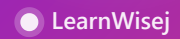


Layout Fundamentals: Docking, Anchoring, AutoSize, AutoScroll, and Shell Composition

Lesson Guide · Lab / Exam Guide



Lesson Guide

The default layout engine, docking order and padding, anchoring for form-like regions, AutoSize and AutoScroll used safely, and composing the shell from nested containers and UserControls instead of resize handlers. What this means for a Wisej.NET developer — and how to read this module.



Production focus

The default layout engine, docking order and padding, anchoring for form-like regions, AutoSize and AutoScroll used safely, and composing the shell from nested containers and UserControls instead of resize handlers.

1. The production problem

The Adaptive Operations Console has a theme and scoped CSS now, but its shell still moves controls by hand: a **Resize** handler recomputes **Bounds** for the toolbar, the navigation rail, the ticket grid and the details editor every time the browser changes size. It works at the width the developer tested, breaks at every other one, and cannot survive the tablet and phone profiles that Module 6 introduces. Adaptive design in Wisej.NET depends on containers and layout rules, not coordinates: regions that claim edges, a workspace that fills what is left, and editors that scroll instead of overlapping.

2. Key concepts

Concept	Why it matters
default layout engine	Every Wisej.NET control has a layout engine; the default one supports absolute position, docking and anchoring, so a screen can mix precise designer placement with regions that resize themselves.
docking priority	Docked children consume space in the order they appear in the parent's control collection, which is why the same five panels produce a correct shell or an overlapping one depending on child order.
DisplayRectangle and Padding	Docked controls lay out inside the parent's <code>DisplayRectangle</code> , which <code>Padding</code> reduces; the default engine ignores <code>Margin</code> for docked children, so gaps come from padding or spacer controls.
anchoring	An anchor keeps a control's distance from selected parent sides, which suits form-like regions: labels top-left, text boxes left-right, command buttons bottom-right.
AutoSize and circular layout	A container that sizes itself from children that are docked or anchored back to it has no stable answer; that loop is the module's main AutoSize warning.
AutoScroll with hidden bars	<code>AutoScroll</code> lets a container scroll when content does not fit, and <code>ScrollBars.Hidden</code> keeps touch scrolling on phones without drawing scrollbars.
UserControl region	A shell region with its own local layout and a small public surface, so the shell shows, hides, docks or moves the whole region per profile instead of knowing its internal coordinates.

3. Lesson narrative

Docking, child order and the space that is left

Docking attaches a child to an edge of its parent (`DockStyle.Top` , `Bottom` , `Left` , `Right`) or fills whatever remains (`DockStyle.Fill`). The engine hands out space to docked children one at a time, following their order in the parent's collection, and each docked child reduces the rectangle available to the next. That is the whole secret of the console shell: the toolbar takes a strip from the top, the status region one from the bottom, the navigation rail one from the left, the details panel one from the right, and the workspace fills what is left. Change the order and the status panel hides the last grid row. In the Visual Studio designer, `BringToFront` , `SendToBack` and the child-index operations change that order, so when a shell looks wrong check the order of the children before touching a size. The pack's shell code is deliberately short, and the lab reproduces it in the designer:

```
// MainPage.cs – the console shell, one Dock per region
this.toolbarPanel.Dock = DockStyle.Top;    this.toolbarPanel.Height = 56;
this.statusPanel.Dock = DockStyle.Bottom; this.statusPanel.Height = 28;
this.navigationPanel.Dock = DockStyle.Left; this.navigationPanel.Width = 240;
this.detailsPanel.Dock = DockStyle.Right;  this.detailsPanel.Width = 360;

this.workspacePanel.Dock = DockStyle.Fill;
this.workspacePanel.MinimumSize = new System.Drawing.Size(320, 240);

// gaps between docked regions come from the container, not from Margin
this.Padding = new Padding(8);
this.workspacePanel.Padding = new Padding(12, 8, 12, 8);
```

Docked children are placed inside the parent's `DisplayRectangle`, and `Padding` shrinks that rectangle from every side. The default layout engine does not use `Margin` for docked controls: a margin on the navigation rail does nothing to the gap beside it, so give the container padding or dock a thin spacer panel. Anchoring covers the controls inside a region. An anchor preserves the distance to the selected parent sides: labels top-left so they stay put, text boxes left-right so they grow with the details panel, the wide notes editor top-left-right, and the Save and Cancel buttons bottom-right. A control with no anchor on an axis floats and keeps a proportional position, right for centring a sign-in card and wrong for a grid.

Containers instead of resize handlers

The console's `MainPage_Resize` is the smell the module exists to remove. Every line that sets `Bounds`, `Location` or `Size` in response to a resize is a rule the layout engine could have applied for you, and unlike the engine it runs after the browser has painted the wrong layout, knows nothing about padding, and does not compose: the moment the details panel becomes a `UserControl`, the handler has to learn its internals too. The refactoring is mechanical: list the regions, decide the edge each one claims, add them in that order with `Dock`, and delete the handler. Absolute coordinates remain useful for precise designer work inside a fixed-size dialog; they are the wrong tool for a screen that must also run on a phone.

A durable shell is a set of nested regions: top command bar, left navigation, centre workspace, right details, bottom status. Each region is a `Panel`, `UserControl`, `TabControl`, `SplitContainer` or `Page` with its own local layout, and the nesting is what makes the shell stable: the workspace does not care how the toolbar arranges its buttons, and the toolbar does not care that the workspace holds metric cards above a ticket grid. Anchoring a large number of controls in complicated combinations is a signal too: a `TableLayoutPanel` or `FlexLayoutPanel`, both covered in Module 5, expresses aligned fields or proportional regions more clearly than twelve anchor settings.

