# Study Guide: Prototyping User Interfaces for Devices

## 1. Overview of Prototyping for Embedded Devices

- **Purpose**: Prototyping UIs refines how users monitor, control, and interpret device behavior before committing to production.
- **Approach spectrum**:
  - **On-device GUIs** (e.g., Qt on a touchscreen).
  - **Web-based GUIs** (HTML/CSS/JS served from or connected to the device).
  - **Non-GUI options** (physical controls, segmented/pixel displays, command lines).
- **Course emphasis**: Practical mechanics of UI construction (widgets, layouts, event wiring, states), not full UX research cycles—though user feedback remains essential.
- **Key idea**: **Micro-interactions**—tiny, focused moments of interaction—often define perceived quality on resource-constrained embedded devices.

## 2. Qt and PyQt

### 2.1 What is Qt?

- A **cross-platform GUI framework** supporting Windows, macOS, Linux, and embedded Linux.
- Rich widget set, layouts, graphics, accessibility, internationalization, and hardware integration.
- Suitable for both **rapid prototypes** and **production applications**.

### 2.2 PyQt (Python bindings for Qt)

- Enables building Qt apps in Python; runs on major OSs and Raspberry Pi OS.
- Commonly paired with **Qt Designer** (visual, drag-and-drop UI builder).

### 2.3 Qt Designer

- Visual editor for placing widgets, organizing layouts, and setting properties.
- Saves interfaces as **.ui design files** that define structure and presentation.
- Two integration pathways: compile a Python representation of the UI, or load the `.ui` at runtime.

### 2.4 Benefits of Qt for Embedded Prototyping

- Fast iteration, strong native look/feel, extensive widget library, and robust event model (signals/slots).
- Supports paper/digital prototypes even before hardware is ready.

### 2.5 Practical Setup Notes

- On Linux/Raspberry Pi, Designer is typically installed under **Qt5/bin** and accessible from system menus.
- Best reliability is on **Pi OS or a Pi desktop VM**; remote desktop/VNC may require configuration tweaks.

### 2.6 Licensing Basics

- **Qt**: open-source (LGPLv3/GPLv3) and **commercial** options.
- **PyQt**: GPL/commercial; **PySide (Qt for Python)** uses LGPL.
- Academic/open-source work usually fits open-source licenses; commercial distribution may require a paid license or adherence to LGPL terms.

## 3. HTML-Based Interfaces

### 3.1 Role and Positioning

- Ideal for **remote access** via browsers (phones, tablets, PCs).
- Can also be **served directly from the device** using a lightweight web server.

### 3.2 Deployment Models

1. **On-device server**: device serves HTML/CSS/JS pages to connected clients.
2. **Remote server + API**: UI hosted elsewhere; communicates with device over a defined API.

### 3.3 Strengths

- Platform-agnostic, easy to share/test, naturally network-ready, excellent for dashboards and control panels.

### 3.4 Challenges

- Requires server/API setup and **security** (authN, authZ, TLS, session management).
- Multiple layers to coordinate (HTML structure, CSS styling, JS behavior).

### 3.5 Core Technologies and Aids

- **HTML** (structure) • **CSS** (layout/typography/responsiveness) • **JavaScript/jQuery** (interactivity, ready-made widgets).
- jQuery provides quick, robust components for prototypes without heavy frameworks.

### 3.6 Learning Resources

- Cheat sheets, tutorials, the jQuery examples gallery, and the book *A Software Engineer Learns HTML, JavaScript, and jQuery* (Dane Cameron).

### 3.7 In-Course Application

- Build simple, clear HTML UIs that interact with devices (often via lightweight APIs).
- Focus on maintainable structure and usable controls; avoid unnecessary complexity.

### 3.8 Extensions

- **React / Angular** (richer front-ends—beyond this course's needs).
- **Flask / Django** (Python backends) when you need device↔web integration beyond static pages.

### 3.9 Key Considerations

- **Security from day one** (credentials, roles, transport security).
- **Scalability** from prototype to production dashboards.
- **Accessibility** and responsive layouts for varied screens.

## 4. Other Tools and Alternatives

### 4.1 Physical Controls & Simple Displays

- **Outputs**: LEDs/indicators, 7-segment or multi-segment LCDs, small pixel displays.
- **Inputs**: pushbuttons, toggles, rotary encoders/dials, keypads, full keyboards.
- **Scanning techniques**:
  - **Row/column scanning** for key matrices (poll rows/columns to detect closures).
  - **Charlieplexing** to control many LEDs/switches with fewer I/O lines (requires tri-state capability).
- **Display refresh**: Avoid visible flicker (typically refresh above ~30 Hz for segment displays).
- **When to choose**: Minimal UI needs, low power, high reliability, tactile feedback requirements.

