# Contributing to Terminal.Gui

We welcome contributions from the community. See [Issues](https://github.com/migueldeicaza/gui.cs/issues) for a list of open [bugs](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Abug) and [enhancements](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Aenhancement). Contributors looking for something fun to work on should look at issues taged as:

- [good first issue](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3A%22good+first+issue%22)

- [up for grabs](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Aup-for-grabs)

- [help wanted](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Aup-for-grabs)

## Coding Style

\*\*Terminal.Gui\*\* follows the [Mono Coding Guidelines](https://www.mono-project.com/community/contributing/coding-guidelines/). `../.editorconfig` uses Visual Studio to help enforce these.

## User Experience Tenets

\*\*Terminal.Gui\*\*, as a UI framework, heavily influences how console graphical user interfaces (GUIs) work. We use the following [tenets](https://ceklog.kindel.com/2020/02/10/tenets/) to guide us:

\*NOTE: Like all tenets, these are up for debate. If you disagree, have questions, or suggestions about these tenets and guideliens submit an Issue using the [design](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Adesign) tag.\*

1. \*\*Honor What's Come Before\*\*. The Mac and Windows OS's have well established GUI idioms that are mostly consistent. We adhere to these versus inventing new ways for users to do things. For example, \*\*Terminal.Gui\*\* adopts the `ctrl/command-c`, `ctrl/command-v`, and `ctrl/command-x` keyboard shortcuts for cut, copy, and paste versus defining new shortcuts.

2. \*\*Consistency Matters\*\*. Common UI idioms should be consistent across the GUI framework. For example, `ctrl/command-q` quits/exits all modal views. See [Issue #456](https://github.com/migueldeicaza/gui.cs/issues/456) as a counter example that should be fixed.

3. \*\*Honor the OS, but Work Everywhere\*\*. \*\*Terminal.Gui\*\* is cross-platform, but we support taking advantage of a platform's unique advantages. For example the Windows Console API is richer than the Unix API in terms of keyboard handling. Thus, in Windows pressing the `alt` key in a \*\*Terminal.Gui\*\* app will activate the `MenuBar`, but in Unix the user has to press the full hotkey (e.g. `alt-f`) or `F9`.

4. \*\*Keyboard first, Mouse also\*\*. Users use consoles primarily with the keyboard; \*\*Terminal.Gui\*\* is optimized for getting stuff done without using the mouse. However, as a GUI framework, the mouse is essential, thus we strive to ensure that everything also works via the mouse.

## Public API Tenets & Guidelines

\*\*Terminal.Gui\*\* provides an API that is used by many. As the project evolves contributors should follow these [tenets](https://ceklog.kindel.com/2020/02/10/tenets/) to ensure consistency and backwards compatabiltiy.

\*NOTE: Like all tenets, these are up for debate. If you disagree, have questions, or suggestions about these tenets and guideliens submit an Issue using the [design](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Adesign) tag.\*

1. \*\*Stand on the shoulders of giants.\*\* Follow the [Microsoft .NET Framework Design Guidelines](https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/) where appropriate.

2. \*\*Don't Break Existing Stuff.\*\* Avoid breaking changes to user behavior or the public API; instead, figure out how to implement new functionality in a parallel way. If a breaking change can't be avoided, follow the guidelines below.

3. \*\*Fail-fast.\*\* Fail-fast makes bugs and failures appear sooner, leading to a higher-quality framework and API.

4. \*\*Standards Reduce Complexity\*\*. We strive to adopt standard API idoms because doing so reduces complexity for users of the API. For example, see Tenet #1 above. A counter example is [Issue #447](https://github.com/migueldeicaza/gui.cs/issues/447).

### Include API Documentation

Great care has been provided thus far in ensuring \*\*Terminal.Gui\*\* has great [API Documentation](https://migueldeicaza.github.io/gui.cs/api/Terminal.Gui/Terminal.Gui.html). Contributors have a responsiblity for continously improving the API Documentation.

- All public APIs must have clear, concise, and complete documentation in the form of [XML Documentation](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/xmldoc/).

- Keep the `<summary></summary>` terse.

