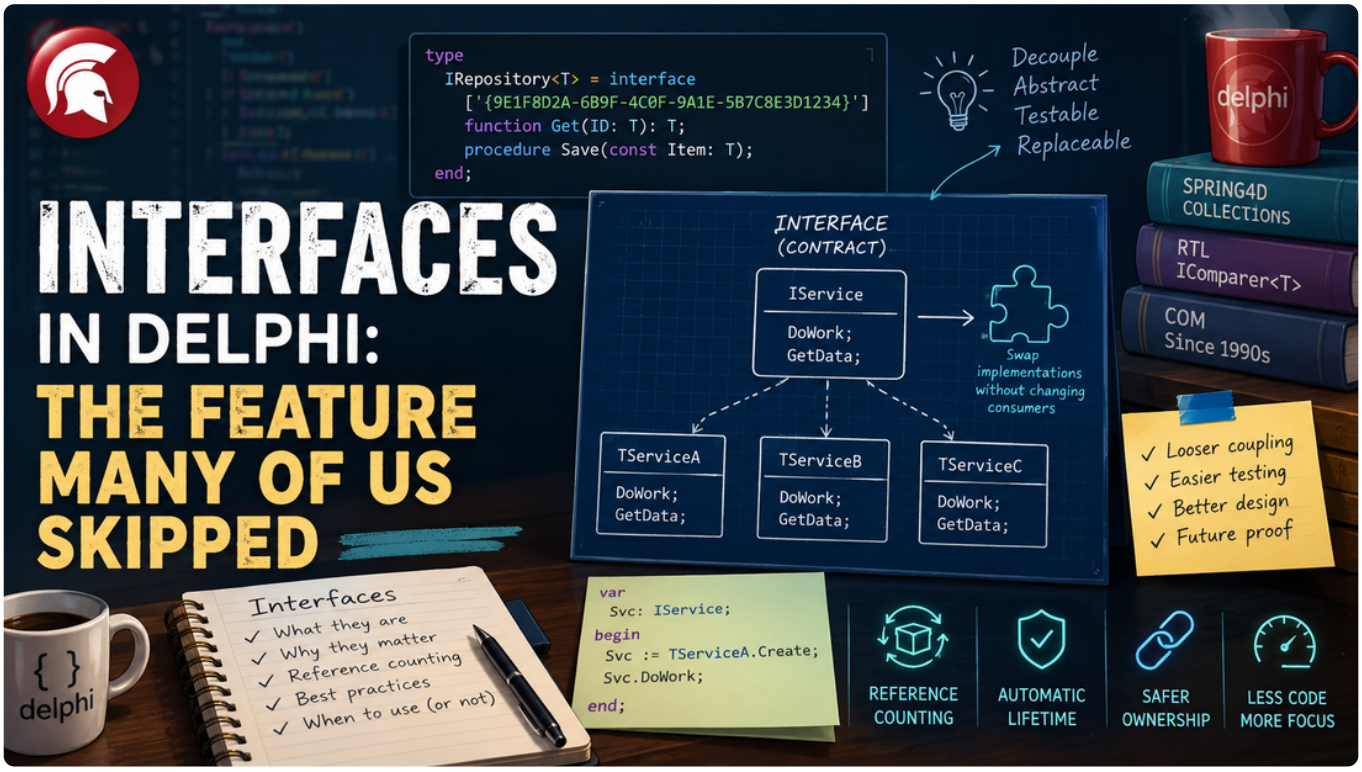
NEWS · JULY 9, 2026

Interfaces in Delphi: Das Feature, das viele von uns übersprungen haben

By Dr. Holger Flick



Eines habe ich in vielen Jahren Arbeit mit Delphi-Teams gelernt: Man kann zwanzig Jahre lang exzellente, ausgelieferte, Geld verdienende Software schreiben, ohne auch nur ein einziges Interface zu deklarieren. Viele meiner Kunden haben genau das getan. Klassen, Formulare, `try/finally`, ein gut organisiertes Datenmodul — dieser Werkzeugkasten trägt bemerkenswert weit, und daran ist nichts auszusetzen.

