Implied Form Design

Using Inkscape, Avalonia, and SvgTools

By Daniel Patterson

Introduction

So, you want to design your form in a fast-and-loose fashion approximating throwing fresh-cooked spaghetti against the wall to see if sticks? If you are the kind of form designer that feels deeply about how a single individual should be able to consistently create several different forms per day rather than one every couple of months, then have we got the tools for you!

Implied visual design allows you to quickly place free-style shapes on a vector editor drawing canvas, perform a little basic approximate alignment on them, then add just a few suggestions to those shapes to help the converter to perform its job of literally creating a beautiful, perfectly functional application form out of your beautiful creation.

This guide is an exploration of the philosophies we will honor, the procedures for building forms in a Rapid Application Development (RAD) design environment, and elemental details available for targeting your designs.

Table of Contents

&TLDR;

While I hope you will find helpful information everywhere in this document, the main take-away is that you can design consistent and completely usable GUI forms for desktop applications in Inkscape using the following simple steps.

1. Open Inkscape.
2. Draw a rectangle, text, or insert an image.
3. Give it a label from the Intents list in the Implied Form Design Object Reference section.
4. Repeat steps 2 and 3 for a few minutes until done.
5. Run the converter to deliver the working form. If you wish, this step can be performed by an IT staff in a separate working role.

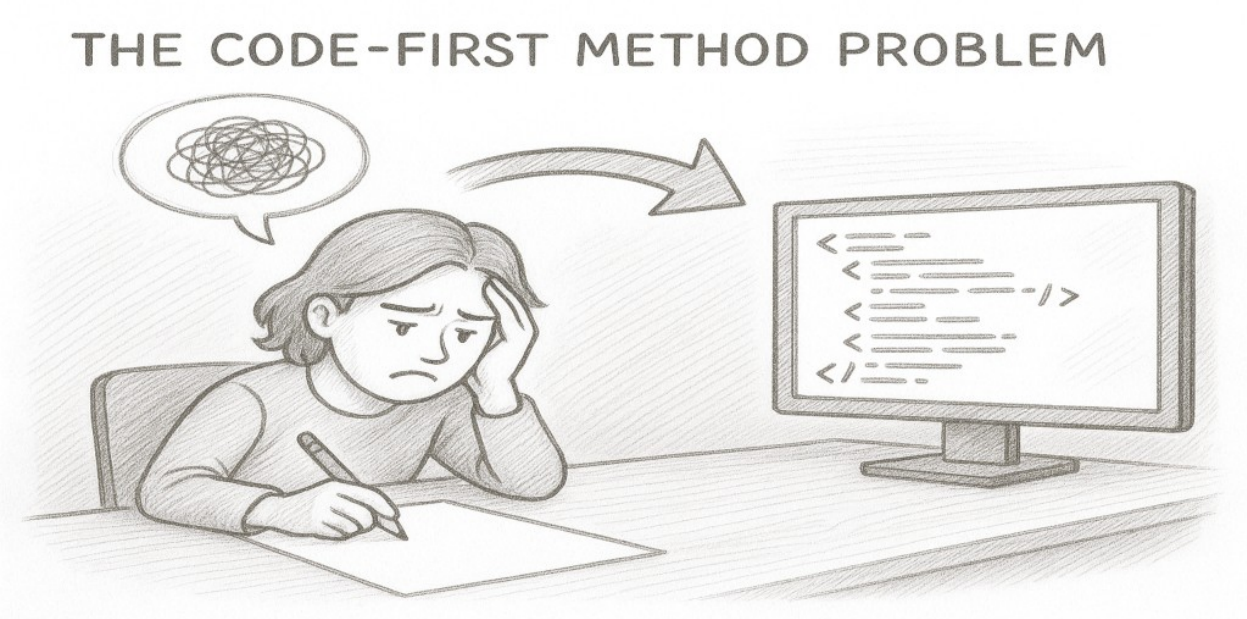
&TLDR;&TLDR;

So really, three steps: **Draw**, **Label**, **Done**.

How On Earth Did We Get Here?

On a slightly more serious note, I know that graphic artists work in a realm of color, shape, and emotion. Your tools are brushes, bezier curves, and layers. You shouldn't be required to think at all in terms of functional containers, data bindings, layout constraint specifications, which control works with which template, or least of all by a long shot: code; of any kind.

The Code-First Method Problem

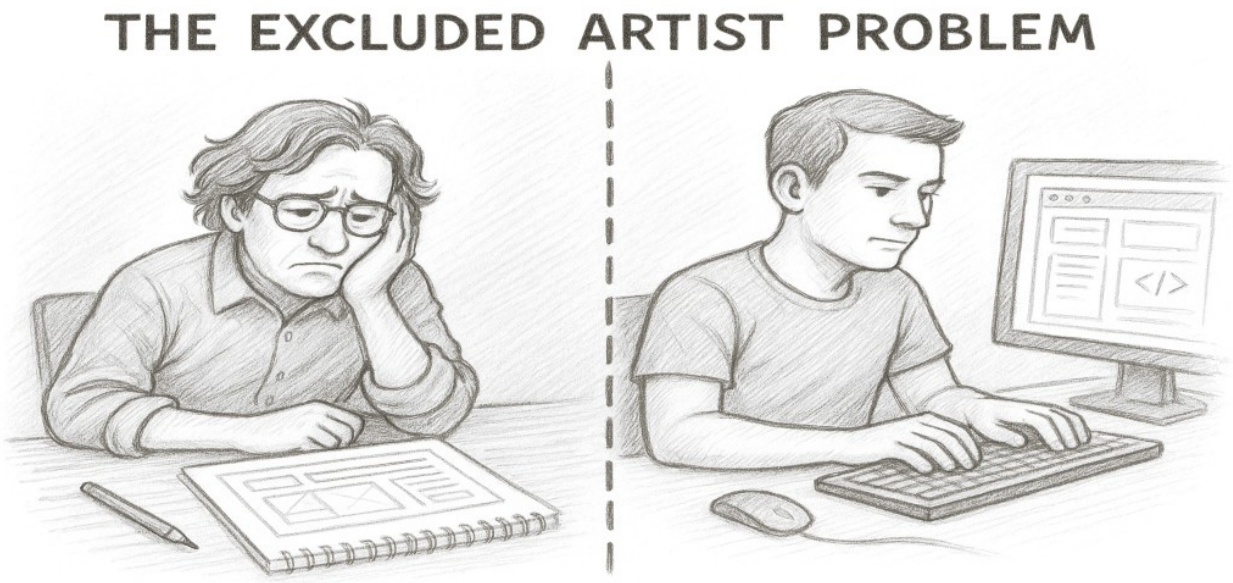


I know that your goal is to evoke, inspire, and communicate visually. I also know that artists are most often known to create SVGs, illustrations, mockups without any concern or curiosity for how those might be wired up or implemented with their underlying logical functionality, and I completely support that. In fact, although I play the part of all of the roles in a lot of my projects, I have always demanded from the industry a complete separation of duties between UI design and the code that defines it. For example, if I am working on a web page, I would prefer not to ever touch HTML/CSS, although currently, that's the only practical way to do it.

You might be surprised to find out that my demand of separation of design and coding roles has literally always landed on deaf ears while the actual industry has gone further every year into what any normal person would have to admit is actually not only a *code-first* approach to UI design, but more accurately, a *code-only* tactic. After all, have you noticed that although the underlying makeup the form is always some version of XML, only the higher quality systems even make the effort to provide working *preview* screens for that code, and none of them other than the ScreenBuilder for JavaFx actually allow you to start drawing the form on a blank canvas then to only preview that work as a very last step when you think it's done?

Even given that current opportunity, unfortunately, Java and JavaFx continue their long, steady decline into oblivion for other related reasons that are also discussed in **The Structure Before Expression Problem**.

The Excluded Artist Problem



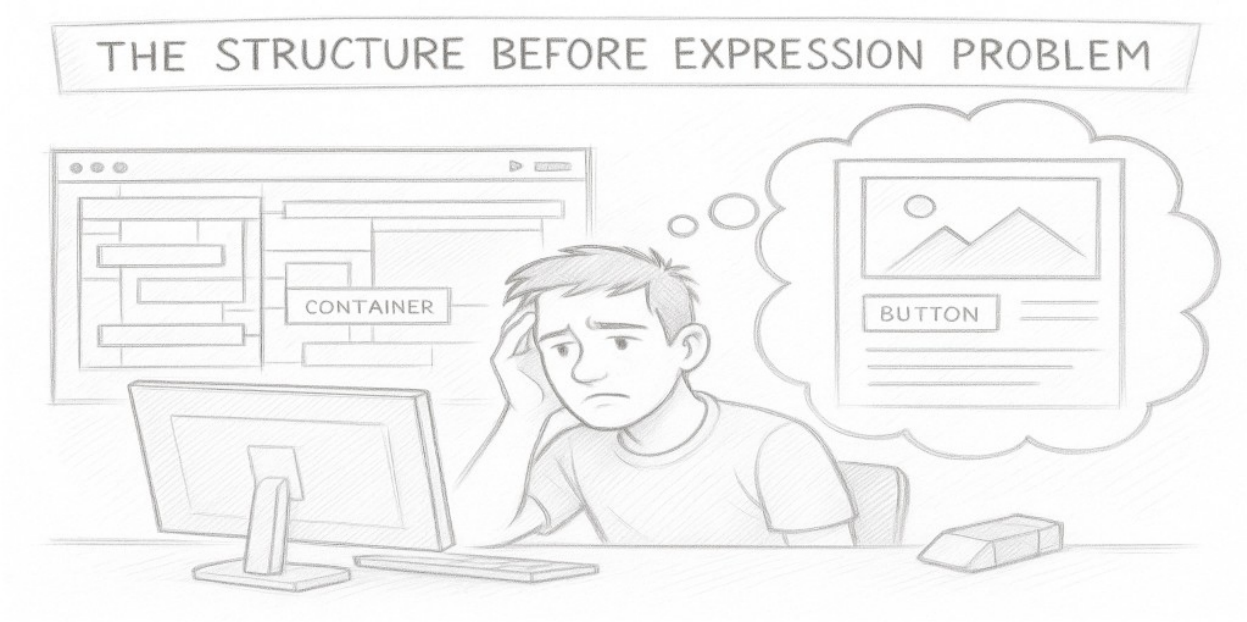
Traditionally, in larger teams, the original artwork of a user interface has been drawn in a separate visually artistic application, then handed-off in a one-time event to so-called front-end developers who have then been tasked with interpreting the visual assets created by the artist, and manually converting them into UI code.

