QUANTO USER GUIDE

Creating add-ins

Data provider, method importer, and integrator contracts; load folders and bootstrap.

Quanto loads **three kinds of add-ins** at startup from folders next to the app. Drop a DLL in the right folder; no registry keys.

Kind	Folder	Contract	Job
Data provider	`AddIns\DataProvider\`	`IDataProvider`	Find samples, read bottle metadata, extract chromatograms
Method importer	`AddIns\MethodImporter\`	`IMethodImporter`	Find and parse vendor methods into compounds / signals / levels
Integrator	`AddIns\Integrator\`	`IIntegrator`	Detect peaks on `x[]` / `y[]`

Contracts live in `Quanto.Shared.Core` (`Quanto.Shared.Core.Interfaces`). The host loader is `Quanto.AddInFactory.AddInLoader`.

How loading works

1. At startup the host calls `AddInLoader.Load(integratorPath, dataProviderPath, methodImporterPath)`.
2. Each folder is scanned for `*.dll`. Assemblies are `Assembly.LoadFrom`.
3. Instantiation is **lazy** — constructors (and heavy vendor init) run on first use, not at discovery.
4. Registration key is the **concrete class name** (e.g. `AgilentMassHunterDataProvider`, `Quanto`, `SciexWiffMethod`).
5. Optional `IAddIn`: `AlternativeDllPath` (vendor DLL search folder) and `Version` (for logs).
6. Optional **bootstrap** class: a tiny type with an `AlternativeDllPath` property (conventionally named `AddInBootstrap`, or any `*Bootstrap`). Instantiated first so vendor paths are registered **before** reflecting types that reference vendor assemblies.

Paths (from `AppPaths`):

```
text {app}\AddIns\ DataProvider\ MethodImporter\ Integrator\ VendorSupportFiles\Agilent\ VendorSupportFiles\Sciex\
```

Relative `AlternativeDllPath` values are resolved from the application directory (examples from shipped add-ins: `AddIns\VendorSupportFiles\Agilent`, `AddIns\VendorSupportFiles\Sciex`).

Shipped projects copy only their **own** primary DLL into the staging add-in folder; vendor natives stay under `VendorSupportFiles\…`.

Project setup

- Target **Windows x64** and the same Quanto Windows TFM as the host (.NET 10 Windows).
- Reference `Quanto.Shared.Core` only for contracts and shared types/enums (do not take a hard dependency on `Quanto.WPF`).
- Implement the matching interface(s) on a **public non-abstract class**.

- Optionally implement `IAddIn`.
- Add a dependency-free bootstrap type if you need vendor DLLs on the probe path before your main type loads.
- Build output: one add-in DLL into the correct `AddIns\…` subfolder.

Example bootstrap (duck-typed; implementing `IAddInBootstrap` is fine but not required for discovery):

```csharp namespace MyVendorData;

public sealed class AddInBootstrap { public string? AlternativeDllPath => @"AddIns\VendorSupportFiles\MyVendor"; } ```

## Data provider

`IDataProvider` extends:

- `ISampleFinder` — `HashSet<SampleFile> FindSamples(string path)`
- `ISampleFileReader` — `void ReadSampleFile(SampleFile sampleFile)`
- `ISignalReader` — chromatogram extraction

```csharp public interface ISignalReader { SignalPoints GetSignalPoints( string filePath, double rt, double beginOffsetTime, double endOffsetTime, double mass1, double beginOffsetMass1, double endOffsetMass1, double mass2, double beginOffsetMass2, double endOffsetMass2, IonPolarity? ionPolarity = null, double? fragmentorVoltage = null, double? collisionEnergy = null, ChromatogramAcquisitionMode preferredMode = ChromatogramAcquisitionMode.Auto);

`SignalPoints GetTotalSignal(string filePath);`
`// optional override:`
`SignalPoints GetTotalSignal(string filePath, ChromatogramAcquisitionMode mode);`

} ```

`ChromatogramAcquisitionMode`: `Auto`, `Scan`, `Sim`, `MsMs`.

`FindSamples` should return `SampleFile` instances with `DataProvider` set to this instance and fill what you can (name, AcqDateTime, instrument, position, dilution, dose hints). The host matches sample type / dose from the method afterward.

**Optional host hooks** (implement if useful): `IMrmTransitionCatalog`, `IDataFilePrefetch`, `IDataFileRelease`.

### Reference implementations

| Class | Project | Format |
|---|---|---|
| `AgilentMassHunterDataProvider` | `Quanto.AddIn.DataProvider.AgilentMHData` | MassHunter `.D` |
| `AgilentChemStationDataProvider` | `Quanto.AddIn.DataProvider.AgilentCSData` | ChemStation `.D` (native MSDChem) |
| `SciexWiff` | `Quanto.AddIn.DataProvider.SciexWiff` | SCIEX `.wiff` |

## Method importer

`csharp public interface IMethodImporter { string Name { get; set; } HashSet<MethodFile> FindMethods(string path); void LoadMethod(string filePath); void ReadMethodFile(MethodFile info); ICompound[] GetCompounds(); }`

Map vendor content onto `ICompound` / `ISignal` / `ILevel`:

- Compound: name, group, expected time, begin/end offsets, ISTD flag and link, curve fit/weight/origin, concentration limits, signals, levels.
- Signal: name, `Mass1` / `Mass2`, collision energy, fragmentor, ion polarity, `AcquisitionMode`, quant/qualifier flags, % and tolerance.
- Optional `ICompoundWithAdditionalQuants` when the vendor has extra quant slices (e.g. MassHunter RA variants).

