Engineering Guide

Architecture, Source Units, Dependencies, Formats, Runtime Contracts, Tests, and Release Controls

Version 1.0

Last changed: September 1, 2026

Windows • Delphi VCL and FireMonkey • Win32 and Win64

Table of Contents

1. Guide Purpose, Plain-Language Architecture, and Vocabulary

This guide explains how Delphi App Translation Studio is built and why its parts are separated. It is for Delphi developers who maintain the Studio or the runtime components used by translated VCL and FireMonkey applications.

Begin with this chapter even if you already know Delphi. It defines the product's vocabulary and gives one simple model of the whole system. Later chapters add implementation detail. Chapters 23 through 27 are reference chapters for individual source files, packages, schemas, forms, and tools.

1.1 Engineering Boundary: Localization Is a Development Workflow

Localization happens while a Delphi source project is under active development. It is not post-processing applied to a finished EXE.

Developers can rescan after source changes and inspect the result inside the real application. They can also make deliberate wording or layout adjustments when an application needs them.

Automation has a defined boundary. Stable keys preserve identity. Catalog merge preserves unchanged work. Provider batching limits network requests. Structural validation protects exported packs. Baseline restoration makes language changes reversible.

Those mechanisms cannot infer every sentence assembled from live data. They also cannot make every terminology, layout, or bidirectional-text decision for the application owner.

Release review must therefore cover dynamic text, specialized terms, custom controls, third-party controls, protected identifiers, right-to-left text, mixed-direction paragraphs, clipping, wrapping, and repeated language changes.

A small and simple project may need almost no manual adjustment. The architecture must still leave a clear path for developer review and correction.

The Studio’s engineering objective is to perform the safe translation work automatically and as accurately as possible, preserve application behavior and source ownership, and provide precise evidence for anything that still requires developer judgment.

Term	Definition	Ownership consequence
ApplicationId	Stable exact identity shared by selected Delphi project, workspace, catalog, pack, runtime manager, and preference.	Never derive it from a translated caption or arbitrary folder alias.

Term	Definition	Ownership consequence
Development catalog	Full editable translation model containing stable keys, source/target text, status, origin, context, locale facts, checksums, scan provenance, and review data.	Studio-owned; not deployed directly as the compact runtime contract.
Canonical source pack	Validated runtime pack containing the authored source language.	Required for switch-back, compatibility comparison, and safe fallback.
Runtime pack	Compact validated JSON used by the offline target application.	Must match ApplicationId, framework, schema/version, locale, checksum, and source-pack contract.
Stable key	Deterministic identity such as form.component.property or unit.resourcestring.	Preserves translation across scans when ownership and source remain compatible.
Semantic contract	Explicit application-owned key for dynamic text, templates, HTML, or other content not represented by one designer property.	Allows safe scanning/application without translating arbitrary live data.
Manager	One nonvisual VCL or FMX component that owns runtime discovery, selected locale, lifecycle, events, errors, and pack application.	Exactly one primary manager normally supervises the application.
Selector	Designer-visible VCL/FMX combo box connected to a manager.	Lists only admitted packs and delegates language switching to the manager.
Applicator	Framework-specific walker that applies text, layout, direction, and restoration to a VCL or FMX object tree.	Contains framework behavior; provider and catalog code remain framework-neutral.

Term	Definition	Ownership consequence
Designer baseline	Snapshot of source-language text, geometry, alignment, direction, and relevant control state.	Restored before every language transition so changes are reversible and do not accumulate.
Localization Review	Operator window and artifact package for linguistic findings, glossary, suggestions, layout proposals, and decisions.	Human decisions remain separate from structural validation.
ComponentSource	Complete current subset of runtime/component PAS units generated for one target framework.	Must be copied as one compatible set; partial copying creates interface drift.
Workspace	Per-ApplicationId Studio state under %LOCALAPPDATA%\DelphiAppTranslationStudio\Workspaces.	Keeps development data outside target source and executable folders.
Pack admission	Runtime validation performed before a pack appears as selectable.	Invalid, wrong-app, wrong-framework, wrong-version, or incompatible packs never become active.

1.2 The system in plain language

The product has two separate halves. The Studio is the developer tool. It scans source, contacts a translation service, supports review, and creates files. The runtime is the code placed in the target Delphi application. It works offline and reads only validated local language packs.

A development catalog is the Studio's working record. It contains much more than translated words: it also records identity, context, review state, source changes, and checksums. A runtime pack is the smaller file produced from that catalog for the finished application.

The target project remains the design authority. Forms stay editable in the Delphi designer. The runtime manager changes language while the application runs, but it must preserve the form's original text, layout, control state, and direction so switching back is reliable.

Part	Plain-language responsibility	Must not do
Project detector and scanner	Find the selected Delphi project and collect text that is safe to translate.	Run the target source or guess that arbitrary data is interface text.
Catalog and workspace	Keep the durable development record outside the target source tree.	Replace a valid file with a partial or corrupt write.
Provider pipeline	Send eligible unresolved text to DeepL or Google and validate the response.	Store a provider key in source, packs, logs, or catalogs.
Localization Review	Show wording and layout concerns that need developer judgment.	Silently approve language quality or move developer-owned controls.
Pack builder	Create one compatible source pack and the translated runtime packs.	Export a structurally invalid or mismatched catalog.
Runtime manager and applicator	Load admitted packs and update open or later forms without losing live state.	Contact a provider, rescan source, or translate the previous translation.
Component and integration kit	Give a VCL or FMX project one compatible set of source units and deployment files.	Mix old and new dependency units or rewrite target source without authorization.

1.3 Follow one caption through the system

1. A caption is saved in a DFM or FMX form. The scanner reads the saved form and assigns a stable key such as form.component.property.
2. Catalog merge compares that key and source text with the prior catalog. Unchanged reviewed work is preserved. New or changed text is marked for attention.
3. The terminology and provider pipeline prepares eligible text, protects placeholders, sends a bounded request, and records the returned text as a machine draft.
4. Localization Review lets the developer inspect wording and any layout finding. Structural validation then checks identity, required text, placeholders, and pack compatibility.
5. The pack builder writes the canonical source pack and target-language pack. Deployment places compatible packs under the executable's Localization\Languages folder.

6. At run time, the manager restores the designer baseline and asks the VCL or FMX applicator to apply the selected pack. A later switch repeats the process from the baseline, not from the prior translation.

1.4 How to read file and dependency entries

Reference label	What it means
Public surface	Types, classes, interfaces, constants, and routines that another unit can use.
Direct DAT dependency	A DAT unit named directly in this unit's uses clause. It does not include dependencies used only through another unit.
Direct production consumer	A production DAT unit that names this unit directly in its own uses clause.
Named test consumer	A test program that imports the unit directly. Other tests may still exercise it indirectly.
ComponentSource closure	The complete group of shared and framework-specific PAS files needed by a target project. It must be refreshed as one set.
Change and validation risk	The minimum subsystem tests and builds required after a change. Shared behavior normally requires both VCL and FMX verification.

A practical reading order. To understand behavior, read Chapters 2 through 22 in order. To change one unit, first read its Chapter 23 entry, then the architecture chapter for its layer, then the tests and consumers named by that entry.

2. Product Scope, Non-Negotiable Invariants, and Trust Boundaries

What rules may never be bypassed? This chapter defines the safety boundary. The Studio may read selected source and create its own artifacts. The recommended workflow does not rewrite the target project's Pascal, form, DPR, or DPROJ files.

- Target Pascal, DFM, FMX, DPR, and DPROJ remain read-only in the recommended Wizard/component workflow.
- Forms and published component properties remain designer-authored and editable in Object Inspector; runtime-only UI construction is not the normal integration mechanism.

- The development Studio may use DeepL or Google online; the deployed VCL/FMX application is offline and contains no provider credential.
- A language transition is reversible: restore the designer/source baseline first, then apply the selected pack once. Never translate the previous translation.
- Live application data, identifiers, object names, file paths, API names, numeric values, selection/focus, and connection state are not language-pack text unless an explicit semantic contract says otherwise.
- VCL and FMX share catalogs, packs, provider, validation, and manager core but use separate applicators, lifecycle hooks, direction rules, text measurement, and design packages.
- Stable keys and source text preserve unchanged work. Provider calls are limited to eligible unresolved entries; reviewed and approved work is never silently overwritten.
- All persistence that can affect catalogs, packs, preferences, settings, glossaries, deployment, or layout decisions is atomic and recoverable.
- RTL changes paragraph/control direction and alignment while preserving protected LTR tokens and explicit application-owned layout.
- No project-specific V5/V6 or named-application exception belongs in universal scanner, runtime, or layout code.

Trust boundary. The Studio trusts only explicitly selected project source and validated provider responses; the runtime trusts only admitted local packs. Neither side treats arbitrary JSON, user data, provider text, or an unverified executable folder as safe input.

3. Repository, Build, and Distribution Layout

Where does each kind of file belong? This chapter maps repository folders to responsibilities. Use it before adding a unit or generated artifact so production, test, documentation, and output files do not become mixed.

Folder	Engineering ownership	What must not be mixed into it
source\core	framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction	FMX/VCL controls, HTTP UI, or target-specific exceptions.
source\scan	project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging	Runtime mutation or provider secrets.

Folder	Engineering ownership	What must not be mixed into it
source\core	framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction	FMX/VCL controls, HTTP UI, or target-specific exceptions.
source\provider	DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation	Target runtime or design-time registration.
source\review	localization findings, layout proposals, text measurement, and application-owned string review	Silent application of unapproved geometry.
source\validation	structural catalog checks that protect runtime export	Linguistic approval decisions.
source\runtime	offline pack loading, preference, translation application, layout restoration, templates, and splash handling	Provider networking or Studio forms.
source\components	designer-visible language-manager components, lifecycle adapters, and language selectors	Studio workflow or catalog editing.
source\design	RAD Studio Tool Palette registration and design-time package exposure	Code required by deployed applications.
source\integration	component-kit generation, controlled Delphi project integration, build, and deployment	Unbounded source rewriting.
source\studio	the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation	Core algorithms that tests/runtime need without FMX UI.

Folder	Engineering ownership	What must not be mixed into it
source\core	framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction	FMX/VCL controls, HTTP UI, or target-specific exceptions.
packages	Runtime and design package projects.	Application-specific compiled BPL copies.
bin / dcu	Platform/configuration output.	Authoritative source or hand-edited files.
docs\guides / docs\pdf	Editable DOCX guides and print PDFs.	Runtime dependencies or credentials.
export	Generated review, pack, kit, and source-distribution artifacts.	Developer secrets or canonical source history.
tools / tools\tests	Build, verification, docs, contracts, test executables, and fixtures.	Production code reachable only from tests.

DelphiAppTranslationStudio.dpr/dproj are the FMX Studio entry point and project metadata. Output is partitioned by platform/configuration. Package projects separate framework-neutral runtime, VCL runtime, FMX runtime, and IDE-only VCL/FMX registration.

Source distributions must contain complete source/package/schema/tool/document sets and must exclude secrets, Local AppData workspaces, provider keys, transient test workspaces, DCUs, EXEs, and unreviewed proprietary catalogs unless explicitly intended.

4. End-to-End Architecture and Ownership

> **How does text move from source to a running translation?** This chapter follows the pipeline from project detection through scanning, translation, review, validation, export, deployment, and run-time application. Each stage has one owner and one output contract.

Stage	Input	Owner	Output / next boundary
Detect	Selected DPR/DPROJ	DAT.Core.ProjectDetection	Framework, ApplicationId, targets, forms, units, output metadata
Scan	Saved DFM/FMX/PAS and authorized resources	DAT.Scan.*	TProjectScanResult with items, diagnostics, snapshots, and counts
Merge	Scan result plus prior catalog	DAT.Scan.CatalogMerge	Preserved/new/changed/ obsolete development catalog
Resolve	Entries, domain profile, glossary, memory, terminology	DAT.Core.* and DAT.Provider.*	Eligible provider inputs and deterministic repairs
Translate	Bounded text/context batches and credential	DAT.Provider.Client / DAT.Core.AITranslation	Machine drafts with status/origin and diagnostics
Review	Translated catalog and source geometry	DAT.Review.* / LocalizationReview form	Findings, glossary, suggestions, layout decisions, HTML/JSON package
Validate	Complete development catalog	DAT.Validation.Catalog	Blocking errors and review warnings
Export	Validated catalog	DAT.Core.RuntimePack	Canonical source pack and translated runtime pack
Integrate	Project profile and packs	DAT.Integration.ComponentPackage	ComponentSource, manifest, README, deployment script, completion report

Stage	Input	Owner	Output / next boundary
Detect	Selected DPR/DPROJ	DAT.Core.ProjectDetection	Framework, ApplicationId, targets, forms, units, output metadata
Admit	Pack files beside target EXE	DAT.Runtime.LanguagePack / Manager	Selectable compatible language descriptors
Apply	Selected pack and open/later forms	DAT.Runtime.VCL/FMX plus components	Immediate reversible translated UI with preserved state

Ownership rule. Each stage validates the contract it receives and emits a narrower explicit contract. Later stages must not reach backward into a UI form or provider response to reconstruct missing state.

5. Core Domain Model and Catalog Lifecycle

What information does one translation record carry? The catalog stores identity, source and target text, context, status, origin, and review facts. Status describes workflow state; origin records where wording came from. They are not interchangeable.

DAT.Core.Types is the vocabulary root. TProjectProfile describes target identity/framework/files/targets. TLocaleProfile describes locale facts. TTranslationEntry carries stable identity, source/target text, status, origin, context, ownership, checksums, and review facts. TTranslationCatalog owns project/locale metadata and its entries.

Status is workflow state, not provenance. Origin records how target text arrived; Reviewed/Approved record human decisions. A machine result must not become Approved merely because structural validation passes.

Catalog event	Required behavior	Forbidden behavior
Same key, same source	Preserve translation, origin, review, approval, and applicable decisions.	Retranslate or reset state without explicit operator action.
Same key, changed source	Retain prior target for reference and mark source-changed/attention-required.	Treat old target as current approved wording.

Catalog event	Required behavior	Forbidden behavior
New key	Add with context, ownership, source snapshot, and unresolved status.	Borrow another key's approval automatically.
Missing key	Retain as obsolete for audit/recovery.	Delete silently.
Equivalent duplicate	Collapse occurrence while retaining provenance/count.	Create conflicting duplicate stable keys.
Semantic recovery	Recreate known application-owned dynamic contract deterministically.	Scan arbitrary runtime values as translatable source.

6. Workspace, Persistence, and Recovery

> **Where is development state stored, and how is it protected?** Project workspaces live under Local AppData, outside the target project. Important JSON and preference files use validated atomic replacement so an interruption cannot silently replace a good file with a partial one.

TTranslationWorkspace roots project state at %LOCALAPPDATA% \DelphiAppTranslationStudio\Workspaces\<ApplicationId>. Development, Languages, Glossaries, and Deployment separate editable data, deployed-format packs, terminology, and destination settings.

TAtomicTextFile writes a candidate, validates it, preserves a prior known-good version, and replaces the active file atomically. Readers can recover .previous, quarantine corrupt input, and avoid converting a process interruption into an empty or half-written catalog.

Artifact	Writer	Validation before commit	Recovery behavior
Development catalog	TCatalogJson	Schema/domain validation and parse round trip.	Previous catalog retained; corrupt input quarantined.
Runtime pack	TRuntimePackBuilder	Required metadata, checksums, locale, entries, layout contract.	Invalid pack never replaces the prior admitted copy.
Provider settings	TProviderSettings	Known provider/plan, bounded timeout/batch, no secret field.	Defaults and previous valid settings remain available.

Artifact	Writer	Validation before commit	Recovery behavior
Language preference	TLanguagePreference	Admitted locale code.	Invalid preference falls back to source/system policy.
Glossary	TProjectGlossary	Term/source/target and match-rule validation.	Last valid glossary can be restored.
Layout overrides	TLayoutOverrides	Application/locale/form/control identity and numeric bounds.	Invalid overrides are ignored/quarantined rather than moving controls unpredictably.

7. Project Detection and Canonical Scanning

> **What exactly does a scan do?** Detection finds the project closure. The scanner reads saved form resources and approved Pascal text. It produces stable records and counts; it does not run the target application or translate arbitrary data.

