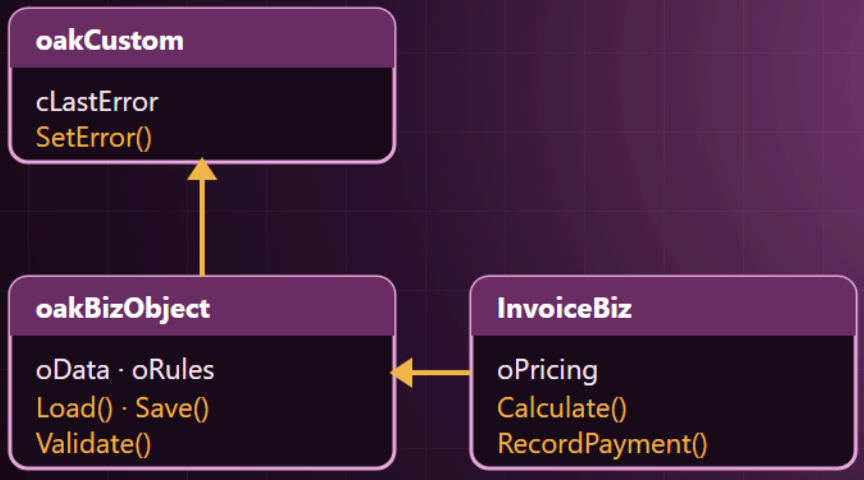
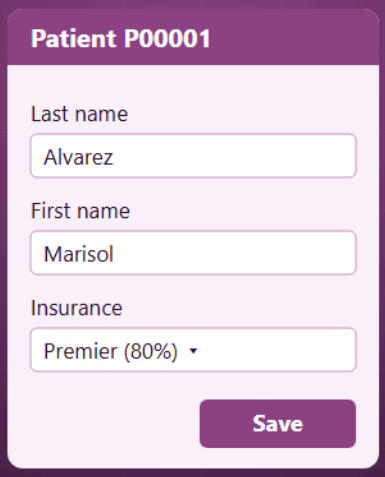
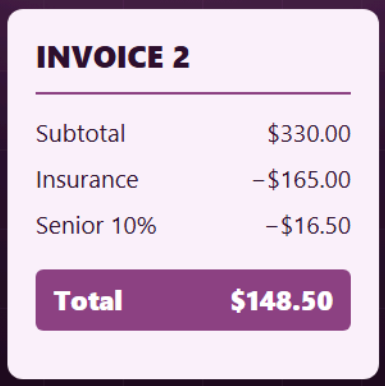



```
DEFINE CLASS oakSeniorDiscount ;
  AS oakFeeDecorator
PROCEDURE Calculate(tcLines)
  loT = DODEFAULT(tcLines)
  loT.yDiscAmt = $16.50
ENDDEFINE
```




```
BINDEVENT(loAppt, "AppointmentChanged", loBoard, "OnChanged")
```

A DEEP DIVE FOR VFP 9 DEVELOPERS

# Object-Oriented Visual FoxPro

## Classes, Forms, and Reusable Application Frameworks in VFP 9

DEFINE CLASS · VCX · Inheritance · Collections · BINDEVENT
Design Patterns · Business Objects · CursorAdapter · Unit Tests

OBJECT MODEL · PATTERNS · LAYERS · FRAMEWORK

THENEXBOOKS

# Object-Oriented Visual FoxPro

*Classes, Forms, and Reusable Application Frameworks in VFP 9*
*From DEFINE CLASS to a tested, layered mini-framework — built step by step*

First Edition

THENEXBOOKS

**Object-Oriented Visual FoxPro**
Classes, Forms, and Reusable Application Frameworks in VFP 9

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Visual FoxPro 9.0 SP2 is the final release of the product, and Microsoft no longer sells or supports it; behavior can differ between service packs and runtime environments. Test every change on a copy of your data, and back up your application and data before applying the techniques in this book to a production system.

Microsoft, Visual FoxPro, Windows, Excel, and SQL Server are trademarks of the Microsoft group of companies. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Microsoft Corporation. Community projects mentioned in this book belong to their respective maintainers.

Oakhurst Dental Group, its clinics, dentists, staff, patients, insurance plans, and the OakDesk application used as the running example in this book are fictional, as is all sample data. Any resemblance to real practices or people is coincidental. Phone numbers use the reserved 555-01xx range and email addresses use reserved example domains. Nothing in this book is dental, medical, insurance, or billing advice.

# Preface

Most Visual FoxPro applications that are still running today began as a handful of programs and forms that grew for years. The code works, the business depends on it, and yet every change is a little frightening: logic lives in the Click methods of buttons, the same calculation is copied into three forms, PUBLIC variables carry state from one screen to the next, and nobody is quite sure what will break when a rule changes.