### 4.2 Command-Line Interfaces (CLI)

- Suited to embedded Linux devices.
- Lightweight, scriptable, excellent for developer/technician workflows.
- Less approachable for general end-users.

### 4.3 Pixel-Based Displays & Low-Level Libraries

- Monochrome/color displays rendering text/icons/bitmaps.
- Typically leverage long-standing embedded graphics libraries (e.g., PEG and others).

- Commonly used from C/C++ on MCUs or SoCs.

### 4.4 Python GUI Alternatives

- **Tkinter**: bundled with Python; good for simple GUIs.
- **Kivy**: cross-platform and touch-friendly; deploys to mobile.
- **PyGTK** and additional bindings exist for niche needs.
- **Selection criteria**: device resources, needed fidelity, development speed, and target platforms.

### 4.5 Summary Guidance

- We emphasize **Qt** and **HTML** for balanced capability on single-board computers.
- Choose simpler physical/CLI approaches when that better fits constraints or environment.

## 5. Development Environments on Raspberry Pi

### IDLE3

- Bundled, minimal friction editor with interactive shell—great for quick tests and import checks.
- Best for early learning and tiny iterations; limited project tooling and debugging depth.

### Geany

- Lightweight IDE with project organization, build/run customization, and plugin support.
- A practical middle ground for ongoing Qt/PyQt or HTML projects on resource-limited hardware.

## 6. Best Practices in GUI Design (Mechanical Focus)

### 6.1 Sources Worth Bookmarking

- **Usability.gov** (evidence-based guidance).
- "**Laws of UI**" style articles (clarity, context, defaults, feedback, guided action).
- **Jeff Johnson**: *GUI Bloopers* and first-principles framing.

### 6.2 Core Principles

- **Simplicity**: remove non-essentials, minimize steps.
- **Consistency**: visual and behavioral uniformity across screens.
- **Purposeful layout**: clear hierarchy, alignment, spacing.
- **Color & typography**: support hierarchy and readability, not decoration.
- **Feedback**: always show system state and action results.
- **Tolerance**: accept reasonable input variance; offer undo/confirm where appropriate.

### 6.3 Frequent Pitfalls

- Misusing controls (e.g., checkbox vs radio, overloading menus/tabs).
- Excessive dynamism in navigation (moving targets confuse users).
- Poor wayfinding (no "where am I?" cues or search).
- Overbearing memory load (forcing users to recall IDs/steps).
- Fragile customization (reverting the user's layout without notice).
- Weak text (inconsistency, jargon, ambiguity).

### 6.4 Text Quality Tactics

- Partner with documentation/tech-writing teams for terminology, tone, and clarity.
- Prefer concise, action-oriented microcopy; maintain glossary consistency.

### 6.5 Process Reminder

- Best practices raise the floor; **iterative user testing** raises the ceiling.

## 7. Micro-Interactions (Expanded)

### 7.1 What They Are

- **Definition** (per Dan Saffer): a **contained product moment** focused on a **single, specific use case**.
- They are **small, simple, brief**, and should feel **effortless**.
- Distinct from "features," which span multiple use cases and sustained engagement (e.g., a music player as a feature vs **volume adjustment** as a micro-interaction).

### 7.2 Why They Matter for Embedded Devices

- Embedded UIs often expose **few critical controls**—each one must feel right.
- On small screens or physical controls, **precision and polish** drive perceived quality, safety, and trust.
- Well-crafted micro-interactions can create **signature moments** that differentiate products (e.g., a satisfying hardware toggle, a clear strength meter).

### 7.3 Four Structural Elements

1. **Trigger**: What starts the micro-interaction (user-initiated or system-initiated; physical switch or on-screen control).
2. **Rules**: The logic that governs what happens from trigger to completion (sequence of states, allowed transitions, constraints).
3. **Feedback**: What the user perceives during/after the interaction (visual, auditory, haptic, textual).

4. **Loops & Modes**: Repetition, duration, and state-dependent behavior (e.g., long-press vs tap; cooldowns; repeated status polling).