TProjectDetector establishes the source closure before scanning. TProjectScanner orchestrates form and Pascal scanners, rules, context, domain profile, quality analysis, progress/cancellation, source snapshots, and the final result.

TTextFormScanner reads text DFM/FMX structure and eligible string properties. TPascalResourceStringScanner reads resourcestring declarations and approved runtime assignments. TScanRuleSet is the universal policy boundary; it must not contain named-project or version-specific exceptions.

Count	Meaning	Why it is separate
Raw observations	Direct form/Pascal/resource observations.	Shows scanner input volume before merge behavior.
Recovered semantic contracts	Explicit dynamic/application-owned strings added by stable rules.	Dynamic coverage is auditable rather than hidden in code.

Count	Meaning	Why it is separate
Equivalent duplicates collapsed	Repeated equivalent ownership represented once.	Prevents inflated translation counts without losing occurrence evidence.
Unique catalog entries	Final stable-key records after recovery/collapse.	This is the canonical Wizard/Maintenance headline.

Scan snapshot invariant. Final processing must reject a source/form save that occurred after the accepted scan. Exporting a stale snapshot can omit new text even when every later stage succeeds.

8. Context, Terminology, Translation Memory, and Glossary

How does the Studio choose the best wording it already knows? Context explains meaning. Terminology protects required words. Translation memory reuses reviewed work. A glossary records project-specific choices. These steps reduce provider ambiguity before new machine translation is requested.

Context and terminology are layered because provider text alone is insufficient for short interface strings. TScanContextAnalyzer records UI role, semantic concept, description, confidence, and surrounding evidence. TDomainProfiler builds project vocabulary and senses. TProjectGlossary and TSharedDictionary carry approved local and shared terms. TTranslationMemory reuses reviewed prior segments with provenance.

Resolution order must preserve the most specific trusted evidence. Project glossary outranks shared/general terminology; reviewed contextual memory outranks an unreviewed machine draft; protected tokens and placeholders are never rewritten as terminology.

Layer	Strength	Allowed effect
Protected token/placeholder	Absolute structural constraint	Preserve exactly and validate restoration.
Project glossary	Application-specific approved terminology	Supply or repair the target term without touching protected Reviewed/Approved exceptions.

Layer	Strength	Allowed effect
Contextual translation memory	Trusted prior segment in compatible context	Suggest or reuse according to explicit confidence/status rules.
Shared dictionary/calendar terms	Vetted cross-project UI vocabulary	Repair common UI/calendar wording where locale support exists.
Provider context	Advisory description supplied with DeepL-capable requests	Improve disambiguation but never override structural validation.
Provider draft	Unreviewed external result	Populate eligible unresolved target text as machine origin.

8.1 Authoritative glossary ownership and path

TTranslationWorkspace.GlossaryFileName derives the authoritative path as %LOCALAPPDATA%\DelphiAppTranslationStudio\Workspaces\<ApplicationId>\Glossaries\<ApplicationId>.<locale>.glossary.json. The filename is built from the detected project identity and normalized target locale. The source and target language plus ApplicationId are also stored inside the JSON, so moving a file into another workspace does not silently change its identity.

TProjectGlossary owns a list of TProjectGlossaryTerm records. Each record carries SourceText, TargetText, optional SemanticConcept, optional ContextKind, optional DeveloperNote, CaseSensitive, and Approved. SaveToFile serializes schema version 1 and delegates to TAtomicTextFile, which validates the candidate before atomic replacement and preserves normal recovery artifacts.

8.2 Matching and catalog-application rules

- Only Approved terms with nonblank target text are eligible.
- Case-sensitive terms require exact source text. Other terms compare trimmed source text without case sensitivity.
- A nonblank ContextKind is a real filter and must match the entry. SemanticConcept is a preference, not a filter: an exact concept match wins, an untagged term is next, and a differently tagged but otherwise matching term remains a fallback.
- ApplyToCatalog never overwrites Reviewed or Approved entries. Matching unresolved entries receive the preferred target, Project Glossary origin, high confidence, a review note, and Machine Translated status so human approval remains a separate decision.

- TProjectGlossary.ApplyToCatalog itself changes the supplied in-memory development catalog only. It does not export, deploy, call a provider, rescan, or edit target source. TGlossaryPublisher deliberately composes that narrow operation with validation, atomic persistence, runtime-pack export, and pack deployment.

8.3 Standalone Maintenance Studio glossary state

MainForm owns FProjectGlossary, FGlossaryFileName, FGlossaryDirty, and FUpdatingGlossaryLanguage. Opening the Glossary page derives the selected language file and loads it, or creates a new in-memory model when no file exists. This is intentionally independent of the Setup Wizard and independent of whether a development catalog has already been created.

The designer-owned page contains the language selector, exact file-path label, term list, source/target/concept/note editors, case-sensitive and approved check boxes, and New, Add / Update Term, Delete, Cancel Edits, Save Glossary, and Apply and Deploy buttons. FGlossaryDirty becomes true after add/update/delete. RefreshGlossary rebuilds the list and reports whether the model is new, saved, or pending.

ConfirmGlossaryChanges protects three transitions: changing the glossary language, opening another Delphi project, and closing Maintenance Studio. Yes saves atomically, No discards the in-memory changes, and Cancel vetoes the transition. Cancel Edits is narrower: after confirmation it reloads the selected glossary and remains on the page.

Apply and Deploy is enabled whenever a project glossary is loaded. The handler saves dirty glossary work first and calls TGlossaryPublisher. The publisher locates the matching workspace catalog itself, so the Translate page does not have to be open. If that catalog is currently displayed, MainForm reloads it after publication so the editor cannot retain a stale in-memory copy.

8.4 Standalone glossary publication transaction

TGlossaryPublisher.Publish is the terminology-only publication boundary. Preflight verifies project identity, glossary identity and approved terms, the matching development catalog, framework and locale compatibility, the managed dependency Languages folder, and the syntax of remembered deployment destinations before replacing any artifact.

After ApplyToCatalog, TCatalogValidator must report no blocking errors. TRuntimePackBuilder serializes the candidate before any write, then TCatalogJson and TRuntimePackBuilder use their atomic persistence contracts to refresh the development catalog, canonical workspace pack, and managed dependency pack. SHA-256 equality proves that the canonical and dependency copies are identical before deployment begins.

TTargetBuildDeployer.ExistingBuildOutputDirectories resolves every existing supported Debug/Release output from DCC_ExeOutput patterns and conventional output trees. The publisher then atomically

replaces Localization\Languages from the complete managed dependency pack directory at each deduplicated existing output and remembered destination. Missing destinations are reported and skipped; deployment failures are counted and reported without hiding successful canonical publication. No compiler is launched and no EXE, DPROJ, DPR, PAS, DFM, or FMX file is replaced.

9. Provider Pipeline and Credential Security

> **What crosses the network?** Only eligible source text and safe context go to the selected provider. Credentials stay in Windows Credential Manager or session memory. Placeholders are protected, requests are bounded, and malformed responses are rejected.

TProviderSettings stores only provider, DeepL plan, remember policy, timeout, and batch size. TProviderCredentialStore stores the selected secret in Windows Credential Manager or session memory. TTranslationProviderClient builds and posts bounded HTTPS requests and maps provider-specific errors into ETranslationProviderError.

TContextBatching groups compatible contexts and limits request size. TPlaceholderProtection replaces immutable tokens before network transmission and requires exact recovery. TProviderRetry retries only transient failures with bounded backoff. TProviderLanguageCodes prevents sending a canonical locale code that a provider does not accept.

Failure class	Examples	Engineering response
Permanent authentication/configuration	401, 403, wrong DeepL plan, disabled Google API.	Fail promptly with actionable account/endpoint guidance; do not loop retries.
Quota/rate	429.	Honor bounded retry policy and surface quota context.
Transient service/network	5xx, temporary transport failure.	Retry with bounded backoff and cancellation.
Timeout	Request exceeds configured seconds.	Cancel/abort the operation and return UI control; never leave the Wizard uncloseable.
Cancellation	Operator stops final processing or closes within allowed boundary.	Raise controlled cancellation, preserve completed atomic artifacts, and report STOPPED safely.

Failure class	Examples	Engineering response
Structural response error	Wrong response count, malformed JSON, missing placeholders.	Reject the batch; never write partial unvalidated translations as complete.

10. Localization Review, Measurement, Layout, and Hyphenation

> **Who decides whether wording and layout are acceptable?** Automated review finds likely problems and can propose conservative layout changes. The developer decides. Structural validation can prove that a pack is safe to load, but it cannot prove that a sentence is the best human translation.

TLocalizationReviewer combines source geometry, code-geometry ownership, translated measurements, text roles, and language behavior into findings and proposals. ITextMeasurer isolates metric acquisition; FMX and GDI implementations provide framework-appropriate measurements.

Layout changes are stored as per-language rules and applied only after baseline restoration. A proposal is not application state until explicitly accepted. Code-positioned controls and complex layout relationships are excluded from automatic movement.

Fitting order	Engineering rationale	Stop condition
Natural space wrapping	Preserves words, typography, and shared column geometry.	Text fits without clipping or harming neighbors.
Intelligent hyphenation	Breaks a long unspaced word at language-aware points.	Readable result fits and remains linguistically acceptable.
Modest font reduction	Uses remaining control capacity without expanding shared layout.	Do not cross the readable minimum or create inconsistent hierarchy.
Conservative size/layout proposal	Adds locale-specific room when ownership and neighbors permit.	Reject if it moves code-owned controls or introduces collision.
Developer redesign	Handles complex collision, grid, graph, or container constraints.	Required when automated proposals cannot preserve the design.

11. Validation and Runtime Pack Export

> **When may a runtime pack be created?** Only a structurally valid catalog can be exported. Source and target packs must agree on application identity, framework, locale relationship, schema, and checksum contract.

TCatalogValidator separates blocking runtime safety from linguistic state. Required metadata, duplicate keys, source changes, placeholder/accelerator integrity, runtime wiring, and eligible target text are validated; Reviewed and Approved remain explicit human states.

TRuntimePackBuilder exports only after structural validation. It creates a compact pack with identity, framework, version/schema, source/target locale, checksum linkage, texts, templates, source maps needed for reversibility, and accepted layout rules.

- An error blocks export.
- A warning requires developer judgment and may describe linguistic or runtime-readiness risk.
- Information explains a condition that is safe but relevant.
- Canonical source and every target pack in one deployment must share compatible application identity and catalog/source contract.

12. Offline Runtime Manager and Pack Admission

> **Why does a pack appear—or not appear—in the selector?** The runtime admits a pack only after compatibility checks pass. The canonical source pack is the fallback anchor and is required for reliable switching back to the authored language.

TRuntimeLanguagePack parses one JSON file and exposes text, templates, source maps, locale, and layout rules. TTranslationRuntime discovers files, validates pack headers/checksums/framework/application identity, admits compatible packs, loads source and selected target, updates format settings, and serves translation lookups.

TLanguagePreference remembers an admitted locale. Startup loads the source pack as the safety anchor, then attempts the stored or system language according to policy. A stale preference cannot force an invalid pack into the process.

Admission check	Reason
ApplicationId	Prevents one product's pack from appearing in another.
Framework	Prevents incompatible layout/runtime assumptions between VCL and FMX.

Admission check	Reason
Schema/version	Prevents a runtime from misreading a newer/older contract.
Locale/source locale	Prevents mislabeled files and invalid fallback relationships.
Checksum/source compatibility	Prevents a translated pack from pairing with a different source catalog.
Required data/layout structure	Prevents incomplete pack activation and partial translation.

13. VCL Runtime Application and Lifecycle

> **What is special about VCL?** The VCL manager follows Windows form and handle lifecycle. Its applicator restores the designer baseline, applies the selected pack, and preserves focus, selection, lists, grids, menus, and other live state.

TDATVCLLanguageManager connects TTranslationRuntime to VCL application lifecycle. Idle inspection, modal boundaries, active-form change, notifications, and explicit ApplyToForm cover existing and later forms without replacing application event handlers destructively.

TVCLTranslationApplicator snapshots source text, geometry, alignment, direction, and writable state; restores the baseline; applies stable-key text/templates/layout; handles menus, dialogs, status panels, lists/grids, wrapping, and RTL; then restores focus, selection, and other live state.

> **VCL handle rule.** Do not force unnecessary handle recreation or overwrite window state merely to change language. Native controls and Windows messages can regenerate captions, so VCL template/caption interception reapplies only the owned text contract.

14. FMX Runtime Application and Lifecycle

> **What is special about FireMonkey?** The FMX manager handles styled controls, tabs, scrolling, browsers, grids, and later repainting. Template repair is important because an FMX style can recreate visible text after the first assignment.

TDATFMXLanguageManager connects TTranslationRuntime to FMX form lifecycle and styled-control behavior. It handles open/later forms, tabs, scroll content bounds, browsers, grids, menus, dynamic refresh timers, and lifecycle-safe application without relying on one form-specific event.

TFMXTranslationApplicator snapshots/restores the designer baseline, applies text and layout to the FMX object tree, preserves state/order, and coordinates direction. Template rewrite is essential because FMX styles can recreate visible text after a property was translated.

> **FMX styled-control rule.** A language switch is not complete merely because Text changed once. Styled controls, tab activation, browser document load, and scroll-layout recalculation can repaint later; lifecycle contracts must reapply only when the owned text is regenerated.

15. Components, Design Packages, and Designer Contract

> **Why are there separate runtime and design packages?** Runtime packages contain code an application may ship. Win32 design packages contain RAD Studio registration code. Keeping them separate prevents deployed applications from depending on IDE-only units.

DAT.Components.Core exposes published properties/events and the framework-neutral manager contract. DAT.Components.VCL/FMX implement framework lifecycle and applicator bridges. The LanguageSelector units provide connected designer combo boxes. Registration units expose those components only to RAD Studio design packages.

The design packages are Win32 IDE packages because RAD Studio is a Win32 design host even when the target application also builds Win64. Runtime packages are separated so a deployed application never links IDE registration code.

Generation copies the exact three-file framework bundle into <Kit>\DesignPackages\Win32\Release. VCL receives CoreRuntime, VCLRuntime, and VCLDesign; FMX receives CoreRuntime, FMXRuntime, and FMXDesign. The developer selects only the design BPL through Component > Install Packages > Add, but the required runtime BPLs must remain beside it. Automatic IDE package registration is intentionally outside the safety boundary.

Package	Role	Contains / requires
DATLanguageManagerCoreRuntime	Framework-neutral runtime package.	Core runtime/manager/component contract used by VCL and FMX runtime packages.
DATLanguageManagerVCLRuntime	VCL runtime package.	VCL applicator, VCL manager, VCL selector, and framework-specific helpers.

Package	Role	Contains / requires
DATLanguageManagerFMXRuntime	FMX runtime package.	FMX applicator, FMX manager, FMX selector, and framework-specific helpers.
DATLanguageManagerVCLDesign	VCL IDE design package.	Requires VCL runtime package and contains VCL Register.
DATLanguageManagerFMXDesign	FMX IDE design package.	Requires FMX runtime package and contains FMX Register.

16. ComponentSource and Target dependencies

> **Why must ComponentSource be copied as one set?** The generated source closure contains matching shared and framework-specific units. Copying only files named by the first compiler error can combine incompatible interfaces from different versions.

ComponentSource is a generated source-closure distribution for one target framework. The package generator computes the exact shared and framework-specific set and copies all of it into the kit. The target compiler must see one internally consistent version of that set.

The target-local dependency is <Target>\dependencies\DelphiAppTranslation. PrepareProjectDependencies creates or replaces this managed folder automatically after snapshotting any prior dependency. source holds the complete PAS closure; deployment\Languages holds the canonical pack source; license and manifests record provenance, integrity, the exact developer-owned Search Path, and the read-only project-file boundary.

Layer	Units	Why inseparable
Shared persistence/diagnostics	DAT.Core.AtomicFile; DAT.Core.Diagnostics	Preference/layout/runtime parsing depends on the same atomic/error contract.
Shared runtime	DAT.Runtime.LanguagePack; Preference; Manager; LayoutOverrides; SplashTranslation; TemplateRewrite	Pack model, active runtime, restoration, early startup, and later template repair share interfaces.