Aber es gibt eine merkwürdige Lücke. Die Bibliotheken, die Delphi-Entwickler am meisten bewundern, stützen sich *massiv* auf Interfaces. In meiner kürzlichen Tour durch die Collections von Spring4D habe ich etwas

Bemerkenswertes im Quellcode verifiziert: Jede einzelne der 127 Factory-Methoden liefert ein Interface zurück, keine Klasse. Das `IComparer<T>` der RTL ist ein Interface. Ganz COM — die Maschinerie hinter Jahrzehnten von Windows-Automatisierung — besteht aus Interfaces. Das Feature, an dem viele von uns nur vorbeigenickt haben, ist also im gesamten Ökosystem still und leise tragend.

Ich hatte in jenem Spring4D-Beitrag versprochen, dass Interfaces eine eigene, ordentliche Einführung bekommen. Hier ist sie: was ein Interface tatsächlich ist, warum es sich 2026 lohnt, sich damit zu beschäftigen, wie Reference Counting die Speicherverwaltung *einfacher* statt beängstigender macht — und, ebenso ehrlich, wo

die scharfen Kanten liegen und in welchen Situationen eine gewöhnliche Klasse nach wie vor die bessere Wahl ist.

Wir beginnen bei dem, was Sie bereits kennen, und bauen langsam darauf auf — mit kompakten, ausführbaren Beispielen bei jedem Schritt.

Was Sie bereits tun: Klassen, Objekte und try/finally

Beginnen wir auf vertrautem Terrain, denn Interfaces versteht man am besten als *Kontrast* dazu.

In Delphi gehört ein aus einer Klasse erzeugtes Objekt Ihnen — Sie müssen es zerstören. Der Compiler überwacht seine Lebensdauer nicht; das tun Sie. Das Idiom, das jeder Delphi-Entwickler tausendfach getippt hat:

```
var
  list: TStringList;
begin
  list := TStringList.Create;
  try
    list.Add('Hello');
    // ... mit list arbeiten ...
  finally
    list.Free;
  end;
end;
```

Das ist *manuelle Speicherverwaltung*, und — mit aufrichtigem Respekt gesagt — es ist ein völlig gutes Modell. Es ist explizit, es ist vorhersagbar, kein Garbage Collector pausiert Ihre Anwendung zu unpassenden Momenten, und die Frage nach dem Besitz ist immer beantwortbar: Wer es erzeugt hat (oder wem es übergeben wurde), gibt es frei. Delphi-Entwickler bauen auf diesem Modell seit drei Jahrzehnten grundsolide Software.

Seit Delphi 10.4 Sydney die Speicherverwaltung über alle Plattformen hinweg vereinheitlicht hat, ist diese Geschichte zudem wunderbar konsistent: Objektinstanzen werden *überall* manuell verwaltet — Windows, macOS, Linux, iOS, Android. Kein separater ARC-Compiler mehr auf Mobilgeräten mit eigenen Regeln. Eine Sprache, ein Modell.

> **Das Buch, das neben Ihre Tastatur gehört**
> Wenn Sie die maßgebliche Abhandlung darüber suchen, wie Objekt- und Interface-Lebensdauern in Delphi wirklich funktionieren — `Free` vs. `Destroy` vs. `FreeAndNil`, Exceptions während der Zerstörung, Ownership-Muster, Referenzzyklen —, dann ist Dalija Prasnikars Buch Delphi Memory Management for Classic and ARC Compilers meiner Meinung nach die beste Referenz, die die Community hat. Vieles, was dieser Beitrag behutsam einführt, behandelt ihr Buch erschöpfend.

Aber beachten Sie eines an dem Codeausschnitt oben: Das `try/finally` ist *Zeremonie*. Es sagt nichts über Ihr Problem aus — es existiert nur, damit ein vergessenes `Free` nicht zum Leak wird. Behalten Sie diesen Gedanken im Hinterkopf.

Ein Interface ist ein Vertrag — nicht mehr, und nichts Beängstigenderes

Hier die ganze Idee in einem Satz: **Eine Klasse sagt,* wie *etwas getan wird; ein Interface sagt nur,* was *getan werden kann.**

Ein Interface ist ein Typ, der Methoden deklariert, ohne eine einzige davon zu implementieren. Es ist ein Versprechen — ein Vertrag —, den irgendeine Klasse irgendwo erfüllen wird:

```
type
  ILogger = interface
    ['{8B3E4C21-9D5A-4F6E-B2C7-1A0D9E8F3B42}']
    procedure Log(const msg: string);
  end;
```

Das ist eine vollständige, gültige Interface-Deklaration (Object Interfaces in der Delphi-Dokumentation)). Keine Felder, keine Implementierung, kein Konstruktor. Nur: *Alles, was behauptet, ein* `ILogger` *zu sein, kann* `Log`.

