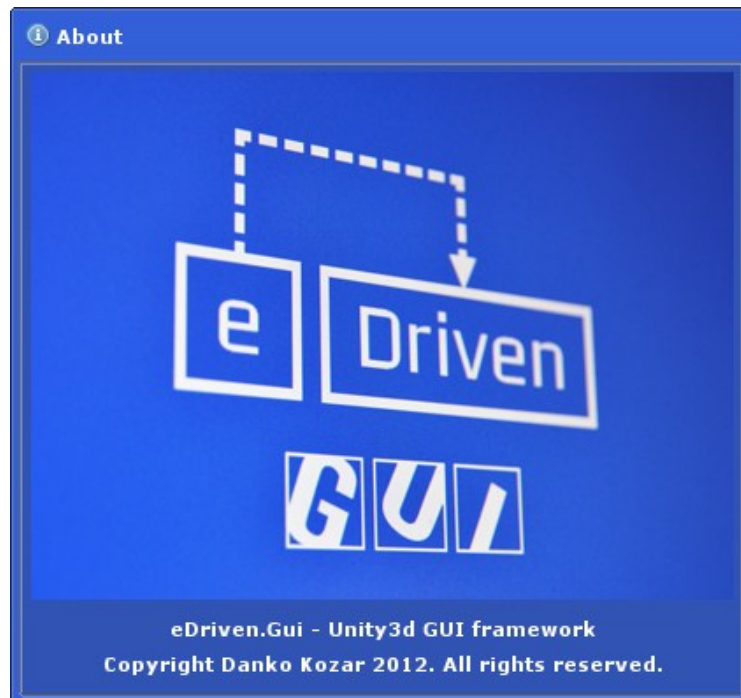


eDriven Framework Programming Manual

eDriven: RIA framework for Unity3d



eDriven RIA Framework Programming Manual

Author: Danko Kozar, SW architect

Version: 1.0.2

Last changed: 01/06/2013

Copyright Danko Kozar 2012-2013. All rights reserved.

eDriven Framework Programming Manual

About.....	5
Getting help	5
Using the API.....	6
eDriven.Core	8
Event systems used by eDriven	8
Events.....	8
Dispatching an event	8
Subscribing to events.....	8
Event handler syntax.....	9
Unsubscribing from events	9
Signals	9
Emitting a signal.....	9
Connecting to the signal	10
Slot syntax.....	10
Disconnecting from the signal	10
Core classes and utilities.....	11
PowerMapper pattern	11
AudioPlayer.....	13
eDriven.Gui	15
eDriven.Gui framework architecture.....	15
Animation.....	17
Tween class	17
TweenFactory class	18
Styling.....	20
Creating custom style mappers	23
Components.....	24
Display list	24
Things to remember	24
Creating a custom component	25
Properties in – events out.....	25
The inheritance decision.....	25

eDriven Framework Programming Manual

About rendering.....	27
About the component lifecycle.....	27
Invalidation/Validation mechanism.....	27
Invalidation and validation methods	30
Component creation	31
The constructor call	31
Configuration	32
Attachment	32
The PREINITIALIZE event.....	32
CreateChildren method	33
The INITIALIZE event	33
The CommitProperties method	34
The Measure method	34
eDriven.Gui Designer	36
eDriven.Gui Designer basics	36
Add control panel.....	37
Creation options	37
Events panel.....	39
About Event handler scripts.....	40
About Event handler panel relation to eDriven event system	41
Children panel	43
Layout panel.....	43
Descriptor layout mode	44
Full layout mode	44
Component inspector	45
Main property panel	47
Position and sizing	49
Constrains	50
Padding	51
Additional panels	52
Designer Components.....	53
Creating designer components.....	53
The control class	53
The ExampleControl class	54
The adapter class	56
The ExampleAdapter class	57
Explaining the ExampleAdapter class	58

eDriven Framework Programming Manual

The editor class	59
The ExampleEditor class	59
Explaining the ExampleEditor class.....	61

About

eDriven is an event-driven / asynchronous framework for Unity3d.

It is the result of 10 years of programming experience, self-education and research. It is inspired by RIA (Rich Internet Applications) frameworks, such as Apache Flex and other Actionscript and Javascript frameworks.

The idea behind building a framework was bringing the event-driven programming paradigm to the table.

Danko Kozar

Software architect, eDriven framework author

Getting help

eDriven is a pretty complex system, having multiple concepts introduced to Unity for the first time. So, it is natural that new users would have many questions.

This is the reason that eDriven installation adds a *Help* menu. This menu is useful for accessing various resources where you can get answers to your questions.

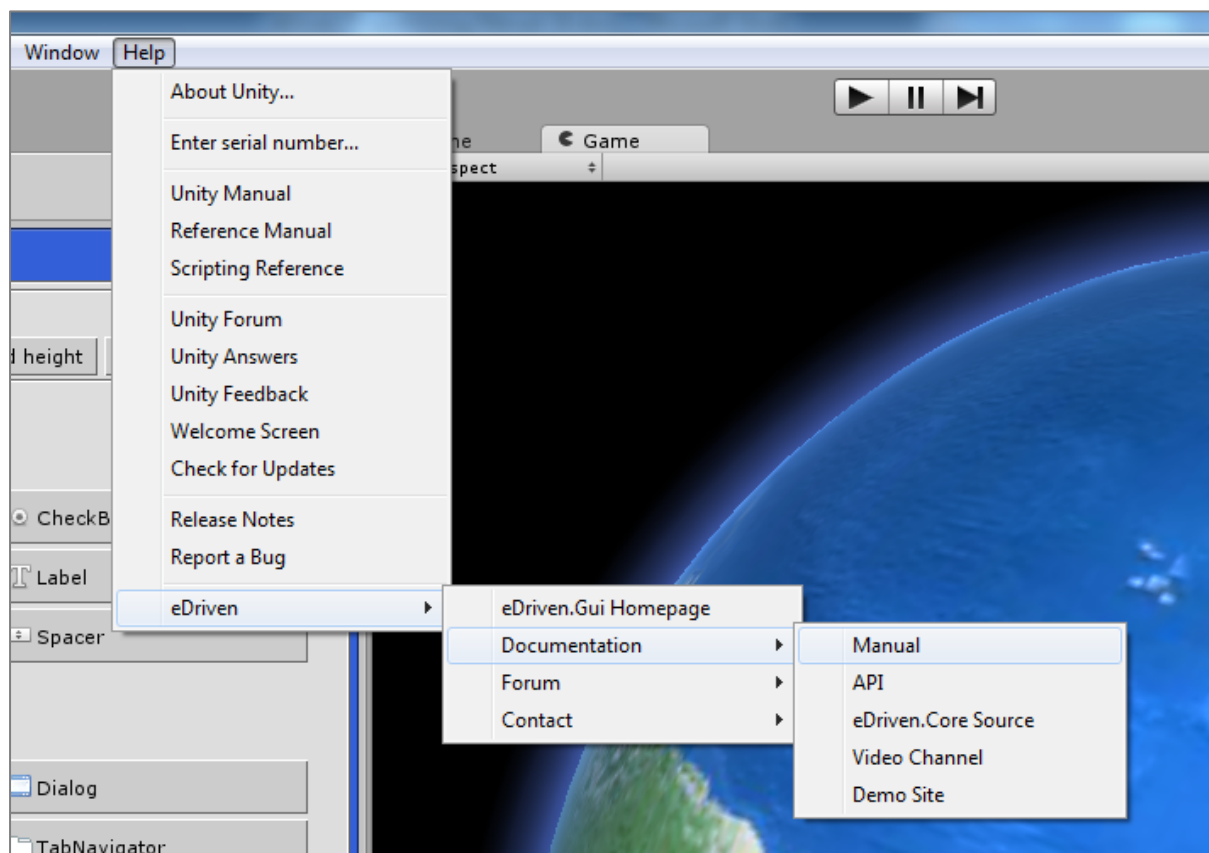


Fig 1 Help menu

eDriven Framework Programming Manual

Using the API

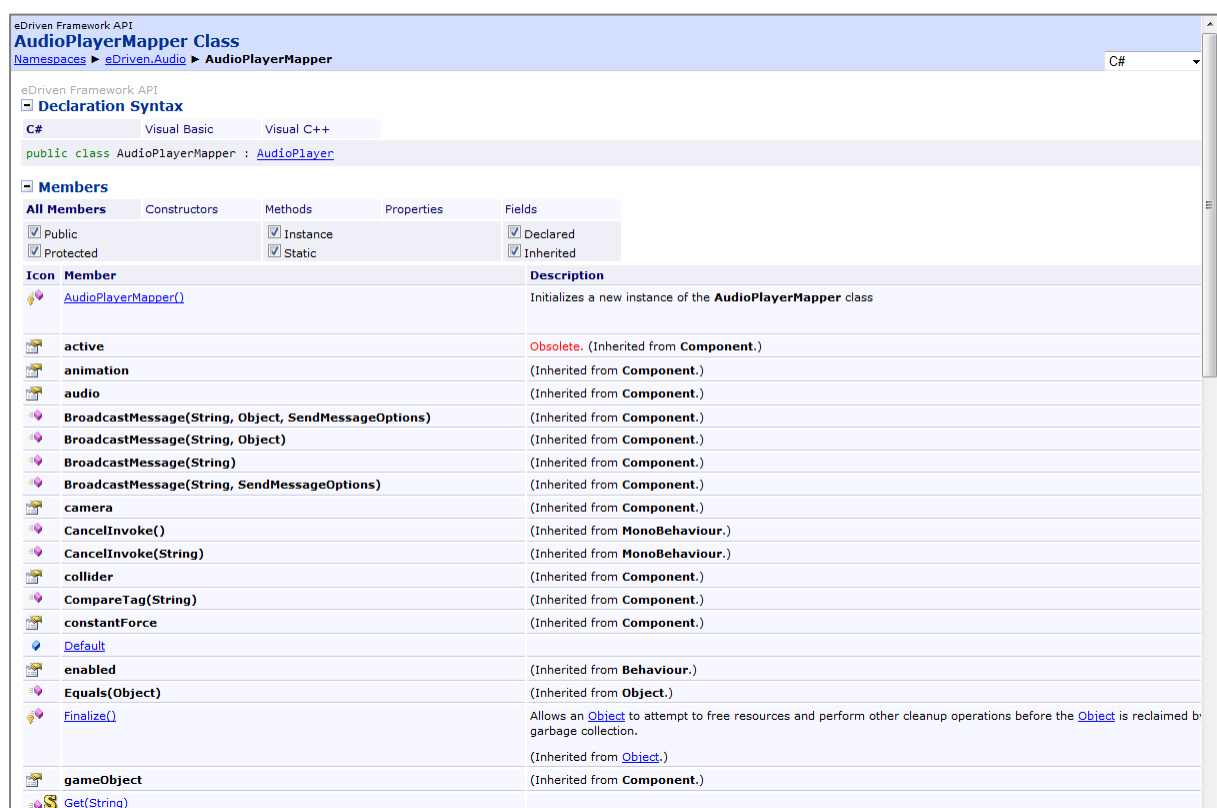
eDriven API is a web resource displaying important data about eDriven classes and their hierarchy.

The API is located at: <http://edrive.dankokozar.com/api/1-0/>.

Note: When using eDriven API, all the inherited properties for a class are being shown by default. It's often better to hide them and concentrate to the properties added by the class itself. This could be easily done by unchecking the "Inherited" checkbox.

Let's illustrate this point with a class inheriting from `MonoBehaviour`.

With *Inherited* turned on:



The screenshot displays the eDriven Framework API interface for the `AudioPlayerMapper` class. The 'Members' tab is active, and the 'Inherited' checkbox is checked, showing a list of members inherited from the `Component` base class. The members include:

Icon	Member	Description
	<code>AudioPlayerMapper()</code>	Initializes a new instance of the <code>AudioPlayerMapper</code> class
	<code>active</code>	Obsolete. (Inherited from <code>Component</code> .)
	<code>animation</code>	(Inherited from <code>Component</code> .)
	<code>audio</code>	(Inherited from <code>Component</code> .)
	<code>BroadcastMessage(String, Object, SendMessageOptions)</code>	(Inherited from <code>Component</code> .)
	<code>BroadcastMessage(String, Object)</code>	(Inherited from <code>Component</code> .)
	<code>BroadcastMessage(String)</code>	(Inherited from <code>Component</code> .)
	<code>BroadcastMessage(String, SendMessageOptions)</code>	(Inherited from <code>Component</code> .)
	<code>camera</code>	(Inherited from <code>Component</code> .)
	<code>CancelInvoke()</code>	(Inherited from <code>MonoBehaviour</code> .)
	<code>CancelInvoke(String)</code>	(Inherited from <code>MonoBehaviour</code> .)
	<code>collider</code>	(Inherited from <code>Component</code> .)
	<code>CompareTag(String)</code>	(Inherited from <code>Component</code> .)
	<code>constantForce</code>	(Inherited from <code>Component</code> .)
	<code>Default</code>	
	<code>enabled</code>	(Inherited from <code>Behaviour</code> .)
	<code>Equals(Object)</code>	(Inherited from <code>Object</code> .)
	<code>Finalize()</code>	Allows an <code>Object</code> to attempt to free resources and perform other cleanup operations before the <code>Object</code> is reclaimed by garbage collection.
	<code>gameObject</code>	(Inherited from <code>Object</code> .)
	<code>Get(String)</code>	(Inherited from <code>Component</code> .)

Fig 2 Inherited turned on

With *Inherited* turned off:

eDriven Framework Programming Manual

eDriven Framework API

AudioPlayerMapper Class

Namespaces ▶ eDriven.Audio ▶ AudioPlayerMapper

eDriven Framework API

Declaration Syntax

C# Visual Basic Visual C++

public class AudioPlayerMapper : [AudioPlayer](#)

Members

All Members Constructors Methods Properties Fields

☒ Public

☒ Protected

☒ Instance

☒ Static

☒ Declared

☐ Inherited

Icon	Member	Description
	AudioPlayerMapper()	Initializes a new instance of the AudioPlayerMapper class
	Default	
	Get(String)	
	GetDefault()	
	GetWithFallback(String)	
	HasDefault(String)	
	Id	
	IsMapping(String)	

Inheritance Hierarchy

[Object](#)
└─ **Object**
 └─ **Component**
 └─ **Behaviour**
 └─ **MonoBehaviour**
 └─ [AudioPlayer](#)
 └─ **AudioPlayerMapper**

eDriven: Event-driven framework for Unity

[Copyright \(c\) 2012 Danko Kozar](#)

Assembly: eDriven.Audio (Module: eDriven.Audio) Version: 1.0.0.0 (1.0.0.0)

Fig 3 Inherited turned off

eDriven.Core

Event systems used by eDriven

There are two parallel event-dispatching systems inside eDriven:

- Signals
- Events

Each of the two systems works better in a particular scenario.

Events

eDriven's event system is inspired by DOM Level 3 Events. It supports event bubbling, which is the important feature for GUI. The class that needs to dispatch events must inherit from *EventDispatcher* class.

Dispatching an event

When a class decides to dispatch an event, it does it by calling the `DispatchEvent` method and passing an instance of `Event` (or some other class extending the `Event` class).

In this example, `ButtonEvent` extends `Event`:

```
ButtonEvent e = new ButtonEvent(PRESS) { ButtonText = Text };
DispatchEvent(e);
```

Subscribing to events

Event handler is a function having the `EventHandler` signature:

```
public delegate void EventHandler(Event e);
```

The event handler receives a single argument (`Event e`) and returns `void`.