The form views are usually little more than screenshots, and there is no feedback or creative round-trip built-in. In other words, after the hand-off, it is out of the artist's hands forever.

This practice often leads to misinterpretation, loss of nuance, and frustration on both sides. Not only does your intent get diluted in the so-called interpretation, but your artistic team are essentially removed from the design loop at the inception stage.

In many cases, a later version of the same application has to be manually redrawn from more recent screenshots to represent the crusty evolution it has gone through between the time the front-end coders have started playing with it until the next serious visually artistic version is needed.

The Structure Before Expression Problem



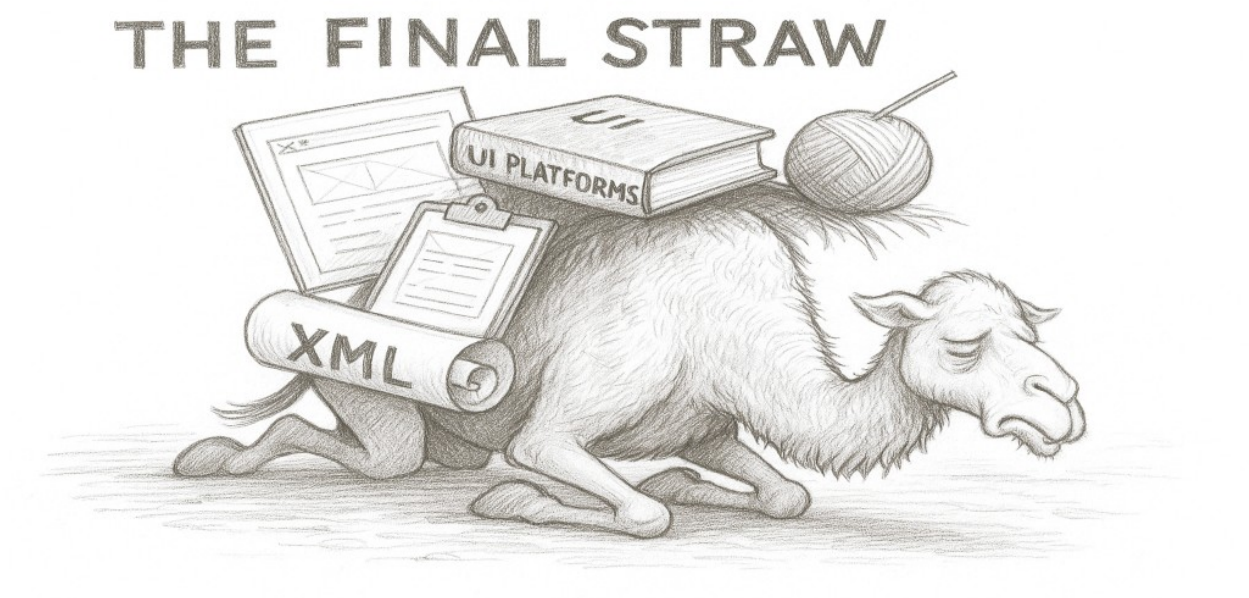
Muddying the waters, in my opinion, are the recent movements of various cloud-based vector drawing services who have gotten the word that modern GUI productivity is an absolute disaster across the board.

In response, they have rolled up their sleeves and created several visually oriented drawing methods and tools to echo exactly what is being done in the code representation of the modern XML-based UI definitions, where all of the multi-layer layouts and form configurations must also be expressed visually before any of the intended shapes can even be placed on the canvas.

The problem is that humans don't think that way - ever. After all, the shapes and their subjective visual placement ARE the art.

Can you imagine having to say that you are going to need a border around something before knowing what you are going to put in that space? How about having to decide how many units of space you want to take up before you know whether that area is going to contain a button or a tree control? Not only do these examples seem convoluted, but wait until you have taken the time and frustration to finally draw a fully containerized panel system in Figma only to realize you would like to rearrange just a couple of small toolbar buttons, before giving up and handing it off because it is going to be too much trouble to tear down this set of containers, figure out how the new blocks have to move against one another, then build all of new layout panels to fit the new button positions where they would look a little better.

The Final Straw



I constantly argue that these factors alone do more to hinder decent user interface design in modern times than all of the tools ever created have ever helped it flourish in the past. I have personally taken several weeks on large UI projects that should have taken only a day or two, had I been able to directly use the basic visual sketches I drew for myself in Inkscape beforehand.

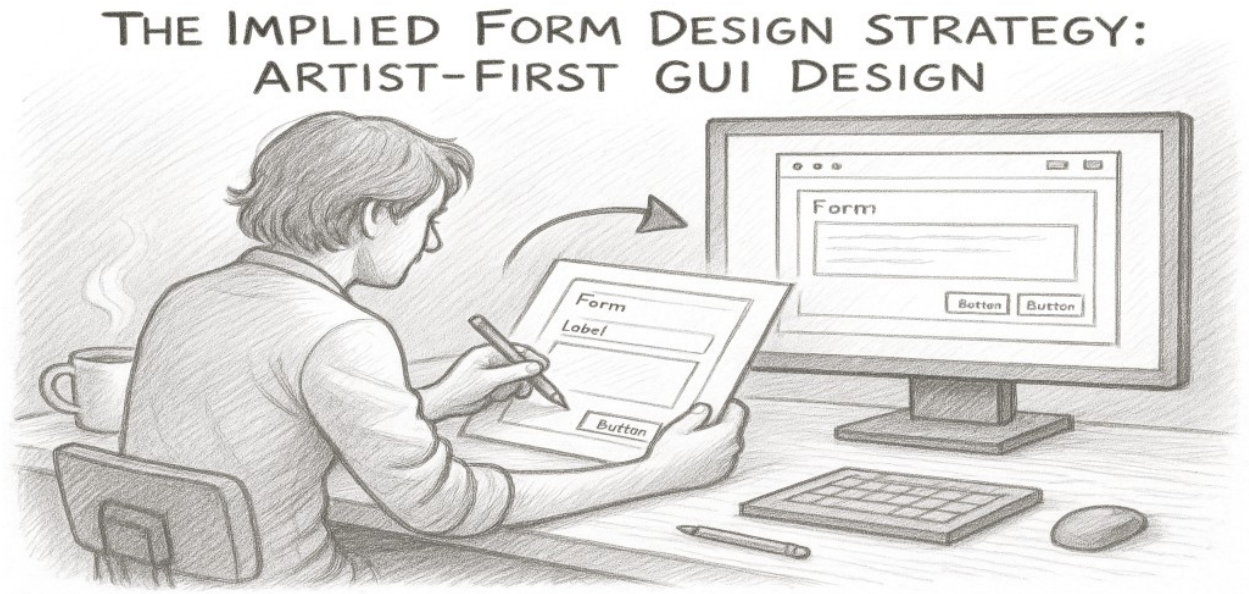
I'll admit that my special way of cheating in the past was to remain active with WinForms for an extraordinarily long time beyond the end of its normal popularity, where I could easily whip up an entire functional and reliable application in less than a day, with the only caveat being that it only worked on Windows. Not only this, but you might be surprised in that meager little editor, I could create fantastically complex control surfaces and interactions that worked at game-level speeds, totally disproving any disparaging messages 'modern' UI platforms were spreading about the so-called disadvantages of widget-based form design and their editors.

Ultimately, however, my perceived stability in the luxury of Microsoft development support that I have directly relied upon for more than 45 years has recently disappeared with the culmination of the enbleepification of all proprietary technology and the death of Windows 10.

Almost surrealistically, I now find myself working exclusively on cross-platform projects while remaining dedicated to desktop applications that can compile and run on every OS, and must accept that all of the major UI platforms in this space use XML variants for their underlying form definitions.

I only tell you this because it is exactly why I finally decided to start working on a tool that helps to make the job of form design so much easier, reliable, and faster to complete; specifically for the graphic artist.

The Implied Form Design Strategy: Artist-First GUI Design



The basic idea of Implied Form Design is that if you can at least indicate what you want on your form, then I can give you that actual, working GUI form that compiles successfully into a desktop application. There shouldn't be any other requirements other than if you feel like becoming more specific about various details. In other words, you imply what you want, and when the conversion runs, I infer a working form from your indications.

SvgTools now supports the philosophy of visual-first design where the entire role of the graphic artist is completely decoupled from any programmatic expectation set up by declarative-style form design, where the artist can remain directly involved throughout the entire evolution of the application, and where there is no expectation whatsoever that the artist memorize anything at all about the variables or coding logic of the end application.

In this setting, there may be one or more individuals specifically in the artist role, one or more individuals working in a type of art translation role, which we'll talk about in a following section, and finally, one or more individuals working in the application role, who ultimately complete the wiring on the functional parts of the underlying application.

In the artist's role, we don't have to have any conversations about abstract data or value handling philosophies, and I think I will even abstain from mentioning any of their names. In this role, our only concern is making the presentation intuitively easy to understand and interact with, in part, or as a whole, and definitely easy, or even fun, to look at during long hours of use.

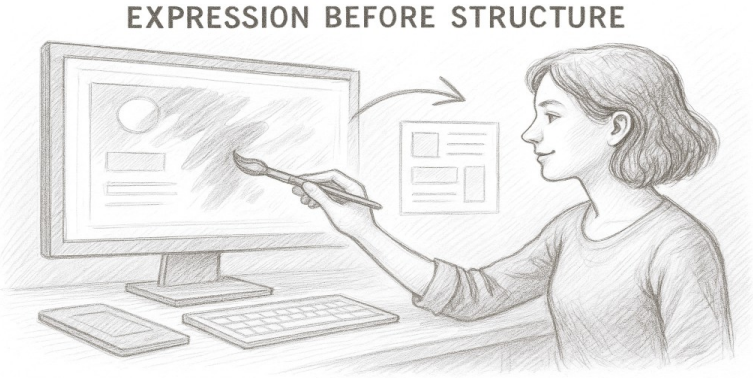
Not only is this approach better than cryptic, bracketed, declarative code, but is also much faster and easier than the previous form of widget-based design.