Visual FoxPro has had a complete object model since its first version. It offers classes with inheritance, encapsulation with PROTECTED and HIDDEN members, Access and Assign methods, visual class libraries, containers, collections, event binding, structured exceptions, and the CursorAdapter class for data access. Many developers use forms and controls every day without ever writing a class of their own. This book is about closing that gap: using VFP's object model deliberately to build applications that are layered, testable, and easy to change.

We do not stop at syntax. We build one real application, OakDesk, from a small root class up to a reusable mini-framework with business objects, a data access layer, form and control classes, an application object, a form manager, object-driven menus and toolbars, and a unit-test harness written in VFP itself. Every class has a clear job, and every worked example in the book was run in Visual FoxPro 9 and checked against its printed output.

## Who This Book Is For

- **VFP developers who write mostly procedural code** and want to design with classes, not just use the ones VFP gives them.
- **Developers who inherited an object-oriented VFP framework** and need to understand how it is put together before they change it.
- **Maintainers of long-running business applications** who want to refactor safely, one rule at a time, without a risky rewrite.
- **Developers coming from C#, Java, or VB** who must work in VFP and want to know exactly where its object rules differ from what they expect.

We assume you know the VFP basics: the Command window, tables and indexes, SELECT-SQL, and simple forms. Every object-oriented idea is explained from first principles.

## What You Will Learn

- How VFP's object model works: DEFINE CLASS and visual classes, properties, methods, events, object lifetime, and references.
- How to use inheritance well: THIS, THISFORM, DODEFAULT(), NODEFAULT, and the scope resolution operator.

- How to protect a class's design with PROTECTED and HIDDEN members, Access and Assign methods, and _MemberData.
- How to compose objects with containers and collections, and how to connect them with RAISEEVENT() and BINDEVENT().
- How the classic design patterns look in VFP: factory, strategy, template method, observer, decorator, and mediator.
- How to separate business objects from the user interface, build a data access layer, and subclass CursorAdapter.
- How to handle errors in classes with TRY...CATCH, the Error event, and exception objects.
- How to build a base class library and reusable form, grid, combo box, list box, pageframe, and container classes.
- How to run an application through an application object, a form manager, and menus and toolbars defined by objects.
- How to unit-test classes with a small harness written in VFP, refactor legacy procedural code into classes, and assemble everything into a mini-framework you can extend.

## How This Book Is Organized

**Part I — The VFP Object Model** (Chapters 1–5) covers classes and objects, DEFINE CLASS in depth, inheritance, encapsulation, and visual class libraries. **Part II — Composition, Collections, and Events** (Chapters 6–10) covers containers, collections, event binding, and six design patterns. **Part III — Business Objects and Data Access** (Chapters 11–15) separates rules from the user interface and builds the data layer, validation, and error handling. **Part IV — Forms and Controls as Classes** (Chapters 16–20) builds the base class library and reusable form and control classes. **Part V — Application Architecture** (Chapters 21–24) adds the application object, the form manager, object-driven menus and toolbars, and unit testing. **Part VI — Refactoring and the Capstone Framework** (Chapters 25–28) refactors legacy code, examines object lifetime and performance, and assembles and extends the finished framework. The appendices provide a cheat sheet, a class reference with templates, a production checklist, and a glossary.

Each chapter ends with Try It Yourself exercises and a **Key Takeaways** box for quick review.

## The Running Example: Oakhurst Dental Group

Throughout the book we work for **Oakhurst Dental Group**, a fictional group of three dental clinics. Its practice application, **OakDesk**, keeps patients, dentists, appointments, treatments, and invoices. Version 1 was written procedurally years ago and still runs every day, but each new rule from the practice manager takes longer to add than the last. We rewrite it as **OakDesk 2**: business objects that know the booking, insurance, discount, and invoicing rules; a data service that is the only code that knows where the data lives; form and control classes that only present and collect data; and an

application layer that ties it together. The sample data is small enough to check by hand — five patients, six appointments, and a handful of treatments — so every number in the book can be verified.

## Versions and Conventions

Examples target **Visual FoxPro 9.0 SP2**, the final release; nearly everything also applies to VFP 8, and a Note marks features that are new in VFP 9. All code was run headless in VFP 9: forms and controls were created with CREATEOBJECT, driven through their methods, and checked by printing results, so no example depends on clicking. Where a step happens in an interactive designer, we describe what you should see. Output is shown with US settings (`SET DATE AMERICAN`, `SET CENTURY ON`), so dates appear as `03/09/2026` and money as `$41.00`; the `?` command prints Currency values with four decimals, such as `41.0000`.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause errors, lost changes, or hard-to-find bugs.