Layer	Units	Why inseparable
Shared component API	DAT.Components.Core	VCL/FMX managers and application code compile against its published properties/events/functions.
VCL layer	DAT.Runtime.VCL; SplashTranslation.VCL; TemplateRewrite.VCL; DAT.Components.VCL; VCL.LanguageSelector	All VCL lifecycle, applicator, splash, selector, and rewrite contracts must match.
FMX layer	DAT.Runtime.FMX; SplashTranslation.FMX; TemplateRewrite.FMX; DAT.Components.FMX; FMX.LanguageSelector	All FMX lifecycle, styles, selector, and rewrite contracts must match.

Partial-copy failure mode. Copying only the units named by the first compiler errors can compile against older interfaces elsewhere on Search path, creating ambiguous unit resolution or runtime behavior. Refresh the complete generated set together.

17. Integration, Build, Deployment, and Project Boundaries

What may integration change? The normal component workflow generates a kit and deploys packs without changing target source. Advanced source editing exists only behind explicit preview, authorization, backup, and restore boundaries.

TComponentIntegrationPackageGenerator.Generate produces a staged, verified kit. PrepareProjectDependencies is the shared Wizard/Maintenance transaction: it validates kit identity, creates a dated snapshot of any prior managed dependency, stages the replacement, verifies copied hashes, atomically promotes it, and verifies that the selected project-file hash did not change. It restores the previous dependency if promotion or post-promotion validation fails.

The Studio never writes DCC_UnitSearchPath or an Import element into the DPROJ. RunHiddenBuild uses /p:DCC_UnitSearchPath with <Target>\dependencies\DelphiAppTranslation\source for that MSBuild process only. BuildAndDeploy then atomically replaces the output Localization\Languages set directly, preventing obsolete locale files from surviving a Studio deployment.

TTargetBuildDeployer runs the baseline Win32 Release build even when no output folder existed, refreshes other supported configurations already in use, locates the configured executable output, and

performs an additional atomic/hash-validated deployment. ExistingBuildOutputDirectories is also the shared non-build discovery contract used by the Wizard and standalone glossary publication, preventing the two paths from drifting. Configured application destinations use the same atomic routine. TIntegrationPackageGenerator and TDelphiIntegrationSourceEditor remain advanced source-edit machinery and are not invoked by the component workflow.

16.1 Persistent IDE Search Path Contract

Exact RAD Studio Search Path $(PROJECTDIR)\dependencies\DelphiAppTranslation\source

1. Open Project > Options > Building > Delphi Compiler for the target project.
2. Select All configurations and All platforms.
3. Append the exact relative path above to Search path while preserving every existing entry. Use a semicolon as the delimiter when other entries are present.
4. Save the Project Options value once. Subsequent Studio dependency refreshes replace the contents of the same managed folder and do not require or authorize another path entry.

$(PROJECTDIR) resolves to the directory containing the target DPROJ. The entry therefore remains portable when the entire target project is moved or cloned. It must not resolve into the Studio repository or a replaceable export kit.

Ownership is deliberately split: the developer owns this normal RAD Studio setting; the Studio owns only the managed dependency folder. For Studio-initiated builds, RunHiddenBuild supplies the absolute dependency source directory through the process-local /p:DCC_UnitSearchPath MSBuild property. No persistent DPROJ mutation, import, managed block, or post-build event is used.

- The Project Options Search path contains one developer-added relative entry. The Studio never inserts, replaces, or duplicates that entry in the DPROJ.
- The dependency folder is Studio-owned as one versioned unit. Do not combine its files with older DAT copies beside the DPR or in a legacy DAT_Runtime folder.
- Pack deployment is an explicit Studio build/deployment operation; stale target JSON is removed before the current set is promoted. Ordinary IDE builds do not import or execute hidden Studio project logic.
- Unavailable optional removable/network destinations warn and skip; they must not fail an otherwise valid local build.
- No integration code writes target Pascal, DPR, DPROJ, DFM, or FMX files. Authorized writes are limited to Studio export/workspace, the target-managed dependency, dependency snapshots, build outputs, and configured destinations.
- The BPL boundary is intentionally manual: the Studio locates and bundles exact files but never registers or unregisters a package in the IDE.

18. Studio Forms, State Machines, and Self-Localization

> **Which form owns which workflow state?** MainForm owns Maintenance Studio. SetupWizard owns the guided state machine. LocalizationReview owns findings and decisions. Their FMX resources remain the editable design authority.

TfrmTranslationStudio is the landing and eight-page Maintenance state owner: Project, Scan, Translate, Glossary, Validation, Export, Integration, and Provider Settings. TfrmSetupWizard is an eight-step validation state machine whose downstream state is invalidated when project/language/provider inputs change. TfrmLocalizationReview is modal within final processing and returns saved decisions to a continuation. DAT.Studio.Translation owns the Studio's own interface-language runtime.

The forms are designer-authored FMX resources. Runtime code updates state, enablement, text, and data; it does not rebuild the form hierarchy that belongs in the FMX designer. Default buttons, Enter/Esc behavior, hidden irrelevant buttons, and asynchronous provider state are explicit UI contracts. MainForm.SetWorkflowStep is the single page-routing contract: Glossary is step 4, later pages are steps 5 through 8, and the designer rail positions and page visibility must remain synchronized.

btnMaintenanceCancel is a persistent designer-owned Cancel button while Maintenance Studio is active and has the FMX Cancel property, so Esc calls Close. FormCloseQuery closes immediately only when no scan/provider operation is active and ConfirmGlossaryChanges permits it. Otherwise it sets atomic cancellation flags, records close-after-operation intent, vetoes the current close, and lets the queued completion handler call Close again after the operation reaches a safe boundary. The second close then performs the glossary Save/Discard/Cancel decision if needed.

Form	Primary state	Critical transition
MainForm	Opened project, scan result, catalog, selected maintenance page, selected project/locale glossary and dirty state, provider/settings, validation/export/integration status.	Landing to Wizard or Maintenance; page changes protect pending glossary work; Cancel/Esc safely bounds active work; language refresh rebuilds current visible content immediately.
SetupWizard	Current/highest step, project/profile, destinations, locales, provider, scan/catalog, authorization, final-processing continuation.	Begin Final Processing disables navigation; Localization Review temporarily suspends and resumes the pipeline.

Form	Primary state	Critical transition
LocalizationReview	Findings, glossary, suggestions, proposals, decisions, output package.	Close saves accepted decisions and returns to Wizard validation/export.

19. Directionality, HTML, Dynamic Text, and Reversible Layout

> **How are direction, HTML, and changing text kept correct?** Every switch begins from the source baseline. Dynamic text and HTML use explicit translation keys and refresh contracts. RTL is applied at the correct control or paragraph boundary while protected identifiers remain LTR.

Text direction is pack locale metadata interpreted by the framework applicator. Direction and alignment are applied at the smallest correct paragraph/control ownership boundary; the whole interface is not blindly mirrored when the application contains protected LTR identifiers, paths, APIs, or numeric content.

Dynamic text uses stable semantic keys or source-map translation through DATTranslateDynamicText. HTML uses DATTranslateHtmlText and must rebuild the current page on language change. A language switch restores source/dynamic baseline before applying the new target, preventing Arabic text from surviving into Greek or another language.

Content class	Translation contract	Refresh contract
Designer property	Stable form.component.property key.	Applicator updates open form and lifecycle handles later forms.
Resourcestring	Stable unit.symbol key plus explicit application use.	Application code reads translated value at the intended message point.
Dynamic status/message	Semantic key/template or source-map contract.	Recompute immediately on language change without altering live data.
HTML/report	Canonical HTML/text templates and semantic keys; table grid owns header/body alignment.	Rebuild current document synchronously on switch; no previous-language DOM/text may remain.
Identifier/ path/API token	Protected LTR/nontranslatable token.	Preserve spelling and isolate direction inside RTL prose.

Content class	Translation contract	Refresh contract
Layout rule	Locale/form/component property decision.	Restore designer baseline, then apply selected locale rule only.

20. Diagnostics, Failure Containment, and Performance

> **How does the system fail without hanging or corrupting state?** Network, build, scan, save, and language-switch operations have explicit bounds and recovery behavior. Errors must return the interface to a usable state and must not expose secrets.

TDATDiagnostics and structured exceptions provide severity, operation, path, provider status, stable key, and recovery context. UI code must convert them into actionable status/progress text without exposing secrets. Background provider/build work must always return the form to a closable and internally consistent state.

Performance-sensitive language switching must use already admitted in-memory packs. It should not rescan the project, call a provider, reread every JSON file repeatedly, or rebuild hidden pages that have no active content. Only open/later managed objects and explicit dynamic/HTML refresh consumers participate.

Operation	Bound	Cancellation / recovery
Provider request	Configured timeout (default 30 seconds) and bounded batch (default 40 strings).	Cancel check and retry policy; no endless wait.
Build/deploy process	Explicit process timeout and termination wait.	Controlled process stop, captured output, destination-specific failure.
Scan	Progress by stage/file and source snapshot.	EProjectScanCancelled; prior catalog remains valid.
Atomic save	One candidate validation and replace operation.	Prior valid file and corruption quarantine.
Language switch	Admitted pack lookup plus managed open objects.	Reentrancy/main-thread guards and error behavior event/exception policy.

Operation	Bound	Cancellation / recovery
Maintenance close	One close request plus any already-active scan/provider boundary.	Atomic cancel flag and deferred close; then glossary Save/Discard/Cancel. No UI-thread busy wait.

21. Security and Privacy Engineering

> **What information is trusted?** Selected source is parsed, provider responses are validated, files are written atomically, and packs are admitted before use. Secrets, arbitrary JSON, and unverified output folders are never trusted merely because they exist.

- Secrets: Credential Manager/session memory only; never serialize or log API-key text.
- Network: development Studio only; HTTPS endpoints determined by selected provider/DeepL plan; response structure and placeholder count validated.
- Filesystem: normalize and validate intended paths; atomic writes; no broad recursive deletion; deployment only to verified application folders.
- Source: scanner parses text; it does not execute target Pascal, forms, scripts, or arbitrary JSON.
- Runtime: admit only compatible packs; reject malformed/wrong-app/wrong-framework/wrong-version/checksum mismatches before selection.
- Logs/reports: actionable identifiers and paths are acceptable; secrets and proprietary unneeded text are not.
- Distribution: exclude Local AppData workspaces, credentials, transient test folders, compiled output when source-only, and proprietary catalogs unless explicitly licensed.
- Dependencies: generated ComponentSource is source code and must be versioned/licensed/reviewed like any vendored library.

22. Test Architecture and Release Gates

> **What proves a change is ready?** Small focused test programs isolate individual contracts. Release scripts then combine those results with builds, real pilot applications, documentation checks, and a Git review.

The test suite is intentionally many small Delphi programs rather than one opaque test executable. Each program names and isolates a contract: scanning, placeholders, batching, retries, memory, validation, pack layout, VCL/FMX runtime, lifecycle, direction, wrapping, splash, templates, design streaming, and Studio form creation.

tools\verify_all.ps1 is the release-level orchestration entry. Contract scripts compile and run groups using the RAD Studio environment. check_shipped_units_complete.ps1 prevents package/kit/source-

distribution omissions; check_build_paths_agree.ps1 detects conflicting outputs; check_source_encoding.ps1 detects unsupported encoding drift.

Gate	Required evidence
Source hygiene	Encoding/path checks, no stale lock or unintended generated files, Git diff limited to approved scope.
Compile	Studio, package set, tools, and affected tests compile for intended platform/configuration.
Foundation	Types, JSON, workspace, atomic persistence, locale, placeholders, retry, terminology, memory, validation.
Scanner	VCL/FMX form discovery, Pascal/resource contracts, counts, context, quality, source snapshot.
Runtime	VCL/FMX load/apply/switch-back/restart, state preservation, lifecycle, templates, splash, wrapping, RTL.
Design-time	VCL/FMX form streaming and package registration with no missing-class or published-property error.
Studio	Main/Wizard/Review form creation, UI transitions, provider timeout/cancel, scan consistency, self-localization.
Pilot matrix	Real target project, all supported languages, HTML/dynamic text, LTR/RTL, actual output folders.
Documentation	DOCX source-driven accuracy, PDF generation, page rendering, visual inspection, accessibility/structure audits.

23. Production Pascal Unit Reference

> **How should the production unit reference be used?** Find the unit by subsystem, read its plain-language role and purpose, then inspect its direct dependencies, consumers, tests, and change risk before editing it.

This chapter is the production PAS inventory. Each entry names its file, purpose, reason for separation, public surface, direct DAT dependencies, direct production consumers, named test consumers, and change risk. Dependencies are parsed from the current source; descriptions state the architectural responsibility rather than merely repeating the filename.

23.1 Core units

These units own framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction. The entries are ordered by unit name.

DAT.Core.AITranslation

File: source\core\DAT.Core.AITranslation.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Coordinates automatic translation at catalog level. It decides eligibility, invokes the provider client, preserves protected work, records origin/status, and produces review information instead of treating machine output as approved.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TAITranslationReview, TAITranslationWorkflow

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.CatalogJson, DAT.Core.Types

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 384 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.AtomicFile

File: source\core\DAT.Core.AtomicFile.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Provides atomic UTF-8 persistence with validation, temporary files, previous-version recovery, corruption quarantine, and controlled replacement. Catalogs, packs, preferences, settings, glossaries, and deployment data rely on it to avoid partial JSON writes.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TAtomicTextFile, EDATAtomicFileError

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Core.AITranslation, DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.Hyphenation, DAT.Core.Interchange, DAT.Core.RuntimePack, DAT.Core.TranslationMemory, DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Provider.Settings, DAT.Review.Localization, DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides, DAT.Runtime.Preference, DAT.Scan.DomainProfile, DAT.Studio.SetupWizard

Named test consumers: AtomicPersistenceSmokeTests, FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXLanguageManagerTests, FoundationSmokeTests, GlossaryPublishSmokeTests, LanguageManagerCoreTests, VCLDesignStreamingTests, VCLLanguageManagerTests

Change and validation risk: The file contains 171 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.BuildInfo

File: source\core\DAT.Core.BuildInfo.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Supplies the compiled Studio build stamp and display description used in the title, Wizard header, reports, and diagnostics so artifacts can be traced to the producing build.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: StudioBuildStamp, StudioBuildDescription

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 61 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.CatalogJson

File: source\core\DAT.Core.CatalogJson.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Serializes and validates development catalogs and implements CSV import/export planning. It preserves stable keys, statuses, provenance, contexts, locale facts, and source checksums across round trips.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TCatalogCsvImportChange, TCatalogCsvImportPlan, TCatalogJson, TCatalogCsv, ECatalogJsonError, ECatalogCsvError

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Core.Types, DAT.Scan.TextCodec

Direct production consumers: DAT.Core.AITranslation, DAT.Integration.BuildDeploy, DAT.Review.Localization, DAT.Scan.CatalogMerge, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 712 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Diagnostics

File: source\core\DAT.Core.Diagnostics.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Centralizes structured information, warning, and error diagnostics so scanners, providers, integration, and runtime code report consistent severity and actionable context.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATDiagnosticSeverity, TDATDiagnostics

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.TranslationMemory, DAT.Provider.Settings, DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides, DAT.Runtime.Manager, DAT.Runtime.Preference

Named test consumers: FMXDesignStreamingTests, GlossaryPublishSmokeTests, LanguageManagerCoreTests, VCLDesignStreamingTests

Change and validation risk: The file contains 111 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Glossary

File: source\core\DAT.Core.Glossary.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Owns the project glossary model, matching rules, suggestions, catalog application, validation, and atomic persistence. It keeps product terminology explicit and reusable across provider runs.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProjectGlossary, TProjectGlossaryTerm, TGlossarySuggestion, TProjectGlossarySuggester

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Core.Types, DAT.Scan.TextCodec

Direct production consumers: DAT.Core.Interchange, DAT.Core.SharedDictionary, DAT.Integration.BuildDeploy, DAT.Provider.Glossary, DAT.Studio.LocalizationReview, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests, ProviderGlossarySmokeTests, TranslationMemorySmokeTests

Change and validation risk: The file contains 440 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Hyphenation