> **Wozu die GUID?**
> Die Zeichenkette in eckigen Klammern ist eine GUID) — ein global eindeutiger Bezeichner. Delphi nutzt sie, um Interfaces zur Laufzeit zu identifizieren; das ist es, was `Supports()` und `QueryInterface` funktionieren lässt (und für COM ist sie Pflicht). In der IDE drücken Sie **Strg+Umschalt+G**, und Delphi generiert eine für Sie. Sie werden selten wieder darüber nachdenken — nehmen Sie einfach immer eine auf.

Eine Klasse erfüllt den Vertrag, indem sie das Interface auflistet und seine Methoden implementiert. Der einfachste Einstieg ist, von `TInterfacedObject` zu erben, das die Basisarbeit für Sie erledigt:

```
type
  TFileLogger = class(TInterfacedObject, ILogger)
  private
    FFileName: string;
  public
    constructor Create(const fileName: string);
    procedure Log(const msg: string);
  end;

procedure TFileLogger.Log(const msg: string);
begin
  TFile.AppendAllText(FFileName, msg + sLineBreak);
end;
```

Und die Verwendung sieht so aus — achten Sie darauf, was *fehlt*:

```
var
  logger: ILogger;                      // die Variable hat den INTERFACE-Typ
begin
  logger := TFileLogger.Create('app.log');
  logger.Log('Application started');
  // kein try/finally, kein logger.Free – siehe nächster Abschnitt
end;
```

Der Typ der Variablen ist `ILogger`, nicht `TFileLogger`. Der aufrufende Code kennt den *Vertrag* und sonst nichts.

Diese eine Verschiebung schaltet beide großen Vorteile frei — gehen wir sie also nacheinander durch.

Vorteil 1: Das „Wie" austauschen, ohne die Aufrufer anzufassen

Weil Aufrufer nur vom Vertrag abhängen, können Sie die Implementierung dahinter frei ändern — und genau hier beginnen Interfaces, sich in echten Projekten zu bezahlen zu machen.

Stellen Sie sich drei Klassen vor, die denselben Vertrag erfüllen:

```
type
  TConsoleLogger = class(TInterfacedObject, ILogger)
    procedure Log(const msg: string);    // schreibt nach stdout
  end;

  TSilentLogger = class(TInterfacedObject, ILogger)
    procedure Log(const msg: string);    // tut nichts – perfekt für Tests
  end;

  // ... und TFileLogger von oben
```

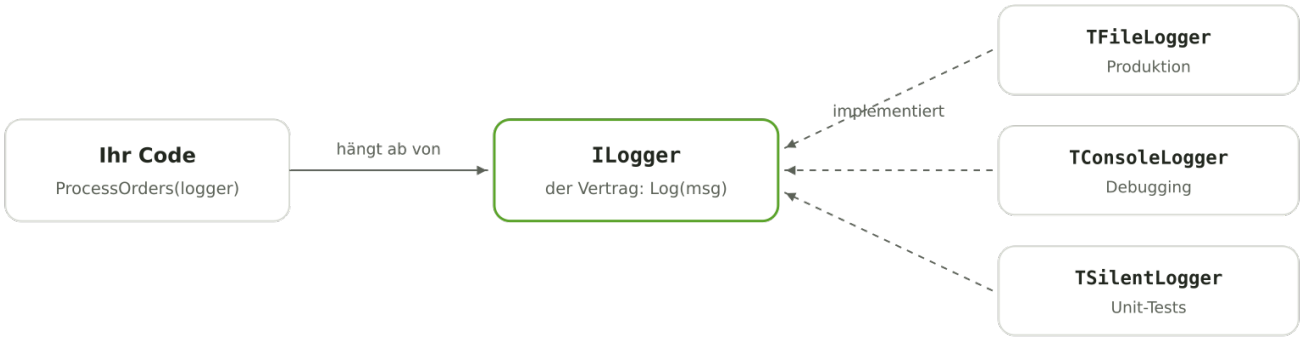
Jeder Code, der gegen `ILogger` geschrieben ist, funktioniert mit allen dreien — unverändert:

```
procedure ProcessOrders(const logger: ILogger);
begin
  logger.Log('Starting order run');
  // ... Geschäftslogik ...
  logger.Log('Done');
end;

// Produktion:
ProcessOrders(TFileLogger.Create('orders.log'));

// im Unit-Test – kein Dateisystemzugriff:
ProcessOrders(TSilentLogger.Create);
```

Hier die Beziehung als Bild — der Aufrufer berührt nur den Vertrag, niemals die konkreten Klassen.