Using the `AddEventListener` method we could subscribe to events of the specified type. The first argument of this method is the event type, and the second is the event handler:

```
button.AddEventListener(ButtonEvent.PRESS, ButtonPressHandler);
```

Note: `ButtonEvent.PRESS` is a constant (having the string value `"press"`). The following expression would work the same:

```
button.AddEventListener("press", ButtonPressHandler);
```

However, whenever we have a choice to use the constant, we choose to use it (instead of using the plain string). This way we could prevent the possible spelling errors.

eDriven Framework Programming Manual

For instance, if we need to subscribe to *"click"*, but make a spelling error and type *"clock"* instead, the event handler would never fire although the events are actually being dispatched.

This type of errors is hard to find, because those are not the compile-time errors.

The constant is usually being defined inside the specific event class.

Event handler syntax

Event handler function example:

```
private void ButtonPressHandler(Event e) // eDriven.Core.Events.Event
{
    Button button = e.Target as Button;
    Debug.Log("Clicked button with the label: " + button.Text);

    // or getting the value right from the event
    ButtonEvent be = (ButtonEvent)e; // casting to ButtonEvent
    Debug.Log("Clicked button with the label: " + be.ButtonText);
}
```

As you can see in this example, event handler always receives the object typed as `Event`.

However, if you know the event specifics, you might cast *e* to specific class to get the details (casting to `ButtonEvent` in the previous example).

Note that you could find the type of the event class by printing it out to console:

```
Debug.Log("Event: " + e);
```

Unsubscribing from events

We could unsubscribe from the particular event type anytime:

```
button.RemoveEventListener(ButtonEvent.PRESS, ButtonPressHandler);
```

Signals

Signals are inspired by Robert Penner's Signals and are used for framework internals. Signals are faster than events because there's no instantiation involved (`new Event(EVENT_TYPE)`).

Additionally, your class isn't required to inherit the `EventDispatcher` class.

Emitting a signal

When a class decides to emit a signal, it does it by emitting the arbitrary number of arguments:

```
button.ClickSignal.Emit(); // no params
button.ClickSignal.Emit(1); // single param (int)
```

eDriven Framework Programming Manual

```
button.ClickSignal.Emit(1, "foo"); // multiple params [int, string]

button.ClickSignal.Emit(button); // object reference
```

Connecting to the signal

The example of connecting a slot to the signal:

```
button.ClickSignal.Connect(ClickSlot);
```

Slot syntax

A slot is the function having the `Slot` signature:

```
public delegate void Slot(params object[] parameters);
```

Slot example:

```
private void ClickSlot(params object[] parameters)
{
    // do something with parameters
}
```

Note that inside our slot we have to know the exact order and type of each parameter dispatched by the signal. Then we have to cast each parameter to the specific type:

```
private void ClickSlot(params object[] parameters)
{
    string value = (int) parameters[0];
    Debug.Log("Value: " + value;

    string name = (string) parameters[1];
    Debug.Log("The name is: " + name;
}
```

Note: Each signal in your application should always emit the same number of parameters, each of the same type.

Disconnecting from the signal

Disconnecting from the signal is similar to unsubscribing an event handler from the event dispatcher:

```
button.ClickSignal.Disconnect(ClickSlot);
```

eDriven Framework Programming Manual

Core classes and utilities

eDriven framework contains a number of classes and utilities:

- `PowerMapper` (maps strings to stuff)
- `SystemManager` (emits system signals)
- `SystemEventDispatcher` (dispatches system events)
- `MouseEventDispatcher` (dispatches mouse events)
- `KeyboardMapper` (easy mapping for keyboard gestures)
- `CallbackQueue` (implements queuing of asynchronous operations)
- `TaskQueue` (sequences asynchronous operations)
- `AudioPlayerMapper` (maps strings to AudioPlayers)
- `Cache` (caches stuff)
- `ObjectPool` (object reusing; prevents instantiation and garbage collection problems)
- `Timer` (the source of timed events)
- `HttpConnector` (loads stuff from the web and triggers responders when finished)
- `JavascriptConnector` (talks to a host web page)

PowerMapper pattern

A PowerMapper pattern is a concept that I conceived because very often I had a need of mapping strings-to-stuff. I'm using it throughout the framework, so it deserves to be explained right at the beginning.

The concept includes looking up for instantiated components by (string) ID, and if not found - returning a default one. Sometimes it also includes the lazy instantiation of objects currently not present in the scene.

The following methods are being used in PowerMapper pattern:

- `IsMapping`: returns a boolean value indicating the presence of the mapper specified by ID in the scene.
- `Get`: returns the reference to the mapper specified by ID. Could return null if the mapper isn't present in the scene.
- `HasDefault`: returns a boolean value indicating that a default mapper is present in the scene (the one with a "Default" checkbox checked)
- `GetDefault`: returns the reference to the default style mapper. Could return null if default mapper isn't present in the scene.
- `GetWithFallback`: returns the reference to the mapper specified by ID. If this reference is null, returns the default mapper.

```
namespace eDriven.Core.Mapper
{
    /// <summary>
    /// Power mapper interface
    /// </summary>
    /// <remarks>Conceived and coded by Danko Kozar</remarks>
    public interface IPowerMapper
    {
        /// <summary>
        /// Returns true if mapping with specified ID exists
        /// </summary>
        /// <param name="id"></param>
        /// <returns></returns>
        bool IsMapping(string id);

        /// <summary>
        /// Gets the mapped item
        /// </summary>
        /// <param name="id">ID of the mapped item</param>
        /// <returns></returns>
        IPowerMapper Get(string id);

        /// <summary>
        /// Returns true if default mapping exists
        /// </summary>
        /// <returns></returns>
        bool HasDefault();

        /// <summary>
        /// Gets the default mapping
        /// </summary>
        /// <returns></returns>
        IPowerMapper GetDefault();

        /// <summary>
        /// Gets the mapping specified with ID, or a default one
        /// </summary>
        /// <param name="id"></param>
        /// <returns></returns>
        IPowerMapper GetWithFallback(string id);
    }
}
```

Fig 4 IPowerMapper interface

Some of the scenarios where this concept is used include:

- Referencing an [AudioPlayer](#) (the default one is often the child of a camera and is used for GUI sounds)
- Referencing style mappers (internally by GUI components). Style mapper instances follow the PowerMapper pattern and are being looked upon by GUI components during the application start.

The ID of the style mapper is being specified by the user when setting up a particular component, using the [StyleMapper](#) property.

AudioPlayer

The Audio part is a wrapper around Unity's audio system, which enables referencing audio players in *PowerMapper* fashion.

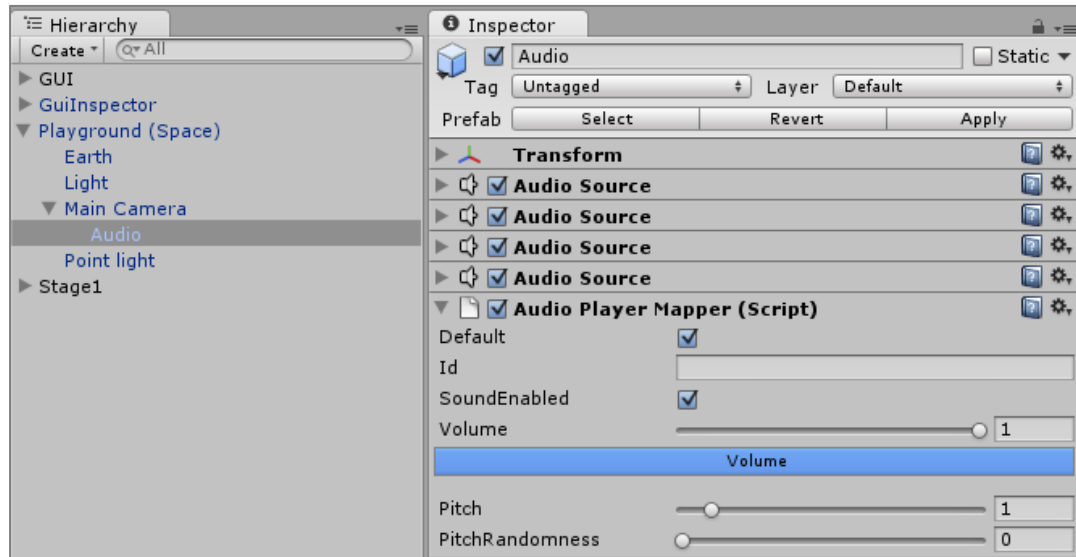


Fig 5 Default AudioPlayerMapper

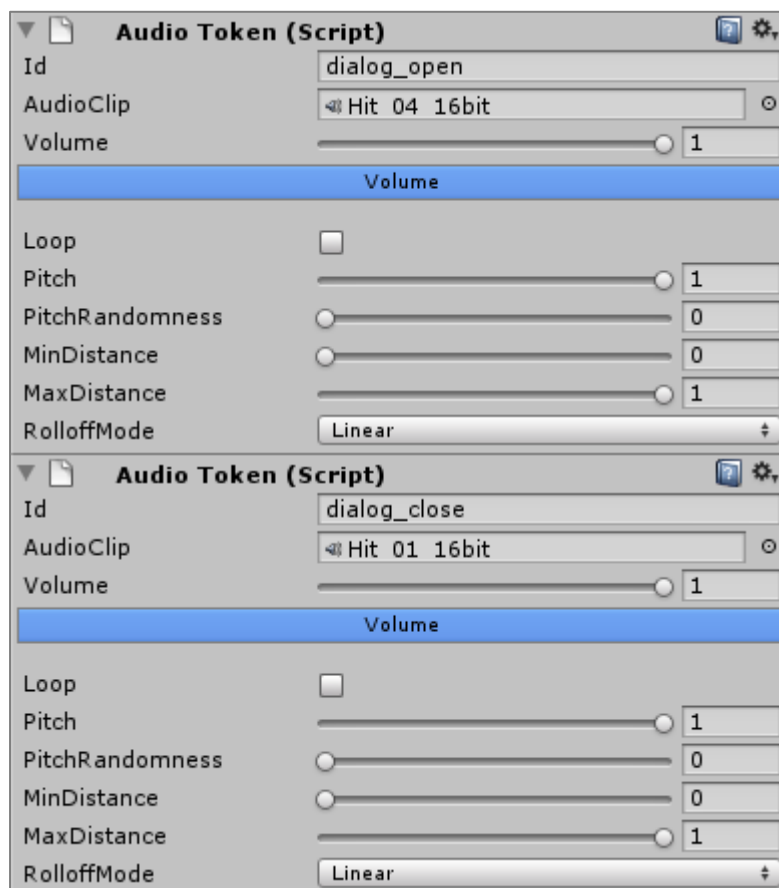


Fig 6 Audio tokens

eDriven Framework Programming Manual

AudioPlayerMapper is a component which has to be instantiated with at least one *AudioSource* on the same game object. That's because it uses sibling *AudioSources* for playing sounds.

Note: the polyphony of sounds played on a single audio player depends of the number of *AudioSources* attached, because *AudioPlayerMapper* looks for idle *AudioSources* and playing the specified sound using the first found idle audio source.

Additionally, a number of audio tokens should be attached to the same game object.

Audio token is a placeholder for sound, which could be tweaked anytime by audio personnel, without touching the code. This is a way of decoupling actual audio clips from the code: the token is being referenced from code using the particular ID and not specifying the exact sound which is going to be played.

To play the sound using the default *AudioPlayer* use:

```
AudioPlayerMapper.GetDefault().PlaySound("your_sound_id");
```

To play the sound using the *AudioPlayer* specified with ID use:

```
AudioPlayerMapper.Get("audio_player_id").PlaySound("your_sound_id");
```

You could also override the default properties used by the specified *AudioPlayer* by using a number of *AudioOptions* params:

```
AudioPlayerMapper.GetDefault().PlaySound("your_sound_id",  
                                           new AudioOption(AudioOptionType.Pitch, 2f));
```

eDriven.Gui

eDriven.Gui framework architecture

eDriven.Gui framework consists of four independent parts:

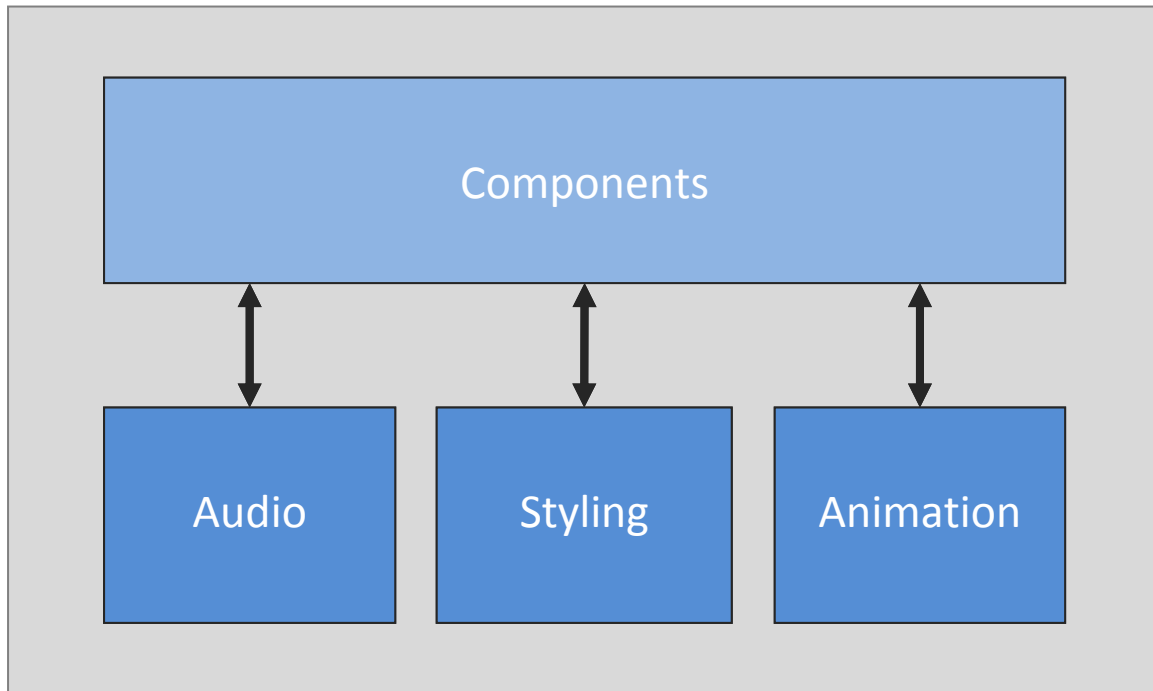


Fig 7 eDriven.Gui framework parts

- *Components* part is responsible of instantiating, managing, rendering, keeping references of, and destroying GUI components.
- The *audio* part is responsible for playing audio using a number of audio sources. *AudioPlayers* are being referenced by GUI components using power mappers.
- *Styling* is used for custom look and feel of GUI components. Styles are being referenced by GUI components using power mappers.
- *Animation* part is used for animation and tweening and is being referenced using tween factories (or playing tween instances directly on components).