VFP keywords appear in uppercase (`DEFINE CLASS`, `LPARAMETERS`), variables and properties use Hungarian prefixes (`lcName`, `tnApptId`, `oData`, `cLastError`), and framework classes start with `oak` (`oakCustom`, `oakForm`, `oakDataService`). Long lines are continued with a semicolon to fit the page. Appendix B lists every OakDesk class with its public methods.

# Contents

Preface

PART I — THE VFP OBJECT MODEL

1 Objects in Visual FoxPro: The Object Model and the OakDesk Rewrite

2 DEFINE CLASS in Depth: Properties, Methods, and Object Lifetime

3 Inheritance: THIS, DODEFAULT, NODEFAULT, and the Scope Resolution Operator

4 Encapsulation: PROTECTED, HIDDEN, Access and Assign Methods, and _MemberData

5 Visual Classes: VCX Libraries, the Class Designer, and PRG Twins

PART II — COMPOSITION, COLLECTIONS, AND EVENTS

6 Composition and Containers: ADD OBJECT, AddObject, and Object References

7 Collections and Object Graphs

8 Events: Native Events, RAISEEVENT, and BINDEVENT

9 Patterns in VFP, Part 1: Factory, Strategy, and Template Method

10 Patterns in VFP, Part 2: Observer, Decorator, and Mediator

PART III — BUSINESS OBJECTS AND DATA ACCESS

11 Business Objects: Separating Rules from the User Interface

12 A Data Access Layer: The oakDataService Class

13 CursorAdapter as a Class

14 Validation and Business Rules: Broken-Rule Collections and Workflow

15 Error Handling in Classes: TRY/CATCH, the Error Event, and Exceptions

PART IV — FORMS AND CONTROLS AS CLASSES

16 Form Classes and the oakbase Library

17 Data-Aware Controls: Binding Controls to Business Objects

18 Grids as Classes: Columns, Dynamic Properties, and a Reusable Grid

19 Combo Boxes, List Boxes, and Lookup Classes

20 Pageframes, Containers, and Composite Controls

**PART V — APPLICATION ARCHITECTURE**

21 The Application Object and Global Services

22 A Form Manager: Opening, Tracking, and Closing Forms

23 Menus and Toolbars Driven by Objects

24 Unit Testing Classes: A Tiny Test Harness in VFP

**PART VI — REFACTORING AND THE CAPSTONE FRAMEWORK**

25 Refactoring Legacy Procedural Code into Classes

26 Object Lifetime, Memory, and Performance

27 Capstone, Part 1: Assembling the OakDesk Framework

28 Capstone, Part 2: Extending, Packaging, and Maintaining the Framework

**APPENDICES**

A OOP in VFP Cheat Sheet

B OakDesk Class Reference and Templates

C Framework Production Checklist

D Glossary

PART I

# The VFP Object Model

Classes and objects the VFP way: defining classes in code and in visual libraries, properties, methods, and events, how objects live and die, inheritance with DODEFAULT() and NODEFAULT, and protecting a design with PROTECTED and HIDDEN members and Access and Assign methods.

1 · Objects in Visual FoxPro

2 · DEFINE CLASS in Depth

3 · Inheritance

4 · Encapsulation

5 · Visual Classes and PRG Twins

CHAPTER 1 · PART I — THE VFP OBJECT MODEL

# Objects in Visual FoxPro: The Object Model and the OakDesk Rewrite

*You have been using objects in Visual FoxPro every day, even if your code is mostly PRGs and form methods. In this chapter we name the pieces of the VFP object model, look honestly at what procedural code costs a real application, meet the dental practice whose system we will rebuild, and write the first class of the OakDesk 2 framework.*

---

## Why a Book About Objects in VFP?

Visual FoxPro has supported real classes since version 3.0, and VFP 9 (SP2), the final release, has a mature object model: inheritance, encapsulation with `PROTECTED` and `HIDDEN`, Access and Assign methods, event binding, collections, exceptions, and visual class libraries. Yet a great deal of production VFP code never uses most of it. Forms are built in the Form Designer and their methods fill up with SQL, business rules, and calls to procedures in a giant PRG. Global state lives in `PUBLIC` variables. When the application grows, every change becomes a careful search for all the places that read `gnTotal` or depend on whatever work area happens to be selected.

That style works, which is exactly why so much of it survives. It also becomes expensive. This book is about the other way: designing with classes so that rules live in one place, forms become thin, and the parts of an application can be tested and reused. We assume you already know the Command window, tables and indexes, `SELECT-SQL`, and how to build a basic form. We will not spend pages on those. Every chapter teaches one part of the object model and applies it to a single running example.

If you inherited an object-oriented VFP application and are trying to understand why it is put together the way it is, this book is for you too. VFP frameworks reuse a small set of ideas — base classes, business objects, a data layer, an application object, a form manager — and once you have built each one, they stop looking like magic.