`FindMethods` should attach `MethodImporter = this` on each `MethodFile` and set a display `Name`.

### Reference implementations

| Class | Project | Source |
|---|---|---|
| `AgilentMhMethod` | `Quanto.AddIn.MethodImporter.AgilentMHMethod` | `.m` / `.quantmethod.xml` |
| `AgilentMassHunterDataMethod` | `Quanto.AddIn.MethodImporter.AgilentMHData` | Method from `.D` |
| `AgilentCsMethod` | `Quanto.AddIn.MethodImporter.AgilentCSMethod` | `.M` + `qdb.mth` |
| `SciexWiffMethod` | `Quanto.AddIn.MethodImporter.SciexWiff` | Method from `.wiff` |

## Integrator

`csharp public interface IIntegrator { PeakValues[] Integrate(double[] x, double[] y, string settings); }`

`PeakValues` fields: `BegX`, `EndX`, `BegB`, `EndB` (peak time bounds and baseline height at those bounds).

`settings` is the free-form string from the channel's integrator settings. Parse what you need; ignore unknown keys.

Channels pick an integrator **by class name** (default shipped name: `Quanto`).

### What the integrator returns vs what Quanto calculates

An integrator **only detects peak boundaries**. It does **not** compute area, height, apex time, width, or symmetry.

After your add-in returns `PeakValues[]`, the host always runs **Quanto's shared peak-metric methods** in `Quanto.Shared.Core.PeakCalculations` (`IPeakMetricsCalculator` / `PeakMetricsCalculator`) on the same `x[]` / `y[]` and your `BegX` / `EndX` / `BegB` / `EndB`. That path is used for:

- automatic integration (`ImportPeakValuesService` after any integrator, including custom ones)
- manual re-integration and boundary edits (`PeakIntegrationService`)

Derived metrics include (see `PeakMetrics`):

| Metric | Meaning |
|---|---|
| `Area` | Trapezoidal integral of the **baseline-corrected** signal |
| `Height` | Apex height above the linear baseline |
| `CenterX` / `Time` | Apex retention time |
| `MaxSignal` | Apex signal value |
| `FwAt50`, `FwAt5` | Peak widths at 50% and 5% of height |
| `Symmetry` | Peak symmetry from those crossings |
| `BegR`, `EndR` | Residuals / edge refinements from the shared calculator |

Response for quantitation is then **area** or **height** according to the analyte's `QuantitateByHeight` flag — still from these shared metrics, not from the integrator DLL.

So when you write an integrator:

1. Focus on finding good start/end times and baseline points.
2. Return accurate `BegX`, `EndX`, `BegB`, `EndB`.
3. **Do not** reimplement area/height in the add-in; Quanto will calculate them consistently for every integrator (Quanto, Slice, or yours).
4. You may call `PeakMetricsCalculator.Instance` yourself for diagnostics or unit tests (same algorithms), but the host will recalculate when importing peaks into the batch.

### Reference implementations

| Class | Project | Notes |
|---|---|---|
| `Quanto` | `Quanto.AddIn.Integrator.Quanto` | Production peak detector (boundaries only; metrics still via shared calculator) |
| `Slice` | `Quanto.AddIn.Integrator.Slice` | Minimal stub (returns no peaks) — useful as a template |

## Minimal skeletons

### Integrator

```csharp using Quanto.Shared.Core.Interfaces; using Quanto.Shared.Core.Types; // Optional for offline checks: // using Quanto.Shared.Core.PeakCalculations;

namespace MyIntegrator;

public sealed class MyIntegrator : IIntegrator, IAddIn { public string? AlternativeDllPath => null; public string? Version => "1.0.0";

```
public PeakValues[] Integrate(double[] x, double[] y, string settings)
{
    // Detect peaks; return only BegX/EndX/BegB/EndB.
```
```

```
    // Host applies PeakMetricsCalculator for Area, Height, Fw*, Symmetry, etc.
    return [];
}

} ```
```

Data provider / method importer

Follow the same pattern: implement the interface + optional `IAddIn`, discover files under the path the host passes, and return shared types from `Quanto.Shared.Core.Types` / `Enums`. Prefer looking at the matching Agilent or SCIEX project above rather than re-deriving vendor binary formats.

Checklist before shipping an add-in

- [ ] DLL builds x64 / Quanto Windows TFM and copies to the correct `AddIns\…` folder
- [ ] Class is public, non-abstract, and assignable to the right interface
- [ ] Class **name** is unique among loaded add-ins of that kind (registry key)
- [ ] Bootstrap + `AlternativeDllPath` set if vendor natives are required
- [ ] Vendor DLLs installed under `VendorSupportFiles\…` (not duplicated beside every add-in)
- [ ] Constructors avoid crashing the process at first use (lazy load still runs your ctor once)
- [ ] Version visible via `IAddIn.Version` or assembly informational version
- [ ] Smoke-test: host log shows Discovered / instantiated lines; Find / Load / Integrate works on a real file
- [ ] Integrators: confirm Area/Height appear after Analyze (shared `PeakMetricsCalculator`), not only raw boundaries

Related

- Formats and modes: Compatibility
- Install tree: Installation and folders
- Analyst import UI: Reports, import, and export