The following image shows how GUI components reference corresponding audio, styling and animation components:

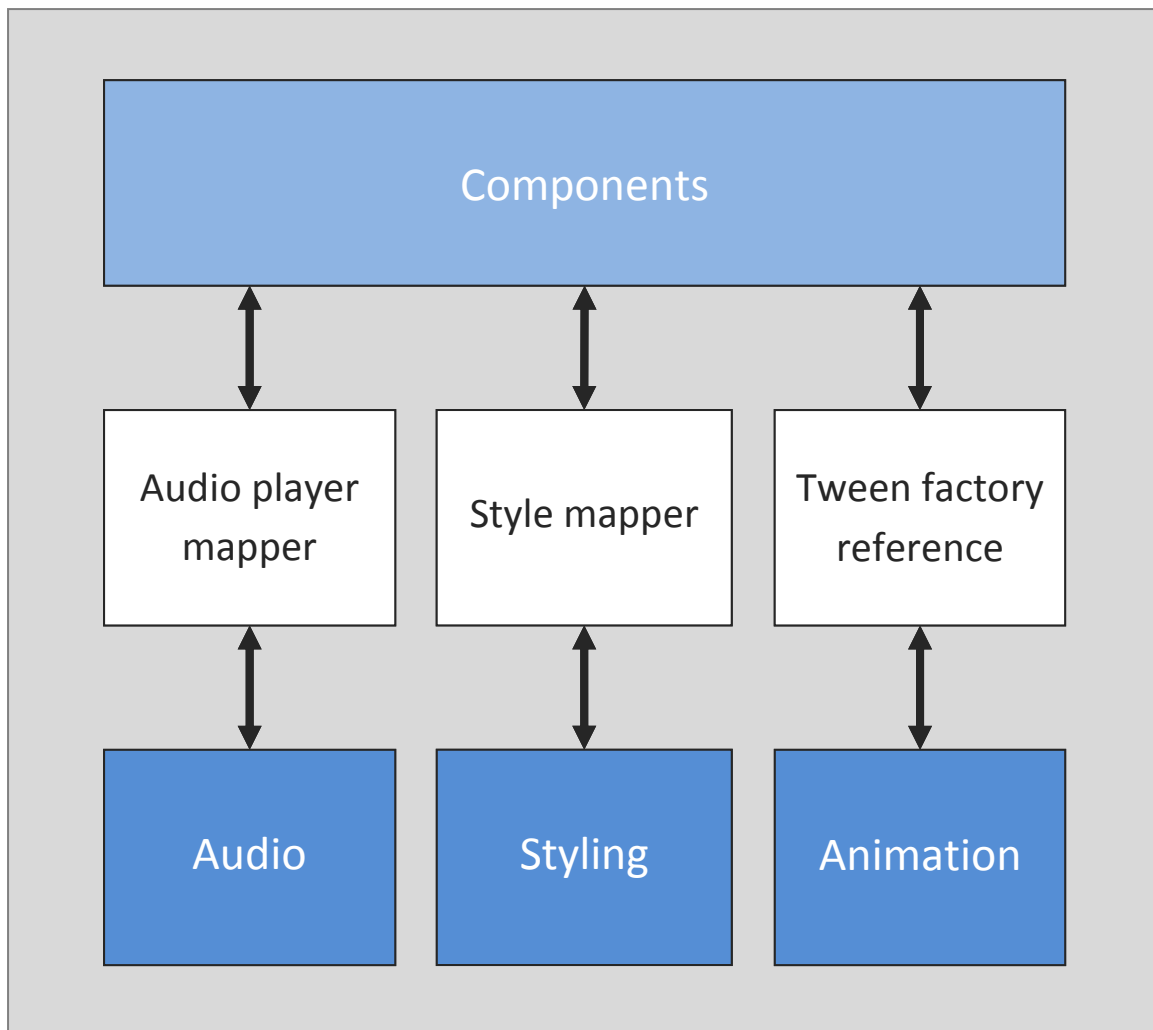


Fig 8 Referencing audio, styling and animation

Animation

Tween class

eDriven.Animation assembly contains the eDriven's tweening engine. It enables the creation of tweens as (reusable) classes.

A tween must extend eDriven.Animation.TweenBase class. The tween could also be a composite one, in which case it should extend Sequence or Parallel class.

Each tween has a number of properties:

- **Target:** The object which has to be tweened
- **Property:** A property which needs to be tweened
- **Duration:** Tween duration (ignored with composite tween)
- **Name:** A callback function executed when a tween is finished
- **Easer:** Easing function
- **Interpolator:** Tweening interpolator, i.e. the object which translates values from [0, 1] range to values applied to a component
- **StartValue:** Tween start value
- **EndValue:** Tween end value
- **StartValueReader:** PropertyReader returning the start value when requested
- **EndValueReader:** PropertyReader returning the end value when requested
- **StartValueReaderFunction:** Function returning the start value when requested
- **EndValueReaderFunction:** Function returning the end value when requested
- **Callback:** A callback function executed when a tween is finished playing

Only first two values are mandatory (Target and Property). Additionally one of the *End* values have to be defined (either EndValue, EndValueReader or EndValueReaderFunction).

Tweens are objects instantiated when the tween is being run (triggered by one of component's events or by developer's code directly).

You are building a tween instance the following way:

```
var tween = new Tween
{
    Property = "Alpha",
    Duration = 0.6f,
    Easer = Linear.EaseNone,
    StartValue = 0f,
    EndValue = 1f
};
```

Before playing a tween, its target has to be defined:

```
tween.Target = _myComponent;
```

And finally, here's how to play the tween (manually):

```
tween.Play();
```

eDriven Framework Programming Manual

TweenFactory class

However, there is another way of playing tweens. This is the way eDriven.Gui components use it. The tween structure is being passed to the instance of a *TweenFactory* class:

```
private readonly TweenFactory _addedEffect = new TweenFactory(  
    new Tween {  
        Property = "Alpha",  
        Duration = 0.6f,  
        Easer = Linear.EaseNone,  
        StartValue = 0f,  
        EndValue = 1f  
    }  
);
```

Note: no target specified in tween passed to the factory constructor (if specified, it is ignored).

The purpose of passing the tween to a factory is to reuse it on multiple objects. The factory makes a copy of the supplied tween each time a new tween is requested by any of the components.

The structure passed to constructor could also be a complex tween (*Sequence* or *Parallel*):

```
private readonly TweenFactory _addedEffect = new TweenFactory(  
    new Sequence(  
        new Tween {  
            Property = "X",  
            Duration = 0.8f,  
            Easer = Expo.EaseInOut,  
            StartValue = 100f,  
            EndValue = 200f  
        },  
        new Tween {  
            Property = "Alpha",  
            Duration = 0.6f,  
            Easer = Linear.EaseNone,  
            StartValue = 0f,  
            EndValue = 1f  
        }  
    )  
);
```

Tweens could be written as classes (inheriting from *TweenBase*). Note a usage of the *Jumpy* class in this example:

```
private readonly TweenFactory _addedEffect = new TweenFactory(  
    new Sequence(  
        new Action(delegate { AudioPlayerMapper.GetDefault().  
                                PlaySound("dialog_open"); }),  
        new Jumpy()  
    )  
);
```

After specifying the factory, the reference is being passed to a component that needs tweening using the *SetStyle* method:

```
Component.SetStyle("addedEffect", _addedEffect);
```

eDriven Framework Programming Manual

The first parameter ("`addedEffect`") is called the trigger. It describes the set of circumstances that have to be met in order for the tween to start playing.

In the case of the "`addedEffect`" trigger, the tween starts playing on the component the moment the component is *added* to the display list.

Styling

eDriven.Gui framework offers the very flexible method of component styling.

The motivation behind this type of styling came from the need of having a CSS-like styling, but utilizing the original concept of Unity GUISkins and GUIStyles.

Also, I wanted to be able to map:

- any style
- from any skin
- to any component
- at any depth

Additionally, I wanted to enable a kind of cascading - so if you change a style mapper on a complex component - this would in turn change styles on component children.

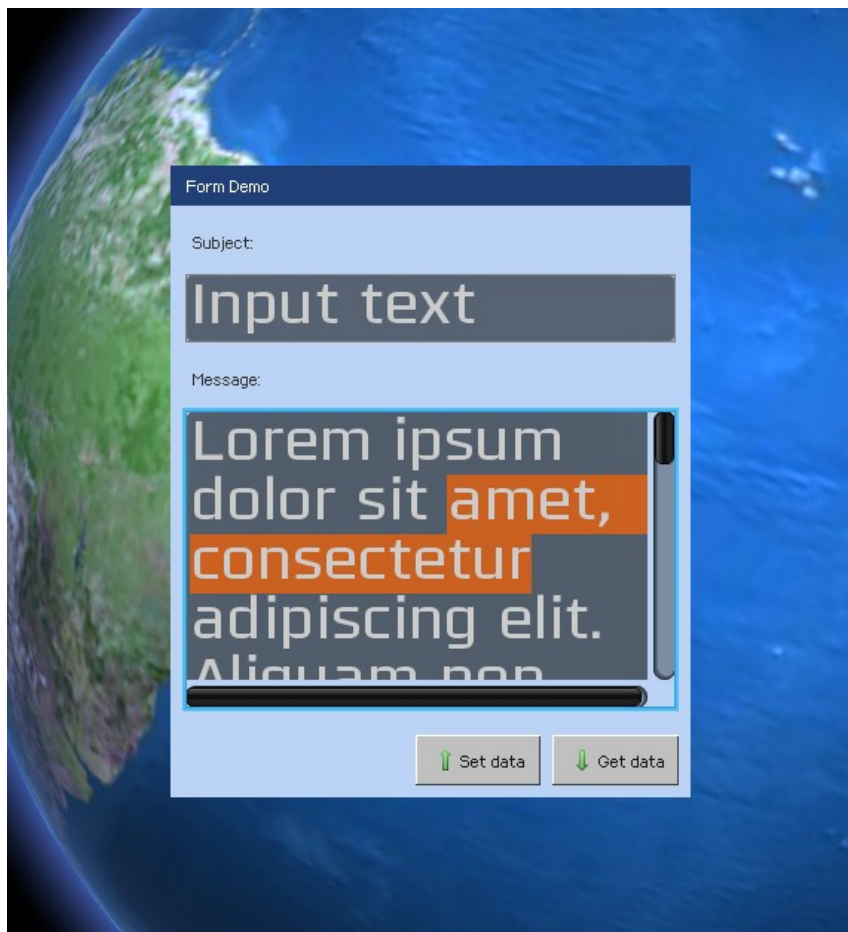


Fig 9 Styling a text field

I also wanted to have a robust type of styling, with the application not crashing if missing any of the style declarations (this was a common case when working with Unity: styles were often missing when copying stuff from project to project).

eDriven Framework Programming Manual

To accomplish it, I've built a system having the default (hardcoded) styles, which could be overridden with custom styles.

The process of mapping styles to components happens the moment the application starts (e.g. the *Play mode* is being run).

During the application start, eDriven looks for all the style mappers present in the scene and compares them with instantiated components. The style processing order that eDriven does is as follows:

1. It looks if the component is having the *StyleMapper* property set (*string*). If this is true, it looks for a mapper having the specified ID. If such mapper found, its style definitions are being applied to the component.

Note: the mapper type should always relate to component's type. For instance, when styling the *Button* instance, eDriven looks for *ButtonStyleMappers* only - because only the *Buttons* skin parts match to those of the *ButtonStyleMapper*.

2. Then it looks for a default style mapper for components still having no style applied. If such found in the scene, it is applied to components (i.e. *ButtonStyleMapper* with *Default* checked is being applied to all non-styled *Buttons*).
3. If no custom style mapper found for the component, the hardcoded default styles are being applied, as a fallback.

Note that styling is an independent process from building the component tree by both C# classes and designer classes (adapters). It works the same using the both approaches - with style mappers being present anywhere in the scene.

C# component developer should normally create a style mapper for its component and distribute it with the component. However, the component would work the same without any style mapper applied (but having the default styles).

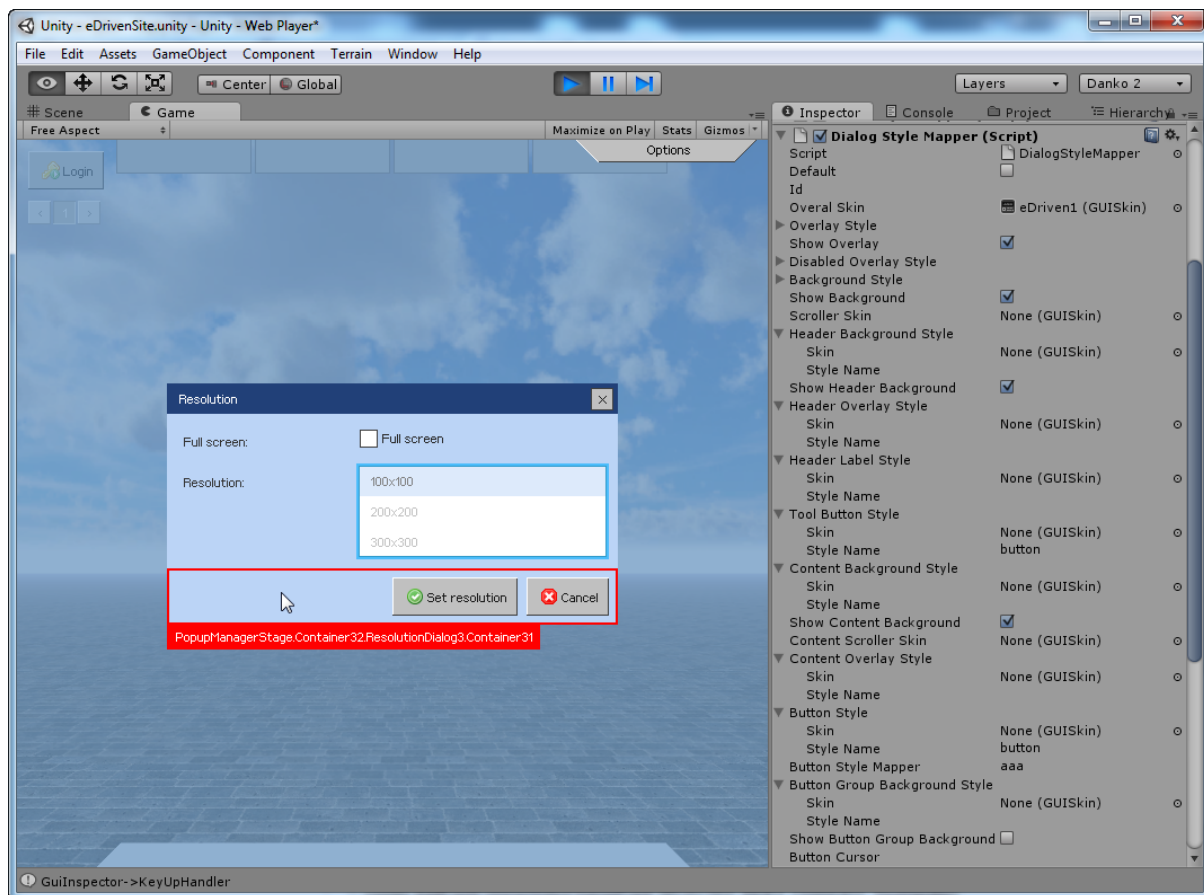


Fig 10 StyleMapper example

Creating custom style mappers

StyleMapper hierarchy is parallel with the component hierarchy.

To illustrate this, let's assume that your component has a class name of *MyComponent* and is extending `eDriven.Gui.Components.Component`.

If wanting to style your component, you need to create a style mapper extending `eDriven.Gui.Mappers.Styles.Component.ComponentStyleMapper`. The name for the style mapper should be *MyComponentStyleMapper*.

Additionally, you have to create a style proxy extending `eDriven.Gui.Mappers.Styles.Component.ComponentStyleProxy`. The name for the proxy should be *MyComponentStyleProxy*.

MyComponent class should specify the *MyComponentStyleProxy* class as its style proxy class within its class metadata, and also the *MyComponentStyleProxy* should specify the *MyComponentStyleMapper* as its style mapper class within its class metadata.

The component gets the styling information from the style mapper *indirectly* - using the proxy.

The reason for using the proxy is because it is always providing the default (hardcoded) styles - even with no style mappers present in the scene.