- Use `<see cref=""/>` liberally to cross-link topics.

- Use `<remarks></remarks>` to add more context and explaination.

- For complex topics, provide conceptual documentation in the `docfx/articles` folder as a `.md` file. It will automatically get picked up and be added to [Conceptual Documenation](https://migueldeicaza.github.io/gui.cs/articles/index.html).

- Use proper English and good grammar.

### Defining Events

The [Microsoft .NET Framework Design Guidelines](https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/) provides these guideliens for defining events:

> Events always refer to some action, either one that is happening or one that has occurred. Therefore, as with methods, events are named with verbs, and verb tense is used to indicate the time when the event is raised.

>

> ?? DO name events with a verb or a verb phrase.

>

> Examples include Clicked, Painting, DroppedDown, and so on.

>

> ?? DO give events names with a concept of before and after, using the present and past tenses.

>

> For example, a close event that is raised before a window is closed would be called Closing, and one that is raised after the window is closed would be called Closed.

>

> ? DO NOT use "Before" or "After" prefixes or postfixes to indicate pre- and post-events. Use present and past tenses as just described.

>

> ?? DO name event handlers (delegates used as types of events) with the "EventHandler" suffix, as shown in the following example:

>

> public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);

>

> ?? DO use two parameters named sender and e in event handlers.

>

> The sender parameter represents the object that raised the event. The sender parameter is typically of type object, even if it is possible to employ a more specific type.

>

> ?? DO name event argument classes with the "EventArgs" suffix.

We are not currently consistent along these lines in `Terminal.Gui` at all. This leads to friction for adopters and bugs. As we take on fixing this we use the following guidelines:

1. We follow the naming guidelines provided in https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/names-of-type-members?redirectedfrom=MSDN

2. We use the `Action<T>` idiom for internal APIs, not for public APIs. For public APIs we use the `event/EventHandler` model.

3. For public APIs, the class that can raise the event will implement:

- A `virtual` event raising function, named as `OnEventToRaise`. Typical implementations will simply do a `EventToRaise?.Invoke(this, eventArgs)`.

- An `event` as in `public event EventHandler<EventToRaiseArgs> EventToRaise`

- Consumers of the event can do `theobject.EventToRaise += (sender, e) => {};`

- Sub-classes of the class implementing `EventToRaise` can override `OnEventToRaise` as needed.

4. Where possible, a subclass of `EventArgs` should be provided and the old and new state should be included. By doing this, event handler methods do not have to query the sender for state.

See also: https://www.codeproject.com/Articles/20550/C-Event-Implementation-Fundamentals-Best-Practices

### Defining new `View` classes

- Support parameterless constructors (see [Issue 102](Parameterless constructors #102)). Do not require callers to use a parametrized constructor except when forcing `Absolute Layout`).

- Avoid doing initialization via constructors. Instead use a property so consumers can use object initialization (e.g. `var foo = new Foo() { a = b };`).

- Ensure the `UICatalog` demo for the new class illustrates both `Absolutle Layout` and `Computed Layout`.

## Breaking Changes to User Behavior or the Public API

- Tag all pull requests that cause breaking changes to user behavior or the public API with the [breaking-change](https://github.com/migueldeicaza/gui.cs/issues?q=is%3Aopen+is%3Aissue+label%3Abreaking-change) tag. This will help project maintainers track and document these.

- Add a `<remark></remark>` to the XML Documentation to the code describing the breaking change. These will get picked up in the [API Documentation](https://migueldeicaza.github.io/gui.cs/api/Terminal.Gui/Terminal.Gui.html).

## Examples & Tests

\*\*Terminal.Gui\*\* has an automated unit or regression test suite. See the [Testing wiki](https://github.com/migueldeicaza/gui.cs/wiki/Testing)

In addition [UI Catalog](https://github.com/migueldeicaza/gui.cs/tree/master/UICatalog) is a great sample app for manual testing.

When adding new functionality, fixing bugs, or changing things, please either add a new `Scenario` to \*\*UICatalog\*\* or update an existing `Scenario` to fully illustrate your work and provide a test-case.