The Graphic Artist's Bill of Development Rights

Let's start off by setting up some concrete expectations for how this is all going to work. Even if a Bill of Rights document sounds a little strong, I think it's applicable. Technical companies everywhere have spent far too long not considering the best interests of the artists who are actually trying to help them, albeit with little appreciation.

Expression Before Structure

UI design must be allowed to begin with freeform expression. Artists should be able to place elements intuitively, as though they were painting on a canvas, without needing to define containers, hierarchies, or layout rules upfront.

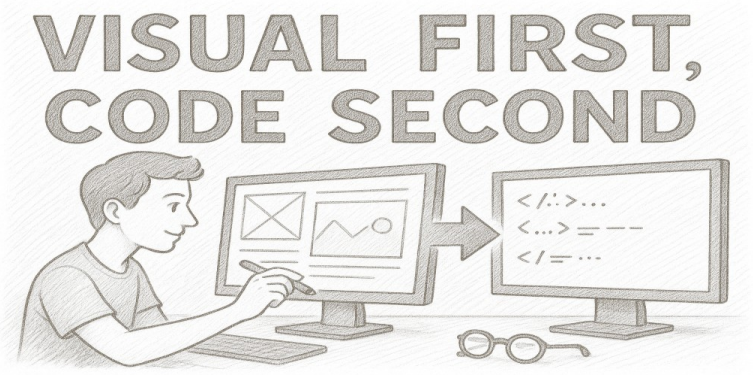


In other words, the artist should be allowed complete freedom to let the design emerge organically. Structure should follow creativity instead of preceding it.

There must also be consideration for the artist's permanent participation throughout the evolution of the application, so that any visual changes needing to be made for the next version of application can be initiated directly by the artist, without requiring that individual to make any changes to the file, in it's interpreted state, before getting started.

Visual First, Code Second

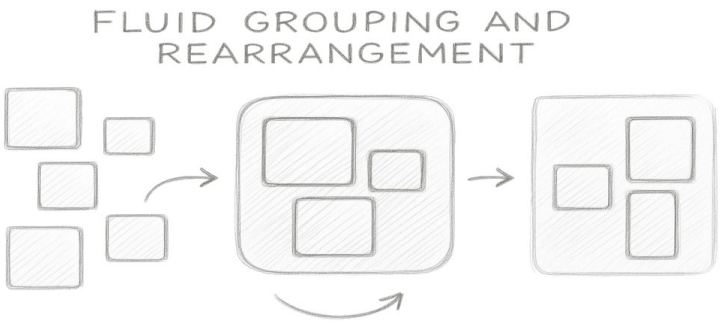
The primary interface for UI creation should be visual, tactile, and itself visually interactive. Code should be an after-the-fact, behind-the-scenes handling of visual intent, and not ever the starting point.



Visual designers should never be forced to learn a markup language to express visual ideas, especially one like XAML, or many other declarative alternatives that follow the same core principles, that have virtually no indexing, modular segregation, or consistency in construction or inter-brand definition. That type of markup language is a complete mess of unorganized characters and inconsistent patterns, and should almost always be considered to be a type of *hands-off* style structure that can be extremely

useful in automation, though not at all helpful for creation or maintenance, let alone casual reading, by humans.

Fluid Grouping and Rearrangement



Elements must be freely movable, groupable, and reconfigurable at any stage of the design process. Grouping should be allowed to be a post-expression act, and should reflect the natural evolution of the layout, help to label important assemblies, or help to quickly align objects on the screen using common alignment tools, depending upon the artist's instantaneous intent. Grouping should also be retractable, meaning that anything that has been grouped can be ungrouped with zero or minimal consequences.

Put another way, user interface design tools must support dynamic visual restructuring without any penalty or complexity.

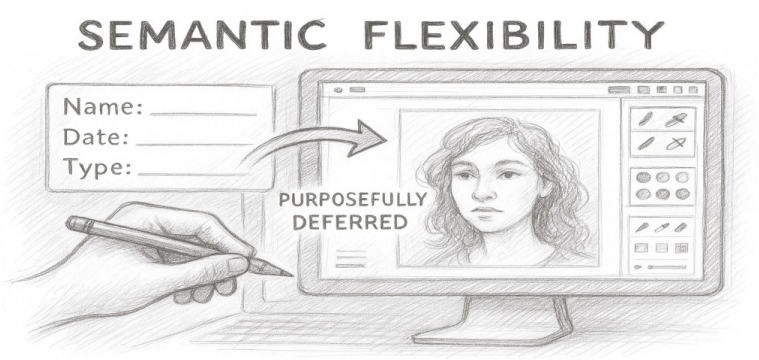
If, in the next version of the user interface, the artist moves several items from one layer or group to another to better represent the visual aspect of the idea, the destruction or loss of previous interpreted objects should not be detrimental to the product, and the individual in the interpretation role should be able to easily re-classify objects that are either new or newly unclassified, without backward harm to the visual art.

Real-Time Feedback

Designers should see immediate visual results of their actions without having to initiate any compile steps or abstract, complicated previews. Just as with painting on a canvas the artist must be able to interact with instant, accurate rendering of the work as it evolves.

The canvas should be alive, interactive, and reflective of every creative decision, as is currently both expected and delivered in the wide variety of vector editors available today.

Semantic Flexibility



Naming, identification, and binding should not only be optional, but purposefully deferred. The system must not demand any rigid declarations while the art is in progress. Phrased in a different way, the function of the file must adapt to, and accept, the artist's form.

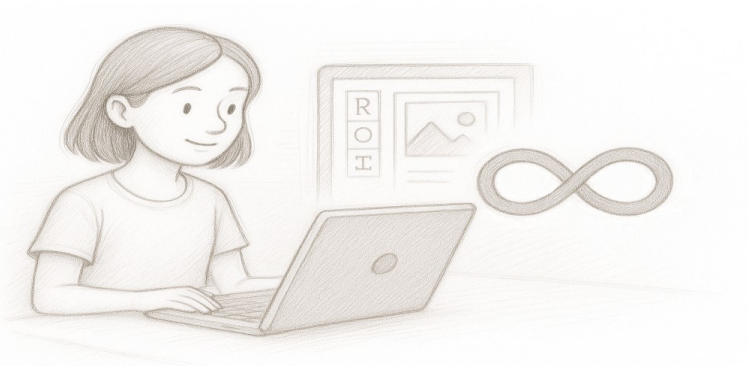
Design as Dialogue

Tools should be collaborators instead of gatekeepers. User interface creation, in general, should feel much more like a conversation between the artist and the tool than a demanding specification that must be met before anything works at all.

If there is any real-time monitoring of the artistic process at all, it should be in the form of suggestions, (actual) intelligent grouping, and adaptive layout hints that only enhance creativity, as opposed to constraining it.

More highly interactive design tool features should not be required at all in the early phases of a successful conversion system.

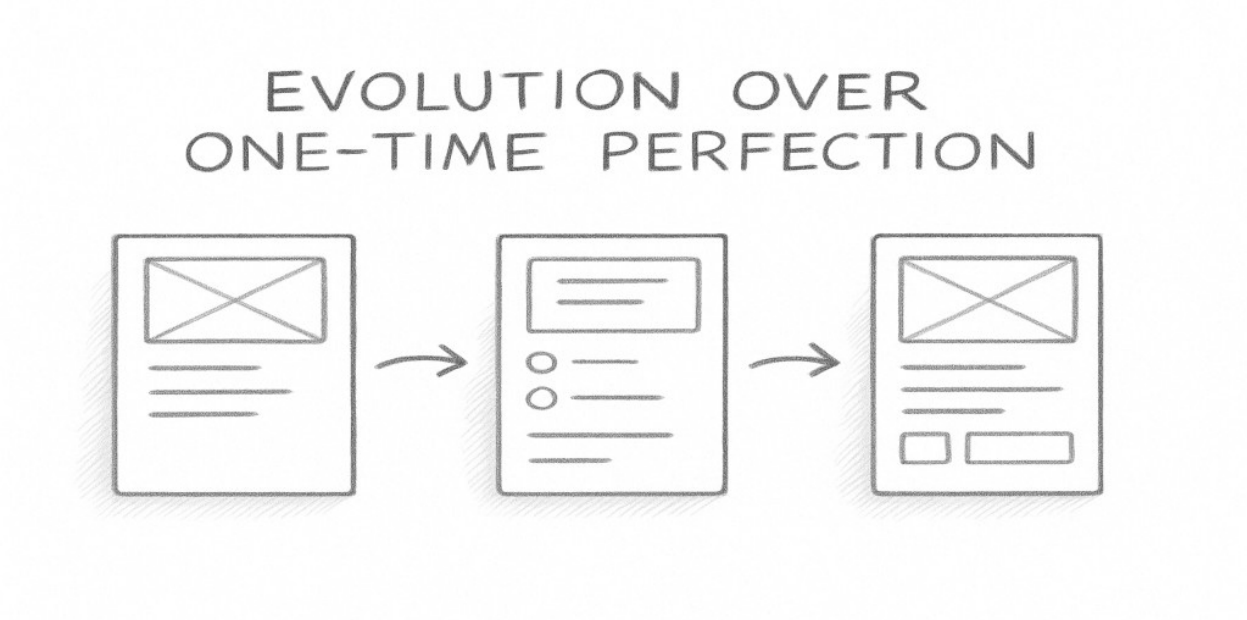
Accessibility of Tools



Creativity should never be gated by technical knowledge. To assure this result, the barrier to entry must be low or even non-existent. Artists should be able to create interfaces without needing to understand ANY programming paradigms, layout engines, or dependency properties.

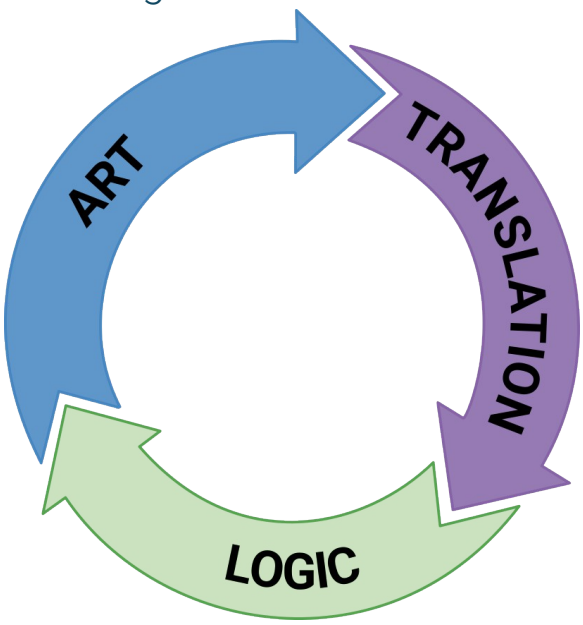
Artists understand what makes a view or visual tool intuitive to another person. They do not need to be involved in how that is going to be achieved in computer logic, which is a different, possibly incompatible, skill.