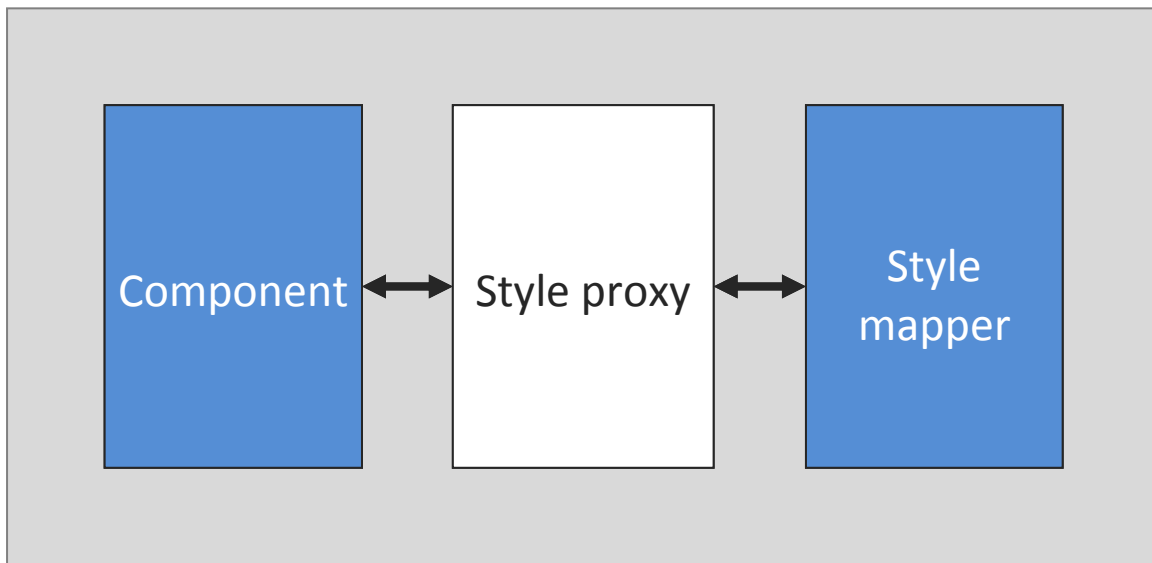


Fig 11 The style chain

Components

Display list

Display list is a component tree based on the [Composite pattern](#). Components are nodes in this tree.

The root node of a display list is called a stage (e.g. the `Stage` class).

The `Stage` class is being adapted to game object hierarchy by:

- Using the subclass of the `Gui` class
- Using the `StageAdapter` class (designer)

Both of these classes inherit from `MonoBehaviour`.

With `eDriven.Gui`, there could be multiple stages in the scene. Each stage has its *ZIndex* - the integer which represents its z-ordering position.

The component could be added to other component as a child using the `AddChild` method (and its variants).

Things to remember

- The component is considered to be attached to the display list the moment it becomes an ancestor of any of the `Stage` objects. Until this moment, some of the component's lifecycle methods are never run.
- A branch of the component tree could be built in advance, and then attached to the display list.
- With `eDriven.Gui`, it's important to know that components are in different initialization state at different points in time.
- You cannot rely on component's children being created the moment the component itself is created (constructed). In fact, the children are usually created the moment a component is attached to the display list (there are exceptions to this rule).

This is because child components are usually instantiated inside the `CreateChildren` method, which is being run upon parent's attachment to the display list.

Creating a custom component

eDriven allows building custom components from code using C#. Developers usually build custom component having the encapsulated logic (i.e. having "*private parts*"), so developers using their custom component see this component as a "black box".

Only a number of properties are being exposed – those really needed for setting up the component up from the outside.

For instance, a very complex *DataGrid* component should – in this scenario – have only a few properties exposed. None of its row or cell objects should be publically available and the developer shouldn't be able to alter these objects directly.

This is also true for designer components (visual components available within Unity Editor). Using the corresponding adapter (made for *DataGrid* component), all of the internals shouldn't be exposed – only the stuff the component author explicitly allows to be used.

Properties in – events out

I coined this phrase - it very much describes the best way of communication between the component and the "outside world". For creating reusable components with eDriven, I suggest following this pattern:

1. You should expose properties (setters) for passing the (input) data to the component.
2. You should dispatch notifications describing component's internal changes using events.
3. The user of the component should subscribe to events from the outside. Then it has to read the component data:
 - using properties (getters)
 - if available, read it directly from event
4. You should never reference component's children directly (they shouldn't be exposed to the outside).

The inheritance decision

Both eDriven.Gui components and containers could have child components. The main difference between the component and the container is:

Containers could have layout (i.e. you could plug-in one of the layout classes to the container and the layout should handle sizing and placement of its children).

So, when creating a custom component you need to extend one of the following classes (or their inheritors):

- eDriven.Gui.Components.[Component](#)
- eDriven.Gui.Containers.[Container](#)

The decision on what to inherit should be made base on the answers to following questions:

1. Should the component have layout (e.g. should it lay out its children automatically)?
 - If the answer is yes - your component should extend `Container` or one of its inheritors (`Panel`, `Dialog` *etc.*).
 - On the other hand, if the answer is no - your component should extend `Component` or one of its inheritors (`Label`, `Button`, `Slider` *etc.*)
2. Should the component have the functionality already implemented in any of the existing components?
 - If the answer is yes- your component should extend this existing component. For instance, if your component is very similar to `Label` component, you should extend `Label`.
 - On the other hand, if your component is so specific that you cannot find a good fit in any of the existing components, you should inherit directly from the `Component` or `Container` class (based on the answer to the 1st question).

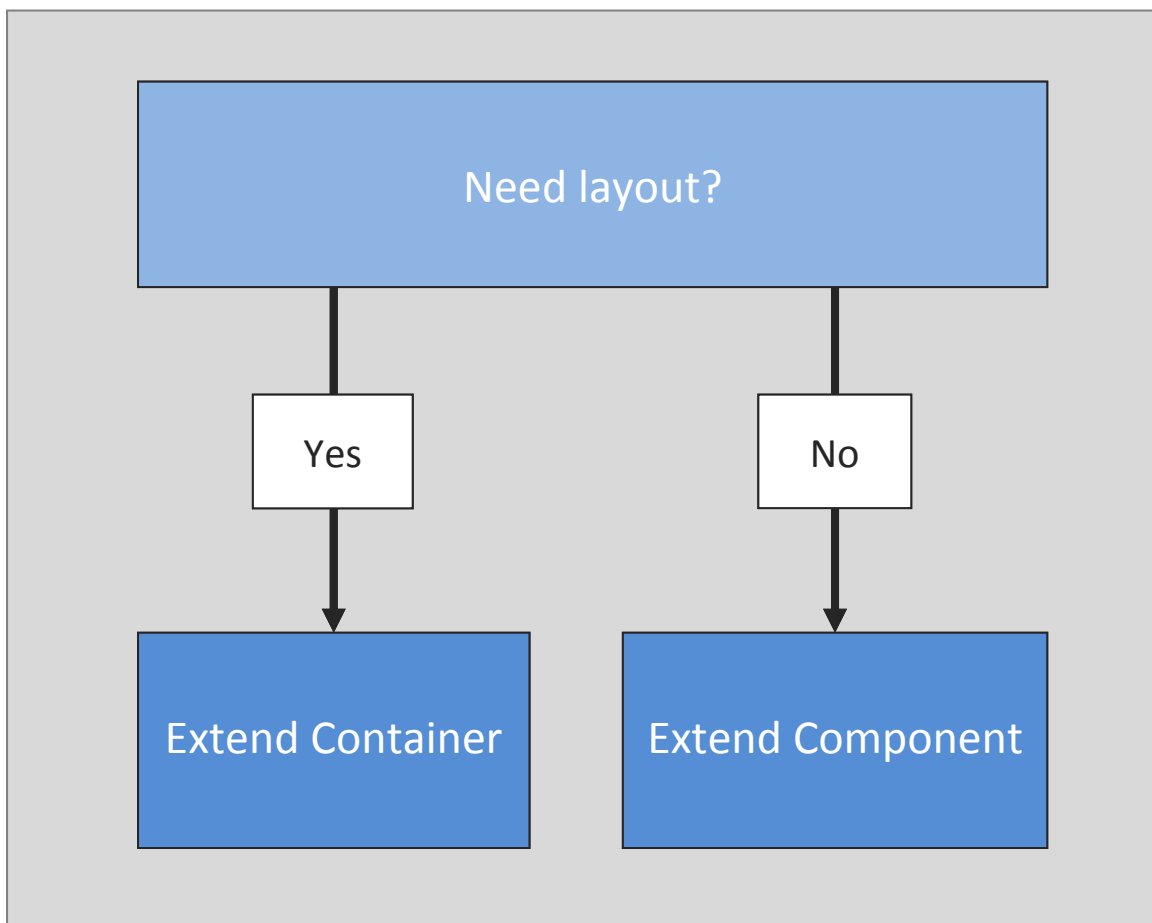


Fig 12 The inheritance decision

About rendering

With eDriven.Gui the word *rendering* stands for the process of *redrawing* the component. This is not the actual rendering process itself (i.e. `OnGUI` method call). Redrawing or *rendering* happens only when component needs to change its graphical presentation (for instance when the component is changing its size or another component is added to it as a child). This process of rendering happens just before the actual `OnGUI` call which draws the component on screen.

The time between two `OnGUI` calls we'll be calling the *rendering cycle*.

About the component lifecycle

All eDriven.Gui components follow the predefined order of events when building themselves. We call it the component lifecycle. These are the component lifecycle stages:

- Creation
 - Construction
 - Configuration
 - Attachment
 - Initialization
- Life
 - Invalidation
 - Validation
- Death
 - Destruction
 - Garbage collection

Invalidation/Validation mechanism

eDriven uses the invalidation/validation mechanism for rendering components in their various states.

The basic idea of this concept is to defer rendering process to the moment it is *optimal* to run.

Let's describe what "optimal" means with an example.

You could have a for-loop that loops 1000 times and is setting component's width:

```
for (int i = 0; i < 1000; i++)  
{  
    component.Width = i;  
}
```

Changing component's width requires the component to redraw itself. Additionally, the change propagates to component's children (their sizing and positions) as well as children's children. This recursive operation is a processor-intensive process.

eDriven Framework Programming Manual

Without using the invalidation/validation, this process would be run directly from within the Width setter, upon each value change (thousand times in example above).

However, only the last change really needs to be processed and is expected to have the impact on actual display:

```
component.Width = 1000;
```

So, here's where the invalidation/validation mechanism comes to place: instead making complex calculations inside a setter, only the flag is being set. Changes are this way *aggregated* - only the last change really counts. The actual redraw of is actually run once, just before the next *OnGUI* call.

This is the Width property without the invalidation/validation mechanism involved:

```
private float _width;
public float Width
{
    get
    {
        return _width;
    }
    set
    {
        if (value == _width)
            return; // no change

        _width = value;
        // processor intensive operations using new _width value
    }
}
```

And here is the invalidation/validation version of the same property:

```
private bool _widthChanged; // flag
private float _width;
public float Width
{
    get
    {
        return _width;
    }
    set
    {
        if (value == _width)
            return; // no change

        _width = value;
        _widthChanged = true; // set flag
        InvalidateProperties(); // run the validation in the next cycle
    }
}
```

This is the central concept in the component lifecycle. In this approach, private variables and `bool` flags are used to defer render-related processing until the proper validation method is reached.

eDriven Framework Programming Manual

The proper validation method in this case is `CommitProperties` method. Within this method, the component checks for a particular flag. If flag is set, the change in value is processed.

```
protected override void CommitProperties()
{
    base.CommitProperties(); // important: call superclass
    if (_widthChanged)
    {
        _widthChanged = false; // important: reset the flag
        // processor intensive operations using new _width value
    }
    if (_heightChanged)
    {
        _heightChanged = false; // important: reset the flag
        // processor intensive operations using new _height value
    }
    // other invalidated properties follow here...
}
```

You might ask yourself: why someone would re-apply the same property in a loop as in the example above?

Well, this example was just for the illustration purposes. The real world example would be adding the data to component's data provider:

```
list.DataProvider.Add(new ListItem(1, "foo1"));
list.DataProvider.Add(new ListItem(2, "foo2"));
// etc....
```

Without the invalidation/validation mechanism the process of redrawing the list component would run each time a new data item is added to its `DataProvider`.

This is not the only reason for using the invalidation/validation mechanism though.

The other important reason is synchronizing between multiple properties setters, because there may be setters that need other values to be set in advance.

The best example would be with `DataProvider` and `SelectedIndex` of a List control:

```
list.DataProvider = _data; // list of ListItems
list.SelectedIndex = 3;
```

`SelectedIndex` obviously relies on `DataProvider` being previously set (because if not there are zero elements in the list so `SelectedIndex` of 3 would be not a valid number). Meaning that this shouldn't work then:

```
list.SelectedIndex = 3;
list.DataProvider = _data;
```

However, using the invalidation/validation mechanism it works no matter of the order. This is because both setters are just setting flags, and deferring the execution to `CommitProperties` method, which looks like this:

```
protected override void CommitProperties()
{
```

eDriven Framework Programming Manual

```
base.CommitProperties(); // important: call superclass
if (_dataProviderChanged)
{
    _dataProviderChanged = false; // important: reset the flag
    // processor intensive operations using new _dataProvider value
}
if (_selectedIndexChanged)
{
    _selectedIndexChanged = false; // important: reset the flag
    // processor intensive operations using new _selectedIndex value
}
// other invalidated properties follow here...
}
```

So, the changes of both properties are being synchronized inside of the `CommitProperties` method and are always being executed in the right order - ensuring that `DataProvider` is always being processed first (the List control has now 4 child rows attached), and `SelectedIndex` is processed after it (the child with index 3 is being styled differently to show the selection).

As a conclusion: the invalidation/validation mechanism is really like a `COMMIT` statement in `SQL` which ends a transaction. It aggregates and synchronizes property changes, deferring the real processing to the optimal time.

Invalidation and validation methods

`CommitProperties` isn't the only validation method available with eDriven. However it is the first one being run.

There are 3 validation methods that utilize deferred validation in total, and the reason for having multiple methods is because there are 3 processes – different in nature - which should be synchronized between them (i.e. executed in the right order).

They are namely (in the order of execution):

- `CommitProperties`
- `Measure`
- `UpdateDisplayList`

Note: These three methods could be implemented by the developer, but should actually never be called by the developer: they are being called automatically by the framework.

Each of the methods above has the corresponding invalidation method (which *should be called by the developer*) that allows you to schedule one of them to be executed before the next `OnGUI` call. They will be discussed later on, but for illustration they are included below:

- `InvalidateProperties`
- `InvalidateSize`
- `InvalidateDisplayList`

The invalidation -> validation relationship between those methods is as follows:

- `InvalidateProperties` -> `CommitProperties`
- `InvalidateSize` -> `Measure`

eDriven Framework Programming Manual

- `InvalidateDisplayList -> UpdateDisplayList`

The difference between the three validation passes is the direction they are being run regarding to a parent-child relationship:

- `CommitProperties` is being run *top-to-bottom*: from parent to children. This means the stage is processed first, then its children, then children's children etc. This works with properties, because most often the component itself has to be configured for being able to instantiate its children and configure them.
- `Measure` is being run *bottom-up*: from children to parent. This means that bottom-most children are processed (measured) first and their parent is processed after its all children are processed (up to the stage).
The reason why this process has to be reversed is because the way that the measuring concept works. To measure component's preferred size, the size of its children has to be known in advance.
- `UpdateDisplayList` is being run *top-to-bottom* (just like `CommitProperties`).
Inside this method component's children are being laid out (placed and sized): child coordinates and sizes depend of the size of the parent.
So, the stage is being processed first (its direct children are placed and sized depending of the stage size and taking into account the preferred sizes of the children). This process goes on recursively all to the bottom-most components.