## 7.4 Designing Each Element Well

- **Trigger**
  - Make it **discoverable**, **predictable**, and **consistent** every time.
  - Consider **context** (when/where users act; environmental constraints such as gloves, glare, noise).
  - For **system triggers** (errors, thresholds, connectivity changes): decide **frequency**, **salience**, and **interruptibility** (soft vs hard alerts).
- **Rules**
  - Keep flows **short** and **bounded** (few actions, few objects).
  - Sequence information so users have the **right detail at the right time** (avoid front-loading).
  - For complex logic, map states/events to verify no dead ends or ambiguous transitions.
- **Feedback**
  - Provide **immediate, perceivable** responses (press states, progress indicators, confirmations).
  - Use **multimodal** feedback on devices where appropriate (LED blink + tone + brief vibration), mindful of context (clinical, quiet, bright).
  - Differentiate **success**, **processing**, **warning**, and **error** with consistent patterns.
- **Loops & Modes**
  - Define repeat behavior (e.g., alarm repeats every 60 s until acknowledged).
  - Clarify **modes** (normal vs setup vs maintenance) and how micro-interactions vary by mode.
  - Prevent "mode errors" by minimizing hidden state and providing clear mode indicators.

## 7.5 Good Micro-Interaction Characteristics

- **Low cognitive load** (recognition over recall).
- **Fast and forgiving** (supports slip tolerance; easy to cancel/undo).
- **Context-appropriate** (screen size, lighting, noise, user posture).
- **Brand-aligned** (tone of messages, motion cadence, sound design).

## 7.6 Examples & Inspiration

- **Hardware toggle** (e.g., ring/silent switch): tactile detent, clear on-device icon change, optional brief haptic.
- **Password strength indicator**: real-time grading, clear thresholds, unobtrusive color/text cues.
- **Volume adjustment**: smooth granularity, audible sample at safe levels, persistent indicator that times out gracefully.

- **“Little Big Details”-style catalogs**: browse curated micro-patterns to spark solutions for your use case.

## 7.7 Evaluation & Iteration

- **Expectation mapping**: Ask users how they think it should work *before* exposure; compare to actual behavior.
- **Think-aloud trials**: Observe first-time use without coaching; then guided walkthrough to reveal mismatches.
- **Failure mode probes**: What happens on long press vs double-tap? On loss of network? On sensor error?
- **Field conditions**: Test in realistic environments (gloves, motion, low light, noise).
- **Refinement cadence**: Iterate quickly; small changes in timing, motion, or copy often yield big gains.

## 7.8 Common Micro-Interaction Pitfalls

- **Inconsistent triggers** (same control behaves differently across screens/modes).
- **Ambiguous feedback** (colors/sounds that don’t clearly map to states).
- **Over-animated** flourishes that delay action completion.
- **Hidden modes** leading to unexpected outcomes.
- **High dependency on memory** (cryptic icons, unexplained gestures).

## 7.9 Deliverables & Checklists (Design-Ready)

- **One-pager per micro-interaction**: purpose, context, trigger, rules, feedback, loops/modes, states, and edge cases.
- **Acceptance criteria**: latency thresholds (e.g., immediate press feedback), accessibility cues, error fallback.
- **Handoff artifacts**: motion timing specs, tone libraries, icon sets, color token references, copy variants.
- **Telemetry plan**: events to log (triggered, completed, canceled, error), thresholds for success/failure.

## 7.10 Strategic Payoff

- Strong micro-interactions can create **signature moments** (memorable, brand-defining), reduce training time, and improve safety/efficiency—especially on constrained embedded UIs.

## 7.11 References for Deeper Study

- **Dan Saffer — *Microinteractions*** (book and companion resources).
- **Little Big Details** (pattern inspiration).
- **Nielsen Norman Group / Interaction Design Foundation** (general interaction guidance that complements micro-interaction design).

## 8. Project 1: Qt + Python User Interface

- **Objective**: Build a Qt/PyQt application for an embedded-style use case.
- **Requirements**: Install Qt/PyQt and Qt Designer; design the UI; integrate and test on Raspberry Pi OS or Pi desktop VM.
- **Expectations**: Apply best practices, include intentional **micro-interaction** decisions for at least one critical control, and document triggers/rules/feedback/loops.

# Study Guide: Laws of UI Design

## 1. Law of Clarity

- **Definition**: Users should be able to understand what every element in the interface does without guessing.
- **Why It Matters**: Ambiguity creates hesitation, slows workflows, and increases error rates.
- **Best Practices**:
  - Use **labels with icons** rather than icons alone.
  - Maintain **visual affordances**: buttons should look clickable, sliders draggable.
  - Remove unnecessary elements to reduce clutter.
- **Embedded Context**: On a small device display, clarity might mean one large, labeled button for a critical function instead of multiple small unlabeled icons.

