Хотя пользовательский элемент можно построить в любом проекте WPF, обычно такие элементы размещаются в специально выделенной сборке — библиотеке классов (DLL). Это позволяет разделять работу с множеством приложений WPF.

Чтобы гарантировать наличие всех необходимых ссылок на сборки и импорт всех нужных пространств имен, при создании приложения в Visual Studio в качестве типа проекта следует выбрать Custom Control Library (WPF) (Библиотека пользовательских элементов управления (WPF)). Внутри библиотеки классов можно создавать сколько угодно элементов управления.

Как при разработке любой библиотеки классов, часто стоит помещать как саму библиотеку классов, так и приложение, использующее ее, в одно и то же решение Visual Studio. Это позволит легко модифицировать и отлаживать обе части вместе.

Первый шаг в создании пользовательского элемента управления — это выбор корректного базового класса для наследования. Ниже перечислены некоторые часто применяемые классы для создания пользовательских элементов управления, а на рисунке показано их расположение в иерархии элементов:

**FrameworkElement**

Это — самый низкий уровень, с которым обычно приходится иметь дело при создании пользовательского элемента. Обычно такой подход выбирается только тогда, когда нужно нарисовать его содержимое "с нуля" посредством переопределения OnRender() и использования System.Windows.Media.DrawingContext. Это похоже на подход, где пользовательский интерфейс конструировался на основе [объектов Visual](https://professorweb.ru/my/WPF/graphics_and_animation/level14/14_1.php). Класс FrameworkElement предоставляет лишь самый базовый набор свойств и событий для элементов, которые не предназначены для взаимодействия с пользователем

**Control**

Этот класс чаще всего служит начальной точкой при построении элемента управления "с нуля". Это — базовый класс для всех взаимодействующих с пользователем графических элементов управления. Класс Control добавляет свойства для установки фона и переднего плана, а также шрифта и выравнивания содержимого. Кроме того, этот класс помещает себя в последовательность обхода по клавише <Tab> (свойством IsTabStop) и получает уведомления о двойном щелчке (через события MouseDoubleClick и PreviewMouseDoubleClick). Но что более важно, так это то, что класс Control определяет свойство **Templat**e, позволяющее заменять его внешний вид с неограниченной гибкостью

**ContentControl**

Это — базовый класс для элементов управления, которые могут отображаться как единое целое с произвольным содержимым. Содержимое может быть элементом пользовательского объекта, применяемого в сочетании с шаблоном. (Содержимое устанавливается свойством Content, а необязательный шаблон может быть представлен в свойстве ContentTemplate.) Многие элементы управления упаковывают специфический, ограниченный тип содержимого (вроде строки текста в текстовом поле). Поскольку эти элементы управления не поддерживают всех элементов, они не должны определяться как элементы управления с содержимым

**UserControl**

Это элемент управления с содержимым, который может быть сконфигурирован с применением поверхности времени проектирования. Хотя такой пользовательский элемент управления не настолько отличается от обычного элемента управления с содержимым, обычно он используется тогда, когда необходимо быстро повторно применить неизменный блок пользовательского интерфейса в более чем одном окне (вместо создания действительно отдельного элемента управления, который может быть перенесен из одного приложения в другое)

**ItemsControl и Selector**

ItemsControl — базовый класс для элементов управления, служащих оболочками для списков элементов, но не поддерживающих выбор позиций, в то время как Selector — более специализированный базовый класс для элементов, поддерживающих выбор. Эти классы нечасто применяются для создания пользовательских элементов управления, поскольку средства шаблонов данных ListBox, ListView и TreeView обеспечивают достаточную гибкость

**Panel**

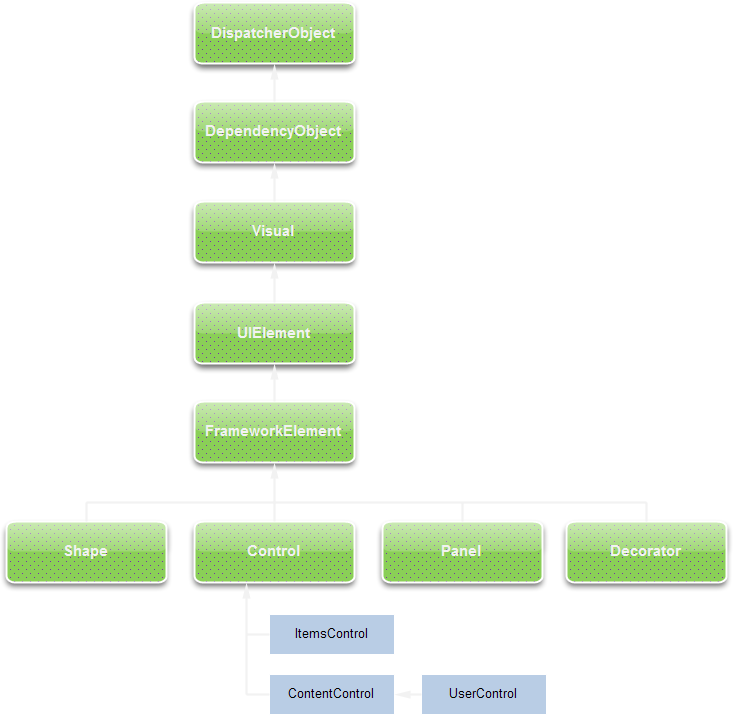
Базовый класс для элементов управления, обладающих логикой компоновки. Элемент с компоновкой может содержать в себе множество дочерних элементов и размещать их в соответствии с определенной семантикой компоновки. Часто панели включают присоединенные свойства, которые могут быть установлены в дочерние элементы для того, чтобы конфигурировать их расположение

**Decorator**

Это базовый класс для элементов, служащих оболочками для других элементов и обеспечивающих графический эффект или определенное средство. Двумя яркими примерами могут служить Border, который рисует линию вокруг элемента, и Viewbox, масштабирующий свое содержимое динамически с использованием трансформаций. Среди других декораций — классы Chrome, служащие для снабжения знакомыми рамками и фоном часто используемых элементов управления, таких как кнопка

**Специфический класс элемента**

Если необходимо усовершенствовать существующий элемент управления, можно наследоваться непосредственно от класса этого элемента, управления. Например, можно создать элемент TextBox со встроенной логикой проверки достоверности. Однако прежде чем предпринять такой шаг, подумайте, нельзя ли достичь той же цели с помощью кода обработки событий или отдельного компонента. Оба подхода позволят отделить логику от элемента управления и применять ее в других элементах



Хотя допускается построить специальный элемент, который не является элементом управления, большинство пользовательских элементов, создаваемых в WPF, будут именно элементами управления. Это значит, что они смогут принимать фокус, а также взаимодействовать с пользовательскими нажатиями клавиш и действиями мыши. По этой причине термины пользовательские элементы и пользовательские элементы управления при разработке WPF-приложений часто являются взаимозаменяемыми.

**Определение маршрутизируемых событий**

Также может понадобиться добавить маршрутизируемые события, которые позволяют уведомлять потребителя элемента управления о том, что что-то произошло. В примере с указателем цвета удобно иметь событие, возбуждаемое при изменении цвета. Хотя это событие может быть определено как обычное событие .NET, применение маршрутизируемого события позволит организовать пузырьковое распространение и туннелирование, так что события смогут обрабатывать более высокоуровневые родители вроде содержащего элемент окна.

Как и в случае свойств зависимости, первый шаг в определении маршрутизируемого события — это создание статического свойства для него, со словом Event, добавленным в конец имени:

public static readonly RoutedEvent ColorChangedEvent;