Evolution Over One-Time Perfection



UI design is not a destination but a journey, and designs should be allowed to evolve continuously. The system must support iterative refinement, versioning, and experimentation without punishing the designer for repeatedly changing their mind.

A Common Workflow for Implied Form Design



Since the design and development roles are so well-defined, there are multiple workflows, the attention to each of which mainly depends upon your team's staffing configuration.

In the next sections, I will present a role-playing walk-through posing the phases as three individual people, The Artist, The Translator, and The Front-End Developer. This is an incremental, cyclical process that can keep repeating both smoothly and indefinitely.

The tasks laid out here reflect actual work being done to build the **Implied Form Design Wizard**, a companion product to SvgTools that not only illustrates the practical tasks involved, but serves as a tool that reduces complication in the conversion task performed by The Translator role.

In each of the following sub-sections, feel free to read only those sections that apply to your role name, skipping the others.

Phase 1 - The Artist

An assignment comes in. Our group is working on a wizard application that will take some basic information from the user, possibly get a couple of options, then run an underlying conversion. We know several details.

- The wizard is called "Implied Form Design Wizard". Its purpose is to help the individual in The Translator role of an Implied Form Design project perform Art-To-GUI conversions quickly and easily without having to use command-line prompts.
- The wizard has several basic pages, each of which have a heading text, a wizard logo and two buttons, all in the same places on each page, which will definitely make copying and pasting the drawings an easy task at the file level. The customer generally knows what they want to see on a couple of the pages, but the rest is up to you.
    - **Page 1**. The customer knows what they want to see here:
      Title: Start Page
      (Heading, large text, bold): **Implied Form Design Wizard**
      (Instructions):
      In this session, we will read your
      graphics-first form design file and
      generate a practical, working form for

your Graphics User Interface application.

Please click **[Next]** to get started.
(Buttons): Cancel, Next
(Image): Customer-supplied. WizardIcon01.png

- **Page 2**.
  Title and heading: **Select Drawing**
  (Prompt text and a text box with ellipsis button helper so the user can either type the path and filename of the source drawing, or click the helper button to open an explorer.)
  (Buttons): Back, Next.

- **Page 3**.
  Title and heading: **Style Worksheet**
  (
  -Checkbox for whether to include a style worksheet.
  -Prompt text and a text box with ellipsis button helper so the user can either type the path and filename of the style worksheet, or click the helper button to open an explorer.
)
  (Buttons): Back, Next.

- **Page 4**.
  Title and heading: **Options**
  (Prompt and text box allowing the user to enter a name for the Unnamed Panel Name. Tool tip or instruction: This is the optional name to provide to an automatically created backing panel that is not known during the visual design.)

- **Page 5**.
  Title and heading: **Output File**
  (Prompt text and a text box with ellipsis button helper so the user can either type the path and filename of the target form, or click the helper button to open an explorer).
  (Checkbox for whether to create a code-backing file).
  (Buttons): Back, Finish.

- **Page 6**.
  Title and heading: **Processing**
  (Progress bar and a message that the converter is running).
  (Message that the conversion process has completed successfully).
  (Message that the conversion process has failed with the following message).
  (Message containing the error message for the operation).
  (Buttons): Cancel, Done.

Note that many of the objects on each page will need to have Labels and specific IDs before the forms can be used in an application. One fast and reliable way to resolve these values is to share draft versions of the original drawings with the front-end developer, who can set those names according to their own policy or taste without making any adjustments to the visual art.

After the art and naming have been completed, the drawings are sent to the translator, who will prepare them for conversion.

Phase 2 - The Translator

Upon receiving notification that the page drawings are available in the shared folder, the translator creates a reusable batch script from the following information that converts each drawing in SvgTools (because the wizard doesn't yet exist). The **SvgTools/Docs/CommandLines.md** file and **SvgTools/Docs/ImpliedFormDesignDemoTutorial.md** both contain helpful information for writing the batch file.

- Prepare the style worksheet: ImpliedFormDesignWizardStyles.json
  - Add CreateBackingFile:True property to FormInformation.
  - Set ProjectName to ImpliedFormDesignWizard.
  - Add a Loaded:window_Loaded property to the Window object.
  - Add the IsVisible:True property to Page1 object.
  - Add the IsVisible:False property to all of the other page objects.
  - Set default TextBox font information to Roboto, 20.

- **Page 1**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page1
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard01.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage01.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- **Page 2**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page2
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard02.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage02.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- **Page 3**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page3
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard03.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage03.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- **Page 4**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page4
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard04.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage04.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- **Page 5**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page5
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard05.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage05.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- **Page 6**.
  Action: ImpliedDesignToAvaloniaXaml
  UnnamedPanelName: Page6
  Working Path: SvgTools/Drawings
  Input File: ImpliedFormDesignWizard06.svg
  Output File: ../Intermediate/ImpliedFormDesignWizardPage06.axaml
  StyleWorksheet: Styles/Avalonia/ImpliedFormDesignWizardStyles.json

- Merge pages into a single Window object.
  Action: XamlMergeContents
  Working Path: SvgTools/Intermediate
  Input File: ImpliedFormDesignWizardPage0?.axaml
  Output File: ../Output/frmImpliedFormDesignWizard.axaml
  Properties: [{ CreateBackingFile:True }]

Run the batch file and verify output. Make any adjustments necessary. Notify the front-end developer that the wizard GUI is available in the shared Output folder.

Phase 3 - The Front-End Developer

At this stage, the front-end developer has received notification that the GUI is ready. Using information supplied by the project manager, the art team, and the translator, they take the following steps.

- Create a full skeleton Avalonia application with the project name **ImpliedFormDesignWizard**.
- Configure the project file settings to taste.
- Update the application file code-behind to pre-load various assets like fonts, and to load the named form (**frmImpliedFormDesignWizard**) as the main window.
- Create **Assets/Images** and **Assets/Fonts** folders and preload them with the project resources.
- Copy the shared form file to the project directory.
- Compile the application to pass its initial smoke-test.
- Use the SvgTools **XamlManifest** action to print out a list of objects and properties present in the XAML form file.
- Referring to the XAML manifest text file, begin wiring the button events and creating the navigation logic for the application.
- On **Page 2**, the developer discovers that they would like to show a warning message on the form if the file wasn't found.

The front-end developer requests that the art team add a message named lblFileNotFound with the text "Error: File not found..." beneath the textbox on page 2.

Phase 1 - The Artist

After receiving a request from the front-end developer that a message named lblFileNotFound reading "Error: File not found..." be added under the text box on page 2, the artist takes the following steps.

- Open the original drawing in Inkscape: ImpliedFormDesignWizard02.svg

- Draw text under the text box reading "Error: File not found...". Then, using the **Object Properties** pane, set the object's ID to **lblFileNotFound**.

- Save the file and notify the translator that the drawing has been changed.

Phase 2 - The Translator

Upon receiving notice from the artist that the drawing has been changed, the translator re-runs the batch script they created earlier, then performs a brief visual inspection on the output file content to verify its consistency.

A notification is sent to the artist that the GUI form file has been changed.

Phase 3 - The Front-End Developer

After notification from the translator that the new GUI form version is ready, the front-end developer takes these steps.

- Copy the new version of the .axaml file from the output directory to the project directory, directly overwriting the previous version.

- Re-run the SvgTools **XamlManifest** action on the file to update the local object and properties list.

- In the code-behind file, complete the wiring of the form to application logic, then connect its notable event activities with calls to the SvgToolsLib library to perform the represented file conversions to complete the functionality.

- Deliver version 1 of the finished project to the project manager.

- Attend after-project party.

Style Extension Worksheet Format

The style extension worksheet is an external JSON file that supplies much more additional information about converting the shapes in the drawing file that is easily drawn or specified in attribute settings.

By using the following uncomplicated structure, extensive adjustments or enhancements can be made on the parsed drawing data to result in a much more tailored form experience requiring even less back-end manual interaction than would even be standard for the basic drawing itself.

```
MainCollection
[
        Comment: string
        MatchPatterns: string[],
        MatchType: string,
        Extensions*: Extension[]
]

Extension
[
        Comment: string
        ExtensionType*: string
        Selector: string
        Settings*: Node[]
]

Node
[
        Name*: string
        Value: string
        Properties: Property[]
        Nodes: Node[]
]

Property
[
        Name*: string
        Value: string
]
```

In the above structure, the MainCollection is the file-level body, consisting of an array of items, each having an identifier, known as a match patterns list, and its own list of extensions to apply when any one of the supplied patterns have been matched.

Extension List Match Pattern

In this configuration file, the idea of a match pattern is loosely similar to the idea of a CSS selector, but takes the more explicit and qualified form.

RegEx:

```
(?i:^(?<qualifier>[a-z][a-z0-9-_\.]+)\:(?<pattern>.*))
```

In the above expression, which is case-insensitive, the first letter of a match pattern will be alpha and followed by one or more alphanumeric, characters, a dash, an underscore, or a period. These characters are interpreted as the *qualifier* of the match pattern, which are followed by a colon (':'), then zero or more characters that reveal the *pattern* to be matched for the given qualifier.

In this version the following qualifiers are supported.

- **class**. Any output shape for which the user has specified the pattern in the customer user property named **Classes**, a property which can specify multiple names, each delimited by your choice of spaces (' ') or semicolons (';').

  Example XAML:
  <Button Classes="Interesting Cool" Text="A Button" />

  Matching Pattern:
  "class:Cool"

- **name**. An output shape whose unique name, as it appears in the property **x:Name**, is equal to the specified pattern.

  Example XAML:
  <Label x:Name="lblGeneric" Content="Generic text." />

  Matching Pattern:
  "name:lblGeneric"

- **property.***{PropertyName}*. An output shape whose specified property value is equal to the given pattern.

  Example XAML:
  <TextBox AcceptsReturn="True" TextWrapping="True" />

  Matching Pattern:
  "property.acceptsreturn:true"

- **tag**. Any output shape whose node tag matches the given pattern.

