Architecting a Lab App (MVC)

The GUI guide ended with one class doing three jobs at once: talking to the instrument, deciding what to measure, and drawing the interface. For a small app that's fine. As the app grows — more instruments, a real measurement sequence, a richer interface — that tangle becomes the thing that slows every change: you can't touch the plot without risking the hardware code, and you can't test the measurement logic without a window on screen.

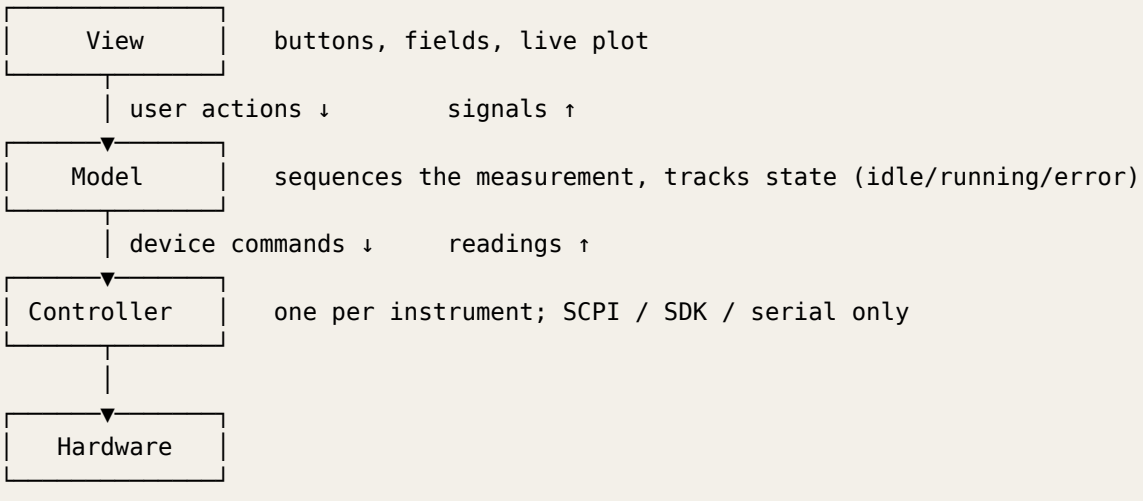
The fix is a deliberate **separation of concerns**, the idea behind the **Model–View–Controller (MVC)** pattern. Split the app into layers, give each one job, and let dependencies flow in only one direction. This page is about the *principles* — they apply whether your interface is a PyQtGraph window, a Tkinter panel, or a web-based UI, and whether the app is 300 lines or 30,000.

> **Prerequisites**
>
> The rest of the building-lab-software guides. This page assumes you've felt the pain a small app creates before reaching for structure to manage a large one.

1 The Layers

Each layer has exactly one responsibility, and never does the others':

Layer	Responsibility	Never does
View	Display data, capture user input	Hardware calls, measurement logic
Model	Orchestrate the experiment, hold state	Draw the GUI, send SCPI commands
Controller	Talk to one piece of hardware	Experiment logic, state management



You've already built two of these layers in earlier guides:

- The **Controller** is the instrument wrapper class — a `Device` subclass that turns SCPI into clean methods and knows *nothing* about your experiment.

- The **Model** is the experiment object from the data-acquisition guide — it owns the acquisition loop and state but doesn't draw anything.
- The **View** is the Qt window.

MVC is mostly the discipline of keeping those three from leaking into each other.

> **The web-UI variant**
>
> When the View is a web interface (HTML/CSS/JS rendered in a Qt WebEngine view rather than native widgets), one more layer appears: an **API/bridge** between View and Model that marshals calls across the JavaScript↔Python boundary. The responsibilities above don't change — there's just an extra translator. If you're using native widgets (PySide6, Tkinter), you don't need it.

2 Dependencies Flow Downward Only

The one rule that makes the rest work:

```
View → Model → Controller → Hardware
```

A layer may call the layer **below** it, never the one above. The Controller doesn't know a Model exists; the Model doesn't know whether its View is a Qt window, a web page, or a test script. Information flows back **up** through signals (or callbacks), not through a layer reaching upward.

Why this specific constraint pays off:

- **Swap a layer without touching the others.** Replace an instrument? Rewrite its Controller to keep the same method names; the Model is untouched. Switch from a Qt window to a web UI? Replace the View; the Model and Controllers don't change.
- **Test without hardware or a screen.** Because the Model only depends downward, you can drive it from a plain script with a mock Controller — no GUI, no instruments — and check that the measurement logic is correct.

