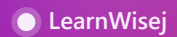


# The Wisej.NET Visual System

Lesson Guide · Lab / Exam Guide



## Lesson Guide

How server-side controls, client-side widgets, theme appearances, states, CSS classes and layout engines together produce what the user sees, which of the three customization layers owns each decision, and the Adaptive Operations Console shell that grows across the course. What this means for a Wisej.NET developer — and how to read this module.



### Production focus

How server-side controls, client-side widgets, theme appearances, states, CSS classes and layout engines together produce what the user sees, which of the three customization layers owns each decision, and the Adaptive Operations Console shell that grows across the course.

## 1. The production problem

A team that treats Wisej.NET as a CSS-first framework ends up with `BackColor` set on two hundred controls, selectors that break on the next upgrade, and Resize handlers that recompute `Bounds` by hand; when a phone version is requested, the only option left is a rewrite. This module fixes where a control's look and layout really come from, names the three customization layers and which one owns each decision, and starts the Adaptive Operations Console shell that every later module extends.

## 2. Key concepts

| Concept                                         | Why it matters                                                                                                                                                                         |
|-------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| server control and client widget                | You write .NET controls on the server; the browser displays JavaScript widgets that Wisej.NET keeps in sync, so a visual decision can live on either side of that line.                |
| theme                                           | A single JSON file of colors, images, fonts, appearances, states, styles, properties and settings that gives every standard control its identity; the contract between the two worlds. |
| appearance and <code>AppearanceKey</code>       | Each control exposes an appearance key; the active theme holds an appearance for that key, and a semantic key such as <code>action-button</code> selects a variant.                    |
| state                                           | A named interaction condition inside an appearance (default, hovered, pressed, focused, invalid, selected) whose styles and properties apply while it is active.                       |
| <code>CssClass</code> and <code>CssStyle</code> | Direct browser styling: a class from a scoped application stylesheet, or a raw style string on the widget element for a rare one-off value.                                            |
| layout engine and <code>Dock</code>             | Position and size come from the container hierarchy, not from the theme; docking stacks regions against the parent's edges and fills what is left.                                     |
| visual inventory                                | The written list of controls needing appearances, color and font tokens, layout regions and breakpoints that the theme, layout and profile work is planned from.                       |

## 3. Lesson narrative

### The theme is the contract between server controls and browser widgets

In Wisej.NET the developer writes .NET controls on the server, but what the user sees is a client-side widget rendered by the browser. A `Button`, `TextBox`, `TabControl`, `DataGridView`, `Panel` or custom widget exposes an appearance key, the active theme contains an appearance for that key, and each appearance contains states, styles, properties and child components. That is why a professional visual change can be made once, centrally, instead of being scattered across individual controls: change the button appearance and every button in every session follows.

This is also why Wisej.NET theming is more than normal CSS. The theme reaches into the controls themselves, into the widget metrics the layout depends on, into the fonts and sizes the Visual Studio designer renders with, and into colors, images and appearances that CSS alone cannot express. The theme engine may generate CSS classes from appearance and state, and theme properties can override styles, so a hand-written rule against the internal widget DOM fights the engine rather than using it. Treat the theme file as a product design asset: named colors and shared fonts are design tokens, appearances model component variants, states express interaction, and a button, grid, editor, panel, toolbar and form look related even when different developers created them.

### Three layers, and which one owns each decision

Layer one is the theme: global visual identity, state behaviour, widget metrics, named colors, fonts and the appearances of standard components. Layer two is the server-side control model: `BackColor`,

`ForeColor`, `Font`, `Padding`, `Margin`, `Dock`, `Anchor`, `Size`, `MinimumSize`, `MaximumSize`, `AppearanceKey` and the responsive properties. Layer three is direct browser styling: `CssClass`, `CssStyle`, the StyleSheet extender in legacy code, custom CSS files, `InitScript` and JavaScript widget customization. An intermediate developer must know which layer owns each decision before touching any of them.

The decision rule the whole course applies is short. Theme first for the identity of standard controls.