Example XAML:
<TextBlock Margin="0,5">Filler text.</TextBlock>

Matching Pattern:
"tag:textblock"

Note that the special matching pattern "**tag:FormInformation**" matches form configuration properties, whose values coincide with the **FormInformation** control found in the Implied Form Design Object Reference section, and doesn't output values to a rendered node. Similarly, the related matching pattern "**tag:Application**" matches configuration properties being managed for the surrounding application. Both of those tag match patterns only support the **Properties** extension type (described below).

Match Type

The match type field is an optional value that helps to distinguish how the MatchPatterns property is reviewed for the current record.

The MatchType field can have any of the following values.

- **And**. Requires that all entries in the **MatchPatterns** array are matched before the list will be processed.
- **Or**. Requires that one or more of the entries in the **MatchPatterns** array are matched before the list is processed. This is the default value if not specified.

Extension Type

For each extension list matched in the base-level match pattern arrays, a series of individual style extensions come into view. The activity taken by the extension is mainly determined by its type, which will be one of the following values.

- **ItemsPanel**. Items panel customization for controls that use underlying ItemsPanel style templates during user interaction.

  Properties:

  - **Settings**. A general **Name**/**Value**/**Nodes**/**Properties** tree.

- **Nodes**. Child nodes will be added to the matching node.

  Properties:

  - **Settings**. A general **Name**/**Value**/**Nodes**/**Properties** tree.

- **Properties**. The Settings list contains property name/value pairs that will be placed upon the target node.

  Properties:

  - **Settings**. List of **Name** and **Value** properties.

- **Setters**. Setter child entries will be added to the target nodes collection.

  Properties:

  - **Settings**. List of **Name** and (**Value** or **Nodes**) properties.

- **Style**. A Style entry will be added for the matched output node.

  Properties:

  - **Selector**. Pattern that will activate the style during form operation.
  - **Settings**. List of **Name** and (**Value** or **Nodes**) properties.

- **Template**. (Reserved). A template entry will be added for the matched output node.

To load an external styling worksheet for the conversion, specify it on the command-line using the **/styleworksheet:***{Filename}* parameter.

Implied Form Design Object Reference

This section identifies all of the explicit control intentions and settings for each of those controls.

To get started, name a layer using the syntax **Form-**{Caption}. For example, 'Form-My First Implicit Design Form' will indicate that the form's title caption is going to be 'My First Implicit Design Form'. Alternatively, you can also simply name the layer **Form**, then drag on a **FormInformation** control, upon which, you can set the **Caption** attribute.

Custom User Attribute Index

The following is a list of defined custom user attributes available for implied design to form conversion in SvgTools. Wherever possible, the items in this list apply to every control in the following sections. See each control intent for attributes available only on that control.

Caption

Visual: Freehand text over the control.
Type: String

Description

The caption text of the control.

Columns

Visual: Controls placed side-by-side.
Type: String

Description

A comma-delimited string defining the columns and their general behaviors. Values for each entry are {*|Auto|{Number}[*]}, where '*' indicates that the column will take up the remaining available space after the other columns have been laid out, 'Auto' indicates that the column will occupy as much space as needed by its child control, and {Number} indicates the number of pixels that will be assigned to the column, as an integer value. If this value is omitted, the columns are defined implicitly as 'Auto' sizing by the objects that are placed in this control's area. If {Number} is followed by an asterisk, that number becomes a ratio portion.

ControlStyle-\d{3}

Type: String

Description

Text attributes of a control using the names ControlStyle-000 through ControlStyle-999. To main a CSS-like syntax for each value, replace the colon character (':') with a period ('.') when specifying pseudo-classes.

BorderThickness

Type: Number

Description

The thickness of the border around the control.

Dock

Type: String
Values: Left | Top | Right | Bottom

Description

The area within the parent at which the control should be docked.

Intent

Alias: inkscape:label
Type: String

Description

The type of control implied by the design artist.

Orientation

Type: String
Values: Horizontal | Vertical

Description

The orientation of this control on its parent control. If this value is provided, it overrides the orientation of the visual object.

Margin

Type: Number

Description

An margin size to apply equally around all of the sides of the control, if supplied as an individual value, or to the left, top, right, and bottom sides of the control, if supplied as four comma-delimited values.

MaxValue

Type: Number
Values: (Integer), (DateTime)

Description

The maximum value allowed.

MinValue

Type: Number
Values: (Integer), (DateTime)

Description

The minimum value allowed.

Padding

Type: Number, comma-delimited

Description

A padding size to apply equally around all of the sides of the control, if applied as an individual value, left/right and top/bottom sizes, if a dual number has been supplied, and left, top, right, bottom sides, if supplied as four comma-delimited values.

Prompt

Type: String

Description

The text to be placed in the control when the Text property is blank.

Rows

Type: String, comma-delimited

Description

A comma-delimited string defining the rows and their general behaviors. Values for each entry are {*|Auto|{Number}}, where '*' indicates that the row will take up the remaining available space after the other rows have been laid out, 'Auto' indicates that the row will occupy as much space as needed by its child control, and {Number} indicates the number of pixels that will be assigned to the row, as an integer value. If this value is omitted, the rows are defined implicitly as 'Auto' sizing by the objects that are placed in this control's area.

StyleSnippetFilename

Type: String

Description

Path and filename of the filename from which snippet information will be loaded.

TabEdge

Type: String
Values: Left | Top | Right | Bottom

Description

The edge at which to place the tabs.

Text

Visual: Freehand text over the control
Type: String

Description

The initial user text in the control.

UseBorders

Type: Boolean

Description

Value indicating whether the borders of the visual representative objects will be used to define the border styles of the target controls. If false, no border information is inferred by the target control from the visual source.

Value

Type: Object

Description

The starting value at which to initialize the control.

Intent Index

Following are the available values for the **Intent** customer user property. Use one of these values on each shape that needs to appear on the target user interface form.

Note that in the definitions for the following controls, control location and size is only applied where applicable, such as in the case of when it is being placed on a StaticPanel. In other cases, the control is sized according to its child role within whatever parent has been indicated.

Discrete User Controls

Following is a list of the controls that offer direct user interaction.

- **Button**. A visual area on the form that can be clicked to initiate an action.
- **CheckBox**. A control that toggles its state when clicked.
- **ComboBox**. A drop-down list that lets users either select an item from a predefined set or type in their own input.
- **GridView**. A table-like control that displays data in rows and columns, making it easy to view, sort, and edit structured information.
- **Label**. A simple text element used to display static information or instructions to guide users within a form.
- **ListBox**. A simple control that displays a vertical list of selectable items, ideal for basic selection tasks without some of the structured layout flexibility of a ListView.
- **ListView**. A flexible control that displays a list of items with customizable layouts, making it ideal for presenting rich, structured data like images, text, or interactive object-based elements.
- **MenuBar**. A horizontal strip of drop-down menus that organizes application commands into categories like File, Edit, or Help, giving users quick access to key features.
- **PictureBox**. A simple visual control used to display images, like icons, photos, or graphics, within a form, making it easy to add visual context or decoration.
- **ProgressBar**. A visual indicator that shows the completion status of a task or process, helping users track progress in a clear and intuitive way.
- **RadioButton**. A selection control that lets users choose one option from a group, ensuring only a single choice is active at a time.
- **StatusBar**. A horizontal panel typically placed at the bottom of a form that displays brief messages, progress updates, or contextual information to keep users informed as they interact with the application.
- **TabControl**. A container that organizes content into multiple tabs, allowing users to switch between different views or sections without cluttering the interface.

- **TextBox**. An input field that lets users enter or edit text, essential for collecting information like names, passwords, or comments in a form.
- **TextWithHelper**. A composite control that combines a TextBox with a button, typically labeled with an ellipsis ("..."), to let users enter text manually or launch a helper dialog for guided input.
- **ToolBar**. A horizontal or vertical strip of buttons and controls that gives users quick access to frequently used commands, tools, or actions within an application.
- **TrackBar**. A slider control that lets users select a value from a continuous or discrete range by dragging a thumb along a track, perfect for adjusting settings like transient position, volume, brightness, or zoom.
- **TreeView**. A hierarchical control that displays nested items in a collapsible tree structure, perfect for organizing complex data like file systems, categories, or organizational charts.
- **UpDown**. A numeric input control that lets users adjust a value by clicking up or down arrows, offering a precise alternative to typing in a TextBox.

Information Only

The following controls are used to provide information to the converter.

- **FormInformation**. A virtual control that structures additional general information about the form upon which it is placed, like design implication styles, dialog behavior, etc.

Layout Surfaces and Containers

Following is the list of controls that help to arrange the discrete controls upon the form.

- **FlowPanel**. A layout container that arranges child elements sequentially, horizontally or vertically, wrapping them as needed, making it ideal for dynamic, content-driven interfaces.
- **Grid**. A layout container that generally occupies the parent's available space and arranges child elements into rows and columns, giving you precise control over positioning and alignment for complex UI designs. Elements drawn within a grid layout are
- **GroupBox**. A container that visually frames a set of related controls under a labeled border, helping organize form elements into logical sections for clarity and structure.
- **HorizontalGrid**. A layout container, typically occupying the parent's available space, that arranges child elements in a horizontal sequence across defined columns, offering structured alignment for side-by-side content presentation.

- **HorizontalScrollPanel**. A layout container that arranges child elements in a horizontal line and enables scrolling when the content exceeds the visible width, making it ideal for carousels or wide tool strips.
- **HorizontalStackPanel**. A layout container that arranges child elements in a single horizontal line, ideal for creating toolbars, button rows, or evenly spaced horizontal content.
- **Panel**. A flexible layout surface that allows designers to position child controls at exact coordinates and sizes, making it ideal for custom arrangements where precise placement matters. NOTE: This control might be replaced by StaticPanel.
- **ScrollPanel**. A layout container that enables vertical and/or horizontal scrolling when its child content exceeds the visible area, making it ideal for displaying large or dynamic content within a constrained space.
- **SplitPanel**. A layout container that divides its space into resizable sections, either horizontally or vertically aligned, and typically with a draggable splitter, allowing users to adjust the relative size of content panes dynamically.
- **StaticPanel**. A flexible layout surface that allows designers to position child controls at exact coordinates and sizes, making it ideal for custom arrangements where precise placement matters.
- **VerticalGrid**. A layout container, typically occupying the parent's available space, that arranges child elements in a vertical sequence across defined rows, offering structured alignment for stacked content with consistent spacing and positioning.
- **VerticalScrollPanel**. A layout container that stacks child elements vertically and enables scrolling when the content exceeds the visible height, making it ideal for long forms, lists, or dynamic vertical layouts.
- **VerticalStackPanel**. A layout container that arranges child elements in a single vertical line, making it ideal for stacking controls like labels, text boxes, buttons, or compound controls, in a top-to-bottom flow.