3 Data Flow: One Round Trip

Tracing a single user action through the layers shows how they cooperate without knowing each other's internals:

```
1. User clicks "Start"            → View
2. View calls model.start(params) → Model
3. Model starts a worker thread, calls controller.measure(v)  → Controller
4. Controller sends SCPI, returns a reading                   → Controller
5. Model emits a `measurement_point` signal                   → Model
6. Signal crosses to the main thread, View's slot redraws     → View
```

Steps 5–6 are the thread-safety pattern from the GUI guide: the Model runs measurements on a worker thread and reports results **upward via signals**, which Qt delivers on the main thread where the View can safely update. The Model never calls a View method directly — it just emits.

```
from PySide6.QtCore import QObject, Signal

```

```
class ExperimentModel(QObject):
    measurement_point = Signal(float, float)   # (x, y) — the View connects to this
    state_changed = Signal(str)                # "idle" / "running" / "error"

    def __init__(self, controller):
        super().__init__()
        self.controller = controller           # depends downward only

    def run(self, setpoints):
        self.state_changed.emit("running")
        for v in setpoints:
            y = self.controller.measure(v)      # no GUI code here
            self.measurement_point.emit(v, y)   # report upward
        self.state_changed.emit("idle")
```

Notice what's *absent*: no `pyqtgraph`, no `setData`, no SCPI strings. The Model is pure experiment logic, which is exactly why you can test it on its own.

4 Hardware Abstraction and Offline Mode

Because every instrument hides behind a Controller with a fixed set of methods, the Model is written against the *interface*, not the device. That gives you a powerful development convenience: an **offline / mock mode** where Controllers return simulated data, so you can build and test the entire app without hardware connected.

```
class InstrumentController:
    def __init__(self, resource, offline=False):
        self.offline = offline
        if not offline:
            self.dev = Device(resource)

    def measure(self, setpoint: float) -> float:
        if self.offline:
            import random
            return setpoint * 0.95 + random.gauss(0, 0.01)   # a linear stand-in
        return float(self.dev.query(f"MEAS? {setpoint}"))
```

Mock data should be *fast* (no artificial delays) and of the right magnitude. But the property that earns the most is one that is easy to miss: **the mock should have the shape of the real measurement, not just its size.**

The stand-in above is linear. It will happily exercise every line of the View and the Model — and it will tell you nothing about whether your axis limits, your log scale, or your curve fit are right, because a straight line is the one shape that makes all of those look fine. Swap in the physics instead:

```
import numpy as np

class DiodeController:
    """Offline mode returns an ideal diode curve rather than a straight line."""

    I0, N, VT = 1e-12, 1.8, 0.02585      # saturation current, ideality, thermal voltage

    def measure(self, volts: float) -> float:
        if self.offline:
```

```
            current = self.I0 * (np.exp(volts / (self.N * self.VT)) - 1)
            return float(current + np.random.normal(0, max(abs(current), 1e-12) * 0.01))
        return float(self.dev.query(f"MEAS:CURR? {volts}"))
```

Now the mock curve has a knee in the right place and spans decades of current, so a linear y-axis looks obviously wrong, an autoscale that clips the interesting region shows up immediately, and a fit that only converges near the origin fails at a desk instead of on the bench.

The distinction is worth naming: a linear mock proves the app **runs**; a mock with the right shape proves the app is **right**. Only the second one saves you bench time, because only the second one can fail.

With either, the View and Model can be developed away from the lab, and a teammate without bench access can still work on the app. See also Developing Without the Hardware.

5 A Directory That Reflects the Layers

Let the file layout mirror the architecture — it makes the separation visible and hard to violate by accident:

```
my_lab_app/
├── __main__.py           # entry point: wires the layers together and shows the window
├── controllers/
│   └── instrument.py     # Controller(s): hardware communication only
├── models/
│   └── experiment.py     # Model: measurement orchestration and state
└── views/
    ├── main_window.py     # View: widgets, plot, signal/slot wiring
    └── ...
```

`__main__.py` is the only place that knows about all three layers — it constructs a Controller, hands it to a Model, hands the Model to a View, and starts the event loop. Everywhere else, each layer sees only the layer below.

6 When Is This Worth It?

MVC is overhead, and overhead you don't need is a cost, not a virtue. A quick acquisition script or a one-screen control panel should stay a script — splitting a 100-line program into four files helps no one.

Reach for this structure when an app shows the signs of outgrowing a single file:

- More than one instrument, or a real multi-step measurement sequence.
- A GUI you expect to keep extending.
- Logic you want to **test without hardware**, or hardware you expect to **swap**.
- More than one person working on it.

Start simple. Let the structure arrive when the app's complexity asks for it — the layers above are where to go *when* it does, not a checklist for day one.

Back to Python for Lab Work