Component creation

A broad overview of the creation part of the component lifecycle is as follows:

- The constructor is called (default properties are set within the constructor).
- Component is being configured (from outside).
- Component is added to the display list.
- `FrameworkEvent.PREINITIALIZE` event is dispatched.
- The `CreateChildren` method is called.
- `FrameworkEvent.INITIALIZE` event is dispatched.
- The `CommitProperties` method is called.
- The `UpdatePosition` method is called.
- The `Measure` method is called (if necessary).
- The `UpdateDisplayList` function is called.
- `FrameworkEvent.CREATION_COMPLETE` event is dispatched.
- `FrameworkEvent.UPDATE_COMPLETE` event is dispatched.

Let's explain each of the phases in details.

The constructor call

The primary purpose of the constructor is to create a new instance of an object and to set initial properties for the object.

eDriven Framework Programming Manual

When extending the component, in the (overridden) constructor you could set a number of defaults for the particular component.

An example of configuring the component's initial properties in the constructor follows:

```
public Button() : base() {  
    MinWidth = 50;  
    MinHeight = 20;  
    AddEventListener(MouseEvent.CLICK, OnClick);  
}
```

In the previous example, *MinWidth* and *MinHeight* properties are being set to a *Button* component (that is, the minimum size for a button should be 50x20px). Additionally the event handler for mouse click event is added (the actual handler not shown for brevity).

Configuration

After the component has been constructed, we could configure it:

```
var button = new Button(); // construction  
button.Text = "Button text";  
button.MinWidth = 100;
```

Attachment

The component could be added to a display list anytime. It isn't necessary for the component to be added to a display list right after it has been constructed - as long as you keep the reference to it (to be able to add it later).

The component is being added to the display list using the *AddChild* method:

```
parent.AddChild(component);
```

The moment the component is being added to the display list – this starts the order of events which is being called the initialization process.

Note: the initialization of the component won't be run if the parent isn't attached to the display list. In this case, the component is waiting until the parent becomes a part of the display list (e.g. is added to Stage). The moment this happens, both the parent and the component will be initialized.

The PREINITIALIZE event

The event is dispatched immediately after the component is added to the display list.

This event should be used sparingly as there are few things you can do with a component early in its life because it is in a very "raw" state. At this moment, *CreateChildren* method hasn't been run, so children created from within that method are not yet created.

eDriven Framework Programming Manual

CreateChildren method

When building a custom component and wanting to add children, this method has to be overridden.

The purpose of this method is to add child components from within the component class. This is the only place where you should use `AddChild` method inside of the custom component class.

Note: calling the base class `CreateChildren` is pretty much mandatory. If base is not called, the component could not work as expected:

```
protected override void CreateChildren()
{
    base.CreateChildren();
    // create children here
}
```

This method is called only once - upon the component creation.

Note that by the time the `CreateChildren` method is being run, the component has already been configured (both from within the constructor and from the outside) so when the creating children we might use the values passed by when configuring the component. This is one of the reasons for deferring the child creation to the moment when the component is completely configured (note that if creating children from within the constructor this wouldn't be the case).

For instance, our component could have the following property:

```
public int NumberOfButtons { get { /* ... */ } set { /* ... */ } };
```

Based on the value of this property we might choose to instantiate a specified number of child buttons.

When configuring the component, the user might then choose what the number of buttons would be:

```
component.NumberOfButtons = 3;
// or
component.NumberOfButtons = 5;
```

Inside the `CreateChildren` method, we create *static* child components, instantiated with the component. But, since the `CreateChildren` method is being run only once (since the component is being initialized once), we can't use this method for instantiating new children *dynamically* (during the component lifetime). We are doing it inside the `CommitProperties` method instead (more on this later).

The INITIALIZE event

This event is dispatched following the execution of the `CreateChildren` method. At this point, child components have been added but not yet sized and laid out (i.e. positioned). You can subscribe to this event to perform additional processing before the sizing and layout processes occur.

The CommitProperties method

The purpose of this method is to commit any changes to custom properties and to synchronize changes so they occur at the same time or in a specific order. This means that regardless of the order of invalidation of certain properties, the order of the validation is specified with the code inside of the `CommitProperties` method.

This is one of the validation methods. You shouldn't call this method by yourself. eDriven calls this method automatically as a response to the previous `InvalidateProperties` method call.

This method is called during the component initialization (once, after the component is added to a display list) and each time during the component lifetime after the `InvalidateProperties` method gets called.

Note: This method will always be called before `Measure` and `UpdateDisplayList`, giving you the chance to influence both of these methods by calling `InvalidateSize` and `InvalidateDisplayList` right from within `CommitProperties`. You might choose that changing the *NumberOfButtons* affects component size, so you might force the component to re-measure itself with calling `InvalidateSize`.

Things to remember

- Never do the performance costly operations within property setters! Performance costly operations are all operations affecting component's children: the number of children, their layout, sizes, color etc.
- Setters should be as lightweight as possible and should generally use flags and `InvalidateProperties` call to schedule `CommitProperties` to run later.
- All of the processing you would have done in the setter is now done during `CommitProperties`. This can have a significant impact on performance because no matter how many times `InvalidateProperties` is called; `CommitProperties` will only be run once between two `OnGUI` calls.

The Measure method

The purpose of this method is to set a default width and height for your custom component. Additionally, you can also set a default minimum width and minimum height on your component.

While this method is run, the `Measure` on all components' children has already been run, so their default and minimum sizes are known.

Note: while this being true for containers, for simple components the children size is still unknown (measuring children doesn't happen automatically). From within this component's `Measure`, the `ValidateNow` method should be called on each child before reading their default and minimum values (to force child components to re-measure). For example:

```
protected void Measure()
{
    var height = 0;
    for (int i = 0; i < NumberOfChildren; i++)
    {
```

eDriven Framework Programming Manual

```
        var child = Children[i];
        child.ValidateNow();
        height += child.Height;
    }
    MeasuredHeight = height;
}
```

This method is called during the component initialization (once, after the component is added to a display list) and each time during the component lifetime after the `InvalidateSize` method gets called.

Note: This method will always be called after `CommitProperties` and before `UpdateDisplayList`, giving you the chance to influence the layout (child positions and sizes) by calling `InvalidateDisplayList` right from within `Measure`. You might choose that changing the container size affects its layout, so this way you might force the container to re-layout itself.

Things to remember

- `Measure` is being run only if no both explicit width and height have been given to the component. For example:

```
var button = new Button { Width = 100, Height = 100 };
```

The button in this example will never be measured (automatically), since it's pointless to do so - the component has both the width and height explicitly set and they shouldn't be changed.

- Within this method four (optional) properties should be set: `MeasuredWidth`, `MeasuredHeight`, `MeasuredMinWidth` and `MeasuredMinHeight`. You might choose not to set any of this; in that case they will be set to by the superclass.
- `Measure` is first called at the bottom of the display list and works its way up to the top.
- There are two convenience methods used for getting the child size:
`child.GetExplicitOrMeasuredWidth()` and
`child.GetExplicitOrMeasuredHeight()`.
- These functions will return either the explicit size or the measured size of the child component.

eDriven.Gui Designer

eDriven.Gui Designer basics

eDriven.Gui Designer is an add-on to eDriven.Gui, which enables visual GUI editing.

To use it, you should open the eDriven.Gui Designer window and dock it to Unity Editor. The window could be opened from the Window menu:

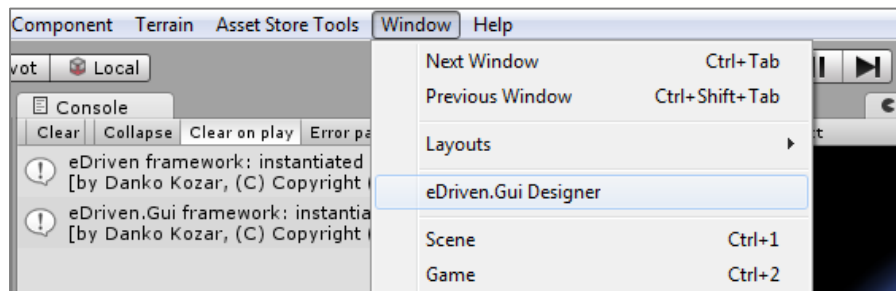


Fig 13 eDriven.Gui Designer in Window menu

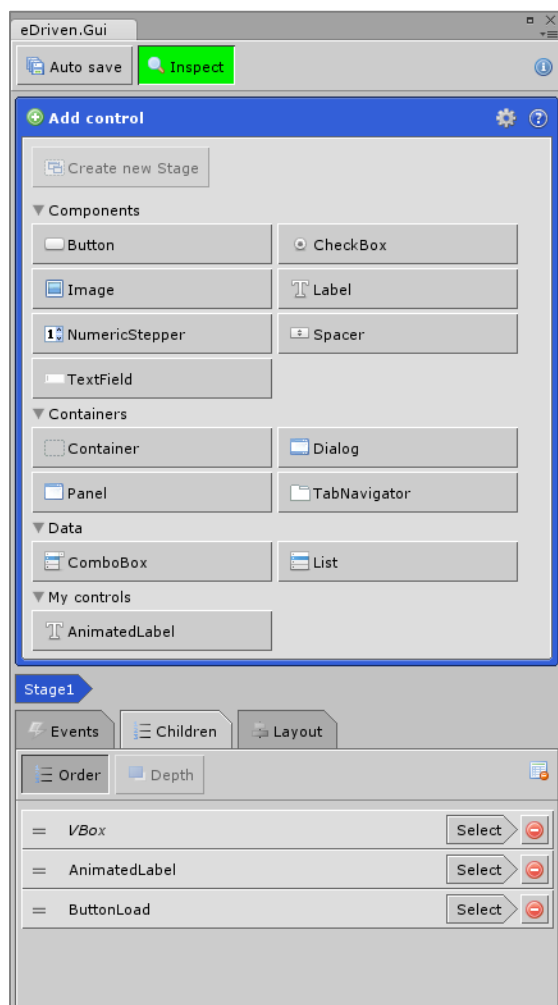


Fig 14 Designer window

Add control panel

This panel is part of the designer window. It is used to add new components to the stage.

Components could be added to containers or to the stage (*Stage* is also a *Container*).

Stage cannot be created inside other eDriven.Gui components. It can only be created on:

1. game objects not being eDriven.Gui components
2. the hierarchy root (game objects having no parent game object)

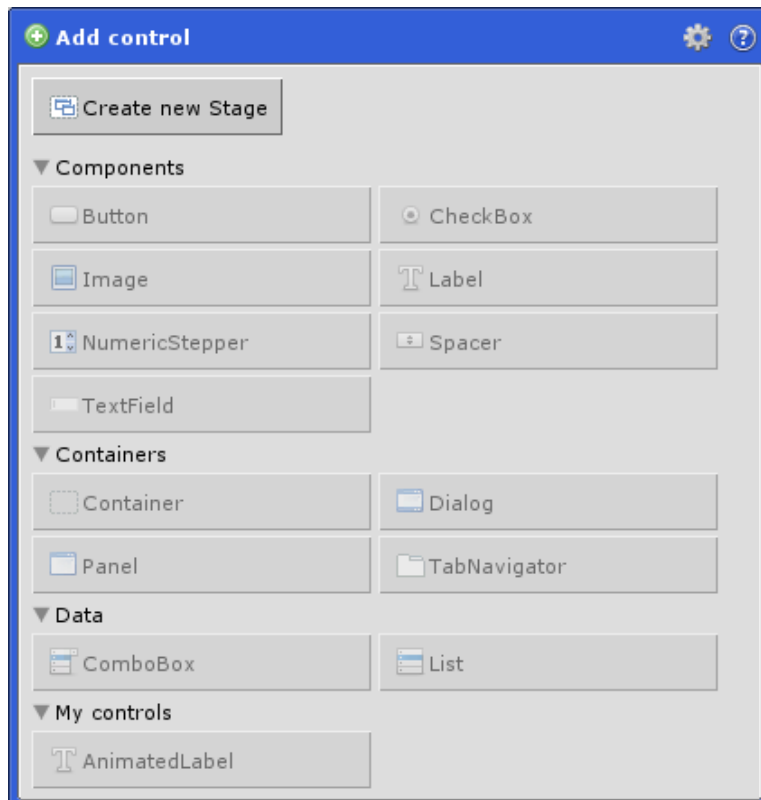


Fig 15 Add control panel

Creation options

You could toggle creation options by clicking the options button in the header of *Add control* dialog.

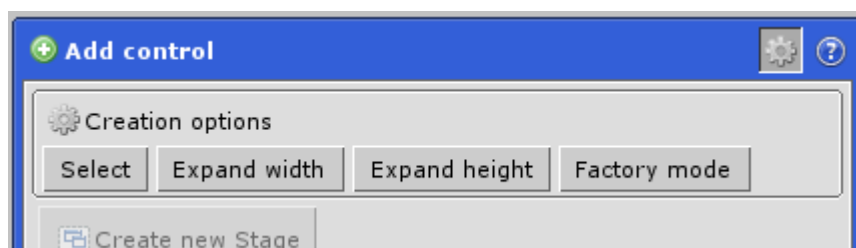


Fig 16 Creation options

eDriven Framework Programming Manual

If *Select* button is on, the newly created component will become selected.

Other 3 buttons are used for automatically configuring newly created components:

- Expand width sets component's `PercentWidth = 100`
- Expand height sets component's `PercentHeight = 100`
- Factory mode sets component adapter to factory mode

Events panel

This panel is part of the designer window. It is used to map events of the selected component (or its ancestors, utilizing the *event bubbling*).

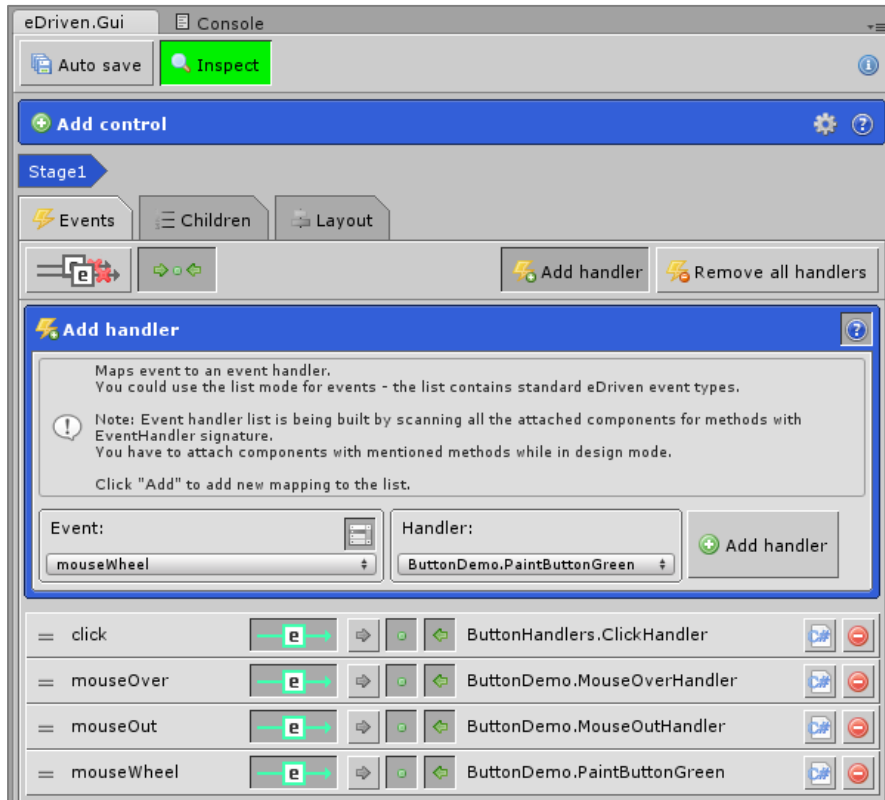


Fig 17 Events panel

To add an event handler, you should toggle the *Add handler* button to turn on the *Add handler* panel.