AutoSize, AutoScroll and the sizing guard rails

`AutoSize` is valuable where content decides the size: a label that must fit translated text, a badge that grows with its number, a card that wraps dynamic content; `AutoSizeMode` decides whether the control may only grow or may also shrink. The danger is circular layout. If a container auto-sizes from children that are docked or anchored to that same container, the parent asks the children how large to be, the children ask the parent how large they are, and the engine has no stable answer. The typical accident is a panel with `AutoSize` that also fills its parent. Keep auto-sizing on leaf controls and on containers whose children have fixed or content-driven sizes.

`AutoScroll` is the safe answer when content does not fit: the container scrolls instead of stretching the shell. The console's details editor has more fields than a phone screen can show, so the lab turns on `AutoScroll` and, at narrow widths, sets `ScrollBars` to `Hidden`, which keeps touch scrolling without drawing scrollbars that steal width. `AutoScrollMargin` adds room after the last control and `AutoScrollMinSize` guarantees a minimum scrollable area. `MinimumSize` and `MaximumSize` are the other guard rails: a `MinimumSize` on the workspace and on the grid means the fill region can never collapse into a sliver when the browser narrows, and a `MaximumSize` keeps the details panel from swallowing an ultra-wide monitor.

UserControl regions the shell can rearrange

The last step of the lab is the one that pays off in Module 6. Once the navigation rail and the details editor are `UserControl`s, each with its own docking and anchoring inside, the shell stops knowing the internal coordinates of any editor. It knows five regions, and per profile it shows, hides, docks or moves whole regions: the rail can become a narrow icon strip, the details editor can move under the grid on a tablet or open in a modal `Form` on a phone, and none of those moves touches a text box. The region exposes a few meaningful members, such as a `Ticket` property and a `Saved` event, and keeps its child controls private so a rename inside never breaks the page.

The layout checklist for the module is short: prefer containers over resize handlers, check the docking order, put `Padding` on docked regions, set `MinimumSize` on fill regions, and let content that does not fit scroll. The pack's refactoring smell list is the mirror image of that checklist, and reviewing a screen against it is a good habit before every profile test.



Refactoring smell list

Manual `Resize` code that repeatedly sets `Bounds` on many controls. `Dock` and `Anchor` mixed on controls that fight each other. Absolute coordinates in a screen that must run on phones. Grids without `MinimumSize`. Panels with `AutoSize` and `Fill` docking creating loops. Form layouts that hide controls instead of moving low-priority content into a scrollable section.



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:

- ✓ Use docking and anchoring intentionally and understand how child order affects docking priority.
- ✓ Compose screens from nested containers rather than resize handlers and manual coordinate updates.

- ✓ Use `AutoSize`, `AutoSizeMode`, `AutoScroll`, `ScrollBars`, `MinimumSize`, `MaximumSize`, `Padding`, and `Margin` safely.
- ✓ Create reusable `UserControl` regions that can be rearranged across desktop, tablet, and phone profiles.

Lab / Exam Guide

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



Hands-on lab

Refactor the Adaptive Operations Console shell into nested containers: a top toolbar, a left navigation rail, a fill workspace, a right details panel and a status region, each a Panel or UserControl with its own local layout. Delete every Resize handler that sets Bounds, fix docking order in the designer, use Padding for gaps between docked regions, give the workspace and grid a MinimumSize, and make the details editor scroll with AutoScroll and hidden scrollbars at narrow widths.

Starting point

Open the `AdaptiveOperationsConsole` solution as you left it after Module 3: the main `Page` carries the AdaptiveOps theme, the scoped stylesheet and the session theme switch, but its five regions are still plain panels whose `Bounds` are recomputed in `MainPage_Resize`, and the details editor overlaps the ticket grid below roughly 1100 pixels. Run it once at desktop width, then drag the browser narrower and note what clips, overlaps or disappears before you change anything.

Suggested approach

- ✓ List every `Resize`, `Layout` or load-time method that sets `Bounds`, `Location` or `Size`, write the five regions and the edge each one claims into your lab notes, then delete that code before touching the designer.
- ✓ In the designer set `Dock` on the five panels (toolbar Top, status Bottom, navigation Left, details Right, workspace Fill) and correct the child order with `BringToFront` and `SendToBack` until the edges are claimed first and the workspace takes what is left.
- ✓ Give the page and the regions `Padding` for the gaps between docked areas, set `MinimumSize` on the workspace and the ticket grid, and confirm the default engine ignores `Margin` on docked children.
- ✓ Inside the details panel anchor labels top-left, editors left-right, the notes editor top-left-right and the command buttons bottom-right; turn on `AutoScroll`, set `ScrollBars` to `Hidden` for narrow widths and tune `AutoScrollMargin`.
- ✓ Extract `NavigationRail` and `DetailsEditor` as `UserControl`s with their own local layout and a small public surface, drop them back into the shell, and resize the browser from desktop to phone width to verify nothing overlaps or clips.

Required deliverables

- ✓ Shell rebuilt from nested containers with toolbar, navigation, workspace, details and status regions
- ✓ All manual `Resize` code that set `Bounds` removed
- ✓ Docking order, `Padding` and `MinimumSize` set intentionally on every region
- ✓ Details editor scrolling through `AutoScroll` with hidden scrollbars at narrow widths

- ✓ Navigation and details regions extracted into UserControl

Student review questions

- ✓ If the workspace panel were docked before the status panel in the child order, what would the user see at the bottom of the grid, and which designer command fixes it without changing a single size?
- ✓ You set a 12-pixel `Margin` on the navigation rail and the gap did not appear. Why not, and which two mechanisms do produce a gap between docked regions?
- ✓ The details editor is a `UserControl` with `AutoScroll`. Which of its members must the shell know about to move it under the grid on a tablet, and which must it never need?



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.