---

## 2. Law of Preferred Action

- **Definition**: The UI should make the most important or recommended action the easiest and most obvious to take.
- **Why It Matters**: Users are more likely to select the option that is visually or physically emphasized. This can guide safe, efficient operation.
- **Best Practices**:
  - Use **visual hierarchy** (color, size, placement) to highlight the primary action.
  - Avoid making destructive or rare actions look equally prominent.
  - Provide clear **call-to-action cues** (e.g., “Start,” “Confirm,” “Next”).
- **Embedded Context**: On a medical pump, “Start Infusion” should be large and central, while “Factory Reset” should be hidden under advanced settings.

---

## 3. Law of Context

- **Definition**: Show users the right information and controls at the right time, depending on their task and environment.
- **Why It Matters**: Overloading users with irrelevant options or displaying controls too early/late leads to confusion.
- **Best Practices**:
  - **Context-sensitive menus** that adapt to the user's current step.
  - Display **error messages near the source** of the problem.
  - Hide or disable unavailable options rather than leaving them active.
- **Embedded Context**: A device could show "Stop" only once "Start" has been pressed, simplifying the interface flow.

---

# 4. Law of Defaults

- **Definition**: Provide sensible default values or states that match the most common, safe, or recommended configuration.
- **Why It Matters**: Most users accept defaults. Well-chosen defaults reduce effort and prevent misconfiguration.
- **Best Practices**:
  - Pre-fill settings with safe, **commonly used values**.
  - Make defaults visible and easy to override.
  - Provide **fail-safe fallbacks** if no user input occurs.
- **Embedded Context**: A ventilator might default to a safe oxygen concentration and alert the user before starting with unsafe parameters.

---

# 5. Law of Guided Action

- **Definition**: The interface should guide the user step by step, making the correct sequence of actions clear and intuitive.
- **Why It Matters**: Users tend to follow the paths suggested by the UI; guidance reduces mistakes and training needs.
- **Best Practices**:
  - **Step-by-step workflows** (wizards) for complex tasks.
  - Highlight or unlock the **next required step** after the previous is complete.
  - Disable irrelevant actions during each step.
- **Embedded Context**: A diagnostic tool may only enable "Analyze" after "Collect Data" has been completed, steering the operator through the process.

---

# 6. Law of Feedback

- **Definition**: Every user action should trigger an immediate and perceivable system response.
- **Why It Matters**: Without feedback, users don't know if their action was registered; this can cause repeated actions or loss of trust.
- **Best Practices**:
  - Visual: button highlight, status bar update.
  - Auditory: confirmation tone, warning beep.
  - Haptic: vibration or tactile click on hardware buttons.
- **Embedded Context**: Pressing "Start" on an infusion pump should beep, flash a light, and show an on-screen confirmation, reassuring the user the process has begun.

---

## 7. Law of Easing

- **Definition**: Tasks should be broken down into manageable, low-effort steps rather than overwhelming the user with complexity at once.
- **Why It Matters**: Users complete tasks more successfully when effort is distributed in small, intuitive steps.
- **Best Practices**:
  - Use **progressive disclosure**: hide advanced options until needed.
  - Divide complex workflows into **short sequences** with clear progress indicators.
  - Provide **graceful error recovery** so users don't need to restart completely.
- **Embedded Context**: A setup wizard for a home medical device might first ask for language, then network, then calibration—rather than displaying all settings at once.

---

# Key Takeaways

- **Clarity** → Every control should be self-explanatory.
- **Preferred Action** → Highlight the best or safest choice.
- **Context** → Present only what is relevant for the current step.
- **Defaults** → Provide safe, common starting points.
- **Guided Action** → Lead users through workflows step by step.
- **Feedback** → Always acknowledge user actions.
- **Easing** → Break down tasks to make them low-effort and achievable.

# 9. Key Takeaways

- **Qt/PyQt**: Native, robust, and ideal for on-device interfaces.
- **HTML/CSS/jQuery**: Flexible and network-friendly for remote dashboards and control.
- **Other options**: Physical controls, CLIs, segmented/pixel displays, and Python alternatives (Tkinter, Kivy) fit simpler or specialized needs.
- **Raspberry Pi tooling**: **IDLE3** for quick checks; **Geany** for organized, ongoing development.
- **Best-practice guardrails**: Simplicity, consistency, clear feedback, and tolerance; avoid control misuse and wayfinding gaps.
- **Micro-interactions**: Design them deliberately (trigger, rules, feedback, loops/modes) to create low-effort, signature moments.
- **Process**: Best practices raise baseline quality; **user testing** unlocks fit and finish.