Aufrufer hängen nur vom Vertrag ab; die Implementierungen dahinter lassen sich frei austauschen

Was das Bild zeigt: Die Pfeile aus Ihrem Code enden am Vertrag. Die konkreten Klassen rechts können hinzugefügt, ersetzt oder neu geschrieben werden, ohne dass Ihr aufrufender Code je davon erfährt — und genau das macht Testen, den Wechsel des Loggings und die Anfrage „wir brauchen ein zweites Backend" schmerzfrei.

Beachten Sie: Das ist **Polymorphie ohne Vererbung**. `TFileLogger` und `TSilentLogger` teilen keinen Vorfahren (außer `TInterfacedObject`); sie teilen ein *Versprechen*. Sie sind nicht länger gezwungen, tiefe Klassenhierarchien zu entwerfen, nur um Austauschbarkeit zu bekommen — und eine einzelne Klasse kann mehrere voneinander unabhängige Interfaces gleichzeitig implementieren, etwas, das einfache Vererbung nie leisten kann.

> **Diese Idee ist größer als Delphi**

> Wenn Sie andere Ökosysteme berührt haben, sind Sie diesem Konzept schon begegnet: Interfaces in C# und Java, das `interface` von TypeScript, die impliziten Interfaces von Go — überall dieselbe Trennung von „Vertrag vs. Implementierung". Wer sie in Delphi lernt, lernt eine übertragbare Entwurfsfertigkeit, kein Hersteller-Feature.

Vorteil 2: Das Objekt gibt sich selbst frei

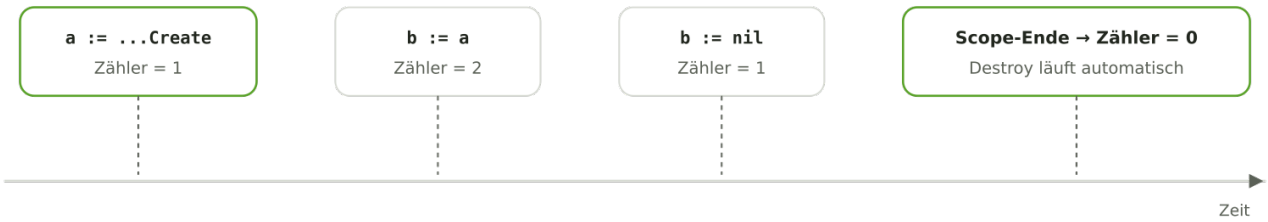
Hier kommt der Teil, der die meisten am stärksten überrascht — und der Grund, warum der frühere Codeausschnitt kein `try/finally` hatte: **Interface-Referenzen werden gezählt, und wenn der Zähler null erreicht, zerstört sich das Objekt selbst.**

Diesem Mechanismus vertrauen Sie übrigens schon jeden Tag — Delphi-`string`s sind seit jeher referenzgezählt, und das blieb nach der Vereinheitlichung in 10.4 auf jeder Plattform so. Niemand schreibt `try/finally` um einen String. Interfaces erweisen Ihren eigenen Typen denselben Dienst.

Die Mechanik, Schritt für Schritt:

```
procedure Demo;
var
  a, b: ILogger;
begin
  a := TFileLogger.Create('app.log');   // Referenzzähler: 1
  b := a;                                // Referenzzähler: 2
  b := nil;                              // Referenzzähler: 1
end;                                     // 'a' verlässt den Gültigkeitsbereich ' Zähler: 0
                                         // ' Destroy wird automatisch aufgerufen
```

Und als Zeitstrahl:



Ein interface-verwaltetes Objekt lebt genau so lange, wie es noch jemand referenziert

Lesen Sie es von links nach rechts: Jede Zuweisung an eine Interface-Variable erhöht den Zähler, jede Variable, die loslässt, verringert ihn — und in dem Moment, in dem niemand mehr eine Referenz hält, räumt das Objekt sich selbst auf. Deterministisch, genau dann, nicht „irgendwann" wie bei einem Garbage Collector. Sie behalten Delphis Vorhersagbarkeit und verlieren die Zeremonie.