`AppearanceKey` for a semantic variant such as an action button or a destructive button, letting the theme define its default, hovered, pressed, focused and disabled states. `CssClass` for a narrowly scoped, app-specific hook: a metric card, embedded HTML, a third-party widget. `CssStyle` only for a small one-off or dynamic value, kept rare and local. Layout containers for positioning, responsive properties for values the designer assigns per profile, and `ResponsiveProfileChanged` for behaviour that a property value cannot express.

The negative rule matters as much: do not patch every standard control with direct CSS. Theme appearances are better for built-in controls because they participate in states, designer rendering and the shared tokens; CSS is for app-specific content, third-party widgets and embedded HTML. When a theme "does not apply", the first suspects are a direct `BackColor`, `ForeColor`, `Font`, `CssStyle` or custom `CssClass` overriding it, a wrong appearance key, or a theme set in the designer but not selected for the running application.

## The visual inventory: tokens, regions, breakpoints

Before anyone opens the Theme Builder, the lab asks for an inventory of the application as it will be. Walk the Adaptive Operations Console and list the controls that need a custom appearance: the command buttons, the panels that become metric cards, the tabs, the grid header, the invalid state of the ticket editor, the tooltips. List the color tokens the theme will define once and reuse everywhere: `brandPrimary`, `brandAccent`, `surface`, `surfaceAlt`, `textMain`, `textMuted`, `danger`, `warning`, `success` and `focusFrame`, plus a default, heading and mono font. List the layout regions: toolbar, navigation rail, workspace with metric cards and ticket grid, details panel, status bar. Finally list the breakpoints: desktop at 1440, small desktop at 1024, tablet at 768 portrait and 1024 landscape, phone at 390 portrait and 844 landscape.

Write it as `VisualInventory.md` inside the project and give every row a fourth column: the layer that owns it. A metric card's rounded corners belong to a scoped `CssClass`; the button's hover color belongs to the theme; the details panel's width belongs to `Dock`; whether the rail is visible on a phone belongs to a responsive property. Record the base theme decision in the same file: Module 2 derives the AdaptiveOps theme from one of the built-in themes and inherits its appearances, states, images and metrics, so record why you chose it, not only which.

## The Adaptive Operations Console shell

The course project is an operations console with a branded theme and a desktop, a tablet and a phone layout. The desktop version has a left navigation rail, a toolbar, metric cards, a ticket grid and a details panel. The mobile version collapses the navigation, stacks the grid and details, hides low-priority text, uses icon-only display where appropriate and keeps touch scrolling comfortable. Module 1 builds only the

desktop shell, with `Dock` and nested `Panel` s and nothing else: toolbar top, status bottom, navigation left, details right, and a workspace docked to fill with a `MinimumSize` so it never collapses.

```
private void BuildShell()
{
    toolbarPanel.Dock = DockStyle.Top;        toolbarPanel.Height = 56;
    statusPanel.Dock = DockStyle.Bottom;      statusPanel.Height = 28;
    navigationPanel.Dock = DockStyle.Left;    navigationPanel.Width = 240;
    detailsPanel.Dock = DockStyle.Right;      detailsPanel.Width = 360;
    workspacePanel.Dock = DockStyle.Fill;
    workspacePanel.MinimumSize = new Size(320, 240);
}

private void MainPage_Resize(object sender, EventArgs e)
{
    widthLabel.Text = $"Width: {this.Width} px";
}
```

The one piece of behaviour in the shell is the status label that reports the current browser width. The main `Page` always fills the browser viewport, so its `Resize` event fires whenever the user resizes the window; writing `this.Width` into the label gives later modules a number to compare against the breakpoints in `ClientProfiles.json` and the name in `Application.ActiveProfile`. Wrap the update in a try / catch that leaves an honest fallback text rather than a stale value. What the shell must not contain is as important as what it does:

- No `BackColor`, `ForeColor` or `Font` overrides: the base theme owns the look until Module 2.
- No `CssStyle` strings and no custom CSS yet: nothing in the shell is app-specific enough to need them.
- No `Resize` code that sets `Bounds`: docking already answers every desktop width.

Take the desktop screenshot when the shell runs and write the note that explains, region by region, which layer owns each visual decision; it is the seed of the architecture note the capstone asks for.



### Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, `AppearanceKey` for semantic variants, `CssClass` for scoped app-specific CSS, `CssStyle` only for one-off dynamic values, layout containers instead of `resize` code, responsive properties for designer-set profile values, and `ResponsiveProfileChanged` for behaviour that is not simply visual.



### Learning objectives

By the end of this module you'll be able to:

- ✓ Explain why Wisej.NET theming is more than normal CSS and how it reaches into controls, widgets, designer metrics, fonts, colors, images, and appearances.

- ✓ Identify when to use a theme file, built-in theme, control property, CssClass, CssStyle, or a layout container.
- ✓ Create a visual inventory for an existing application and define theme tokens, layout regions, and responsive breakpoints.
- ✓ Start the course project, Adaptive Operations Console, with a baseline desktop shell and target mobile behavior.

## Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



### Hands-on lab

Create the Adaptive Operations Console solution and its first desktop shell: a main Page with a top toolbar panel, a left navigation panel, a fill workspace, a right details panel and a bottom status panel built with Dock and nested Panels. Choose a built-in base theme and record the choice, write a `VisualInventory.md` that lists the controls needing custom appearances, the named color and font tokens, the layout regions and the desktop, tablet and phone breakpoints, and add a status label that reports the current browser width so later modules can verify profiles.

## Starting point

Open Visual Studio and create a new Wisej.NET web application named `AdaptiveOperationsConsole`; this is the solution every later lab extends, so keep each module's work in its own folder. The template gives you an empty main `Page`, a `Program.cs` that opens it and the default theme selection; no panels, no theme file and no inventory exist yet.

## Suggested approach

- ✓ Sketch the five regions on paper first, then drop five `Panel` s on the main Page and dock them in this order: `toolbarPanel` top, `statusPanel` bottom, `navigationPanel` left, `detailsPanel` right, and `workspacePanel` last with Dock set to Fill and a `MinimumSize` of 320 by 240.
- ✓ Pick one of the built-in themes as the base, select it for the running application (not only in the designer), and write the reason for the choice in your lab notes before you change anything else.
- ✓ Add a `widthLabel` to the status panel, handle the main Page's `Resize` event and write  `$"Width: {this.Width} px"`  into it from a single `ReportWidth()` method wrapped in try / catch; call it once from the constructor so the label is never empty.
- ✓ Create `VisualInventory.md` in the project with four tables: controls that need custom appearances, color and font tokens (brandPrimary, brandAccent, surface, surfaceAlt, textMain, textMuted, danger, warning, success, focusFrame; default, heading, mono), layout regions, and the desktop, tablet and phone breakpoints; give every row an "owned by" column naming theme, control property, CssClass, CssStyle or layout container.
- ✓ Run the shell, resize the browser from desktop to tablet to phone width while watching the label, take the desktop screenshot and write the note that explains which layer owns each visual decision in the shell.

## Required deliverables

- ✓ Adaptive Operations Console solution with a docked desktop shell of toolbar, navigation, workspace, details and status panels
- ✓ Base theme chosen and configured, with the reason recorded in the lab notes

- ✓ VisuallyInventory.md listing controls that need appearances, theme tokens, layout regions and breakpoints
- ✓ Status label that reports the current browser width
- ✓ Desktop screenshot and a note explaining which layer owns each visual decision

## Student review questions

- ✓ The toolbar's primary command needs the brand colour on every screen. Which of the three layers owns that decision, and what would go wrong if you set `BackColor` on the button instead?
- ✓ Your width label reads 1024 px and the details panel is still visible. Which mechanism from a later module will decide whether it should be, and why is a CSS media query the wrong first answer in Wisej.NET?
- ✓ If the base theme were swapped for a different built-in theme tomorrow, which rows of your VisuallyInventory.md would change and which would not, and what does that tell you about where the inventory's decisions live?



### Visual-system habit

Use the Wisej.NET mechanism that owns the problem: the theme for the identity of standard controls, AppearanceKey for semantic variants, CssClass for scoped app-specific CSS, CssStyle only for one-off dynamic values, layout containers instead of resize code, responsive properties for designer-set profile values, and ResponsiveProfileChanged for behaviour that is not simply visual.