## What "Object-Oriented" Means in Visual FoxPro

Object-oriented programming has a vocabulary that sounds abstract until you attach it to things you already use. Here are the terms as VFP uses them.

| Term | Meaning in VFP | Example you know |
|---|---|---|
| Class | A definition: properties with default values, plus methods and events | The `TextBox` base class |
| Object | A live instance of a class, created at run time | `Thisform.txtName` on a running form |

| | | |
|---|---|---|
| Property | A named value that belongs to an object | `Caption`, `Value`, `Visible` |
| Method | Code that belongs to an object and that you call | `Refresh()`, `SetFocus()` |
| Event | Code VFP calls when something happens | `Click`, `Init`, `InteractiveChange` |

A **class** is a blueprint. When you run `CREATEOBJECT("Form")`, VFP builds an **object** from the `Form` class and hands you a reference to it. You can create as many objects from one class as you like, and each one has its own copy of the properties. Change `Caption` on one form and the other forms do not notice.

The difference between a **method** and an **event** matters in VFP more than in most languages. An event is a method that VFP itself calls at a specific moment: when the object is created (`Init`), when it is destroyed (`Destroy`), when the user clicks (`Click`), when a value is typed (`InteractiveChange`). You can also call event methods yourself, like any method, and that is the foundation of how we test forms without showing them. Methods you add yourself, such as `CalculateTotal()`, are only ever called by code. Chapter 8 explores events in depth.

Three ideas sit on top of this vocabulary, and each gets its own chapter early in the book:

- **Inheritance** (Chapter 3) — a class can be defined as a subclass of another and receives all its properties and methods, changing only what differs.
- **Encapsulation** (Chapter 4) — an object hides the details of how it works and exposes a small public surface. VFP enforces this with `PROTECTED` and `HIDDEN` members.
- **Polymorphism** — different classes can answer the same method call in their own way. A coverage object for a BASIC insurance plan and one for a PREMIER plan both answer `CoverAmount()`, with different results, and the code that calls them does not need an `IF` for each plan.

A fourth idea, **composition** — objects that hold references to other objects and delegate work to them — is in practice the most important of all, and Chapter 6 is devoted to it.

## The Object Model You Already Use

Even the most procedural VFP program runs inside an object model. The main VFP window is an object, available as `_SCREEN`. The VFP application itself is an object, available as `_VFP`. Every form is an object, and every control on it is an object contained in the form. You have probably written `_SCREEN.Caption = "My App"` without thinking of it as object-oriented code, but it is: you set a property of an object through a reference.

The quickest way to get a feel for the model is to ask objects about themselves. Every VFP object has a `BaseClass` property (the VFP base class it is ultimately built from), a `Class` property (the class it was created from), and a `Name`. The following program asks a few questions and creates a form without ever showing it.

objmodel.prg

```
? VERSION()
? _SCREEN.BaseClass, _SCREEN.Name
? _VFP.Name, _VFP.Version
? _VFP.Forms.Count

LOCAL loForm
loForm = CREATEOBJECT("Form")
? loForm.BaseClass, loForm.Visible
loForm.Caption = "Hello from an object"
loForm.AddObject("txtName", "TextBox")
? loForm.ControlCount, loForm.Controls(1).Name, loForm.txtName.BaseClass
loForm.txtName.Value = "Marisol"
? loForm.txtName.Value, loForm.txtName.Parent.Caption
? _SCREEN.FormCount
loForm = .NULL.
? _SCREEN.FormCount
```

Output

```
Visual FoxPro 09.00.0000.5815 for Windows
Form Screen
Microsoft Visual FoxPro 9.0
         0
Form .F.
         1 TXTNAME Textbox
Marisol Hello from an object
         1
         0
```

Several things in that output are worth slowing down for.

- `_SCREEN.BaseClass` is `Form`. The main VFP window is a form object, which is why you can add controls to it with `_SCREEN.AddObject()` and why its properties look familiar.
- `_VFP` is a different kind of object: an application object that VFP also exposes to other programs through automation. It has properties such as `Name` and `Version`, but it does not behave like a form. Asking for `_VFP.BaseClass` fails with an OLE "Unknown name" error, because `_VFP` is not built from a VFP base class at all.
- A form created with `CREATEOBJECT()` has `Visible = .F.`. It exists, it has controls, and it counts in `_SCREEN.FormCount`, but nothing appears on screen until you call `Show()` or set `Visible`. This is the property that lets us test forms from a program.
- `AddObject()` created a text box inside the form at run time, and the text box knows its container through `Parent`. Containers and their contents form a tree, and you navigate it with dots: `loForm.txtName.Value`.
- Setting the only reference to `.NULL.` released the form: `FormCount` went back to 0. Objects live as long as something refers to them. Chapter 2 covers object lifetime in detail.