File: source\core\DAT.Core.Hyphenation.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Loads language-aware hyphenation dictionaries and applies conservative word/text hyphenation. Runtime layout uses it only when ordinary wrapping and readable font fitting are insufficient.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATHyphenationDictionary, TDATHyphenation

Direct DAT dependencies: DAT.Core.AtomicFile

Direct production consumers: DAT.Core.RuntimePack, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests, HyphenationSmokeTests, PackLayoutSmokeTests

Change and validation risk: The file contains 411 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Interchange

File: source\core\DAT.Core.Interchange.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Implements TMX translation-memory and TBX terminology interchange so reviewed material can move between the Studio and standards-aware localization tools.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTmxInterchange, TTbxInterchange

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Glossary, DAT.Core.TranslationMemory

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: TranslationMemorySmokeTests

Change and validation risk: The file contains 471 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.LocaleFacts

File: source\core\DAT.Core.LocaleFacts.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Reads Windows/Delphi locale facts such as native name, direction, date/time formats, decimal and thousands separators, and currency data for catalog metadata and runtime formatting.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TLocaleFacts, TLocaleFactsReader

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: GlossaryPublishSmokeTests

Change and validation risk: The file contains 135 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.ProjectDetection

File: source\core\DAT.Core.ProjectDetection.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Identifies a selected DPR/DPROJ, determines VCL or FMX, application ID, targets, forms, units, and output metadata, and rejects ambiguous or unsupported projects before scanning.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProjectDetector, EProjectDetectionError

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 118 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.RuntimePack

File: source\core\DAT.Core.RuntimePack.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Builds the compact offline runtime JSON pack from a validated development catalog. It filters non-runtime records, carries locale/layout metadata, and writes checksum-compatible source and target packs.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TRuntimePackBuilder, ERuntimePackError

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Hyphenation, DAT.Core.Types, DAT.Runtime.LanguagePack, DAT.Validation.Catalog

Direct production consumers: DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests, HyphenationSmokeTests, PackLayoutSmokeTests

Change and validation risk: The file contains 415 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.SharedDictionary

File: source\core\DAT.Core.SharedDictionary.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Maintains the shared vetted wording dictionary used to repair or standardize recurring interface terms without overwriting reviewed or approved project-specific work.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TSharedDictionary

Direct DAT dependencies: DAT.Core.Glossary, DAT.Core.Types

Direct production consumers: DAT.Studio.SetupWizard

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 208 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Terminology

File: source\core\DAT.Core.Terminology.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Resolves project glossary, shared dictionary, calendar terminology, protected tokens, and contextual wording in a deterministic priority order before or after provider translation.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTerminologyResolver

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 457 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.TranslationMemory

File: source\core\DAT.Core.TranslationMemory.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Stores and retrieves reviewed translations by exact, normalized, and similarity match. It preserves application and context provenance so later catalogs can reuse trusted wording without blind cross-key approval.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TMemoryMatchKind, TTranslationMemoryUnit, TMemoryMatch, TTranslationMemory, NormalizeSegment, SegmentSimilarity

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Core.Types

Direct production consumers: DAT.Core.Interchange, DAT.Studio.SetupWizard

Named test consumers: TranslationMemorySmokeTests

Change and validation risk: The file contains 582 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.TranslationWorkspace

File: source\core\DAT.Core.TranslationWorkspace.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Defines the authoritative per-application workspace below Local AppData and returns paths for development catalogs, runtime packs, glossaries, deployment settings, and recovery artifacts.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTranslationWorkspace

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests

Change and validation risk: The file contains 123 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Core.Types

File: source\core\DAT.Core.Types.pas

Where it fits: This layer defines the shared data and file rules used by the Studio, the scanners, and the deployed runtime.

What it is: Defines the central domain model: framework and status enumerations, project and locale profiles, translation entries, catalogs, ownership roles, and string conversions used by every pipeline stage.

Why it is separate: This unit exists to keep framework-neutral catalogs, persistence, locale data, terminology, workspace ownership, and pack construction in the core layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTargetFramework, TTranslationStatus, TTranslationOrigin, TRuntimeApplicationKind, TRuntimeTextRole, TTextOwnershipKind, TProjectProfile, TLocaleProfile, TTranslationEntry, TTranslationCatalog, TargetFrameworkToString, StringToTargetFramework, TranslationStatusToString, StringToTranslationStatus; plus 16 additional public declarations

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Core.AITranslation, DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.ProjectDetection, DAT.Core.RuntimePack, DAT.Core.SharedDictionary, DAT.Core.Terminology, DAT.Core.TranslationMemory, DAT.Core.TranslationWorkspace, DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.MenuResource, DAT.Integration.Package, DAT.Review.ApplicationStrings, DAT.Review.Localization, DAT.Review.TextMeasurement, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI, DAT.Scan.CatalogMerge, DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project, DAT.Scan.Quality, DAT.Scan.Rules, DAT.Scan.Types, DAT.Studio.LocalizationReview, DAT.Studio.MainForm, DAT.Studio.SetupWizard, DAT.Validation.Catalog

Named test consumers: ApplicationStringSmokeTests, FormScanContracts, FoundationSmokeTests, GlossaryPublishSmokeTests, HyphenationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, PackLayoutSmokeTests, ProposalDecisionSmokeTests, ProviderGlossarySmokeTests, ScannerSmokeTests, TranslationMemorySmokeTests

Change and validation risk: The file contains 579 lines. Changes require the core contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.2 Scan units

These units own project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging. The entries are ordered by unit name.

DAT.Scan.CatalogMerge

File: source\scan\DAT.Scan.CatalogMerge.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Merges a new scan into an existing catalog by stable key and source text, detects collisions, preserves valid translations, recovers semantic contracts, and marks removed records obsolete.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TCatalogMergeSummary, TScanCatalogMerger, EScanKeyCollision

Direct DAT dependencies: DAT.Core.CatalogJson, DAT.Core.Types, DAT.Scan.Types

Direct production consumers: DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 440 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.Context

File: source\scan\DAT.Scan.Context.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Infers UI role, semantic concept, description, confidence, and surrounding context for each scanned string so provider output and human review are less ambiguous.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TScanContextAnalyzer

Direct DAT dependencies: DAT.Scan.DomainProfile, DAT.Scan.Types

Direct production consumers: DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project

Named test consumers: ContextSmokeTests, FormScanContracts, FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 403 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.DomainProfile

File: source\scan\DAT.Scan.DomainProfile.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Builds an application-specific vocabulary and sense profile from project text, helping distinguish overloaded terms such as Play, Close, Schedule, and application-specific nouns.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TResolvedTerm, TApplicationDomainProfile, TDomainProfiler

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Scan.Types

Direct production consumers: DAT.Scan.Context, DAT.Studio.SetupWizard

Named test consumers: ContextSmokeTests, FormScanContracts, FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 625 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.FormText

File: source\scan\DAT.Scan.FormText.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Parses text DFM/FMX resources, walks component/property structure, applies eligibility rules, repairs encodings, and emits stable form.component.property scan items.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTextFormScanner

Direct DAT dependencies: DAT.Core.Types, DAT.Scan.Context, DAT.Scan.Quality, DAT.Scan.Rules, DAT.Scan.TextCodec, DAT.Scan.Types

Direct production consumers: DAT.Scan.Project

Named test consumers: FormScanContracts, FoundationSmokeTests

Change and validation risk: The file contains 405 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.PascalResources

File: source\scan\DAT.Scan.PascalResources.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Parses Pascal resourcestring declarations and approved runtime assignments while avoiding arbitrary code/data strings that cannot be safely localized.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TPascalResourceStringScanner

Direct DAT dependencies: DAT.Core.Types, DAT.Scan.Context, DAT.Scan.Quality, DAT.Scan.TextCodec, DAT.Scan.Types

Direct production consumers: DAT.Scan.Project

Named test consumers: FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 1,765 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.Project

File: source\scan\DAT.Scan.Project.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Orchestrates complete project scanning across detected form and source files, progress/cancellation, context analysis, quality checks, and result provenance.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProjectScanner

Direct DAT dependencies: DAT.Core.Types, DAT.Scan.Context, DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Quality, DAT.Scan.TextCodec, DAT.Scan.Types

Direct production consumers: DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 671 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.Quality

File: source\scan\DAT.Scan.Quality.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Analyzes scan output for suspicious keys, duplicate occurrences, encoding problems, dynamic/runtime ownership, missing context, and other conditions requiring operator attention.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TScanQualityAnalyzer

Direct DAT dependencies: DAT.Core.Types, DAT.Scan.Types

Direct production consumers: DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project

Named test consumers: FormScanContracts, FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 106 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.Rules

File: source\scan\DAT.Scan.Rules.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Centralizes the universal include/exclude/property rules used by both VCL and FMX scanning so a project is not governed by application-specific exceptions.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TScanRuleSet

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Scan.FormText

Named test consumers: FormScanContracts, FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 188 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.TextCodec

File: source\scan\DAT.Scan.TextCodec.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Loads Delphi text safely across BOM/code-page variants, repairs known mojibake, and decodes Delphi string expressions without executing source code.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: LoadDelphiTextFile, RepairMisdecodedText, TryDecodeDelphiStringExpression

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Integration.MenuResource, DAT.Review.Localization, DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project

Named test consumers: FormScanContracts, FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 210 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Scan.Types

File: source\scan\DAT.Scan.Types.pas

Where it fits: This layer reads saved Delphi forms and Pascal source and turns eligible text into stable translation records.

What it is: Defines scan items, diagnostics, kinds, progress/cancellation callbacks, source snapshots, counts, and the result object passed into catalog merge and UI reporting.

Why it is separate: This unit exists to keep project discovery, DFM/FMX and Pascal text extraction, context, rules, quality, and stable-key catalog merging in the scan layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TScannedTextKind, TScanDiagnosticSeverity, TScanItem, TScanDiagnostic, TProjectScanResult, EProjectScanCancelled, ScannedTextKindDisplayName

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Scan.CatalogMerge, DAT.Scan.Context, DAT.Scan.DomainProfile, DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project, DAT.Scan.Quality, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: ContextSmokeTests, FormScanContracts, FoundationSmokeTests, ScannerSmokeTests

Change and validation risk: The file contains 183 lines. Changes require the scan contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.3 Provider units

These units own DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation. The entries are ordered by unit name.

DAT.Provider.Batching

File: source\provider\DAT.Provider.Batching.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Groups entries by compatible context and splits them into bounded request batches so provider limits, latency, cancellation, and context quality remain predictable.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TContextGroup, TContextBatching

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client

Named test consumers: ContextBatchingSmokeTests, FoundationSmokeTests

Change and validation risk: The file contains 122 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.CalendarTerms

File: source\provider\DAT.Provider.CalendarTerms.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Supplies locale-specific calendar vocabulary and protects weekday/month meanings from ambiguous short-word provider output.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TCalendarTerms

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client

Named test consumers: CalendarTermSmokeTests, FoundationSmokeTests

Change and validation risk: The file contains 285 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Client

File: source\provider\DAT.Provider.Client.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Implements the bounded HTTPS client for DeepL API Free/Pro and Google Cloud Translation Basic v2, including request bodies, response parsing, contexts, cancellation, timeout, retries, and normalized errors.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTranslationCancelled, TTranslationProviderClient

Direct DAT dependencies: DAT.Provider.Batching, DAT.Provider.CalendarTerms, DAT.Provider.LanguageCodes, DAT.Provider.Placeholders, DAT.Provider.Retry, DAT.Provider.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 600 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.CredentialStore

File: source\provider\DAT.Provider.CredentialStore.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Stores, retrieves, and removes provider secrets as Windows Generic Credentials. It deliberately separates secrets from provider-settings JSON, catalogs, packs, source, logs, and exports.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProviderCredentialStore, ECredentialStoreError

Direct DAT dependencies: DAT.Provider.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 166 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Glossary

File: source\provider\DAT.Provider.Glossary.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Converts vetted project terminology into provider-facing glossary behavior where supported and local pre/post-processing where the provider API does not offer a glossary feature.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TGlossaryEntry, TProviderGlossary

Direct DAT dependencies: DAT.Core.Glossary

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: ProviderGlossarySmokeTests

Change and validation risk: The file contains 230 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.LanguageCodes

File: source\provider\DAT.Provider.LanguageCodes.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Maps canonical locale codes to the distinct source/target language codes accepted by DeepL and Google, including provider-specific aliases and unsupported-locale checks.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProviderLanguageCodes

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client

Named test consumers: FoundationSmokeTests, LanguageCodeSmokeTests

Change and validation risk: The file contains 86 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Placeholders

File: source\provider\DAT.Provider.Placeholders.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Protects Delphi format placeholders, accelerators, markup, identifiers, and other immutable tokens before a provider call and verifies exact restoration afterward.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TPlaceholderProtection

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client

Named test consumers: FoundationSmokeTests, PlaceholderSmokeTests

Change and validation risk: The file contains 239 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Retry

File: source\provider\DAT.Provider.Retry.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Defines bounded retry policy for transient provider and network failures, including backoff and retryable status classification without retrying permanent authentication errors indefinitely.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProviderRetry

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client

Named test consumers: FoundationSmokeTests, RetrySmokeTests

Change and validation risk: The file contains 92 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Settings

File: source\provider\DAT.Provider.Settings.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Owns non-secret provider configuration under Local AppData: selected provider, DeepL plan, remember policy, timeout, and batch size. API key text is intentionally excluded.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TProviderSettings

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Provider.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 141 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Provider.Types

File: source\provider\DAT.Provider.Types.pas

Where it fits: This layer is the only part that talks to DeepL or Google and handles provider credentials and limits.

What it is: Defines provider and DeepL-plan enumerations, display/string conversions, and the structured provider exception carrying status and retry context.

Why it is separate: This unit exists to keep DeepL/Google configuration, credential protection, batching, placeholders, retries, and HTTPS translation in the provider layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTranslationProvider, TDeepLPlan, ETranslationProviderError, TranslationProviderDisplayName, TranslationProviderToString, StringToTranslationProvider, DeepLPlanToString, StringToDeepLPlan

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Provider.Client, DAT.Provider.CredentialStore, DAT.Provider.Settings, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 93 lines. Changes require the provider contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.4 Review units

These units own localization findings, layout proposals, text measurement, and application-owned string review. The entries are ordered by unit name.

DAT.Review.ApplicationStrings

File: source\review\DAT.Review.ApplicationStrings.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Classifies strings owned by the application rather than by static form resources, providing focused review of live status, generated messages, templates, and runtime text contracts.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TApplicationOwnedStrings

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Studio.LocalizationReview

Named test consumers: ApplicationStringSmokeTests, FoundationSmokeTests

Change and validation risk: The file contains 164 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Review.CodeGeometry

File: source\review\DAT.Review.CodeGeometry.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Extracts control geometry assignments from Pascal so layout review can distinguish designer-owned positions from deliberately code-positioned controls that must not be moved automatically.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TCodeGeometry

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: DAT.Review.Localization

Named test consumers: FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 233 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Review.Localization

File: source\review\DAT.Review.Localization.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Runs linguistic and layout review, creates findings and conservative per-language layout proposals, records accept/reject decisions, and prepares the HTML/JSON review package.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TLocalizationFindingSeverity, TLocalizationFinding, TLayoutControl, TLayoutProposal, TLocalizationReview, TLocalizationReviewer, LocalizationFindingSeverityText

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.CatalogJson, DAT.Core.Types, DAT.Review.CodeGeometry, DAT.Review.TextMeasurement, DAT.Runtime.LanguagePack, DAT.Scan.TextCodec

Direct production consumers: DAT.Studio.LocalizationReview, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 5,657 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Review.TextMeasurement

File: source\review\DAT.Review.TextMeasurement.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Defines the platform-neutral text-measurement interface and fitting calculations used to compare translated text against available control geometry.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTextMeasurement

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Review.Localization, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI

Named test consumers: FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 89 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Review.TextMeasurement.FMX

File: source\review\DAT.Review.TextMeasurement.FMX.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Provides FMX canvas-based text measurement so font, wrapping, and control-size proposals reflect FireMonkey rendering characteristics.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: No separate public type or routine; the unit supplies registrations or implementation bindings.