Für ein vollständigeres Bild, wie sich Interface-Referenzen) verhalten — einschließlich der Regeln zur Zuweisungskompatibilität —, ist die offizielle Dokumentation gut; Dalijas Buch (oben verlinkt) ist das tiefe Ende des Beckens.

> **Die Lage 2026 ist erfrischend einfach**

> Seit der Vereinheitlichung in 10.4 gelten auf jeder Plattform, die Delphi unterstützt, dieselben Regeln: **Objekte werden manuell freigegeben; Interfaces und Strings geben sich per Reference Counting selbst frei.** Zwei Lebensdauermodelle, sauber getrennt, keine Plattform-Ausnahmen. Wenn Sie Delphi zu einer Zeit gelernt haben, als Mobile noch eigene ARC-Regeln hatte: Es ist heute einfacher als dort, wo Sie aufgehört haben.

Wo Sie Interfaces längst begegnet sind (vielleicht ohne es zu merken)

Interfaces sind kein exotisches Anhängsel von Delphi; sie sind in das Ökosystem eingewoben, das Sie bereits benutzen — und es lohnt sich zu sehen, wie etabliert sie sind.

- **Die RTL selbst.** Jedes Mal, wenn Sie einer Sortierung einen eigenen Comparer übergeben, benutzen Sie `IComparer<T>` aus `System.Generics.Defaults` — ein Interface.
- **COM und Windows-Automatisierung.** Excel oder Word aus Delphi steuern, Shell-Erweiterungen, OLE — das gesamte COM-Modell ist auf Interfaces gebaut. Das ist der historische Grund, warum Delphis Interface-Unterstützung so ausgereift ist.
- **Spring4D.** Wie in der Collections-Tour behandelt, besteht seine gesamte öffentliche Collection-API aus Interfaces — `IList<T>`, `IDictionary<K,V>`, `IEnumerable<T>` —, erzeugt über Factories und freigegeben per Reference Counting. Sobald sich die Konzepte aus *diesem* Beitrag vertraut anfühlen, liest sich diese Bibliothek ganz natürlich.

Das ist die praktische Motivation, diesen Stoff zu lernen: Es ist keine Nischentechnik, sondern das Idiom, das die besten Werkzeuge des Ökosystems bereits sprechen.

Die scharfen Kanten — drei Regeln, die Sie schützen

Ich würde Ihnen einen schlechten Dienst erweisen, wenn ich so täte, als hätten Interfaces keine Fallstricke. Es gibt genau drei große — und alle drei lassen sich mit einfachen Gewohnheiten vermeiden. Lernen Sie diese, bevor Sie irgendetwas Echtes refaktorieren.

Regel 1: Halten Sie dasselbe Objekt niemals gleichzeitig über eine Klassen- und eine Interface-Referenz

Das ist der klassische Anfängerunfall, also schauen wir ihn uns direkt an:

```
var
  obj: TFileLogger;      // Klassenreferenz – zählt NICHT
  log: ILogger;          // Interface-Referenz – zählt
begin
  obj := TFileLogger.Create('app.log');
  log := obj;            // Zähler: 1
  log := nil;            // Zähler: 0 ' Objekt wird HIER zerstört
  obj.Log('crash');      // obj zeigt jetzt auf freigegebenen Speicher!
end;
```

Die Objektreferenz `obj` ist für den Referenzzähler unsichtbar. Wenn die letzte *Interface*-Referenz loslässt, stirbt das Objekt — ganz gleich, welche Klassenreferenzen noch darauf zeigen.

Eine Klassenreferenz zählt nicht mit — erreichen die gezählten Referenzen null, zeigt sie ins Leere

Die Lehre aus dem Bild: Nur der durchgezogene Pfeil hält das Objekt am Leben. Der gestrichelte ist bloß ein roher Zeiger ohne jedes Mitspracherecht.