> **NOTE**
>
> The version string reads `09.00.0000.5815` because the machine we use to verify this book runs VFP 9 SP1. The book targets VFP 9 (SP2), the final release, and everything shown behaves the same in SP1 and SP2 unless a Note says otherwise. Your version string will differ if you run SP2.

### Base Classes: Visual and Non-Visual, Containers and Controls

VFP ships with a fixed set of **VFP base classes**: `Form`, `TextBox`, `Grid`, `Custom`, `Collection`, `Session`, `Exception`, `CursorAdapter`, `Toolbar`, and others. You cannot change a base class itself; you can only create classes based on it. That single fact shapes every VFP framework, because it means the first thing a framework does is subclass each base class it uses once, so that the team has a place to put shared behavior. We do that for OakDesk in Chapter 5 and Chapter 16.

Base classes divide two ways. **Visual** classes have a screen presence (forms, text boxes, grids, labels). **Non-visual** classes have none (`Custom`, `Collection`, `Session`, `Exception`, `Timer`, `CursorAdapter`). Independently, some classes are **containers** that can hold other objects (`Form`, `Container`, `PageFrame`, `Grid`, `Custom`), while others are **controls** that cannot (`TextBox`, `Label`, `CommandButton`).

| Base class | Visual? | Holds objects? | Typical OakDesk use |
|---|---|---|---|
| `Custom` | No | Yes | Business objects, services |
| `Collection` | No | Items | Lists of rules, open forms |
| `Session` | No | No | A private data session without a form |
| `Form` | Yes | Yes | Every OakDesk screen |
| `Container` | Yes | Yes | Reusable composite controls |
| `TextBox` | Yes | No | Data entry fields |
| `Grid` | Yes | Columns | The appointment day book |

The class most of this book revolves around is the humble `Custom`. It has no visual appearance, very few built-in properties, and it can hold other objects. It is the natural parent for business objects, data services, loggers, and everything else that is about logic rather than pixels.

> **TIP**
>
> When you are not sure what an object is, ask it. `? loX.BaseClass, loX.Class` in the Command window answers most questions about an unfamiliar framework object, and `AMEMBERS()`, which we meet in Chapter 2, lists everything it can do.

## Meet Oakhurst Dental Group

Our running example is a fictional practice, **Oakhurst Dental Group**, a small group of three dental clinics: **Oakhurst Northgate** in Oakhurst, **Oakhurst Riverside** in Millbrook, and **Oakhurst Harborview** in Port Ellis. The group was founded by **Dr. Miriam Castell**, who still sees patients at Northgate. Four dentists work across the clinics, and the practice manager, **Renata Holloway**, owns the business rules: how insurance plans pay, who gets the senior discount, what happens when a patient does not show up.

The practice runs on an application called **OakDesk**. It handles patients, dentists, appointments, treatments, and invoices. **OakDesk 1.x** was written years ago by a contractor, **Victor Lindqvist**, who has since moved on. He left software that works, which deserves respect, and code that is tangled, which deserves a plan. The practice now has an in-house developer, **Jonah Pierce**, who is rewriting the system as **OakDesk 2** with classes. Jonah is your stand-in: every decision he makes in this book is one you will make in your own applications.

Renata is the kind of user developers learn to value. She reports problems precisely. Her latest note to Jonah reads: "George Whitfield was charged $165.00 for appointment 2 last month. He is 68. He should have had the senior discount, so the bill should be $148.50. The clinic in Millbrook had the same thing happen twice this year." Finding out why takes Jonah into the billing code.

All people, clinics, plans, and data in this book are fictional. Phone numbers use the `555-01xx` range and email addresses use `example.com`.

## Why OakDesk 1's Procedural Code Hurts

Here is the heart of OakDesk 1's billing, the procedure `MAKEBILL` from `legacy_billing1.prg`. This is the first twenty lines; the rest scans the appointment's treatment lines and adds up the totals. Chapter 25 refactors the whole procedure, so do not try to fix it yet. Just read it the way Jonah did.