This panel brings the *Event* and *Handler* drop-downs.

The *Event* drop-down displays the standard set of GUI event types used with eDriven. You can also input any arbitrary event type using the text field mode:

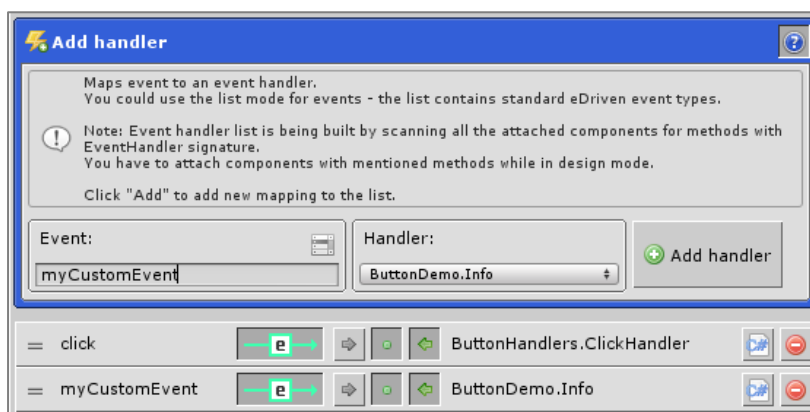


Fig 18 Text field mode

eDriven Framework Programming Manual

The *Handler* drop-down displays names of all the methods having the `eDriven.Core.Events.EventHandler` signature from all the attached scripts.

Note: methods from scripts inheriting `ComponentAdapter` are not displayed in the list.

The attached scripts are visible below the adapter script in the component inspector (ButtonDemo script in this example):

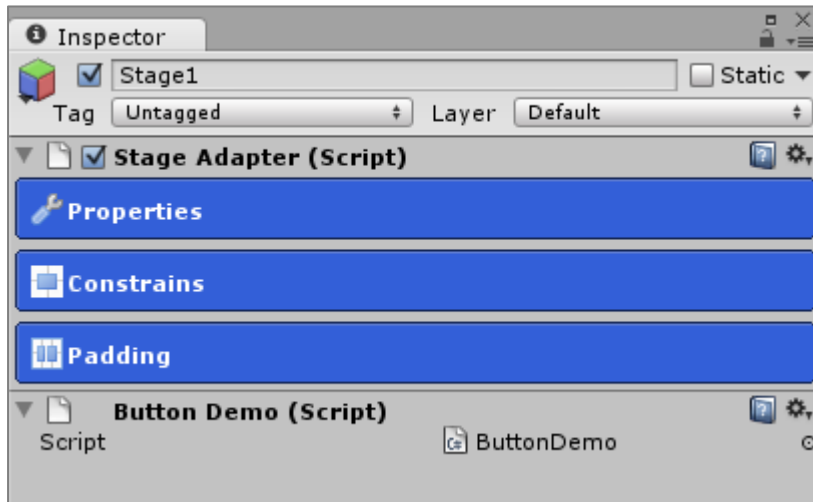


Fig 19 Handler script in inspector panel

About *Event handler* scripts

Here's an example of the *event handler* script:

```
using System;
using eDriven.Gui.Components;
using eDriven.Gui.Designer;
using eDriven.Gui.Dialogs.Alert;
using eDriven.Playground.Demo.Tweens;
using UnityEngine;
using Action=eDriven.Animation.Action;
using Event=eDriven.Core.Events.Event;

public class ButtonDemo : MonoBehaviour {

    public void PaintButtonGreen(Event e)
    {
        Button button = GuiLookup.GetComponent("button1") as Button;
        if (null != button)
            button.Color = Color.green;
    }

    public void RemoveButtonColor(Event e)
    {
        Button button = GuiLookup.GetComponent("button1") as Button;
        if (null != button)
            button.Color = null;
    }

    public void MouseOverHandler(Event e)
    {

```

eDriven Framework Programming Manual

```
        Debug.Log("MouseOverHandler: " + e.Target);
    }

    public void MouseOutHandler(Event e)
    {
        Debug.Log("MouseOutHandler: " + e.Target);
    }

    public void LoadLevel(Event e)
    {
        Debug.Log("Loading level 1");
        Application.LoadLevel(1);
    }

    public void Info(Event e)
    {
        Alert.Show(
            "Info",
            "This is the info message. The time is: " +
            DateTime.Now.ToLongTimeString(),
            AlertButtonFlag.Ok,
            new AlertOption(AlertOptionType.HeaderIcon,
                Resources.Load("Icons/information")),
            new AlertOption(AlertOptionType.AddedEffect, _alertEffect)
        );
    }
}
```

As you can see, the script consists of a number of methods having the `EventHandler` signature. Inside event handlers, the user of the component writes code for programming actions caused by the GUI interaction. Those actions could trigger other GUI screen elements to change, or could interact with the 3D world – there are no restrictions to what could be accomplished using event handlers.

About Event handler panel relation to eDriven event system

Event handler panel is a graphical representation of assigning the event handlers with C# code using the following syntax:

```
component.AddEventListener(Button.PRESS, ButtonPressHandler);
```

Or `+=` syntax, which is basically the *syntactic sugar* wrapping the `AddEventListener` method:

```
Component.Press += PressHandler;
```

Upon the application start, all the event handler mappings specified for each adapter are being pushed to instantiated components.

Important: When having multiple components instantiated by the same adapter (in *factory mode*), all instantiated components map to handlers attached to the adapter instantiating them. If adapter is being disposed (open dialog after scene change) event handlers won't fire anymore.

In the *Events panel*, mapped event handlers are displayed as rows. Each row could be removed from the list using the removed button. The script containing the event handler method could be opened by clicking the script button. The script is then being opened by default code editor.

eDriven Framework Programming Manual

Using the *Disable* button, each event handler could be disabled separately:



Fig 20 Enabling/disabling event handlers

Using the *Disable All* button all the handlers for selected adapter could be disabled:



Fig 21 Enabling/disabling all event handlers

As well as with C# eDriven component, you can also rely on event bubbling when using the designer components. For instance, you could subscribe for a particular event on Stage. If the event bubbles, the event handler attached to Stage would fire for events dispatched by its children.

One additional parameter when subscribing event handlers is the bubbling phase to which we subscribe, giving us the combination of *capture*, *target* and *bubbling* phases.

Clicking the *Show bubbling phases* reveals bubbling phases in all event handler rows:



Fig 22 Showing/hiding bubbling phases

Each of the handlers displays bubbling phases in which it listens for events. This is the place where we can fine-tune our subscriptions:



Fig 23 Enabling/disabling particular bubbling phase subscriptions

In the example above, the first row shows the *target + bubbling phase* subscription (which is default), while the second row shows the *capture phase* subscription.

There are 8 possible combinations of bubbling phases setting:

- 000
- 001
- 010
- 011
- 100
- 101
- 110
- 111

Children panel

This panel is part of the designer window. It is used to change the order of children.

The order of children is needed by layout classes, which use this order to lay the children the particular way (for instance top-to-bottom with vertical box layout).

Note: the order cannot be specified using the hierarchy panel, because this panel ignores any kind of order and orders child transforms by game object name (this is sure something that cannot be relied on).

Additionally, the *Children panel* is used to select a particular child, as well as remove a child.

Note: if selected component is not a container, this panel won't be visible.

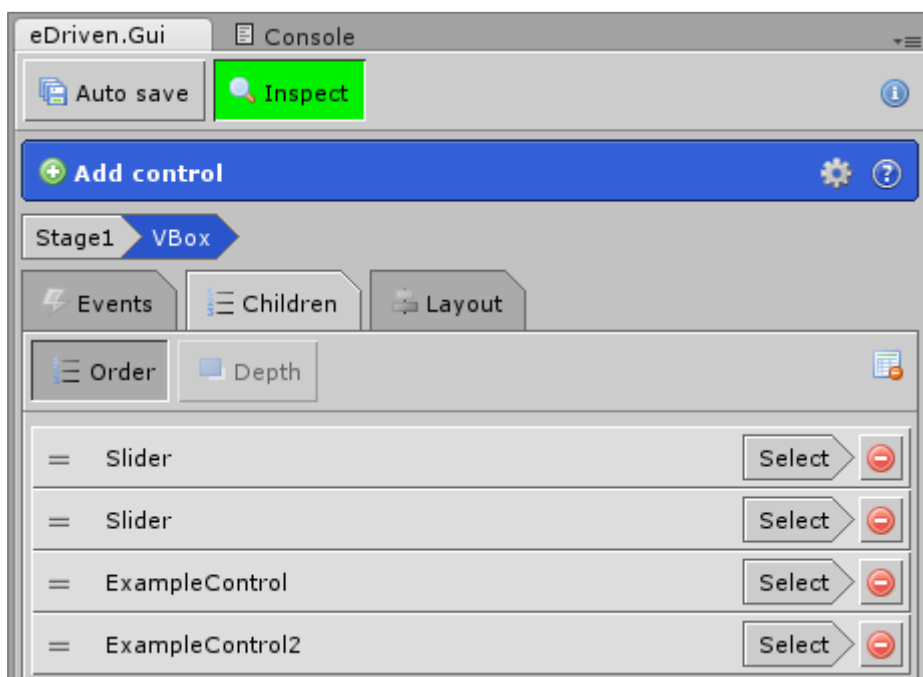


Fig 24 Children panel

Layout panel

This panel is part of the designer window. It is used to change the layout of the selected container.

Note: if selected component is not a container, this panel won't be visible.

The layout of a container could be specified in two ways:

- Descriptor mode
- Full mode

Descriptor layout mode

This mode is often used for prototyping, e.g. for having visible results quickly. It uses the default values of spacing between child controls (10px).

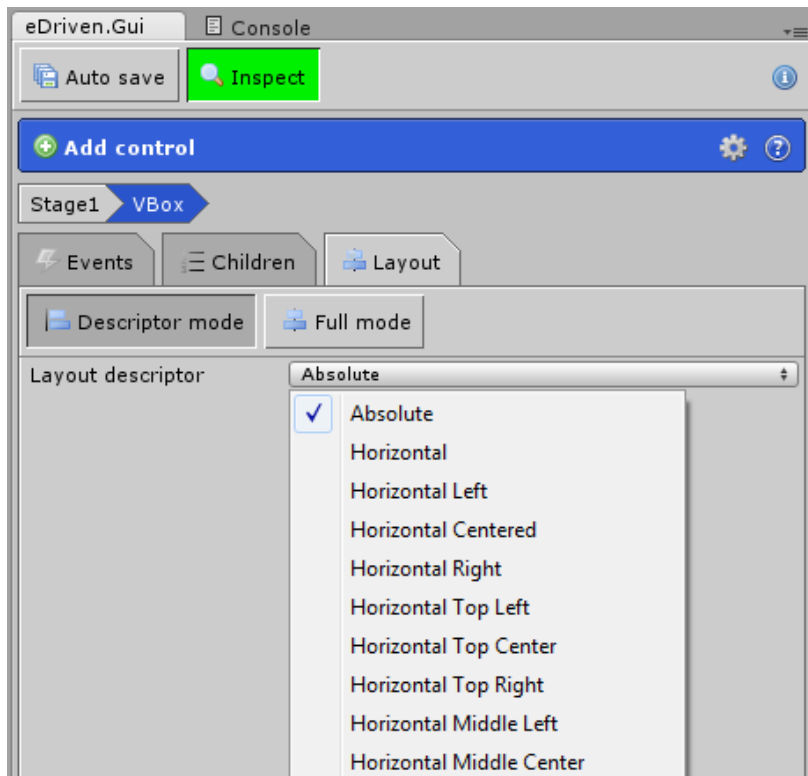


Fig 25 Descriptor layout mode

Full layout mode

This mode offers a full layout customization. All values specific to the selected layout class could be tweaked using this mode.

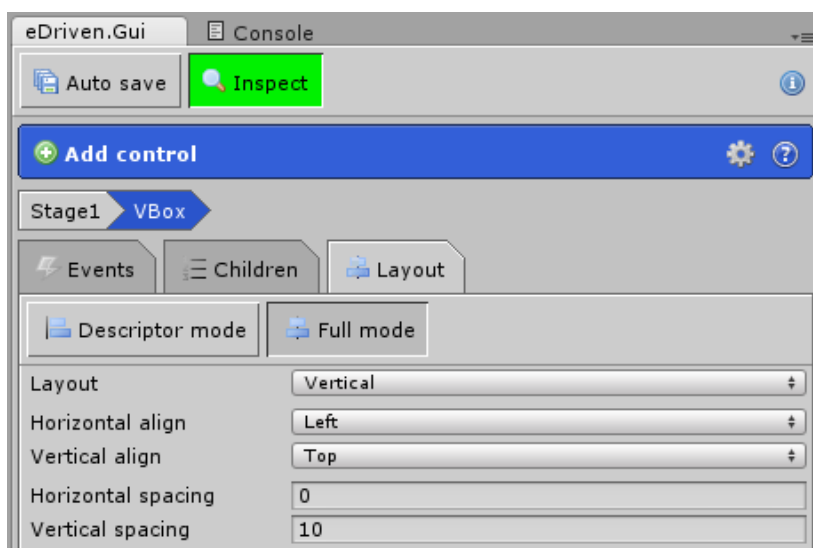


Fig 26 Full layout mode

Component inspector

Properties of the selected component are being edited inside the component inspector (the same way all the Unity components are).

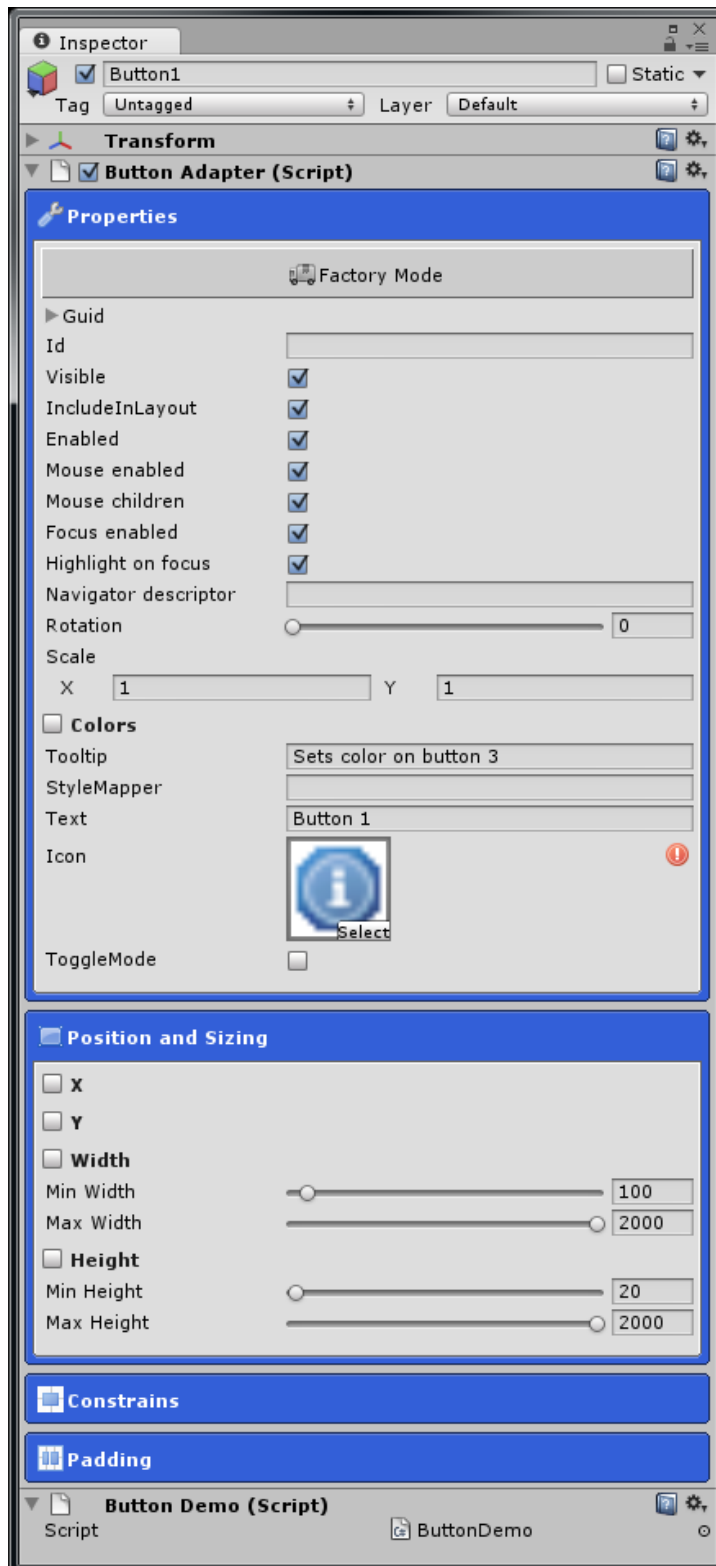


Fig 27 Component inspector

eDriven Framework Programming Manual

eDriven.Gui inspector is divided into multiple sections (panels) which are easy to notice because of their specific bluish border color.