> **Eine Gewohnheit verhindert das vollständig**
> Sobald eine Klasse am Reference Counting teilnimmt (typischerweise durch Ableitung von `TInterfacedObject`), **halten Sie sie ausschließlich über Interface-Variablen — vom Moment der Erzeugung an**.
>
> Weisen Sie `TFileLogger.Create(...)` direkt einer `ILogger`-Variablen zu und behalten Sie niemals eine `TFileLogger`-Variable. Ein Typ pro Objekt, und diese ganze Fehlerklasse kann nicht auftreten.

Regel 2: Referenzzyklen brauchen einen [weak]-Link

Wenn Objekt A eine Interface-Referenz auf B hält und B eine zurück auf A, können ihre Zähler nie null erreichen — beide warten für immer aufeinander, und Sie haben ein Leak. Die Lösung ist, eine Richtung des Zyklus *ungezählt* zu machen. Seit [Delphi 10.1 Berlin unterstützt der klassische Compiler die Attribute `[weak]` und

`[unsafe]` auf Interface-Referenzen](https://blog.marcocantu.com/blog/2016-april-weak-unsafe-interface-references.html) — genau dafür:

```
type
  TChild = class(TInterfacedObject, IChild)
  private
    [weak] FParent: IParent;   // zählt nicht – und wird automatisch auf nil
  end;                          // gesetzt, wenn der Parent zuerst stirbt
```

Eine `[weak]`-Referenz erhöht den Zähler nicht, und Delphi verfolgt sie: Wird das Ziel zerstört, wird die schwache Referenz `nil`, statt ins Leere zu zeigen. (`[unsafe]` überspringt das Zählen ebenfalls, aber ohne diese Verfolgung — es ist für seltene Expertenfälle.) Die Faustregel: **Die „Besitzer"-Richtung ist stark, die „Rückzeiger"-Richtung ist `[weak]`.** Parent'Child stark, Child'Parent weak.

Regel 3: Komponenten spielen nach ihren eigenen (älteren) Regeln

`TComponent`-Nachfahren — Formulare, Datenmodule und alles, was Sie darauf ablegen — implementieren Interfaces *ohne* referenzgezählte Lebensdauer. Ihre Lebensdauer gehört dem Ownership-Modell: Der Owner gibt sie frei, genau wie eh und je. Das ist ein bewusstes und vernünftiges Design — Ihr Formular soll nicht verschwinden, weil eine Interface-Variable ihren Gültigkeitsbereich verlassen hat. Merken Sie sich nur: „Interfaces = automatische Lebensdauer" gilt für Klassen im Stil von `TInterfacedObject`, nicht für die Welt der VCL-Komponenten.

Wann Sie Interfaces *nicht* verwenden sollten

Nun die ehrliche Abgrenzung, denn eine gute Werkzeugempfehlung sagt immer auch, wo sie endet. Meine Sicht, aus echten Projekten; betrachten Sie sie als Ausgangspunkt.

- **Komponenten und alles mit einem Owner.** Wie in Regel 3 beschrieben, verwaltet das Ownership-Modell diese Lebensdauern bereits elegant. Interfaces fügen dort nichts hinzu — kämpfen Sie nicht gegen das Framework.
- **Funktionierender Code mit klarem, lokalem Besitz.** Eine `TStringList`, die in derselben Methode mit `try/finally` erzeugt und freigegeben wird, ist *völlig in Ordnung*. Es gibt keinerlei Grund, Interfaces nachträglich auf Code zu übertragen, dessen Besitzverhältnisse ohnehin offensichtlich sind. Refaktorieren Sie in Richtung Interfaces, wo Sie den Schmerz spüren (Testen, Austauschen, unklare Lebensdauern) — nicht aus Prinzip.
- **Winzige Datenträger.** Für kleine, wertartige Daten ist ein `record` oft besser als Klasse oder Interface — keine Heap-Allokation, überhaupt keine Lebensdauerfrage.
- **Die heißesten aller Hot Paths.** Reference Counting leistet echte (kleine) Arbeit — die Zähleraktualisierungen sind threadsichere atomare Operationen, und Interface-Methodenaufrufe werden immer virtuell dispatcht. Beim allergrößten Teil Ihres Codes werden Sie das nie bemerken. Wenn Sie in einer wirklich performancekritischen inneren Schleife stecken: Messen Sie, bevor Sie sich auf ein Design festlegen — der Profiler steht über jedem Blogbeitrag, diesen eingeschlossen.
- **Ein Team, das noch nicht an Bord ist.** Konsistenz innerhalb einer Codebasis ist mehr wert als jede einzelne Technik. Führen Sie Interfaces an natürlichen Nahtstellen ein — ein neuer Service, eine testbare Grenze — statt als Big-Bang-Rewrite.