Direct DAT dependencies: DAT.Core.Types, DAT.Review.TextMeasurement

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 57 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Review.TextMeasurement.GDI

File: source\review\DAT.Review.TextMeasurement.GDI.pas

Where it fits: This layer turns translation and layout concerns into findings a developer can inspect and decide.

What it is: Provides Windows GDI text measurement for VCL controls and report/header fitting using the actual font metrics available on Windows.

Why it is separate: This unit exists to keep localization findings, layout proposals, text measurement, and application-owned string review in the review layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: No separate public type or routine; the unit supplies registrations or implementation bindings.

Direct DAT dependencies: DAT.Core.Types, DAT.Review.TextMeasurement

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FoundationSmokeTests, LayoutContracts, LayoutFittingSmokeTests, ProposalDecisionSmokeTests

Change and validation risk: The file contains 124 lines. Changes require the review contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.5 Validation units

These units own structural catalog checks that protect runtime export. The entries are ordered by unit name.

DAT.Validation.Catalog

File: source\validation\DAT.Validation.Catalog.pas

Where it fits: This layer blocks structurally unsafe catalogs and packs without pretending to judge human language quality.

What it is: Performs blocking structural validation and non-blocking review warnings for locale metadata, required text, placeholders, accelerators, duplicate keys, source changes, runtime wiring, and export readiness.

Why it is separate: This unit exists to keep structural catalog checks that protect runtime export in the validation layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TValidationSeverity, TValidationIssue, TCatalogValidationResult, TCatalogValidator, ValidationSeverityDisplayName

Direct DAT dependencies: DAT.Core.Types

Direct production consumers: DAT.Core.RuntimePack, DAT.Integration.BuildDeploy, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests

Change and validation risk: The file contains 403 lines. Changes require the validation contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.6 Runtime units

These units own offline pack loading, preference, translation application, layout restoration, templates, and splash handling. The entries are ordered by unit name.

DAT.Runtime.FMX

File: source\runtime\DAT.Runtime.FMX.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Applies runtime packs to FMX object trees while preserving live state, snapshots, designer geometry, control order, styled controls, grids, menus, scrolling, wrapping, and language direction.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: No separate public type or routine; the unit supplies registrations or implementation bindings.

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides

Direct production consumers: DAT.Components.FMX, DAT.Components.FMX.Spike, DAT.Runtime.SplashTranslation.FMX, DAT.Studio.Translation

Named test consumers: FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXGridOrderSmokeTests, FMXLanguageManagerTests, FMXManagerLifecycleSpikeTests, FMXMenuOrderSmokeTests, FMXRightToLeftSmokeTests, FMXRuntimeSmokeTests, FMXSplashHookSmokeTests, FMXStyledRuntimeSmokeTests, FMXWrapSmokeTests

Change and validation risk: The file contains 3,058 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.LanguagePack

File: source\runtime\DAT.Runtime.LanguagePack.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Parses and validates deployed JSON packs, indexes stable keys/source text/templates/layout rules, supports dynamic and HTML translation, and exposes locale and compatibility metadata to the runtime manager.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TRuntimeLocale, TRuntimeLayoutRule, TRuntimeLanguagePack, TLanguagePackDescriptor, TLanguagePackDiscovery, ELanguagePackError, CanonicalNativeLanguageName, IsRuntimeLayoutProperty, IsAutomaticallySafeLayoutProperty

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics

Direct production consumers: DAT.Components.Core, DAT.Components.FMX, DAT.Components.FMX.LanguageSelector, DAT.Components.FMX.Spike, DAT.Components.VCL, DAT.Components.VCL.LanguageSelector, DAT.Components.VCL.Spike, DAT.Core.RuntimePack, DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.MenuResource, DAT.Integration.Package, DAT.Review.Localization, DAT.Runtime.FMX, DAT.Runtime.Manager, DAT.Runtime.SplashTranslation, DAT.Runtime.SplashTranslation.FMX, DAT.Runtime.SplashTranslation.VCL, DAT.Runtime.TemplateRewrite, DAT.Runtime.TemplateRewrite.FMX, DAT.Runtime.TemplateRewrite.VCL, DAT.Runtime.VCL, DAT.Studio.LocalizationReview, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FMXCaptionRewriteSmokeTests, FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXGridOrderSmokeTests, FMXLanguageManagerTests, FMXManagerLifecycleSpikeTests, FMXMenuOrderSmokeTests, FMXRightToLeftSmokeTests, FMXRuntimeSmokeTests, FMXSplashHookSmokeTests, FMXStyledRuntimeSmokeTests, FMXWrapSmokeTests, FoundationSmokeTests, GlossaryPublishSmokeTests, LanguageManagerCoreTests, LayoutOverrideSmokeTests, TemplateRewriteSmokeTests, VCLCaptionInterceptSmokeTests, VCLDesignStreamingTests, VCLDiscoverySmokeTests, VCLLanguageManagerTests, VCLManagerLifecycleSpikeTests, VCLManagerMDILifecycleSpikeTests, VCLRightToLeftSmokeTests, VCLRuntimeSmokeTests, VCLSplashHookSmokeTests, VCLSplashSmokeTests, VCLStatusPanelInterceptTests, VCLWrapSmokeTests

Change and validation risk: The file contains 1,305 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.LayoutOverrides

File: source\runtime\DAT.Runtime.LayoutOverrides.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Persists developer-entered per-language control overrides separately from generated packs so hand-tuned positions, sizes, and font choices remain explicit and recoverable.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TLayoutOverride, TLayoutOverrides

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics

Direct production consumers: DAT.Runtime.FMX, DAT.Runtime.VCL

Named test consumers: FMXStyledRuntimeSmokeTests, LayoutOverrideSmokeTests

Change and validation risk: The file contains 399 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.Manager

File: source\runtime\DAT.Runtime.Manager.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Owns offline pack discovery, admission, checksum/source-pack compatibility, selected locale, formatting settings, translation lookup, preference fallback, and the active source/target runtime pair.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TTranslationRuntime

Direct DAT dependencies: DAT.Core.Diagnostics, DAT.Runtime.LanguagePack, DAT.Runtime.Preference

Direct production consumers: DAT.Components.Core, DAT.Integration.Package, DAT.Studio.Translation

Named test consumers: FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXLanguageManagerTests, FoundationSmokeTests, LanguageManagerCoreTests, VCLDesignStreamingTests, VCLLanguageManagerTests, VCLSplashSmokeTests

Change and validation risk: The file contains 636 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.Preference

File: source\runtime\DAT.Runtime.Preference.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Reads and atomically writes the selected language code in the configured Local AppData or executable-folder preference location, with safe fallback from stale or invalid values.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TLanguagePreference

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.Diagnostics

Direct production consumers: DAT.Runtime.Manager, DAT.Runtime.SplashTranslation

Named test consumers: FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXLanguageManagerTests, FMXSplashHookSmokeTests, FoundationSmokeTests, LanguageManagerCoreTests, VCLDesignStreamingTests, VCLLanguageManagerTests, VCLSplashHookSmokeTests, VCLSplashSmokeTests

Change and validation risk: The file contains 105 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.SplashTranslation

File: source\runtime\DAT.Runtime.SplashTranslation.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Defines the framework-neutral splash translation contract and keys so early startup text can be translated before the main form and standard lifecycle hooks exist.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATSplashTranslation

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.Preference

Direct production consumers: DAT.Components.FMX, DAT.Components.VCL, DAT.Runtime.SplashTranslation.FMX, DAT.Runtime.SplashTranslation.VCL

Named test consumers: FMXSplashHookSmokeTests, VCLSplashHookSmokeTests

Change and validation risk: The file contains 176 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.SplashTranslation.FMX

File: source\runtime\DAT.Runtime.SplashTranslation.FMX.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Implements early FMX splash translation and coordinates the selected runtime before ordinary FMX form discovery begins.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATFMXSplashTranslation

Direct DAT dependencies: DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.SplashTranslation

Direct production consumers: DAT.Components.FMX

Named test consumers: FMXSplashHookSmokeTests

Change and validation risk: The file contains 110 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.SplashTranslation.VCL

File: source\runtime\DAT.Runtime.SplashTranslation.VCL.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Implements early VCL splash translation while respecting VCL handle creation and startup ordering.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATVCLSplashTranslation

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.SplashTranslation, DAT.Runtime.VCL

Direct production consumers: DAT.Components.VCL

Named test consumers: VCLSplashHookSmokeTests

Change and validation risk: The file contains 92 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.TemplateRewrite

File: source\runtime\DAT.Runtime.TemplateRewrite.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Provides the framework-neutral contract for translating templated or owner-drawn text whose final string is produced after normal property translation.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATTemplateRewriter

Direct DAT dependencies: DAT.Runtime.LanguagePack

Direct production consumers: DAT.Runtime.TemplateRewrite.FMX, DAT.Runtime.TemplateRewrite.VCL, DAT.Runtime.VCL

Named test consumers: FMXCaptionRewriteSmokeTests, TemplateRewriteSmokeTests, VCLCaptionInterceptSmokeTests

Change and validation risk: The file contains 339 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.TemplateRewrite.FMX

File: source\runtime\DAT.Runtime.TemplateRewrite.FMX.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Applies FMX-specific template and caption rewrites after styles or control templates regenerate visible text.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATFMXTemplateRewrite

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.TemplateRewrite

Direct production consumers: DAT.Components.FMX

Named test consumers: FMXCaptionRewriteSmokeTests

Change and validation risk: The file contains 74 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.TemplateRewrite.VCL

File: source\runtime\DAT.Runtime.TemplateRewrite.VCL.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Intercepts and reapplies VCL caption/template text that Windows messages or owner-drawn controls can recreate after the initial translation pass.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATVCLTemplateIntercept

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.TemplateRewrite

Direct production consumers: DAT.Components.VCL

Named test consumers: VCLCaptionInterceptSmokeTests

Change and validation risk: The file contains 149 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Runtime.VCL

File: source\runtime\DAT.Runtime.VCL.pas

Where it fits: This layer loads local language packs and applies them inside the finished offline VCL or FMX application.

What it is: Applies packs to VCL windows and controls while preserving handles, focus, selections, list/grid state, designer layout, status panels, menus, dialogs, wrapping, and reversible RTL state.

Why it is separate: This unit exists to keep offline pack loading, preference, translation application, layout restoration, templates, and splash handling in the runtime layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: No separate public type or routine; the unit supplies registrations or implementation bindings.

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides, DAT.Runtime.TemplateRewrite

Direct production consumers: DAT.Components.VCL, DAT.Components.VCL.Spike, DAT.Runtime.SplashTranslation.VCL

Named test consumers: LayoutOverrideSmokeTests, VCLDesignStreamingTests, VCLDiscoverySmokeTests, VCLLanguageManagerTests, VCLManagerLifecycleSpikeTests,

VCLManagerMDILifecycleSpikeTests, VCLRightToLeftSmokeTests, VCLRuntimeSmokeTests, VCLSplashHookSmokeTests, VCLSplashSmokeTests, VCLStatusPanelInterceptTests, VCLWrapSmokeTests

Change and validation risk: The file contains 2,775 lines. Changes require the runtime contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.7 Components units

These units own designer-visible language-manager components, lifecycle adapters, and language selectors. The entries are ordered by unit name.

DAT.Components.Core

File: source\components\DAT.Components.Core.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Defines the framework-neutral component contract shared by VCL and FMX managers: published configuration, language-change events, form identity mapping, error policy, and the process-wide translation functions used by dynamic text and HTML producers.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATPreferenceLocation, TDATMissingPackBehavior, TDATErrorBehavior, TDATCustomLanguageManager, EDATLanguageManagerError, EDATLanguageManagerConfigurationError, EDATLanguageManagerMainThreadError, EDATLanguageManagerReentrancyError, EDATFormIdentityError, DATTranslateText, DATFormatText, DATActiveLanguageCode, DATActiveTextDirection, DATTranslateDynamicText; plus 1 additional public declarations

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.Manager

Direct production consumers: DAT.Components.FMX, DAT.Components.VCL

Named test consumers: FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXLanguageManagerTests, LanguageManagerCoreTests, VCLDesignStreamingTests, VCLDiscoverySmokeTests, VCLLanguageManagerTests, VCLSplashSmokeTests

Change and validation risk: The file contains 1,221 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.FMX

File: source\components\DAT.Components.FMX.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Implements the production FMX language manager and its application lifecycle integration. It discovers forms, preserves state and designer geometry, applies packs, refreshes tabs, scroll boxes, browsers, grids, and dynamic content, and enforces reversible LTR/RTL behavior.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATScrollBoundsSubscription, TDATBrowserLifecycleSubscription, TDATTabChangeSubscription, TDATFMXLanguageManager

Direct DAT dependencies: DAT.Components.Core, DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.SplashTranslation, DAT.Runtime.SplashTranslation.FMX, DAT.Runtime.TemplateRewrite.FMX

Direct production consumers: DAT.Components.FMX.LanguageSelector, DAT.Design.FMX.Register

Named test consumers: FMXDesignStreamingTests, FMXDialogTranslationSmokeTests, FMXDiscoverySmokeTests, FMXLanguageManagerTests

Change and validation risk: The file contains 2,275 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.FMX.LanguageSelector

File: source\components\DAT.Components.FMX.LanguageSelector.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Implements the FMX TDATFMXLanguageComboBox. It binds to one FMX language manager, lists admitted packs, optionally shows locale codes, and changes language without duplicating manager logic.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATFMXLanguageComboBox

Direct DAT dependencies: DAT.Components.FMX, DAT.Runtime.LanguagePack

Direct production consumers: DAT.Design.FMX.Register

Named test consumers: FMXDesignStreamingTests

Change and validation risk: The file contains 170 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.FMX.Spike

File: source\components\DAT.Components.FMX.Spike.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Retains the focused FMX lifecycle spike used to prove form discovery and exactly-once application behavior before those contracts were promoted into the production FMX manager.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATFMXLanguageManagerSpike

Direct DAT dependencies: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FMXManagerLifecycleSpikeTests

Change and validation risk: The file contains 168 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.VCL

File: source\components\DAT.Components.VCL.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Implements the production VCL language manager. It observes application idle, modal, and active-form transitions; translates open and later forms; preserves live control state; restores source geometry; and applies VCL-specific RTL and layout contracts.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATVCLLanguageManager

Direct DAT dependencies: DAT.Components.Core, DAT.Runtime.LanguagePack, DAT.Runtime.SplashTranslation, DAT.Runtime.SplashTranslation.VCL, DAT.Runtime.TemplateRewrite.VCL, DAT.Runtime.VCL

Direct production consumers: DAT.Components.VCL.LanguageSelector, DAT.Design.VCL.Register

Named test consumers: VCLDesignStreamingTests, VCLDiscoverySmokeTests, VCLLanguageManagerTests, VCLSplashSmokeTests

Change and validation risk: The file contains 352 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.VCL.LanguageSelector

File: source\components\DAT.Components.VCL.LanguageSelector.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Implements the VCL TDATVCLLanguageComboBox. It connects through a published LanguageManager property, refreshes available packs, and selects the locale while remaining fully designer-editable.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATVCLLanguageComboBox

Direct DAT dependencies: DAT.Components.VCL, DAT.Runtime.LanguagePack

Direct production consumers: DAT.Design.VCL.Register

Named test consumers: VCLDesignStreamingTests

Change and validation risk: The file contains 170 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Components.VCL.Spike

File: source\components\DAT.Components.VCL.Spike.pas

Where it fits: This layer gives a Delphi application the nonvisual language manager and the visible language selector used at run time.

What it is: Retains the VCL lifecycle proof harness that established idle discovery, notification cleanup, and exactly-once form application before the production VCL component was finalized.

Why it is separate: This unit exists to keep designer-visible language-manager components, lifecycle adapters, and language selectors in the components layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDATVCLLanguageManagerSpike

Direct DAT dependencies: DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: VCLManagerLifecycleSpikeTests, VCLManagerMDILifecycleSpikeTests

Change and validation risk: The file contains 123 lines. Changes require the components contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.8 Design units

These units own RAD Studio Tool Palette registration and design-time package exposure. The entries are ordered by unit name.

DAT.Design.FMX.Register

File: source\design\DAT.Design.FMX.Register.pas

Where it fits: This layer makes the language components appear in the RAD Studio Tool Palette and Object Inspector.