C:\OakDesk\legacy\legacy_billing1.prg (excerpt)

```
PROCEDURE MAKEBILL
PARAMETERS pnAppt
PUBLIC gnSub, gnIns, gnDisc, gnTot
STORE 0 TO gnSub, gnIns, gnDisc, gnTot
cov = 0
fees = 0
SELECT 0
USE appointments AGAIN ALIAS ap
LOCATE FOR appt_id = pnAppt
SELECT 0
USE patients AGAIN ALIAS pt
LOCATE FOR pat_id = ap.pat_id
pct = IIF(pt.ins_plan = "PREMIER", 80, IIF(pt.ins_plan = "BASIC", 50, 0))
age = YEAR(ap.appt_date) - YEAR(pt.birth_date)
IF MONTH(ap.appt_date) * 100 + DAY(ap.appt_date) < ;
      MONTH(pt.birth_date) * 100 + DAY(pt.birth_date)
   age = age - 1
```

```
ENDIF
SELECT 0
USE apptlines AGAIN ALIAS al
```

The procedure is not wrong, and when it runs on clean data it produces the right numbers. Called for appointment 2, it computes George's bill correctly: subtotal $330.00, insurance $165.00, discount $16.50, total $148.50. So why did George get the wrong bill? Because the answer lives in four `PUBLIC` variables, and the invoice form in OakDesk 1 read `gnTot` after another routine had already reset it. That is the first problem on a long list.

- **Results travel through globals.** `gnSub`, `gnIns`, `gnDisc`, and `gnTot` are `PUBLIC`. Any code anywhere can read or overwrite them, and nothing tells you who did. A two-line test makes the point: after `DO MAKEBILL`, the statement `gnTot = "oops"` runs without complaint and the total is now a string.
- **Undeclared variables.** `cov`, `fees`, `pct`, and `age` are never declared, so they are created as private variables. Private variables are visible to every routine this one calls. If a called routine happens to use a variable named `age`, the two silently share it.
- **Hidden dependence on the environment.** The code assumes the database is open and the right folder is the default. It changes the current work area with `SELECT 0` and opens tables under short aliases. The caller's selected work area after the call is whatever the procedure left behind.
- **Magic numbers.** 80, 50, 65, and 0.10 are business rules: the PREMIER and BASIC insurance percentages, the senior age, and the senior discount. They are written inline. When Renata adds a plan, someone has to find every `IIF()` like this one, and OakDesk 1 has several.
- **One routine, many jobs.** Opening data, finding records, computing an age, choosing an insurance percentage, and summing fees are all mixed together. None of those jobs can be reused or tested on its own. The age calculation, for example, also appears in the patient form and in a report.
- **No error contract.** If the appointment number does not exist, `LOCATE` fails quietly, the patient lookup uses an empty value, and the procedure returns zeros that look like a real bill.

The second test makes the global-state problem concrete. We run the legacy procedure, then check what kinds of variables are left behind.

`legacy_peek.prg`

```
OPEN DATABASE oakdesk SHARED
DO MAKEBILL IN legacy_billing1 WITH 2
? gnSub, gnIns, gnDisc, gnTot
? TYPE("gnTot"), TYPE("cov"), TYPE("pct")
gnTot = "oops"
? gnTot
```

Output

```
330.0000   165.0000   16.5000   148.5000
Y U U
oops
```

The private helpers `cov` and `pct` are gone after the procedure returns (type `U`, undefined), but the four results remain as globals of type `Y` (Currency) that any line of code can replace with anything. The procedure communicates through shared mutable state, and shared mutable state is exactly what made George's bill wrong.

> **WARNING**
>
> Do not start a rewrite by deleting code like `MAKEBILL`. It encodes years of business decisions, some of which nobody remembers making. In OakDesk 2 we keep the legacy procedure around and, in Chapter 25, prove with a characterization test that the new classes return the same totals for appointments 1 to 4 ($41.00, $148.50, $425.00, and $35.00) before anything is switched over.

## The OakDesk 2 Plan

Jonah's plan is not to rewrite everything at once. It is to give each kind of responsibility a home, so that each piece of code has one reason to change. OakDesk 2 is organized in layers. Each layer talks only to the layer directly below it, through objects with small public interfaces.

```
+-------------------------------------------------------------+
|  USER INTERFACE        frmPatient, frmDayBook, frmAppointment |
|  (forms, controls)     oakForm, oakTextBox, oakGrid ...       |
+----------------------------+--------------------------------+
                             | calls methods, binds to oData
+----------------------------v--------------------------------+
|  BUSINESS OBJECTS      PatientBiz, AppointmentBiz, InvoiceBiz |
|  (rules, workflow)     oakPricingFactory, coverage classes    |
+----------------------------+--------------------------------+
                             | Query(), Execute(), SqlValue()
+----------------------------v--------------------------------+
|  DATA SERVICE          oakDataService, oakCursorAdapter       |
+----------------------------+--------------------------------+
                             | OPEN DATABASE, SELECT-SQL
+----------------------------v--------------------------------+
|  DATABASE              oakdesk.dbc: patients, appointments,   |
|                        apptlines, invoices, treatments ...    |
+-------------------------------------------------------------+
   Across all layers: goApp (oakApplication) with oData, oForms,
   oLog, oMenu -- the only PUBLIC variable in OakDesk 2
```

Read the diagram from the top. A form, such as `frmPatient`, displays and edits data but contains no business rules. It asks a business object, such as `PatientBiz`, to load, validate, and save a patient. The business object knows the rules ("last name is required", "BASIC pays 50% of covered fees") but not where the data lives. It asks the data service, `oakDataService`, to run queries. The data service is the only class that knows the database is a DBC in a particular folder. Underneath sits `oakdesk.dbc`.