Intents: Button

Category: **Discrete User Control**

The Button control receives a user click to initiate an action, like raising an event that can be intercepted, for example, to perform a sequence of operations.

The button's caption is established by placing a text object inside that button's area. This control is aware of text position, meaning that if you want the caption text to be centered within the control, you should position that text roughly in the center within the button's outline.

Instructions

Follow these steps to use the Button control.

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** of the control and press [Enter] to record it.
- Set the **Label** of the new object to **Button**. Press [Enter] to record the new value.
- Draw text, and/or optionally an image, over the rectangle to set the button's caption.

Text Attributes

The following custom user attributes can be set for this control.

- **ToolTip**. Help text that displays when the mouse is hovered over the control.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.

- Text font size.
- Text font weight.

Intents: CheckBox

Category: **Discrete User Control**

When enabled, the CheckBox control can be on or off. Clicking it once toggles the state. When disabled, the state of the control can be shown but not changed by the user.

Instructions - No Integrated Label

Follow these steps to use the CheckBox control.

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** of the control and press [Enter] to record it.
- Set the **Label** of the new object to **CheckBox**. Press [Enter] to record the new value.

Instructions - Integrated Label

Alternatively If you wish to directly associate a label with the checkbox, follow this slightly longer series of steps.

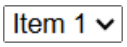
- Create a small rectangle that will represent the visual checkbox.
- Place the text that will serve as the label.
- Group the two objects.
- In the **Object Properties** dialog, with the new group selected, set the **ID** of the checkbox and press [Enter] to record it.
- Set the **Label** of the new object to **CheckBox**. Press [Enter] to record the new value.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).

Intents: ComboBox



Category: **Discrete User Control**

The ComboBox control can contain zero or more items, and when one or more items are present, any one of them can be selected by clicking the down-arrow, then picking it from the drop-down list.

Instructions

You can approach this control in any way from defining a group with the ComboBox intent that contains all of the visual elements of that control, to a series of ungrouped shapes that include a rectangle set to the intent of ComboBox. This example describes the construction of a ComboBox group that contains text and images illustrative of a basic combo box control.

- Draw a rectangle that will visually define the text area of the control.
- If you want a selected text value upon initialization, draw that text directly over the rectangle.
- Draw another rectangle representing the drop-down button.
- Draw a multi-point line representing the down arrow.
- Optionally, group the button rectangle with the down arrow so they can be moved as a single unit.
- Create a group from the above objects.
- In the **Object Properties** dialog, with the new group selected, set the **ID** to a name unique upon the form, and press [Enter] to record the value.
- Set the **Label** of the new object to **ComboBox**. Press [Enter] to record the new value.

Instructions - Add Predefined Items

Predefined items can be added to a combo box by creating a list of text values on a separate layer whose label is identical to the control's ID.

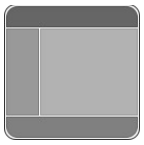
- In the **Layers** dialog, create a new layer and set its label to the value of the combo box's **ID**.
- On the new layer, place one or more text objects. If you want those items to have individual recognizable IDs, make sure to set the **ID** property on each one of them.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: DockPanel



Category: **Layout Surface**

A DockPanel layout container arranges its child elements by attaching them to one of its edges, namely Top, Bottom, Left, or Right, then by filling the width or height directly along that edge. Each element is positioned sequentially along the specified edge, and the remaining space can optionally be filled by a designated child. This layout is well-suited for interfaces that combine fixed-position elements like toolbars or side panels with flexible central content.

Instructions

Unlike the grid-based layout controls, which determine the child's row and column from its position over the parent, the dock panel is a simple container, and any docked children will specify their own docking attributes. Follow these steps to design with a DockPanel control.

- Draw a rectangle on the canvas.
- In the Object Properties dialog, set the unique **ID** of the control. Press [Enter] to record the value.
- Set the **Label** to **DockPanel** and press [Enter] to record the value.
- Draw other controls in front of the DockPanel.
- In each of the base members of the dock panel use the XML Editor to add an attribute named **Dock** with the value **Left**, **Top**, **Right**, **Bottom**, or **Fill**.

Text Attributes

The following custom user attributes can be set for this control.

- .

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: FlowPanel

Category: **Layout Surface**

The flow panel can be oriented horizontally or vertically, and dynamically arranges as many of the child objects as will fit on the current row or column before wrapping to the next row or column, respectively.

Instructions

By default, the orientation of the flow panel is horizontal, meaning that items will be placed in the top row until the width of the control has been filled, when placement will be wrapped to the second row. If you want layout to be processed in vertical order, use the **XML Editor** dialog to set an **Orientation** property to **Vertical**.

Follow these steps to draw a general flow layout panel.

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, optionally set the **ID** of the object.
- Set the **Label** property to **FlowPanel**.
- Either draw child controls over the rectangle area or create a layer whose label corresponds to the ID of this object and draw the child controls on that layer.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control border color.
- Control location (x, y).
- Control size (width, height).

Intents: FormInformation

Category: **Information**

The FormInformation control provides text attribute information for the current form.

Text Attributes

The following custom user attributes can be set for this control.

- **Caption**. The text to be displayed at the top of the form. This value, if supplied, overrides the caption suffix on the **Form-***{Caption}* layer.
- **FormName**. Formal name of the form.
- **ProjectName**. Name of the target project.
- **ThemeName**. Name of the theme to use on the target form.
- **UseBackgroundColor**. True|False. Value indicating whether to use background colors on dropped objects by default on this form.
- **UseBorderColor**. True|False. Value indicating whether to use border colors of drawing objects by default on this form.
- **UseBorderWidth**. True|False. Value indicating whether to use border widths of drawing objects by default on this form.
- **UseCornerRadius**. True|False. Value indicating whether to use border corner radii of drawing objects by default on this form.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control size (width, height).

Intents: Grid

Category: **Layout Surface**

The Grid control is a container that will typically fill the parent area with the specified number of rows and columns.

Text Attributes

The following custom user attributes can be set for this control.

- .

Child Text Attributes

Directly contained members of this layout surface can indicate column widths or row heights for the grid.

- **ColumnSpan**. Number of grid columns to be spanned by this object.
- **ColumnWidth**. Indicates the width of the grid column occupied by this item and others in the same column area. Valid values are '**auto**', '*****', or an integer number to indicate a fixed width in pixels. If no value is specified for any control in the column, '*****' is used. This value should only appear for one item in a column. Otherwise, the first item found during conversion will be used.
- **RowHeight**. Indicates the height of the grid row occupied by this item and others in the same row area. Valid values are '**auto**', '*****', or an integer number to indicate a fixed height in pixels. If no value is specified for any control in the row, '*****' is used. This value should only appear for one item in a row. Otherwise, the first item found during conversion will be used.
- **RowSpan**. Number of grid rows to be spanned by this object.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: GridView

Category: **Discrete User Control**

A control that facilitates table-based and list-based editing, filtering, and sorting.

Instructions

Follow these steps to use the GridView control.

- Draw a rectangle on the canvas.
- In the Object Properties dialog, set the unique **ID** of the control. Press [Enter] to record the value.
- Set the **Label** to **GridView** and press [Enter] to record the value.
- To add initial column definitions to the control, drop one or more text shapes near the inside top of the control.

Text Attributes

The following custom user attributes can be set for this control.

- **AutoGenerateColumns**. Boolean. A value indicating whether columns should be auto-generated from data. Default = False.
- **CanUserSortItems**. Boolean. A value indicating whether columns can be sorted by the user. Default = True.

Child Text Attributes

Directly contained members of this layout surface can indicate column headers and widths for the grid.

- **ColumnWidth**. Indicates the width of the grid column occupied by this item and others in the same column area. Valid values are '**auto**', '*****', or an integer number to indicate a fixed width in pixels. If no value is specified for any control in the column, '*****' is used. This value should only appear for one item in a column. Otherwise, the first item found during conversion will be used.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: GroupBox

Category: **Layout Surface**

The GroupBox control is a container with a border decoration and a caption.

Instructions

Follow these steps to design with the GroupBox control.

- Draw a rectangle on the canvas.
- In the Object Properties dialog, set the unique **ID** of the control. Press [Enter] to record the value.
- Set the **Label** to **GroupBox** and press [Enter] to record the value.
- To add the title, place a text element near the top left corner of the control.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: HorizontalGrid

Category: **Layout Surface**

The HorizontalGrid control is used to fill the width of the parent by distributing it into the specified number of columns.

Instructions

Follow these steps to use the HorizontalGrid control.

- Draw a rectangle on the canvas.
- In the **Layers** dialog, set the new object's label to **HorizontalGrid**.
- Add other controls in front of the object to place them on the surface.

Child Text Attributes

Directly contained members of this layout surface can indicate column widths for the grid.

- **ColumnWidth**. Indicates the width of the grid column occupied by this item and others in the same column area. Valid values are '**auto**', '*****', or an integer number to indicate a fixed width in pixels. If no value is specified for any control in the column, '*****' is used. This value should only appear for one item in a column. Otherwise, the first item found during conversion will be used.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: HorizontalScrollPanel

Category: **Layout Surface**

The HorizontalScrollPanel behaves as a HorizontalStackPanel whose width is limited. Optionally, when the child control is narrower than the width of the scroll panel, no scroll bar is shown. The range of the scroll bar is adjusted automatically according to the difference in widths between the scroll panel window and the child control.

Text Attributes

The following custom user attributes can be set for this control.

- **Spacing**. Number. Number of pixels by which to space immediate members. Default = 10.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: HorizontalStackPanel

Category: **Layout Surface**

The HorizontalStackPanel control is a container for one or more other controls that will be distributed horizontally across the display according to the widths of each of the child controls.

Text Attributes

The following custom user attributes can be set for this control.