The main script on each eDriven.Gui game object is the adapter script. eDriven.Gui designer uses component adapters because components don't extend the `MonoBehaviour` class. Using adapters, components are being "adapted" to Unity's game object hierarchy.

Each component adapter inherits from `ComponentAdapter` class (which in turn inherits from `MonoBehaviour`).

Besides the adapter (which is mandatory and you shouldn't remove it from the game object) a number of scripts could be attached to the same game object. Examples of such scripts are:

- data provider scripts (used by data controls)
- event handler scripts (containing a number of methods having `eDriven.Core.Events.EventHandler` signature)
- arbitrary scripts (directly or indirectly extending `MonoBehaviour`)

To reference the instantiated component from the arbitrary script, you should implement `ComponentInstantiated` method.

Each component adapter calls two methods on each script attached to the same game object:

- `ComponentInstantiated` – this method is called just upon the component instantiation (the adapter instantiated the component)
- `InitializeComponent` – this method is called during the component initialization, giving you the chance to tweak some of its values

Both methods should receive `eDriven.Gui.Components.Component` as an argument.

You could implement the following methods in your arbitrary scripts:

```
void ComponentInstantiated(Component component)
{
    Debug.Log("Initializing component: " + component);
}

void InitializeComponent(Component component)
{
    Debug.Log("Initializing component: " + component);
}
```

Additionally, if you are keeping the adapter reference, you could reference the component using `adapter.Component` (but only after the component has been instantiated).

Note: you shouldn't add or remove component adapters by hand. It's better to leave that job to eDriven.Gui editor.

Also note that only a single component adapter should be added to a game object.

Main property panel

Although the adapter could contain multiple panels, the main one is the most important one, since it is always present. It is used to edit the most used properties of eDriven.Gui components.

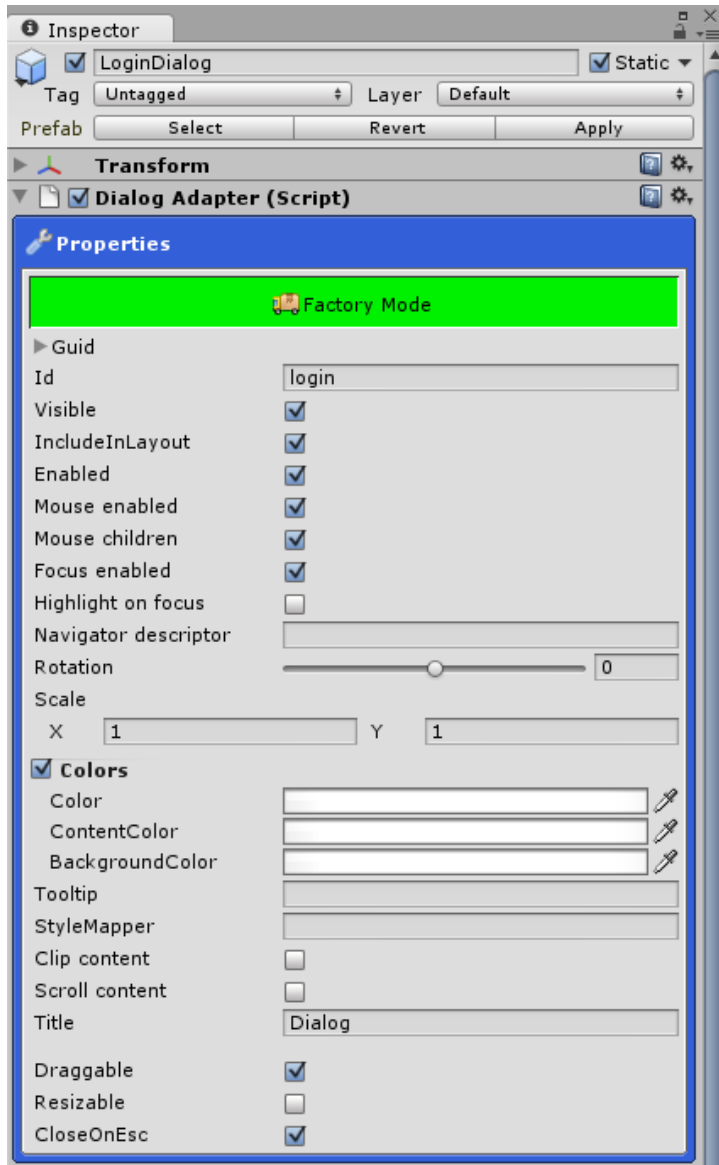


Fig 28 Main property panel

Note: each component adapter could add editable values inside the main panel, or do this within its own, separate panel.

Values edited using this panel are:

- **FactoryMode:** Boolean value indicating that this component adapter acts as a factory, meaning the component isn't instantiated during the scene start but is waiting to be referenced when needed to produce a component (the most common usage cases are popup windows)
- **Visible:** Sets the visibility of the selected component

- `IncludeInLayout`: Includes/excludes component in layout
- `Enabled`: Enables/disables interaction with the component
- `MouseEnabled`: Enables/disables mouse interaction with the component
- `MouseChildren`: Enables/disables mouse interaction component's children
- `FocusEnabled`: The flag indicating that the component can receive focus (and is processing keyboard events while in focus)
- `HighlightOnFocus`: The flag specifying should the component display focus overlay when in focus
- `NavigatorDescriptor`: String value used as a title of *TabNavigator* tab when component a direct child
- `Rotation`: Component rotation in range [-360, 360]. Note: this is the post mouse-processing rotation, meaning component bounds are the same as when the component wouldn't be rotated at all (bounds are a horizontal rectangle).
This is subject to change. Until then, use the `Rotation` property for animation purposes only, not for mouse detection.
- `Scale`: Component scale (`Vector2`). Note: this is the post mouse-processing scale, meaning component bounds are the same as when the component wouldn't be scaled at all (bounds are a horizontal rectangle).
This is subject to change. Until then, use the `Scale` property for animation purposes only, not for mouse detection.
- `Color`: Overall component tint color
- `ContentColor`: Icon/text tint color
- `BackgroundColor`: Component background color
- `Tooltip`: Component tooltip
- `StyleMapper`: The ID of the style mapper located anywhere in the scene (but for this component type). If not found, the default component skin is being applied (see *style chain*).

Container specific properties:

- `ClipContent`: Clip children outside the container bounds
- `ScrollContent`: Show scrollbars if not enough place to display children inside the container

Other properties - for instance, `Dialog` class adds the following properties:

- `Draggable`: A flag indicating the dialog is draggable
- `Resizable`: A flag indicating the dialog is resizable
- `CloseOnEsc`: A flag indicating the dialog could be closed by pressing the *Esc* key

Position and sizing

This panel is used to specify component coordinates and sizing rules.

Note: The effect of applying coordinates and sizing rules to a component isn't always noticable. It really depends of a parent container's layout. For instance, [BoxLayout](#) ignores *X* and *Y* properties.

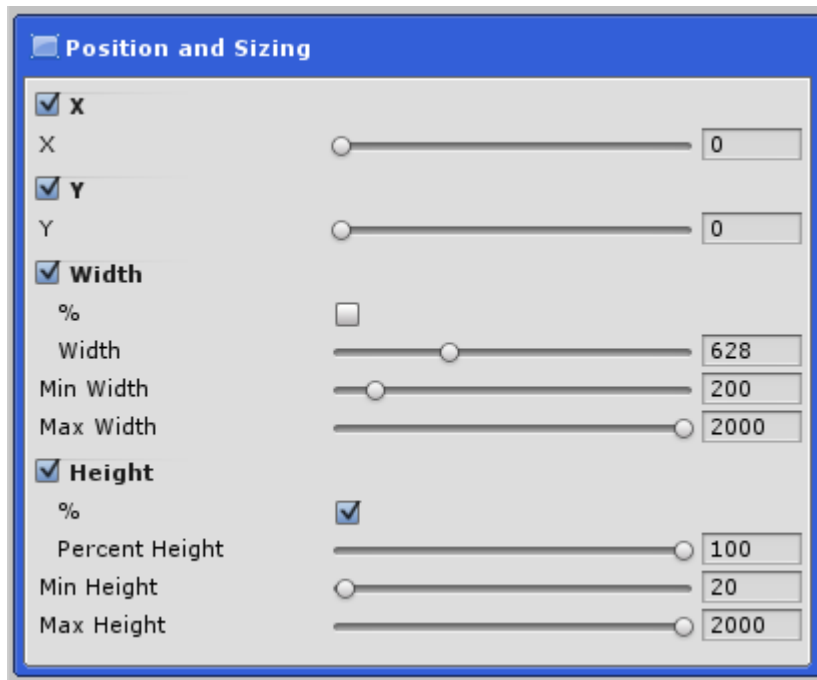


Fig 29 Position and Sizing panel

Values edited using this panel are:

- **X**: The X coordinate of the component within the parent container
- **Y**: The Y coordinate of the component within the parent container
- **Width**: Explicit width of the component. If set, the component width won't be measured
- **PercentWidth**: The width of the component as the percentage of the container width. The component will have its final width set based on the sizing rules
- **MinWidth**: Min width of the component. Used by the sizing rules
- **MaxWidth**: Max width of the component. Used by the sizing rules
- **Height**: Explicit height of the component. If set, the component height won't be measured
- **PercentHeight**: The height of the component as the percentage of the container height. The component will have its final height set based on the sizing rules
- **MinHeight**: Min height of the component. Used by the sizing rules
- **MaxHeight**: Max height of the component. Used by the sizing rules

Note that some of these values are setters-only (you can't rely on *Width* displaying the actual current *Width* of the component).

Also note that Min/Max values are being used by sizing mechanism exclusively, meaning they won't stop you from setting the explicit width and height values to a value outside the Min/Max range.

Constrains

This panel is used to specify component constrains used exclusively by `AbsoluteLayout` (if specified in container). In each other case tweaking values inside this panel has no purpose.

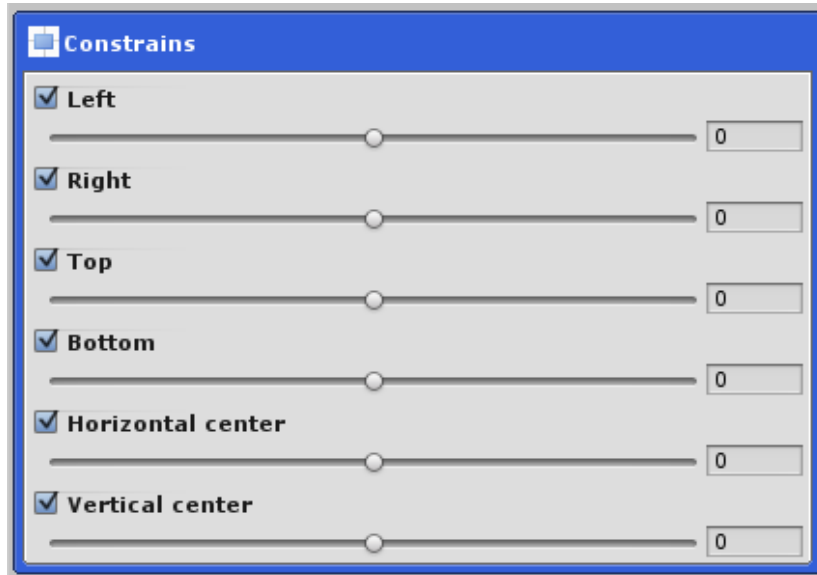


Fig 30 Constrains panel

Values edited using this panel are:

- `Left`: The distance from the left edge of the container to the left edge of the component
- `Right`: The distance from the right edge of the container to the right edge of the component
- `Top`: The distance from the top edge of the container to the top edge of the component
- `Bottom`: The distance from the bottom edge of the container to the bottom edge of the component
- `HorizontalCenter`: The distance from the horizontal center edge of the container to the horizontal center of the component
- `VerticalCenter`: The distance from the vertical center edge of the container to the vertical center of the component

Padding

This panel is used to specify component or padding.

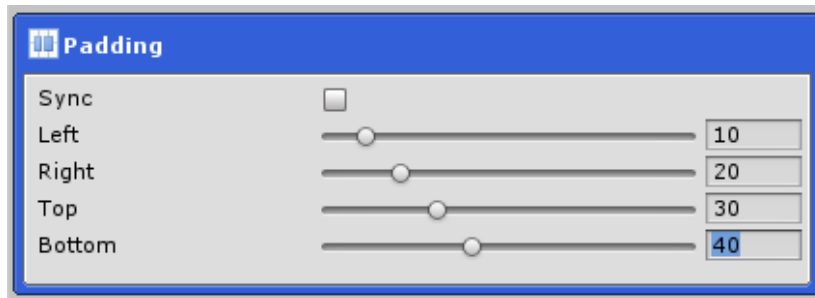


Fig 31 Padding panel

Values edited using this panel are:

- **Left:** Spacing between the left edge of the container to the closest component
- **Right:** Spacing between the right edge of the container to the closest component
- **Top:** Spacing between the top edge of the container to the closest component
- **Bottom:** Spacing between the bottom edge of the container to the closest component

Note: the padding could also be applied if component is not a container. Than it adds up to padding specified with component background style (GUIStyle).

Additional panels

There are additional panels usually added by component editors inheriting `ComponentEditor` or `ContainerEditor`.

For example, `PanelEditor` adds a *Tools* panel, and `DialogEditor` adds the *Button group* panel.