What it is: Registers the FMX language manager and selector on the DAT Localization Tool Palette page. It is compiled only into the FMX design-time package, never into a deployed application.

Why it is separate: This unit exists to keep RAD Studio Tool Palette registration and design-time package exposure in the design layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: Register

Direct DAT dependencies: DAT.Components.FMX, DAT.Components.FMX.LanguageSelector

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 20 lines. Changes require the design contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Design.VCL.Register

File: source\design\DAT.Design.VCL.Register.pas

Where it fits: This layer makes the language components appear in the RAD Studio Tool Palette and Object Inspector.

What it is: Registers the VCL language manager and selector on the DAT Localization Tool Palette page. It is the VCL design-time bridge between package installation and Object Inspector use.

Why it is separate: This unit exists to keep RAD Studio Tool Palette registration and design-time package exposure in the design layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: Register

Direct DAT dependencies: DAT.Components.VCL, DAT.Components.VCL.LanguageSelector

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 20 lines. Changes require the design contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.9 Integration units

These units own component-kit generation, controlled Delphi project integration, build, and deployment. The entries are ordered by unit name.

DAT.Integration.BuildDeploy

File: source\integration\DAT.Integration.BuildDeploy.pas

Where it fits: This layer prepares the files a target Delphi project needs and copies approved outputs to approved destinations.

What it is: Builds selected Delphi platform/configuration targets, discovers existing configured output folders, atomically deploys complete compatible pack sets, and publishes standalone glossary corrections through the matching catalog without rebuilding or editing the target project. It reports command, output, timeout, catalog, pack, and destination results.

Why it is separate: This unit exists to keep component-kit generation, controlled Delphi project integration, build, and deployment in the integration layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TGlossaryPublishResult, TGlossaryPublisher, TTargetBuildDeployer

Direct DAT dependencies: DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.RuntimePack, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Runtime.LanguagePack, DAT.Validation.Catalog

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests, GlossaryPublishSmokeTests

Change and validation risk: The file contains 1,032 lines. Changes require the integration contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Integration.ComponentPackage

File: source\integration\DAT.Integration.ComponentPackage.pas

Where it fits: This layer prepares the files a target Delphi project needs and copies approved outputs to approved destinations.

What it is: Generates and validates the complete framework-appropriate kit, including ComponentSource, packs, Apache license, manifest, README, deployment support, and the exact Win32 Release BPL bundle. PrepareProjectDependencies then stages and atomically installs only the project-local dependency while target project files remain read-only.

Why it is separate: This unit exists to keep component-kit generation, controlled Delphi project integration, build, and deployment in the integration layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TComponentIntegrationPackageGenerator

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.RuntimePack, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Runtime.LanguagePack, DAT.Scan.CatalogMerge, DAT.Scan.Project, DAT.Scan.Types

Direct production consumers: DAT.Studio.MainForm, DAT.Studio.SetupWizard

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 807 lines. Changes require the integration contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Integration.DelphiSource

File: source\integration\DAT.Integration.DelphiSource.pas

Where it fits: This layer prepares the files a target Delphi project needs and copies approved outputs to approved destinations.

What it is: Implements the explicitly advanced source-integration editor with preview, authorization, transaction backup, bounded changes, and restoration; it is not used by the recommended component workflow.

Why it is separate: This unit exists to keep component-kit generation, controlled Delphi project integration, build, and deployment in the integration layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TDelphiIntegrationSourceEditor

Direct DAT dependencies: No DAT unit dependency; this is a leaf/foundation unit.

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 536 lines. Changes require the integration contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Integration.MenuResource

File: source\integration\DAT.Integration.MenuResource.pas

Where it fits: This layer prepares the files a target Delphi project needs and copies approved outputs to approved destinations.

What it is: Plans and applies controlled language-menu resource changes for advanced integration, including safe detection and reversal of the generated menu block.

Why it is separate: This unit exists to keep component-kit generation, controlled Delphi project integration, build, and deployment in the integration layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TMenuResourceEdit, TLanguageMenuResourceEditor

Direct DAT dependencies: DAT.Core.Types, DAT.Runtime.LanguagePack, DAT.Scan.TextCodec

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 563 lines. Changes require the integration contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Integration.Package

File: source\integration\DAT.Integration.Package.pas

Where it fits: This layer prepares the files a target Delphi project needs and copies approved outputs to approved destinations.

What it is: Builds the broader integration plan and coordinates package/source actions, generated artifacts, reset behavior, and safety transactions across supported integration modes.

Why it is separate: This unit exists to keep component-kit generation, controlled Delphi project integration, build, and deployment in the integration layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TIntegrationPackageGenerator

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.RuntimePack, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Scan.CatalogMerge, DAT.Scan.Project, DAT.Scan.Types

Direct production consumers: DAT.Studio.MainForm

Named test consumers: FoundationSmokeTests

Change and validation risk: The file contains 492 lines. Changes require the integration contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

23.10 Studio units

These units own the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation. The entries are ordered by unit name.

DAT.Studio.LocalizationReview

File: source\studio\DAT.Studio.LocalizationReview.pas

Where it fits: This layer is the developer-facing FMX application: its forms, buttons, pages, Wizard, and workflow state.

What it is: Implements the Localization Review FMX window: findings, glossary terms, translation suggestions, layout proposals, decision persistence, HTML package generation, and controlled return to Wizard processing.

Why it is separate: This unit exists to keep the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation in the studio layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TfrmLocalizationReview

Direct DAT dependencies: DAT.Core.Glossary, DAT.Core.Types, DAT.Review.ApplicationStrings, DAT.Review.Localization, DAT.Runtime.LanguagePack

Direct production consumers: DAT.Studio.SetupWizard

Named test consumers: StudioFormSmokeTests

Change and validation risk: The file contains 610 lines. Changes require the studio contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Studio.MainForm

File: source\studio\DAT.Studio.MainForm.pas

Where it fits: This layer is the developer-facing FMX application: its forms, buttons, pages, Wizard, and workflow state.

What it is: Implements the landing screen and eight-page Maintenance Studio. It coordinates project detection, scanning, catalog editing, direct project-glossary maintenance, provider translation, validation, export, integration, settings, safe cancellation, keyboard defaults, and status reporting.

Why it is separate: This unit exists to keep the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation in the studio layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TfrmTranslationStudio

Direct DAT dependencies: DAT.Core.BuildInfo, DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.LocaleFacts, DAT.Core.ProjectDetection, DAT.Core.RuntimePack, DAT.Core.Terminology, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.Package, DAT.Provider.Client, DAT.Provider.CredentialStore, DAT.Provider.Settings, DAT.Provider.Types, DAT.Runtime.LanguagePack, DAT.Scan.CatalogMerge, DAT.Scan.Project, DAT.Scan.Types, DAT.Studio.SetupWizard, DAT.Studio.Translation, DAT.Validation.Catalog

Direct production consumers: No production DAT unit imports it directly; it is an executable/package entry point or optional adapter.

Named test consumers: StudioFormSmokeTests

Change and validation risk: The file contains 3,174 lines. Changes require the studio contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Studio.SetupWizard

File: source\studio\DAT.Studio.SetupWizard.pas

Where it fits: This layer is the developer-facing FMX application: its forms, buttons, pages, Wizard, and workflow state.

What it is: Implements the eight-step Setup Wizard state machine, prerequisite validation, scan/translation/review/final-processing sequence, cancellation boundaries, safety backup, kit generation, deployment, and Finish behavior.

Why it is separate: This unit exists to keep the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation in the studio layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: TfrmSetupWizard

Direct DAT dependencies: DAT.Core.AtomicFile, DAT.Core.BuildInfo, DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.Hyphenation, DAT.Core.LocaleFacts, DAT.Core.ProjectDetection, DAT.Core.RuntimePack, DAT.Core.SharedDictionary, DAT.Core.Terminology, DAT.Core.TranslationMemory, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Provider.Client, DAT.Provider.CredentialStore, DAT.Provider.Settings, DAT.Provider.Types, DAT.Review.Localization, DAT.Runtime.LanguagePack, DAT.Scan.CatalogMerge, DAT.Scan.DomainProfile, DAT.Scan.Project, DAT.Scan.Types, DAT.Studio.LocalizationReview, DAT.Validation.Catalog

Direct production consumers: DAT.Studio.MainForm

Named test consumers: StudioFormSmokeTests

Change and validation risk: The file contains 2,425 lines. Changes require the studio contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

DAT.Studio.Translation

File: source\studio\DAT.Studio.Translation.pas

Where it fits: This layer is the developer-facing FMX application: its forms, buttons, pages, Wizard, and workflow state.

What it is: Self-localizes the Studio interface by loading its own packs, applying them to FMX forms, maintaining the interface-language menu, and keeping Studio language state separate from target-project catalogs.

Why it is separate: This unit exists to keep the FMX operator interface, Setup Wizard state machine, Localization Review, and Studio self-translation in the studio layer instead of coupling that responsibility to unrelated UI or framework code.

Public surface: InitializeStudioTranslation, ApplyStudioTranslation, SelectStudioLanguageMenuItem, StudioTranslationRuntime

Direct DAT dependencies: DAT.Runtime.FMX, DAT.Runtime.Manager

Direct production consumers: DAT.Studio.MainForm

Named test consumers: Coverage is indirect through subsystem and integration tests.

Change and validation risk: The file contains 103 lines. Changes require the studio contract tests, a clean package/application build, and a runtime regression pass for both VCL and FMX whenever shared behavior changes.

24. Test, Spike, and Contract Pascal Program Reference

> **What are the test and fixture files for?** Tests are intentionally separate programs so a failure names one contract. Fixtures provide controlled forms, source, and behavior; they are evidence, not production dependencies.

Every Delphi test DPR is listed below. Test-support PAS/DFM/FMX units and formal contract fixtures follow. These files are not runtime dependencies; they supply isolated evidence and intentionally artificial application behavior.

24.1 Test executable programs

ApplicationStringSmokeTests

File: tools\tests\ApplicationStringSmokeTests.dpr

What it verifies: Verifies application-owned and dynamic string contracts that are not ordinary designer properties.

DAT units exercised directly: DAT.Core.Types, DAT.Review.ApplicationStrings

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

AtomicPersistenceSmokeTests

File: tools\tests\AtomicPersistenceSmokeTests.dpr

What it verifies: Exercises atomic save, previous-file recovery, corruption quarantine, and interrupted-write behavior.

DAT units exercised directly: DAT.Core.AtomicFile

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

CalendarTermSmokeTests

File: tools\tests\CalendarTermSmokeTests.dpr

What it verifies: Checks locale-aware calendar terminology and ambiguous weekday/month handling.

DAT units exercised directly: DAT.Provider.CalendarTerms

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

ContextBatchingSmokeTests

File: tools\tests\ContextBatchingSmokeTests.dpr

What it verifies: Checks that provider batches preserve compatible contexts and obey configured size limits.

DAT units exercised directly: DAT.Provider.Batching

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

ContextSmokeTests

File: tools\tests\ContextSmokeTests.dpr

What it verifies: Checks semantic role, concept, description, and confidence inference for representative UI text.

DAT units exercised directly: DAT.Scan.Context, DAT.Scan.DomainProfile, DAT.Scan.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXCaptionRewriteSmokeTests

File: tools\tests\FMXCaptionRewriteSmokeTests.dpr

What it verifies: Exercises the F M X Caption Rewrite contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.TemplateRewrite, DAT.Runtime.TemplateRewrite.FMX

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXDesignStreamingTests

File: tools\tests\FMXDesignStreamingTests.dpr

What it verifies: Exercises the F M X Design Streaming contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.FMX, DAT.Components.FMX.LanguageSelector, DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXDialogTranslationSmokeTests

File: tools\tests\FMXDialogTranslationSmokeTests.dpr

What it verifies: Exercises the F M X Dialog Translation contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.FMX, DAT.Core.AtomicFile, DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXDiscoverySmokeTests

File: tools\tests\FMXDiscoverySmokeTests.dpr

What it verifies: Exercises the F M X Discovery contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.FMX, DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXGridOrderSmokeTests

File: tools\tests\FMXGridOrderSmokeTests.dpr

What it verifies: Exercises the F M X Grid Order contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXLanguageManagerTests

File: tools\tests\FMXLanguageManagerTests.dpr

What it verifies: Exercises the F M X Language Manager contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.FMX, DAT.Core.AtomicFile, DAT.LifecycleSpike.Trace, DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXManagerLifecycleSpikeTests

File: tools\tests\FMXManagerLifecycleSpikeTests.dpr

What it verifies: Exercises the F M X Manager Lifecycle Spike contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.FMX.Spike, DAT.LifecycleSpike.Trace, DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXMenuOrderSmokeTests

File: tools\tests\FMXMenuOrderSmokeTests.dpr

What it verifies: Exercises the F M X Menu Order contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXRightToLeftSmokeTests

File: tools\tests\FMXRightToLeftSmokeTests.dpr

What it verifies: Exercises the F M X Right To Left contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXRuntimeSmokeTests

File: tools\tests\FMXRuntimeSmokeTests.dpr

What it verifies: Exercises the F M X Runtime contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXSplashHookSmokeTests

File: tools\tests\FMXSplashHookSmokeTests.dpr

What it verifies: Exercises the F M X Splash Hook contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.Preference, DAT.Runtime.SplashTranslation, DAT.Runtime.SplashTranslation.FMX

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXStyledRuntimeSmokeTests

File: tools\tests\FMXStyledRuntimeSmokeTests.dpr

What it verifies: Exercises the F M X Styled Runtime contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FMXWrapSmokeTests

File: tools\tests\FMXWrapSmokeTests.dpr

What it verifies: Exercises the F M X Wrap contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.FMX, DAT.Runtime.LanguagePack

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FormScanContracts

File: tools\tests\FormScanContracts.dpr

What it verifies: Runs the formal DFM/FMX scan contract fixtures and compares the canonical expected results.

DAT units exercised directly: DAT.Core.Types, DAT.Scan.Context, DAT.Scan.DomainProfile, DAT.Scan.FormText, DAT.Scan.Quality, DAT.Scan.Rules, DAT.Scan.TextCodec, DAT.Scan.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

FoundationSmokeTests

File: tools\tests\FoundationSmokeTests.dpr

What it verifies: Covers foundational types, JSON, locale, workspace, validation, and runtime-pack construction contracts.