This arrangement buys three things that OakDesk 1 never had:

- **Testability.** A business object can be created and exercised from a PRG with no form at all. Renata's senior-discount complaint becomes a test that creates an `InvoiceBiz`, calculates appointment 2, and checks for $148.50.
- **Replaceability.** If the practice later moves its data to SQL Server, the data service changes; the business objects and forms do not. Chapter 13 shows a CursorAdapter version of the same idea.
- **One place for each rule.** The insurance percentages live in three small coverage classes. The senior age lives in one `#DEFINE`. Changing a rule means changing one class.

The last box in the diagram is the application object. OakDesk 2 has exactly one `PUBLIC` variable, `goApp`, an `oakApplication` object created only in `main.prg` or in a test driver. It holds the shared services: the data service (`goApp.oData`), the form manager (`goApp.oForms`), the logger (`goApp.oLog`), and the menu builder (`goApp.oMenu`). Everything else is reached through it or passed in as a parameter. Chapter 21 builds it.

## The Classes, Layer by Layer

You do not need to remember these names yet, but it helps to see the whole cast once. Framework classes start with `oak`; application classes are named after what they handle.

| File | Main classes | Built in |
|---|---|---|
| `oakbase.prg` | `oakCustom`, `oakForm`, base controls | Ch 1, 5, 16 |
| `oakpricing.prg` | Coverage strategies, fee calculator, factory | Ch 2, 3, 9, 10 |
| `oakbiz.prg` | `oakBizObject`, `PatientBiz`, `AppointmentBiz`, `InvoiceBiz` | Ch 3, 11, 14 |
| `oakdata.prg` | `oakDataService`, `oakCursorAdapter` | Ch 12, 13 |
| `oakapp.prg` | `oakApplication`, logger, form manager, menus | Ch 21–23 |
| `oakui.prg` | `frmPatient` and other forms | Ch 16–20 |
| `oaktest.prg` | `oakTestCase`, `oakTestRunner` | Ch 24 |

## The Folder Layout

OakDesk 2 lives in `C:\OakDesk\`. The layout is deliberately boring, because boring layouts are easy to explain to the next developer.

```
C:\OakDesk\
   main.prg                && startup program (Chapter 21)
   oakdesk.pjx             && project, builds oakdesk.exe (Chapter 28)
   oakdesk.log             && written by oakLogger
   data\                   && oakdesk.dbc and its tables
   classes\                && oakbase.prg + oakbase.vcx, oakdata.prg,
                           && oakpricing.prg, oakbiz.prg, oakapp.prg,
                           && oakui.prg + oakui.vcx
   legacy\                 && legacy_billing1.prg (kept for Chapter 25)
   tests\                  && oaktest.prg, test_oakdesk.prg
```

The database, `oakdesk.dbc`, holds seven tables: `clinics`, `dentists`, `patients`, `treatments`, `appointments`, `apptlines` (the treatments planned or done in each appointment), and `invoices`. A setup program, `setup.prg`, creates the database and fills it with the same sample data every time: three clinics, four dentists, five patients, seven treatments, and six appointments. Invoices start empty. Because the data is identical on every run, the numbers in this book are reproducible: when we say appointment 1 produces an invoice total of $41.00, you will see $41.00.

| Sample record | Key facts |
|---|---|
| Patient P00001 | Marisol Alvarez, Northgate, PREMIER plan |
| Patient P00002 | George Whitfield, Riverside, BASIC plan, born 11/03/1957 |
| Dentist D001 | Dr. Miriam Castell, Northgate, GENERAL |
| Appointment 2 | George with D002 on 03/02/2026: exam, X-rays, filling |
| Treatment EXAM01 | Comprehensive exam, $85.00, covered by insurance |

## How This Book Verifies Its Code

Every program in this book was run in Visual FoxPro 9, and the output you see was printed by that run. Most books about visual development show screenshots of designers. We take a different route, for two reasons. First, a screenshot cannot prove that code works; printed output can. Second, the habit we want you to build is the habit of exercising objects directly from code.

The technique is simple. Create the object with `CREATEOBJECT()`, call its methods, and print the results with `?`. This works for non-visual classes, obviously, but it also works for forms: a form created with `CREATEOBJECT()` runs its `Load` and `Init` events, builds its controls, and binds them to data, all without appearing on screen. You can then set a control's `Value`, call `loForm.cmdSave.Click()`, and print what happened. By the end of Part IV we will open a patient form, change Marisol's first name to "Marisol Ann", close the form through the form manager, and print the saved value from the table, all without a single window appearing.