- **Spacing**. Number. Number of pixels by which to space immediate members. Default = 10.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: Label

Label Text

Category: **Discrete User Control**

The Label control is used to display non-interactive text upon the form.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: ListBox

Category: **Discrete User Control**

The ListBox control contains zero or more items, each of which can be selected. In multi-select mode, more than one item can be selected at any time, while in the default single-select mode, only one or zero items can be selected.

Instructions

Follow these steps to use the ListBox control.

- Draw a rectangle on the canvas.
- In the **Layers** dialog, set the new object's label to **ListBox**.
- Add other controls in front of the object to place them on the surface.

Note that you can add images or text, either in the front of the listbox control, or upon an extension layer sharing a label with the ID of your listbox. If you use the extension layer, you can also apply one or more user variables to the elements displayed present on that layer by using the following special syntax.

listBoxName-*varName1*:*varValue1*[;*varName2*:*varValue2*] ...

... where listBoxName is the ID you have assigned to your listbox shape, and varNameN:varValueN are name/value assignments to set on the elements in that layer. For example, the label:

lstNavigation-Tag:Hover

...defines the layer as containing list items that will be given a Tag="Hover" property assignment.

This naming technique is extremely useful for cases where you are defining multiple sets of duplicate elements for cases where the image might change under a certain condition, as in the example of button images that might be defined to have Hover, Normal, and Selected Tag values.

Text Attributes

The following custom user attributes can be set for this control.

- **ImageFilename**. Each of the image members of the ListBox control can be assigned a ImageFilename attribute that determines the image file to be loaded at runtime.
- **SelectionMode**. {Single|Multiple|Toggle|AlwaysSelected}. The selection mode for the list. Default = Single.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: ListView

Category: **Discrete User Control**

The ListView control is similar to the ListBox in general functionality, but has more built-in presentation capabilities, mainly in that an icon and a label can be assigned to any item, and the control can be viewed in multiple display modes, including Extra Large, Large, Normal, and Small Icon View, Item View, and Detail View, the last of which is typically a multi-column list with an icon in the left column.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: MenuBar

Category: **Discrete User Control**

The MenuBar control is a classic form design element that provides users with a structured way to access commands and features. Think of it as the row of options you see at the top of most desktop applications, like File, Edit, or Help.

Instructions

Follow these general steps to design with this control.

- Draw a new rectangle object onto the canvas.
- Visually align the control to the top, left, bottom, or right side of the parent area to indicate its docking position.
- In the Object Properties dialog for this object, set the **ID** to a memorable name, like **mnuMain** (The following steps will assume that mnuMain is the control's name. Wherever that name appears, replace it with whatever name you have chosen for your own MenuBar). Press [Enter] to record the value, then [Tab] to move to the **Label** textbox.
- Set the **Label** value to **MenuBar**
- Switch to the Layers dialog and create a new layer.
- Set the label of the layer to **mnuMain**
- With the new layer active, drop a text with the caption **&File**. In the Object Properties panel, set the ID to **mnuFile**. This indicates that what follows is a menu panel named File, and the default navigation key for activating it is Alt+F.
- Under the text, draw a rectangle.
- Onto the rectangle, draw one or more text entries. They will be interpreted as they appear vertically on the canvas. Using the **ID** property in the Object Properties dialog, give each item a unique name, prefixing it with **mnuFile**, as in **mnuFileSave** for the **File / Save** option, and **mnuFileOpen** for the **File / Open** option.
- Whenever sub-menus need to be designed, use a similar approach in a slightly different area of the canvas, and continuing the naming convention as started on its parent menu. For example, if the **File / Export** option is named **mnuFileExport**, and that entry leads to a sub-menu, then drop a new text item on a blank area of the screen with the text **Export** and the **ID menuFileExport**, create a rectangle panel directly beneath it, and a set of items that continue the naming chain, as in **mnuFileExportAsText** for the option **Export /**

As Text, and **mnuFileExportAsBinary** for the option **Export / As Binary**, the latter of which will be fully expressed in the application context as **File / Export / As Binary**.

Text Attributes

The following custom user attributes can be set for this control.

- **Dock**. {Left|Top|Right|Bottom}. The area within the parent at which the control should be docked.

Visual Attributes

There are no specific visual attributes transferred from the drawing to the resulting target control.

Intents: Panel

Category: **Layout Surface**

A Panel control is a container with static positioning that can be used within any other statically positioned container to provide organization.

Because the panel control is able to automatically determine the structures of objects laid out in front of it as being shaped in the forms of **Vertical**, **Horizontal**, or **Grid**, it is able to save a lot of time when creating a layout.

- Vertically aligned members are fitted into a **VerticalStackPanel** control.
- Horizontally aligned members are fitted into a **HorizontalStackPanel** control.
- Grid-aligned members, in other words, members that occupy more than one column and one row as a group, are automatically fitted into a **Grid** control.

A panel can either be created as a stand-alone rectangle shape, or it can be represented by a group.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).

Intents: PictureBox

Category: **Discrete User Control**

The picture control displays a picture at the specified location on the form.

Text Attributes

The following custom user attributes can be set for this control.

- **ImageFilename**. Name of the image file to be loaded at runtime.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).

Intents: ProgressBar

Category: **Discrete User Control**

The ProgressBar control indicates the amount of completion a process has reached.

Instructions

Follow these steps to use the ProgressBar control.

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** of the control and press [Enter] to record it.
- Set the **Label** of the new object to **ProgressBar**. Press [Enter] to record the new value.
- The default values of the drawn control are Minimum:0; Maximum:100; Value:0. If you wish to define a different range or value open the XML Editor and add your own values for those attributes.

Text Attributes

The following custom user attributes can be set for this control.

- **Maximum**. The maximum allowable value.
- **Minimum**. The minimum allowable value.
- **Value**. The initial value of the control.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).

Intents: RadioButton

Category: **Discrete User Control**

The RadioButton, typically used in groups of more than one, is similar to a CheckBox control in that it has on and off states and that clicking upon one can toggle its status to on. However, it differs from the CheckBox in that only one item in the group can be on at any given time. Turning one item on effectively results in all other items in the group being turned off.

Instructions

Follow these steps to use the RadioButton control.

- Draw an oval on the canvas.
- In the **Object Properties** dialog, set the **ID** of the control and press [Enter] to record it.
- Set the **Label** of the new object to **RadioButton**. Press [Enter] to record the new value.
- In the **XML Editor** dialog, set the text attributes for group name and label content..

Text Attributes

The following custom user attributes can be set for this control.

- **Content**. (Optional). The label text to be included directly with this control.
- **GroupName**. The name of the selection group to which the button belongs. Only one button per group can be checked at any time. Checking one automatically unchecks all of the others.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).

Intents: ScrollPanel

Category: **Layout Surface**

The ScrollPanel control is a container that can handle the scrolling and relative in-window positioning of the child control.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: Slider

Category: **Discrete User Control**

The Slider control allows the user to set a value within a range between minimum and maximum values. The control can be displayed in horizontal (default) or vertical orientations.

Text Attributes

The following custom user attributes can be set for this control.

- **Frequency**. The tick frequency on the scale.
- **Maximum**. Maximum allowable value. Default = 100.
- **Minimum**. Minimum allowable value. Default = 0.
- **Orientation**. {Horizontal|Vertical}. The orientation of this control on its parent control. If this value is provided, it overrides the orientation of the visual object.
- **Snap**. {True|False}. Value indicating whether snap to ticks is enabled.
- **Value**. The initializing value. Default = 0.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control orientation (horizontal, vertical).
- Control size (width, height).

Intents: SplitPanel

Category: **Layout Surface**

The SplitPanel control provides a splitter bar between two child controls, allowing the user to resize both objects by dragging the splitter bar. Horizontal and vertical modes are available, and determined by which orientation is used when placing the child controls.

Instructions

Follow these steps to use the SplitPanel layout control.

- Draw a rectangle in available area, set its location and size as appropriate.
- In the **Layouts** dialog, locate the object and set its label to **SplitPanel**.
- Draw child controls on the left and right, if the panel is to be oriented horizontally, or top and bottom, if the panel is to be oriented vertically.

Text Attributes

The following custom user attributes can be set for this control.

- **Orientation**. (Optional). The orientation of the layout direction within the panel, **Vertical** indicates a top panel, horizontal splitter, and bottom panel, while **Horizontal** indicates a left panel, vertical splitter, and right panel. If this value is specified, it overrides the visual layout.
- **PanelASize**. The size of the left or top panel. Default = "Auto".
- **PanelBSize**. The size of the right or bottom panel. Default = "*".
- **SplitterSize**. The size of the splitter bar. Default = "5".

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: StaticPanel

Category: **Layout Surface**

The StaticPanel control provides a surface upon which specific, coordinate-based layout of multiple child objects can be performed.

Instructions

Follow these steps to use the StaticPanel layout control.

- Draw a rectangle in available area, set its location and size as appropriate.
- In the **Layouts** dialog, locate the object and set its label to **StaticPanel**.
- Draw child controls anywhere in front of this panel.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).

Intents: StatusBar

Category: **Discrete User Control**

The StatusBar control is typically aligned at the bottom of the form or parent control, and contains basic controls, like labels and progress bars, help the user to remain aware of current application states.

In this version, the status bar only has one text message at the left side of the control.

Instructions

Follow these steps to use the StatusBar control.

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** of the control and press [Enter] to record it.
- Set the **Label** of the new object to **StatusBar**. Press [Enter] to record the new value.
- Draw text over the control to indicate the initial message.

Text Attributes

The following custom user attributes can be set for this control.

- .

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control size (height).

Intents: TabControl

Category: **Discrete User Control**

The TabControl is a multi-page container control having selection tabs at one of the edges, often the top.

Instructions

Use layers to place child controls in the pages of a TabControl.

First, create one layer for each of the control's tabs. Name that layer using the TabControl's id value, a hyphen character, the Header keyword, and the Text to display on the tab. For example, if the TabControl id="tctlMain" and you want the tab to display the text "Book Sales", then create a new layer named "tctlMain-Header:Book Sales".

Next, treat the new layer somewhat as a form by adding controls and appropriate settings.

Text Attributes

The following custom user attributes can be set for this control.

- **TabEdge**. {Left|Top|Right|Bottom}. The edge at which to place the tabs.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).

Intents: TextBox

Category: **Discrete User Control**

The TextBox is a control that allows the user to enter or edit text, which helps to maintain the value of a field.