DAT units exercised directly: DAT.Core.AITranslation, DAT.Core.AtomicFile, DAT.Core.CatalogJson, DAT.Core.Glossary, DAT.Core.Hyphenation, DAT.Core.ProjectDetection, DAT.Core.RuntimePack, DAT.Core.Terminology, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Integration.BuildDeploy, DAT.Integration.ComponentPackage, DAT.Integration.DelphiSource, DAT.Integration.MenuResource, DAT.Integration.Package, DAT.Provider.Batching, DAT.Provider.CalendarTerms, DAT.Provider.Client, DAT.Provider.LanguageCodes, DAT.Provider.Placeholders, DAT.Provider.Retry, DAT.Provider.Types, DAT.Review.ApplicationStrings, DAT.Review.CodeGeometry, DAT.Review.Localization, DAT.Review.TextMeasurement, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference, DAT.Scan.CatalogMerge, DAT.Scan.Context, DAT.Scan.DomainProfile, DAT.Scan.FormText, DAT.Scan.PascalResources, DAT.Scan.Project, DAT.Scan.Quality, DAT.Scan.Rules, DAT.Scan.TextCodec, DAT.Scan.Types, DAT.Validation.Catalog

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

GlossaryPublishSmokeTests

File: tools\tests\GlossaryPublishSmokeTests.dpr

What it verifies: Exercises the Glossary Publish contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.AtomicFile, DAT.Core.CatalogJson, DAT.Core.Diagnostics, DAT.Core.Glossary, DAT.Core.Hyphenation, DAT.Core.LocaleFacts, DAT.Core.RuntimePack, DAT.Core.TranslationWorkspace, DAT.Core.Types, DAT.Integration.BuildDeploy, DAT.Runtime.LanguagePack, DAT.Validation.Catalog

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

HyphenationSmokeTests

File: tools\tests\HyphenationSmokeTests.dpr

What it verifies: Exercises the Hyphenation contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.Hyphenation, DAT.Core.RuntimePack, DAT.Core.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

LanguageCodeSmokeTests

File: tools\tests\LanguageCodeSmokeTests.dpr

What it verifies: Exercises the Language Code contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Provider.LanguageCodes

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

LanguageManagerCoreTests

File: tools\tests\LanguageManagerCoreTests.dpr

What it verifies: Exercises the Language Manager Core contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

LayoutContracts

File: tools\tests\LayoutContracts.dpr

What it verifies: Exercises the Layout Contracts contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.CatalogJson, DAT.Core.Types, DAT.Review.CodeGeometry, DAT.Review.Localization, DAT.Review.TextMeasurement, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI, DAT.Scan.TextCodec

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

LayoutFittingSmokeTests

File: tools\tests\LayoutFittingSmokeTests.dpr

What it verifies: Exercises the Layout Fitting contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.CatalogJson, DAT.Core.Types, DAT.Review.CodeGeometry, DAT.Review.Localization, DAT.Review.TextMeasurement, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI, DAT.Scan.TextCodec

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

LayoutOverrideSmokeTests

File: tools\tests\LayoutOverrideSmokeTests.dpr

What it verifies: Exercises the Layout Override contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.LayoutOverrides, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

PackLayoutSmokeTests

File: tools\tests\PackLayoutSmokeTests.dpr

What it verifies: Exercises the Pack Layout contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.Hyphenation, DAT.Core.RuntimePack, DAT.Core.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

PlaceholderSmokeTests

File: tools\tests\PlaceholderSmokeTests.dpr

What it verifies: Exercises the Placeholder contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Provider.Placeholders

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

ProposalDecisionSmokeTests

File: tools\tests\ProposalDecisionSmokeTests.dpr

What it verifies: Exercises the Proposal Decision contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.CatalogJson, DAT.Core.Types, DAT.Review.CodeGeometry, DAT.Review.Localization, DAT.Review.TextMeasurement, DAT.Review.TextMeasurement.FMX, DAT.Review.TextMeasurement.GDI, DAT.Scan.TextCodec

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

ProviderGlossarySmokeTests

File: tools\tests\ProviderGlossarySmokeTests.dpr

What it verifies: Exercises the Provider Glossary contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.Glossary, DAT.Core.Types, DAT.Provider.Glossary

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

RetrySmokeTests

File: tools\tests\RetrySmokeTests.dpr

What it verifies: Exercises the Retry contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Provider.Retry

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

ScannerSmokeTests

File: tools\tests\ScannerSmokeTests.dpr

What it verifies: Exercises the Scanner contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.Types, DAT.Scan.Context, DAT.Scan.DomainProfile, DAT.Scan.PascalResources, DAT.Scan.Quality, DAT.Scan.Rules, DAT.Scan.TextCodec, DAT.Scan.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

StudioFormSmokeTests

File: tools\tests\StudioFormSmokeTests.dpr

What it verifies: Streams and creates the production Studio forms to detect missing components, invalid properties, form-resource regressions, missing Glossary/Cancel event wiring, incomplete language lists, and workflow pages or actions that do not fit or activate.

DAT units exercised directly: DAT.Studio.LocalizationReview, DAT.Studio.MainForm, DAT.Studio.SetupWizard

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

TemplateRewriteSmokeTests

File: tools\tests\TemplateRewriteSmokeTests.dpr

What it verifies: Exercises the Template Rewrite contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.TemplateRewrite

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

TranslationMemorySmokeTests

File: tools\tests\TranslationMemorySmokeTests.dpr

What it verifies: Exercises the Translation Memory contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Core.Glossary, DAT.Core.Interchange, DAT.Core.TranslationMemory, DAT.Core.Types

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLCaptionInterceptSmokeTests

File: tools\tests\VCLCaptionInterceptSmokeTests.dpr

What it verifies: Exercises the V C L Caption Intercept contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.TemplateRewrite, DAT.Runtime.TemplateRewrite.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLDesignStreamingTests

File: tools\tests\VCLDesignStreamingTests.dpr

What it verifies: Exercises the V C L Design Streaming contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.VCL, DAT.Components.VCL.LanguageSelector, DAT.Core.AtomicFile, DAT.Core.Diagnostics, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLDiscoverySmokeTests

File: tools\tests\VCLDiscoverySmokeTests.dpr

What it verifies: Exercises the V C L Discovery contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.VCL, DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLLanguageManagerTests

File: tools\tests\VCLLanguageManagerTests.dpr

What it verifies: Exercises the V C L Language Manager contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.VCL, DAT.Core.AtomicFile, DAT.LifecycleSpike.Trace, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLManagerLifecycleSpikeTests

File: tools\tests\VCLManagerLifecycleSpikeTests.dpr

What it verifies: Exercises the V C L Manager Lifecycle Spike contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.VCL.Spike, DAT.LifecycleSpike.Trace, DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLManagerMDILifecycleSpikeTests

File: tools\tests\VCLManagerMDILifecycleSpikeTests.dpr

What it verifies: Exercises the V C L Manager M D I Lifecycle Spike contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.VCL.Spike, DAT.LifecycleSpike.Trace, DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLRightToLeftSmokeTests

File: tools\tests\VCLRightToLeftSmokeTests.dpr

What it verifies: Exercises the V C L Right To Left contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLRuntimeSmokeTests

File: tools\tests\VCLRuntimeSmokeTests.dpr

What it verifies: Exercises the V C L Runtime contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLSplashHookSmokeTests

File: tools\tests\VCLSplashHookSmokeTests.dpr

What it verifies: Exercises the V C L Splash Hook contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.Preference, DAT.Runtime.SplashTranslation, DAT.Runtime.SplashTranslation.VCL, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLSplashSmokeTests

File: tools\tests\VCLSplashSmokeTests.dpr

What it verifies: Exercises the V C L Splash contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Components.Core, DAT.Components.VCL, DAT.Runtime.LanguagePack, DAT.Runtime.Manager, DAT.Runtime.Preference, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLStatusPanelInterceptTests

File: tools\tests\VCLStatusPanelInterceptTests.dpr

What it verifies: Exercises the V C L Status Panel Intercept contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

VCLWrapSmokeTests

File: tools\tests\VCLWrapSmokeTests.dpr

What it verifies: Exercises the V C L Wrap contract as an isolated Delphi console or GUI regression program.

DAT units exercised directly: DAT.Runtime.LanguagePack, DAT.Runtime.VCL

Why it is important: This executable isolates one contract so a regression produces a focused failure rather than being hidden inside a full application run.

Required use: Compile and run it through the maintained verification scripts or documented build matrix whenever its named subsystem changes.

24.2 Test-support Pascal units

FMXDesignHost.pas

File: tools\tests\design\FMXDesignHost.pas

Program or unit: FMXDesignHost

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.Components.FMX, DAT.Components.FMX.LanguageSelector

VCLDesignHost.pas

File: tools\tests\design\VCLDesignHost.pas

Program or unit: VCLDesignHost

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.Components.VCL, DAT.Components.VCL.LanguageSelector

DAT.LifecycleSpike.Trace.pas

File: tools\tests\lifecycle\DAT.LifecycleSpike.Trace.pas

Program or unit: DAT.LifecycleSpike.Trace

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

FMXLifecycle.Form.pas

File: tools\tests\lifecycle\FMXLifecycle.Form.pas

Program or unit: FMXLifecycle.Form

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.LifecycleSpike.Trace

FMXLifecycle.InheritedForm.pas

File: tools\tests\lifecycle\FMXLifecycle.InheritedForm.pas

Program or unit: FMXLifecycle.InheritedForm

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

VCLLifecycle.Form.pas

File: tools\tests\lifecycle\VCLLifecycle.Form.pas

Program or unit: VCLLifecycle.Form

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.LifecycleSpike.Trace

VCLLifecycle.InheritedForm.pas

File: tools\tests\lifecycle\VCLLifecycle.InheritedForm.pas

Program or unit: VCLLifecycle.InheritedForm

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

VCLLifecycle.MDIChild.pas

File: tools\tests\lifecycle\VCLLifecycle.MDIChild.pas

Program or unit: VCLLifecycle.MDIChild

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.LifecycleSpike.Trace

VCLLifecycle.MDIMain.pas

File: tools\tests\lifecycle\VCLLifecycle.MDIMain.pas

Program or unit: VCLLifecycle.MDIMain

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: DAT.LifecycleSpike.Trace

24.3 Formal scan/layout Pascal fixtures

35_code_positioned_control_is_left_alone.pas

File: contracts\layout\35_code_positioned_control_is_left_alone.pas

Program or unit: code_positioned_control_is_left_alone_fmx

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

vcl_34_code_positioned_control_is_left_alone.pas

File: contracts\layout\vcl_34_code_positioned_control_is_left_alone.pas

Program or unit: vcl_34_code_positioned_control_is_left_alone

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

scan_fixture.pas

File: contracts\pascalscan\scan_fixture.pas

Program or unit: ScanFixture

Purpose: Support source or fixture used by lifecycle, design-streaming, scan, or layout contract tests; it is not shipped as an application runtime dependency.

DAT dependencies: None; this fixture supplies target behavior or source text for another test.

25. Delphi Project and Package File Reference

> **Which files start builds and packages?** The DPR starts a program. The DPROJ supplies RAD Studio build metadata. DPK files define packages and their contained units. A change to one can alter what is built even when Pascal implementation code is unchanged.

25.1 Application, command-line, and package entry points

DelphiAppTranslationStudio.dpr

File: DelphiAppTranslationStudio.dpr

What it is: FMX application entry point that initializes the Studio translation runtime and creates the main Studio form.

Why it matters: Defines process startup ordering and the first form/language initialization boundary.

DelphiAppTranslationStudio.dproj

File: DelphiAppTranslationStudio.dproj

What it is: RAD Studio project metadata, unit search paths, platforms/configurations, resources, outputs, and main form identity.

Why it matters: A build can compile different source or write to a different output when DPROJ configuration drifts, even if DPR is unchanged.

DATBatch.dpr

File: tools\DATBatch.dpr

What it is: Command-line/batch entry point for supported noninteractive translation operations.

Why it matters: Provides automation without coupling scripts directly to Studio form classes.

DATLanguageManagerFMXDesign.dpk

File: packages\design\DATLanguageManagerFMXDesign.dpk

Package kind: Design-time

Requires: rtl, fmx, designide, DATLanguageManagerCoreRuntime, DATLanguageManagerFMXRuntime

Contains: DAT.Design.FMX.Register in '..\..\source\design\DAT.Design.FMX.Register.pas

Why it exists: Separates framework-neutral, framework-runtime, and IDE-only registration code so deployed applications never depend on design-time units and RAD Studio can stream the published components correctly.

Companion DPROJ: packages\design\DATLanguageManagerFMXDesign.dproj - platform/configuration/output metadata for this package.

DATLanguageManagerVCLDesign.dpk

File: packages\design\DATLanguageManagerVCLDesign.dpk

Package kind: Design-time

Requires: rtl, vcl, designide, DATLanguageManagerCoreRuntime, DATLanguageManagerVCLRuntime

Contains: DAT.Design.VCL.Register in '..\..\source\design\DAT.Design.VCL.Register.pas

Why it exists: Separates framework-neutral, framework-runtime, and IDE-only registration code so deployed applications never depend on design-time units and RAD Studio can stream the published components correctly.

Companion DPROJ: packages\design\DATLanguageManagerVCLDesign.dproj - platform/configuration/output metadata for this package.

DATLanguageManagerCoreRuntime.dpk

File: packages\runtime\DATLanguageManagerCoreRuntime.dpk

Package kind: Runtime

Requires: rtl

Contains: DAT.Core.AtomicFile in '..\..\source\core\DAT.Core.AtomicFile.pas, DAT.Core.Diagnostics in '..\..\source\core\DAT.Core.Diagnostics.pas, DAT.Runtime.LanguagePack in '..\..\source\runtime\DAT.Runtime.LanguagePack.pas, DAT.Runtime.Preference in '..\..\source\runtime\DAT.Runtime.Preference.pas, DAT.Runtime.LayoutOverrides in '..\..\source\runtime\DAT.Runtime.LayoutOverrides.pas, DAT.Runtime.Manager in '..\..\source\runtime\DAT.Runtime.Manager.pas, DAT.Runtime.SplashTranslation in '..\..\source\runtime\DAT.Runtime.SplashTranslation.pas, DAT.Runtime.TemplateRewrite in '..\..\source\runtime\DAT.Runtime.TemplateRewrite.pas, DAT.Components.Core in '..\..\source\components\DAT.Components.Core.pas

Why it exists: Separates framework-neutral, framework-runtime, and IDE-only registration code so deployed applications never depend on design-time units and RAD Studio can stream the published components correctly.

Companion DPROJ: packages\runtime\DATLanguageManagerCoreRuntime.dproj - platform/configuration/output metadata for this package.

DATLanguageManagerFMXRuntime.dpk

File: packages\runtime\DATLanguageManagerFMXRuntime.dpk

Package kind: Runtime

Requires: rtl, fmx, DATLanguageManagerCoreRuntime

Contains: DAT.Runtime.FMX in '..\..\source\runtime\DAT.Runtime.FMX.pas, DAT.Runtime.SplashTranslation.FMX in

'..\..\source\runtime\DAT.Runtime.SplashTranslation.FMX.pas, DAT.Runtime.TemplateRewrite.FMX in '..\..\source\runtime\DAT.Runtime.TemplateRewrite.FMX.pas, DAT.Components.FMX in '..\..\source\components\DAT.Components.FMX.pas, DAT.Components.FMX.LanguageSelector in '..\..\source\components\DAT.Components.FMX.LanguageSelector.pas

Why it exists: Separates framework-neutral, framework-runtime, and IDE-only registration code so deployed applications never depend on design-time units and RAD Studio can stream the published components correctly.

Companion DPROJ: packages\runtime\DATLanguageManagerFMXRuntime.dproj - platform/configuration/output metadata for this package.

DATLanguageManagerVCLRuntime.dpk

File: packages\runtime\DATLanguageManagerVCLRuntime.dpk

Package kind: Runtime

Requires: rtl, vcl, DATLanguageManagerCoreRuntime

Contains: DAT.Runtime.VCL in '..\..\source\runtime\DAT.Runtime.VCL.pas, DAT.Runtime.SplashTranslation.VCL in '..\..\source\runtime\DAT.Runtime.SplashTranslation.VCL.pas, DAT.Runtime.TemplateRewrite.VCL in '..\..\source\runtime\DAT.Runtime.TemplateRewrite.VCL.pas, DAT.Components.VCL in '..\..\source\components\DAT.Components.VCL.pas, DAT.Components.VCL.LanguageSelector in '..\..\source\components\DAT.Components.VCL.LanguageSelector.pas

Why it exists: Separates framework-neutral, framework-runtime, and IDE-only registration code so deployed applications never depend on design-time units and RAD Studio can stream the published components correctly.

Companion DPROJ: packages\runtime\DATLanguageManagerVCLRuntime.dproj - platform/configuration/output metadata for this package.

26. JSON Schema and Form Resource Reference

> **Why do schemas and form resources need their own review?** Schemas define serialized compatibility. FMX and DFM resources define designer-owned layout and published properties. Both are contracts, not incidental generated text.