> **Die Faustregel nach Einsatzzweck (meine Sicht)**
> **Greifen Sie zu einem Interface, wenn Ihr Code eine* Naht *braucht: Sie wollen Implementierungen austauschen (Tests, Backends, Plattformen), Sie wollen Aufrufer von einer konkreten Klasse entkoppeln, oder Sie wollen gemeinsame, automatische Lebensdauer für etwas, das weit herumgereicht wird. Bleiben Sie bei einer gewöhnlichen Klasse, wenn der Besitz lokal und offensichtlich ist**, wenn Sie sich in `TComponent`-Territorium bewegen oder wenn das Objekt nie eine Grenze überquert, die eine Abstraktion lohnt. Beides ist erstklassiges Delphi. Die Kunst besteht darin zu wissen, welche Frage Sie gerade beantworten.

Fazit

Interfaces sind keine fortgeschrittene Kuriosität, die an Delphi angeschraubt wurde — sie sind die eine Hälfte davon, wie die Sprache über Objekte denkt, und 2026 ist die Lage klarer denn je.

- **Ein Interface ist ein Vertrag**: Es deklariert das *Was*, eine Klasse liefert das *Wie*. Aufrufer, die von Verträgen abhängen, sind leichter zu testen, zu erweitern und zu ändern.
- **Reference Counting ist deterministische Bequemlichkeit**: Über Interfaces gehaltene Objekte geben sich in dem Moment selbst frei, in dem die letzte Referenz loslässt — derselbe Mechanismus, dem Sie bei Strings schon immer vertraut haben, seit Delphi 10.4 einheitlich auf allen Plattformen.
- **Drei Gewohnheiten schützen Sie**: ein Referenztyp pro Objekt, `[weak]` auf Rückzeigern — und denken Sie daran, dass Komponenten stattdessen dem Ownership-Modell folgen.
-

Grenzen Sie ehrlich ab: Lokaler `try/finally`-Code, Komponenten, Records und gemessene Hot Paths sind allesamt Orte, an denen der *Verzicht* auf Interfaces die richtige ingenieursmäßige Entscheidung ist.

- **Das Ökosystem belohnt Sie**: Von `IComparer<T>` über COM bis zu Spring4Ds vollständig interface-basierten Collections — das ist das Idiom, das der beste Delphi-Code bereits spricht. Für den Tiefgang in Sachen Speicherverwaltung halten Sie Dalija Prasnikars Buch griffbereit.

> Eine Klasse sagt wie. Ein Interface sagt was. Lernen Sie, die beiden zu trennen, und sowohl Ihre Architektur als auch Ihre Speicherverwaltung werden einfacher — ein Vertrag nach dem anderen.

INDEPENDENT · AD-FREE · READER-SUPPORTED

Free to read. Not free to make.

Every tutorial, article, and line of example code on Holger's Code is published without a paywall and without ads. This page is about what keeps it that way.

Professional work, published free

The content here is held to the same standard as professional client work: every tutorial is built as a real project, tested against current versions, documented step by step, and revisited when the frameworks move on. A single in-depth tutorial routinely takes days — and that's before the bills behind it: servers, storage, build infrastructure, licenses, and the hardware everything is verified on.

If an article or tutorial here has saved you an afternoon of debugging, taught you a technique, or helped you ship, it created real value for you. Supporting it isn't charity — it's paying for professional work you found useful, at whatever price you choose. And if you can't, keep reading anyway. That's what the free part means.

Ways to support

KO-FI

Buy me a coffee

A one-time tip — no account, no subscription, any amount.

BOOKS & COURSES

Buy the paid work

The paid tier of everything published free here — and you get something lasting in return.

SPREAD THE WORD

Share an article

Send a tutorial to a colleague, cite an article, subscribe to the RSS feed.

Nothing here goes behind a paywall, whether you chip in or not. But if something on this site earned its keep today, this is where to say so.

Support on Ko-fi —