A typical test program in this book looks like this:

C:\OakDesk\tests\try_invoice.prg

```
SET PATH TO C:\OakDesk\classes, C:\OakDesk\data
SET PROCEDURE TO oakbase, oakdata, oakpricing, oakbiz, oakapp
PUBLIC goApp
goApp = CREATEOBJECT("oakApplication")
? goApp.Startup("C:\OakDesk\data")

LOCAL loInv, loTotals, loPat
loInv = CREATEOBJECT("InvoiceBiz")
loTotals = loInv.Calculate(2)              && George, appointment 2
? loTotals.ySubtotal, loTotals.yInsAmt, loTotals.yDiscAmt, loTotals.yTotal

loPat = CREATEOBJECT("PatientBiz")
? loPat.Load("P00002"), loPat.FullName(), loPat.AgeOn({^2026-03-02})
? loPat.Load("P09999"), loPat.cLastError
goApp.Shutdown()
RELEASE goApp
```

Output

```
.T.
330.0000   165.0000   16.5000   148.5000
.T. George Whitfield         68
.F. No patients record P09999
```

This is a preview of where we are going, so do not worry about the classes yet. Notice the shape instead. The same calculation as `MAKEBILL` comes back as one object, `loTotals`, with four named properties, instead of four global variables. The patient object reports failure by returning `.F.` and explaining itself in `cLastError`, instead of displaying a message box. And the whole thing runs from a PRG, which means it can run as an automated test. That is the test Renata's complaint deserved.

Notice also how VFP prints Currency values: `?` shows four decimal places and no dollar sign, so $148.50 appears as `148.5000`. In prose we write money the way people read it, as $148.50; in output we show what VFP actually printed. Dates appear in US format with the century, such as `03/02/2026`.

> **WARNING**
>
> Never put anything that waits for a user into a test program: `MESSAGEBOX()`, `WAIT WINDOW` without `NOWAIT` or `TIMEOUT`, `BROWSE`, `READ EVENTS`, a form shown with `Show(1)`, or `QUIT`. When tests run unattended, a single modal dialog stops the whole run. This is one more reason OakDesk 2 classes report problems through return values and `cLastError` rather than by talking to the user.

If you want to run the programs yourself, use a lab folder kept apart from the real application. Throughout the book that folder is `C:\OakDeskLab\`: the chapter test programs live there (captions such as `C:\OakDeskLab\t09a.prg`), and the sample data lives in `C:\OakDeskLab\data`, built with `DO`

`setup WITH "C:\OakDeskLab\data"`. Any other empty folder works just as well if you adjust the paths. Rebuild the data whenever you want to start over; the worked numbers in each chapter assume fresh data unless the chapter says otherwise.

## Your First Class: oakCustom

Every non-visual class in OakDesk 2 descends from one root class, `oakCustom`, which is itself based on the VFP base class `Custom`. Why not base business objects on `Custom` directly? Because the base class cannot be changed, and sooner or later every framework needs behavior that all its objects share. With a root class of our own, adding that behavior later means editing one class, and every descendant inherits it.

The first shared behavior is a consistent way to report failure. OakDesk 2 adopts a simple contract: **a method that can fail returns `.F.` (or 0, or `.NULL.`) and stores a readable message in `cLastError`. It never shows user interface.** The caller decides what to do with the message: a form shows it, a test asserts on it, a batch job logs it. Here is the first version of `oakCustom`.

C:\OakDesk\classes\oakbase.prg (first version)

```
*-- oakbase.prg  --  OakDesk framework base classes.
*-- Load with:  SET PROCEDURE TO oakbase ADDITIVE

DEFINE CLASS oakCustom AS Custom
   cLastError = ""

   PROCEDURE SetError(tcMessage)
      *-- Stores the message and returns .F. so callers can write
      *-- RETURN This.SetError("...")
      This.cLastError = tcMessage
      RETURN .F.
   ENDPROC

   PROCEDURE ClearError()
      This.cLastError = ""
      RETURN .T.
   ENDPROC

   PROCEDURE HasError()
      RETURN NOT EMPTY(This.cLastError)
   ENDPROC
ENDDEFINE
```

The class has one property and three methods. `cLastError` starts as an empty string; the prefix `c` says it holds characters. `SetError()` stores a message and returns `.F.`, which allows the idiom `RETURN This.SetError("Unknown patient P09999")`: one line that records the reason and reports failure at the same time. `ClearError()` resets the message at the start of an operation, and `HasError()` answers whether a message is present.