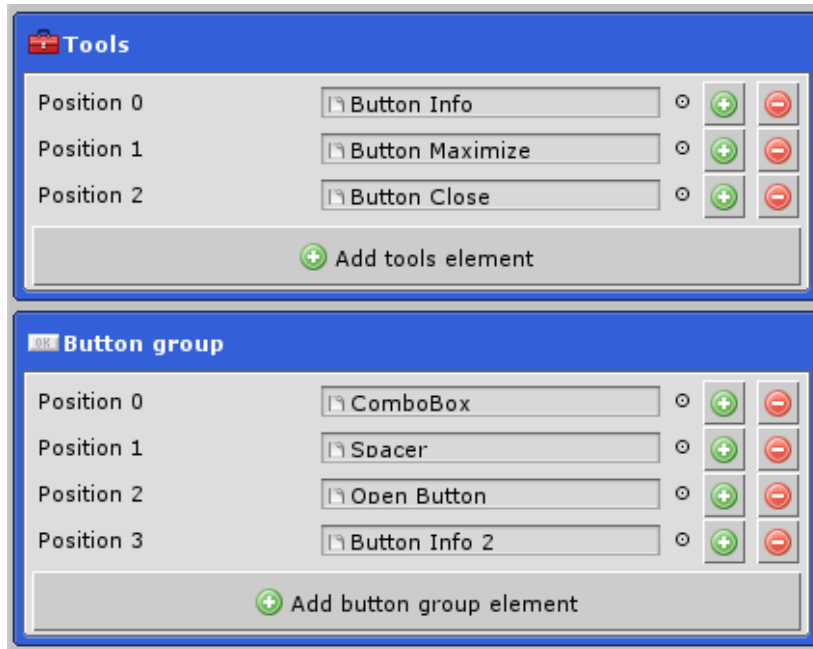


Fig 32 Tools and Button group panels

Usage: developer might decide he wants a particular component to be displayed inside the tools group (in panel header) or inside the dialog button group. He should add these components as dialog children, but after addition those components should be dragged to slots inside these panels.

Note: It isn't possible to drag a single component to multiple slots.

Designer Components

Creating designer components

Chances are that most of the eDriven.Gui users aren't GUI experts. They don't know, want or like to build their GUIs using C#. Most of them expect to use eDriven using the designer.

However, experienced GUI developers usually build their components with C#.

What average users expect is using full-power C# components made by experts inside the designer.

The problem is that C# components cannot be just dragged & dropped to the scene. That's because they are a different kind of components.

All eDriven.Gui components inherit from `eDriven.Core.Events.EventDispatcher` as a base class, so cannot inherit `UnityEngine.MonoBehaviour` at the same time.

To be attached to game objects, they have to be "adapted" to a component system used by Unity.

There isn't much work involved to adapt the component for designer. It's a 2 step process: 2 additional classes should be created:

- The adapter class
- The editor class

Before seeing the examples for both classes, let's first examine the example component class (which I've build as a starting point for C# component developers).

The control class

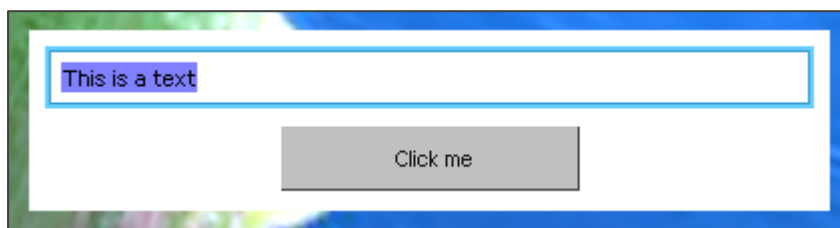


Fig 33 Custom (complex) control

The example control extends a container, having the vertical layout and two children: a text field and a button.

This is the eDriven.Gui C# control created by the rules described in *Components* section.

It has two exposed properties

- `Text`
- `ButtonText`

eDriven Framework Programming Manual

Both of the properties setters trigger the invalidation mechanism, to be able to set properties on children after them being instantiated.

Also, the component implements the custom focus and tabbing support, as well as the button press handler.

This is an excellent example of encapsulation: component children are private and cannot be referenced from the outside (either from C# or from designer). The component could then be manipulated from the outside only through its exposed properties (`Text` and `ButtonText` namely).

This illustrates the main difference between complex components built with C# and with designer:

- Components build with C# are *monolithic* when in designer: their children cannot be selected and manipulated using the inspector. The component is being represented with a single game object in the *hierarchy view*.
- Components build with designer have a tree structure when viewed in the *hierarchy view*. They are not monolithic since their children could be manipulated: added, removed, moved, inspected and edited.
The children of this kind of component could be other components: complex components build with C#, or trees containing other GUI components or prefabs.

The ExampleControl class

```
using System.Collections.Generic;
using eDriven.Gui.Components;
using eDriven.Gui.Containers;
using eDriven.Gui.Dialogs.Alert;
using eDriven.Gui.Layout;
using eDriven.Gui.Plugins;
using Event=eDriven.Core.Events.Event;

namespace eDriven.Extensions.ExampleControl
{
    /// <summary>
    /// Simple component example
    /// </summary>
    public class ExampleControl : Container
    {
        private TextField _textField;
        private Button _button;

        /// <summary>
        /// Overriding constructor for setup
        /// </summary>
        public ExampleControl()
        {
            Layout = new BoxLayout { Direction = LayoutDirection.Vertical,
            VerticalSpacing = 10 };
            SetStyle("showBackground", true);
            MinWidth = 150;
            MinHeight = 100;
            Padding = 10;
            Plugins.Add(new TabManager {CircularTabs = true});
            AddEventListener(Button.PRESS, PressHandler);
        }
    }
}
```

eDriven Framework Programming Manual

```
/// <summary>
/// Create children
/// Use AddChild inside this method exclusively
/// </summary>
override protected void CreateChildren()
{
    base.CreateChildren();

    _textField = new TextField { PercentWidth = 100, ProcessKeys =
true, StyleMapper = "examplecontrol_textfield" };
    AddChild(_textField);

    _button = new Button { Text = "Click me", MinWidth = 150,
StyleMapper = "examplecontrol_button" };
    AddChild(_button);
}

/// <summary>
/// Setting focus
/// </summary>
public override void SetFocus()
{
    _textField.SetFocus();
}

/// <summary>
/// Run when pressing tab
/// </summary>
/// <returns></returns>
public override List<DisplayListMember> GetTabChildren()
{
    return new List<DisplayListMember>(new DisplayListMember[] {
_textField, _button });
}

private bool _textChanged;
private string _text;
public string Text
{
    get
    {
        return _text;
    }
    set
    {
        if (value == _text)
            return;

        _text = value;
        _textChanged = true;
        InvalidateProperties();
    }
}

private bool _buttonTextChanged;
private string _buttonText;
public string ButtonText
{
    get
    {

```


eDriven Framework Programming Manual

```
        return _buttonText;
    }
    set
    {
        if (value == _buttonText)
            return;

        _buttonText = value;
        _buttonTextChanged = true;
        InvalidateProperties();
    }
}

protected override void CommitProperties()
{
    base.CommitProperties();

    if (_textChanged)
    {
        _textChanged = false;
        _textField.Text = _text;
    }

    if (_buttonTextChanged)
    {
        _buttonTextChanged = false;
        _button.Text = _buttonText;
    }
}

private void PressHandler(Event e)
{
    Alert.Show("Info", string.Format(@"The button was pressed. The
input text is:
""{0}""", _textField.Text), AlertButtonFlag.Ok, delegate { SetFocus(); });
}

/// <summary>
/// Cleanup
/// </summary>
public override void Dispose()
{
    base.Dispose();

    RemoveEventListener(Button.PRESS, PressHandler);
}
}
```

The adapter class

The adapter class has to extend [ComponentAdapter](#).

The moment it is created, it should be visible inside the *Add control* panel (since eDriven scans the project for getting all the [ComponentAdapter](#) inheritors):

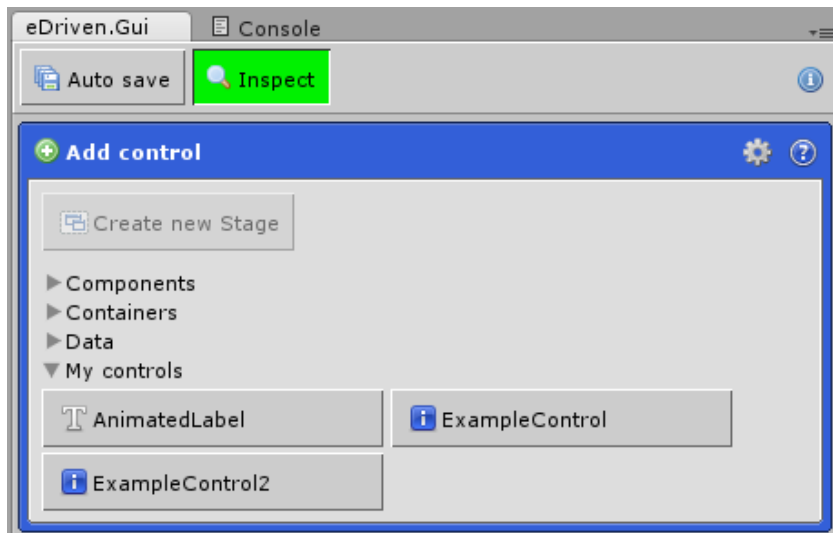


Fig 34 Add control panel displaying the ExampleControl

The ExampleAdapter class

```
using System;
using System.Reflection;
using eDriven.Extensions.ExampleControl;
using eDriven.Gui.Components;
using eDriven.Gui.Designer;
using Component=eDriven.Gui.Components.Component;

[Toolbox(Label = "ExampleControl", Group = "My controls", Icon =
"eDriven/Editor/Controls/example")]

public class ExampleAdapter : ComponentAdapter
{
    #region Saveable values

    [Saveable]
    public string Text = "This is a text";

    [Saveable]
    public string ButtonText = "Click me";

    [Saveable]
    public bool BoolExample;

    [Saveable]
    public SliderOrientation EnumExample = SliderOrientation.Horizontal;

    #endregion

    public ExampleAdapter()
    {
        // setting default values
        MinWidth = 400;
        PaddingLeft = 10;
        PaddingRight = 10;
        PaddingTop = 10;
        PaddingBottom = 10;
    }
}
```

eDriven Framework Programming Manual

```
public override Type ComponentType
{
    get { return typeof(ExampleControl); }
}

public override Component NewInstance()
{
    return new ExampleControl();
}

public override void Apply(Component component)
{
    base.Apply(component);

    ExampleControl exampleControl = (ExampleControl)component;
    exampleControl.Text = Text;
    exampleControl.ButtonText = ButtonText;
}
}
```

Explaining the ExampleAdapter class

```
[Toolbox(Label = "ExampleControl", Group = "My controls", Icon =
"eDriven/Editor/Controls/example")]
```

This is a class attribute specifying a few details about placement and look of toolbox button for the specific component:

- Label - the text on toolbox button
- Group - toolbox group
- Icon - toolbox icon path (relative to *Resources* folder)

```
public class ExampleAdapter : ComponentAdapter
```

The adapter should extend `ComponentAdapter` class, if your component extends `Component` directly.

If your component for instance extends `Button`, your adapter should extend `ButtonAdapter`.

However, when the component itself extends `Container` (or subclasses) – which most custom components do - your adapter shouldn't necessarily extend `ContainerAdapter` or its subclasses.

You should extend `ContainerAdapter` only if you want to allow children manipulation from the designer. This is because using the `ContainerAdapter` signals that child components are handled from designer (using the hierarchy tree). This should not always be the case, since a C# component is a monolithic structure and shouldn't have its children manipulated from the designer.

```
[Saveable]
```

This is the attribute indicating that the following member could be persisted by eDriven after the play mode is stopped

```
public ExampleAdapter()
```

eDriven Framework Programming Manual

The constructor, which is the place to set default values of properties defined in superclass

```
public override Type ComponentType
{
    get { return typeof(ExampleControl); }
}
```

Getter indicating the type of the component this adapter operates with (used by the [GuiLookup](#) class for various lookups).

```
public override Component NewInstance()
{
    return new ExampleControl();
}
```

A factory method used for creating an instance of the component. This method is being called once per adapter when during the application start, or multiple times if adapter working in factory mode.

When not in factory mode, the instantiated component is then being referenced by the adapter for easy access.

When in factory mode, the adapter won't instantiate any component by itself, but should be called explicitly to create a component instance (the component won't be referenced by the adapter). The factory mode is usually used for components instantiated multiple times such as popup dialogs.

```
public override void Apply(Component component)
{
    base.Apply(component);

    ExampleControl exampleControl = (ExampleControl) component;
    exampleControl.Text = Text;
    exampleControl.ButtonText = ButtonText;
}
```

This method is being called each time the any of the editor GUI elements is being changed by the user. It applies the current adapter values to actual control.

You should always call base class from within this method, because its values should also be applied.

The editor class

The editor class has to extend [ComponentEditor](#).

[ComponentEditor](#) class extends Unity's [Editor class](#) and creates the custom inspector used for:

- editing component properties by the user
- serializing and saving values with the scene by UnityEditor

The ExampleEditor class

```
using System;
using System.Reflection;
using eDriven.Gui.Components;
```

eDriven Framework Programming Manual

```
using eDriven.Gui.Editor.Editors;
using UnityEditor;

[CustomEditor(typeof(ExampleAdapter))]
[CanEditMultipleObjects]

public class ExampleEditor : ComponentEditor
{
    public SerializedProperty Text;
    public SerializedProperty ButtonText;
    public SerializedProperty BoolExample;
    public SerializedProperty EnumExample;

    /// Note: the method previously called OnEnableImpl (eDriven.Gui 1.6)
    /// will be renamed to Initialize (eDriven.Gui 1.7)
    protected override void Initialize()
    {
        base.Initialize();

        Text = serializedObject.FindProperty("Text");
        ButtonText = serializedObject.FindProperty("ButtonText");

        #region Example

        BoolExample = serializedObject.FindProperty("BoolExample");
        EnumExample = serializedObject.FindProperty("EnumExample");

        #endregion

    }

    /// <summary>
    /// Rendering controls at the end of the main panel
    /// </summary>
    protected override void RenderMainOptions()
    {
        base.RenderMainOptions();

        Text.stringValue = EditorGUILayout.TextField("Text",
Text.stringValue);
        ButtonText.stringValue = EditorGUILayout.TextField("Button text",
ButtonText.stringValue);

        #region Example

        BoolExample.boolValue = EditorGUILayout.Toggle("BoolExample",
BoolExample.boolValue);
        EnumExample.enumValueIndex =
(int) (SliderOrientation) EditorGUILayout.EnumPopup(
            "EnumExample",

(SliderOrientation) Enum.GetValues(typeof(SliderOrientation)).GetValue(EnumE
xample.enumValueIndex)
        );

        #endregion

        EditorGUILayout.Space();
    }
}
```

eDriven Framework Programming Manual

Explaining the ExampleEditor class

```
[CustomEditor(typeof(ExampleAdapter))]
```

This is a class attribute specifying a type of the adapter this editor is used for editing of.

```
public class ExampleEditor : ComponentEditor
```

The adapter should extend `ComponentEditor` class, if your component extends `Component` directly.

If your component for instance extends `Button`, your adapter should extend `ButtonEditor`.

```
protected override void Initialize()
```

This method is used for initialization of the adapter using the previously saved values, kept in `serializedObject`.

```
Text = serializedObject.FindProperty("Text");
```

Inside the `Initialize` method we should read each of the saved values we are interested in.

```
protected override void RenderMainOptions()
```

Everything inside of this method is being rendered at the end of the main inspector panel (the *Properties* panel).

You should always call base class from within this method, because its controls should also be rendered in the inspector.

```
protected override void RenderExtendedOptions()
```

Optional method used for rendering editor controls inside the separate inspector panel (used with `PanelRenderer` helper class).

```
Text.stringValue = EditorGUILayout.TextField("Text", Text.stringValue);
```

The "classic" way of editing inspector values described in [EditorGUILayout class docs](#).