26.1 JSON schemas

development-project.schema.json

File: source\schemas\development-project.schema.json

What it defines: Delphi App Translation Studio Project

Top-level properties: applicationId, applicationVersion, entries, framework, locale, schemaVersion, sourceLanguage

Required top-level properties: schemaVersion, applicationId, framework, sourceLanguage, entries

Change rule: A schema change requires compatible reader/writer changes, version handling, fixtures, round-trip tests, and release notes; never change emitted structure alone.

layout-proposal.schema.json

File: source\schemas\layout-proposal.schema.json

What it defines: Delphi App Translation runtime layout proposal

Top-level properties: acceptedSafeCount, advisoryOnly, applicationId, languageCode, proposals, runtimeApplication, schemaVersion, targetSourceModified

Required top-level properties: schemaVersion, applicationId, languageCode, advisoryOnly, targetSourceModified, proposals

Change rule: A schema change requires compatible reader/writer changes, version handling, fixtures, round-trip tests, and release notes; never change emitted structure alone.

multilingual-layout-envelope.schema.json

File: source\schemas\multilingual-layout-envelope.schema.json

What it defines: Delphi App Translation multilingual layout envelope

Top-level properties: advisoryOnly, controls, languages, purpose, schemaVersion

Required top-level properties: schemaVersion, purpose, languages, advisoryOnly, controls

Change rule: A schema change requires compatible reader/writer changes, version handling, fixtures, round-trip tests, and release notes; never change emitted structure alone.

project-glossary.schema.json

File: source\schemas\project-glossary.schema.json

What it defines: Delphi App Translation project glossary

Top-level properties: applicationId, schemaVersion, sourceLanguage, targetLanguage, terms

Required top-level properties: schemaVersion, applicationId, sourceLanguage, targetLanguage, terms

Change rule: A schema change requires compatible reader/writer changes, version handling, fixtures, round-trip tests, and release notes; never change emitted structure alone.

runtime-language-pack.schema.json

File: source\schemas\runtime-language-pack.schema.json

What it defines: Delphi App Translation Runtime Language Pack

Top-level properties: applicationId, applicationVersion, fontColors, framework, language, layout, locale, schemaVersion, sourceCatalogChecksum, sourceLanguage, sourceStrings, sourceTemplates, sources, strings, templates

Required top-level properties: schemaVersion, applicationId, framework, sourceLanguage, sourceCatalogChecksum, language, strings

Change rule: A schema change requires compatible reader/writer changes, version handling, fixtures, round-trip tests, and release notes; never change emitted structure alone.

26.2 Production FMX form resources

DAT.Studio.MainForm.fmx

File: source\studio\DAT.Studio.MainForm.fmx

What it is: Designer-authored landing screen and eight Maintenance pages, including the standalone Glossary page, expanded navigation rail, persistent Maintenance Cancel button, dialogs, status, and control defaults.

Why it matters: It is the editable IDE layout authority paired with the same-base-name PAS class. Runtime code must not replace this designer ownership without explicit approval.

DAT.Studio.SetupWizard.fmx

File: source\studio\DAT.Studio.SetupWizard.fmx

What it is: Designer-authored eight-step Wizard, step rail, page controls, keyboard defaults, authorization, and final-processing layout.

Why it matters: It is the editable IDE layout authority paired with the same-base-name PAS class. Runtime code must not replace this designer ownership without explicit approval.

DAT.Studio.LocalizationReview.fmx

File: source\studio\DAT.Studio.LocalizationReview.fmx

What it is: Designer-authored findings/glossary/suggestions/layout-decision review window.

Why it matters: It is the editable IDE layout authority paired with the same-base-name PAS class. Runtime code must not replace this designer ownership without explicit approval.

26.3 Test form resources

VCLDesignHost.dfm

File: tools\tests\design\VCLDesignHost.dfm

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

VCLLifecycle.Form.dfm

File: tools\tests\lifecycle\VCLLifecycle.Form.dfm

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

VCLLifecycle.InheritedForm.dfm

File: tools\tests\lifecycle\VCLLifecycle.InheritedForm.dfm

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

VCLLifecycle.MDIChild.dfm

File: tools\tests\lifecycle\VCLLifecycle.MDIChild.dfm

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

VCLLifecycle.MDIMain.dfm

File: tools\tests\lifecycle\VCLLifecycle.MDIMain.dfm

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

FMXDesignHost.fmx

File: tools\tests\design\FMXDesignHost.fmx

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

FMXLifecycle.Form.fmx

File: tools\tests\lifecycle\FMXLifecycle.Form.fmx

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

FMXLifecycle.InheritedForm.fmx

File: tools\tests\lifecycle\FMXLifecycle.InheritedForm.fmx

What it is: Designer resource streamed by a lifecycle or design test.

Why it matters: Proves published properties, inheritance, object names, lifecycle behavior, and form-resource compatibility.

27. Build, Verification, and Documentation Tool Reference

> **Which tool should be run for a given engineering task?** This chapter explains the maintained build, verification, distribution, and documentation scripts. Prefer these repeatable entry points to improvised one-off commands.

27.1 Build, verification, distribution, and guide tools

build_packages.ps1

File: tools\build_packages.ps1

What it is: Builds the five runtime/design package projects in controlled platform/configuration order.

Why it matters: Package installation and ComponentSource correctness depend on reproducible package outputs.

verify_all.ps1

File: tools\verify_all.ps1

What it is: Runs the broad engineering verification suite.

Why it matters: Primary pre-release gate across foundation, scanner, runtime, components, Studio, and artifacts.

check_shipped_units_complete.ps1

File: tools\check_shipped_units_complete.ps1

What it is: Compares required runtime/component units with shipped packages/kits/distributions.

Why it matters: Prevents a compile/runtime failure caused by an omitted dependency.

check_build_paths_agree.ps1

File: tools\check_build_paths_agree.ps1

What it is: Checks project, script, and documentation output/search-path agreement.

Why it matters: Prevents testing one binary while shipping another.

check_source_encoding.ps1

File: tools\check_source_encoding.ps1

What it is: Audits source encoding and known text hazards.

Why it matters: Scanner/provider/UI correctness depends on stable Unicode source.

build_source_distribution.ps1

File: tools\build_source_distribution.ps1

What it is: Creates the source-only distribution with approved files and structure.

Why it matters: Distribution must be complete while excluding secrets, transient workspaces, and unintended binaries.

run_form_scan_contracts.ps1

File: tools\run_form_scan_contracts.ps1

What it is: Builds/runs formal DFM/FMX scan contracts.

Why it matters: Protects canonical scan keys, counts, inclusion/exclusion, and context behavior.

run_layout_contracts.ps1

File: tools\run_layout_contracts.ps1

What it is: Builds/runs layout and direction contracts.

Why it matters: Protects designer ownership, wrapping, fitting, RTL, and code-positioned exclusions.

run_pascal_scan_contracts.ps1

File: tools\run_pascal_scan_contracts.ps1

What it is: Builds/runs Pascal resourcestring/runtime-assignment contracts.

Why it matters: Protects text extraction without executing target source or translating arbitrary literals.

Install-DATLanguageManagerComponents.ps1

File: tools\Install-DATLanguageManagerComponents.ps1

What it is: Builds/locates and guides installation of design components.

Why it matters: Keeps Tool Palette setup repeatable without copying arbitrary BPLs into RAD Studio folders.

render_guides_pdf.py

File: tools\render_guides_pdf.py

What it is: Creates alternate print PDFs from guide content through an HTML and Playwright rendering path.

Why it matters: Provides a separate print-rendering and page-map utility; it is not the maintained LibreOffice release-export path.

build_user_guide.py

File: tools\build_user_guide.py

What it is: Builds the detailed source-driven User Guide and its static printable TOC.

Why it matters: Documents actual application behavior and provides deterministic regeneration.

build_wizard_engineering_guides.py

File: tools\build_wizard_engineering_guides.py

What it is: Builds this Wizard Guide and Engineering Guide from current source inventories and curated architecture descriptions.

Why it matters: Keeps exhaustive file/dependency reference synchronized with repository content.

finalize_guides.ps1

File: tools\finalize_guides.ps1

What it is: Exports the maintained companion PDFs directly from the editable DOCX guides through LibreOffice.

Why it matters: Keeps the release PDFs tied to the reviewed DOCX sources and makes finalization repeatable.

27.2 Important engineering reference documents

Total Stabilization Contracts.md

File: docs\guides\Total Stabilization Contracts.md

What it is: Canonical cross-cutting runtime, layout, language-switch, and universal-fix contracts.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

Translation Studio Completed Test Report and Remediation Plan.md

File: docs\guides\Translation Studio Completed Test Report and Remediation Plan.md

What it is: Evidence, resolved defects, residual risks, and verification results.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

TDATLanguageManager Deep-Dive and Implementation Plan.md

File: docs\guides\TDATLanguageManager Deep-Dive and Implementation Plan.md

What it is: Manager architecture and staged implementation rationale.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

STAGES-1-3-REVIEW-AND-SCANNING-ENGINEERING-NOTES.md

File: docs\STAGES-1-3-REVIEW-AND-SCANNING-ENGINEERING-NOTES.md

What it is: Detailed scan/context/review engineering notes.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

STAGES-4-10-RUNTIME-LAYOUT-ENGINEERING-NOTES.md

File: docs\STAGES-4-10-RUNTIME-LAYOUT-ENGINEERING-NOTES.md

What it is: Detailed runtime/layout/direction/stabilization engineering notes.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

Release Checklist.md

File: docs\guides\Release Checklist.md

What it is: Human release verification checklist complementing automated tests.

Why it matters: Read before changing the named subsystem; these records explain contracts that a source signature alone cannot express.

28. Dependency Graph Reading Guide

How do dependencies show change impact? A direct dependency is imported by a unit. A direct consumer imports that unit. Trace both directions, then consider package and generated-source closure before deciding that a change is local.

Every production unit entry in Chapter 23 includes direct DAT dependencies and direct production consumers computed from current source. A dependency means the unit imports another DAT unit; a consumer means another production unit imports it. The list is direct, not the complete transitive closure.

Leaf/foundation units may have no DAT dependencies but many consumers. Adapter/entry units may have many dependencies and no production consumer because a DPR/DPK or application code selects them. Test consumers show named DPRs that import the unit directly; indirect coverage is broader.

Graph signal	Likely meaning	Required engineering response
Many consumers	Shared API or foundational contract.	Treat signature/behavior changes as cross-subsystem and run full verification.
Framework-neutral unit imports VCL/FMX	Layer violation.	Move framework behavior into the applicator/component adapter.
Runtime imports provider/studio	Offline trust-boundary violation.	Remove network/UI dependency from shipped runtime.
Design unit in runtime package	IDE-only code can leak into deployment.	Correct DPK containment/requirements.
ComponentSource missing a direct/transitive dependency	Generated kit cannot compile consistently.	Fix closure computation and shipped-unit check; do not hand-copy one file.
No named tests for high-risk unit	Coverage may be only indirect.	Add focused contract test when behavior cannot be proven by existing subsystem tests.

29. Safe Change Procedure

What is the safest order for making a change? Define the contract, trace ownership, make the smallest universal correction, run focused checks, run the full gate, test real VCL and FMX applications, update documentation, and review the final Git diff.

1. Define the contract and affected ownership layer before editing. Read the relevant reference document and unit entries in Chapter 23.
2. Check Git status and remote currency; preserve unrelated work. Make a pre-change backup for material changes.
3. Trace direct dependencies, consumers, package membership, ComponentSource closure, schemas, and tests. Do not edit only the first file named by a compiler error.
4. Make the smallest universal change. Do not add named-project/version exceptions, runtime UI construction, or helper layers without explicit approval.
5. Update schema/version or compatibility handling when serialized contracts change. Preserve atomic persistence and older valid recovery files.
6. Compile affected foundation/runtime/package/Studio targets and run focused tests first, then the complete verification suite.
7. Run real VCL and FMX pilot applications across LTR/RTL, language-to-language, switch-back, restart, dynamic/HTML, layout, and supported platforms.
8. Update User, Wizard, Engineering, help, release, and test documentation as applicable. Render and visually inspect DOCX/PDF outputs.
9. Review Git diff, stage only intended files, commit clearly, and push configured public/private remotes without rewriting history.

30. Engineering Release Checklist

What must be true before release? The checklist converts architecture promises into evidence: builds, tests, pack compatibility, state preservation, direction, dynamic content, deployment, security, distribution, and documentation.

- [ ] Product invariants and trust boundaries remain true.
- [ ] All production PAS units compile in their intended packages/projects.
- [ ] Five package projects build; Win32 Release design BPLs stream components in RAD Studio.
- [ ] Complete ComponentSource closure contains matching shared and framework units.
- [ ] Development catalogs, settings, preferences, glossaries, layout overrides, and packs pass atomic/recovery tests.

- [ ] GlossaryPublishSmokeTests proves catalog correction, canonical/dependency pack identity, complete deployment to existing Win32/Win64 Debug/Release outputs, unchanged DPROJ content, and no executable build or replacement.
- [ ] Canonical scanner totals agree between Wizard and Maintenance for identical inputs.
- [ ] Provider timeout, cancellation, retries, placeholders, language codes, quota/auth errors, and credential isolation pass.
- [ ] Localization Review findings, glossary, suggestions, layout decisions, HTML package, and resume continuation pass.
- [ ] Validation blocks structural errors without pretending to approve language quality.
- [ ] Pack admission rejects wrong application/framework/version/checksum/source contract.
- [ ] VCL and FMX preserve state and designer baseline through every language transition.
- [ ] LTR, RTL, protected LTR tokens, space wrapping, intelligent hyphenation, readable font fitting, and neighbor alignment pass.
- [ ] English/source switch-back repaints current forms/HTML immediately and previous-language content is absent.
- [ ] Startup preference, missing/stale pack fallback, splash, templates, later forms, dialogs, menus, grids, reports, and dynamic text pass.
- [ ] Build/deployment output and actual running executable folder contain the identical intended pack set.
- [ ] Source distribution excludes secrets/transient work and includes required source, packages, schemas, tools, guides, and licenses.
- [ ] Documentation is source-current, rendered, visually inspected, accessible, committed, and pushed.

31. Quick Path and Identifier Reference

Where are the most-used engineering paths? This is a lookup page for the project, compiler, source layers, schemas, tests, workspaces, generated kits, deployed packs, and credential targets.

Purpose	Path / identifier
Main project	DelphiAppTranslationStudio.dproj / DelphiAppTranslationStudio.dpr
RAD environment	C:\Program Files (x86)\Embarcadero\Studio\37.0\bin\rsvars.bat
Win32 compiler	C:\Program Files (x86)\Embarcadero\Studio\37.0\bin\dcc32.exe
Production source	source\core, scan, provider, review, validation, runtime, components, design, integration, studio
Schemas	source\schemas
Packages	packages\runtime and packages\design
Tests	tools\tests
Release verification	tools\verify_all.ps1
Studio settings	%LOCALAPPDATA%\DelphiAppTranslationStudio
Project workspace	%LOCALAPPDATA% \DelphiAppTranslationStudio\Workspaces\<ApplicationId>
Generated kit	export\component-integration\<ApplicationId>
Review package	export\localization-review\<ApplicationId>\<locale>
Vendored target dependencies	<Target>\dependencies\DelphiAppTranslation\source
Target packs	<EXE>\Localization\Languages
DeepL credential target	DelphiAppTranslationStudio/Providers/DeepL

Purpose	Path / identifier
Main project	DelphiAppTranslationStudio.dproj / DelphiAppTranslationStudio.dpr
Google credential target	DelphiAppTranslationStudio/Providers/Google Cloud Translation

32. Appendix A - License Information

Unless a particular file or third-party component states otherwise, Delphi App Translation Studio and its project documentation are licensed under the Apache License, Version 2.0. The license permits broad use while preserving attribution, notices, and the stated warranty limitations.

License grant

Subject to the complete license terms, you may use, reproduce, modify, prepare derivative works from, publicly display, publicly perform, sublicense, and distribute the licensed material. The Apache License 2.0 also includes an express patent license from contributors for their applicable contributions.

Important conditions

- Give recipients a copy of the Apache License 2.0 when distributing covered material.
- State significant changes made to modified files.
- Retain applicable copyright, patent, trademark, attribution, and NOTICE information.
- Do not use project or contributor names and trademarks as an endorsement except as the license permits.

Warranty and liability

The licensed material is provided on an AS IS basis, without warranties or conditions of any kind. The complete Apache License 2.0 controls the warranty, liability, contribution, redistribution, and patent provisions.

Your applications and generated output

Using the Studio does not transfer ownership of the Delphi application being translated. The application developer remains responsible for the license and distribution terms of the target application, translated content, generated language packs, and any third-party components.

Studio runtime or component source included with a project remains subject to the Apache License 2.0 unless its file states different terms. Third-party software retains its own license.

Full legal text

The complete controlling license is the LICENSE file in the repository and source distribution root. An official reference copy is available at https://www.apache.org/licenses/LICENSE-2.0. If this summary and the complete license differ, the complete Apache License 2.0 text controls.