Instructions

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** to the unique ID of the control on the form. For example, 'txtDisplayName'. Press [Enter] to record the value.
- Set the Label to **TextBox** and press [Enter] to record the value.
- Optional. To configure the TextBox with existing text, drop a general purpose text object inside its area.
- Optional. To indicate an in-control label, such as those found in text boxes in the style of Material Design, drop a text object inside the TextBox area and set its label to **Label**.

Text Attributes

The following custom user attributes can be set for this control.

- **AcceptsReturn**. Value indicating whether the textbox accepts presses of the [Enter] key, as well as Carriage Return and Line Feed characters. Default = False.
- **PasswordChar**. The character to use as a password-style mask. If blank, normal text is displayed.
- **Prompt**. The text to be placed in the control when the Text property is blank.
- **Text**. The initial user text in the control. If present, this value overrides the text found in the visual design.
- **TextWrapping**. (NoWrap | Wrap | WrapWithOverflow). Controls text word-wrap. Use this property in conjunction with AcceptsReturn. Default = NoWrap.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.

- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.
- Text font name.
- Text font size.
- Text font weight.

Intents: TextWithHelper

Category: **Discrete User Control**

The TextWithHelper control is a combination of the standard TextBox and Button controls.

Instructions

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** to the unique ID of the control on the form. For example, 'txtFilename'. Press [Enter] to record the value.
- Set the Label to **TextWithHelper** and press [Enter] to record the value.
- Optional. To configure the TextBox with existing text, drop a rectangle with the label of **TextBox**, then a general purpose text object inside its area.
- Optional. To configure the Button with a text value other than '...', drop a rectangle with the label of **Button**, then a general purpose text object inside its area.

Text Attributes

The following custom user attributes can be set for this control.

- **Caption**. The text caption to be displayed on the button. The default value is '...'.
- **Prompt**. The text to be placed in the control when the Text property is blank.
- **Text**. The active user text in the control. This value overrides any visual text value placed on the control.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Button control background color.
- Button control border color.
- Button control corner radius.
- Control location (x, y).
- Control size (width, height).

- Text font color.
- Text font name.
- Text font size.
- Text font weight.
- TextBox control background color.
- TextBox control border color.
- TextBox control corner radius.

Intents: ToolBar

Category: **Discrete User Control**

The ToolBar control, typically docked to the top, bottom, left, or right sides, contains one or more toolbar buttons, which are typically similarly sized, icon-based buttons arranged in a single row or column.

Instructions

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** to the unique ID of the control on the form. For example, 'mnuMain'. Press [Enter] to record the value.
- Set the Label to **ToolBar** and press [Enter] to record the value.
- Optional. Add a toolbar button by drawing a rectangle across the top of the toolbar, setting its Label to **Button**, then drawing either text, an image, or both across the top of that button.

Text Attributes

The following custom user attributes can be set for this control.

- **Dock**. {Left|Top|Right|Bottom}. The area within the parent at which the control should be docked.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Item Button Icon.

Intents: TreeView

Category: **Discrete User Control**

The TreeView control is a user interface element that displays data in a hierarchical structure, much like folders and files in a file explorer application. It's perfect for organizing and navigating complex sets of related items.

Instructions

Follow these steps to use the TreeView control.

- Draw a rectangle area on the canvas.
- In the **Object Properties** dialog, set the **ID** to the unique name for the control, like **trvMain**, for example. Press [Enter] to record the value.
- Set the **Label** to **TreeView**. Press [Enter] to record the value.

Pre-Loading Instructions

Follow these steps if you wish to pre-load items to display in the tree when the form initializes.

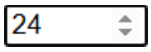
- In the Layers dialog, create a new layer and set its label to the ID of the TreeView to represent. In the above example, the label of the new layer will be **trvMain**.
- Anywhere on the canvas, but usually within the area of the tree, drop a series of hierarchically arranged texts or image / text grouped items.
- If the nodes will have notable identifications, set the **ID**s of either the stand-alone texts or the groups containing the image and text combinations.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: UpDown



Category: **Discrete User Control**

The UpDown control lets users adjust a value by clicking up or down arrows, rather than typing it manually. It's ideal for situations where you want to restrict input to a specific range or step size.

Instructions

- Draw a rectangle on the canvas.
- In the **Object Properties** dialog, set the **ID** to the unique ID of the control on the form. For example, 'txtAValue'. Press [Enter] to record the value.
- Set the Label to **UpDown** and press [Enter] to record the value.
- Optional. To configure the control with an existing value, drop a general purpose text object inside its area.

Text Attributes

The following custom user attributes can be set for this control.

- **Maximum**. The maximum value allowed. Default is 100.
- **Minimum**. The minimum value allowed. Default is 0.
- **Value**. The starting value at which to initialize the control. If this value is supplied, the visual text value will be overridden.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).
- Text font color.

- Text font name.
- Text font size.
- Text font weight.

Intents: VerticalGrid

Category: **Layout Surface**

The VerticalGrid control is used to fill the width of the parent by distributing it into the specified number of rows.

Instructions

Follow these steps to use the VerticalGrid control.

- Draw a rectangle on the canvas.
- In the **Layers** dialog, set the new object's label to **VerticalGrid**.
- Add other controls in front of the object to place them on the surface.

Text Attributes

The following custom user attributes can be set for this control.

- .

Child Text Attributes

Directly contained members of this layout surface can indicate row heights for the grid.

- **RowHeight**. Indicates the height of the grid row occupied by this item and others in the same row area. Valid values are '**auto**', '*****', or an integer number to indicate a fixed height in pixels. If no value is specified for any control in the row, '*****' is used. This value should only appear for one item in a row. Otherwise, the first item found during conversion will be used.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: VerticalScrollPanel

Category: **Layout Surface**

The VerticalScrollPanel control behaves as a vertical stack panel whose height is limited. Optionally, when the child control is shorter than the height of the scroll panel, no scroll bar is shown. The range of the scroll bar is adjusted automatically according to the difference in heights between the scroll panel window and the child control.

Instructions

Follow these steps to define the vertical scroll panel.

- Draw a rectangle onto the canvas.
- In the **Object Properties** dialog, set the **ID** of the object, then press [Enter] to record the new value.
- Set the **Label** of the new object to **VerticalScrollPanel**. Press [Enter] to record the new value.

Preloading Instructions

Since this control's content can potentially extend far beyond the top and bottom of its viewport, the technique for loading items can either follow a standard panel-on-panel approach, where you can drop child shapes directly over the base control, or you can create a separate layer whose label is set to the ID of this control. Follow these steps to preload the control with child controls on a separate layer.

- In this example, the **ID** of the **VerticalScollPanel** will be **scrollMain**.
- In the **Layers** dialog, create a new layer and set its label to **scrollMain**.
- Drop child shapes to be displayed onto the new layer.

Text Attributes

The following custom user attributes can be set for this control.

- **Spacing**. Number. Number of pixels by which to space immediate members. Default = 10.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: VerticalStackPanel

Category: **Layout Surface**

The VerticalStackPanel control is a container for one or more other controls that will be distributed vertically across the display according to the heights of each of the child controls.

Text Attributes

The following custom user attributes can be set for this control.

- **Spacing**. Number. Number of pixels by which to space immediate members. Default = 10.

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control location (x, y).
- Control size (width, height).

Intents: WidgetPanel

Category: **Layout Surface**

The WidgetPanel control provides a surface upon which aligned layout of multiple child objects can be performed relative to one another.

By default, the widget panel is assumed if you don't specify any kind of panels on a form. Most notably, any controls you draw on the widget panel can be aligned in different ways.

- **Aligned with Parent**. The control can be aligned by its implied distance from the Top, Left, Bottom, or Right sides of the form or parent panel using the Anchor property.
- **Aligned with Sibling**. The control can be aligned by its implied distance from another control on the form, using the Above, LeftOf, Under, and RightOf properties.

If no alignments are specified for any of the child controls on the panel, Anchor="Top, Left" is assumed for each one. If an alignment is specified for only one axis, the other axis is assumed to be Left for horizontal or Top for vertical.

You can specify multiple parent alignments on the same axis. For example, if you specify Anchor="Left, Right" the position of the control will remain the same horizontally, but its width will change with the parent. On the other hand, if you specify Anchor="Top, Bottom", the control will remain stationary in the same position horizontally, but its height will change with the parent.

Instructions

If you are starting with an empty form and wish to follow a widget-style design procedure, you can just start drawing controls with no backing panel shape. Otherwise, follow these steps to use the WidgetPanel layout control.

- Draw a rectangle in available area, set its location and size as appropriate.
- In the **Layouts** dialog, locate the object and set its label to **WidgetPanel**.
- Draw child controls anywhere in front of this panel. On each child control, you can optionally add zero or more property settings from the Child Text Attributes list below.

Child Text Attributes

Directly contained members of this layout surface can indicate their alignment preferences.

- **Above**. Specifies that the bottom side of this control is always the current distance above the top of the other control named in the property value.
- **Below**. Specifies that the top side of this control is always the current distance under the bottom of the other control named in the property value.
- **AlignBottom**. Indicates the bottom side of this control will be aligned with the bottom side of the other control specified in the property value.
- **AlignCenter**. Indicates the center horizontal point of this control will be aligned horizontally with the center horizontal point of the other control specified in the property value.
- **AlignLeft**. Indicates the left side of this control will be aligned with the left side of the other control specified in the property value.
- **AlignMiddle**. Indicates the middle vertical point of this control will be aligned vertically with the center vertical point of the other control specified in the property value.
- **AlignRight**. Indicates the right side of this control will be aligned with the right side of the other control specified in the property value.
- **AlignTop**. Indicates the top side of this control will be aligned with the top side of the other control specified in the property value.
- **Anchor**. Indicates the alignment to the parent panel or form. Valid choices are any combination of **Top**, **Left**, **Bottom**, or **Right**. If no anchors are specified for any of the controls on the panel, Anchor="Top, Left" is assumed. If an alignment is specified for only one axis, the other axis is assumed to be Left for horizontal or Top for vertical.
-
- **LeftOf**. Specifies that the right side of this control is always the current distance to the left of the left side of the other control named in the property value.
- **RightOf**. Specifies that the left side of this control is always the current distance to the right of the right side of the other control named in the property value.
-

Visual Attributes

The following visual attributes are transferred from the drawing to the resulting target control.

- Control background color.
- Control border color.
- Control corner radius.
- Control location (x, y).
- Control size (width, height).