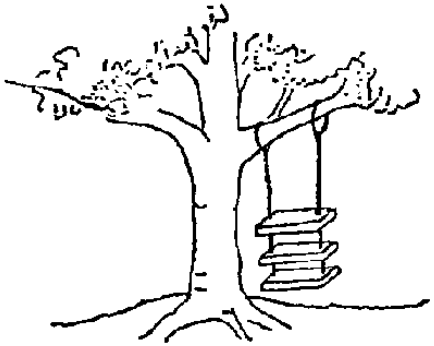
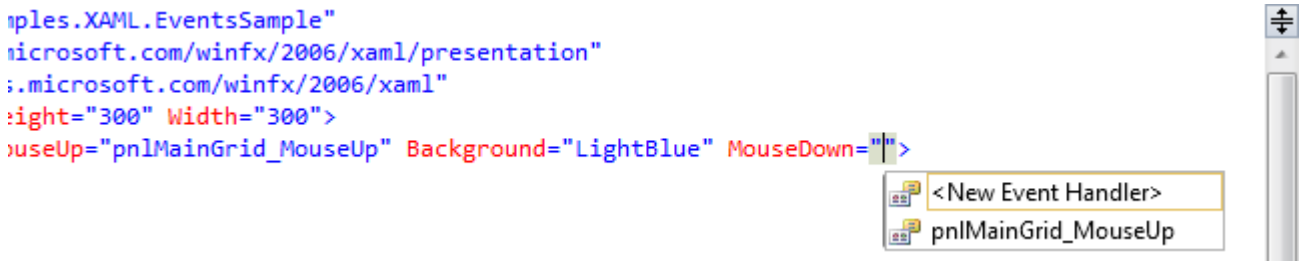
ICommand

Software Entwicklung



Overview

- WPF Basics
  - New Window
  - MessageBox
- WPF Panels
- WPF Data Binding
  - Model View ViewModel (MVVM)
- WPF Controls
  - Events in WPF
  - Control - Button
- WPF ICommand
  - ICommand
  - BaseCommand vs RelayCommand
  - Bind Command to Button – CommandParameter
- WPF Exercises

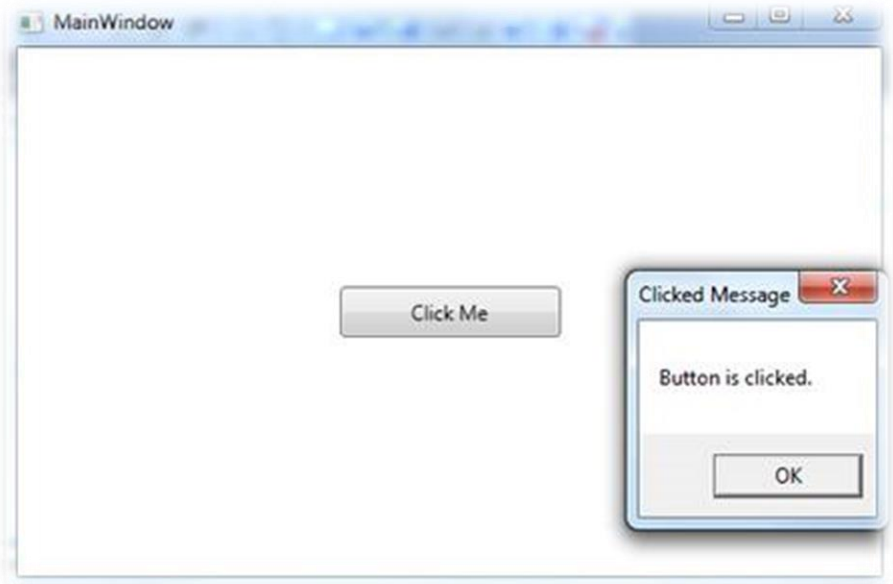


Events

Click, MouseMove, MouseDown, MouseUp …

```
private void pnlMainGrid_MouseUp(object sender, MouseButtonEventArgs e)
{
        MessageBox.Show("You clicked me at " + e.GetPosition(this).ToString());
}
```

https://wpf-tutorial.com/xaml/events-in-xaml/



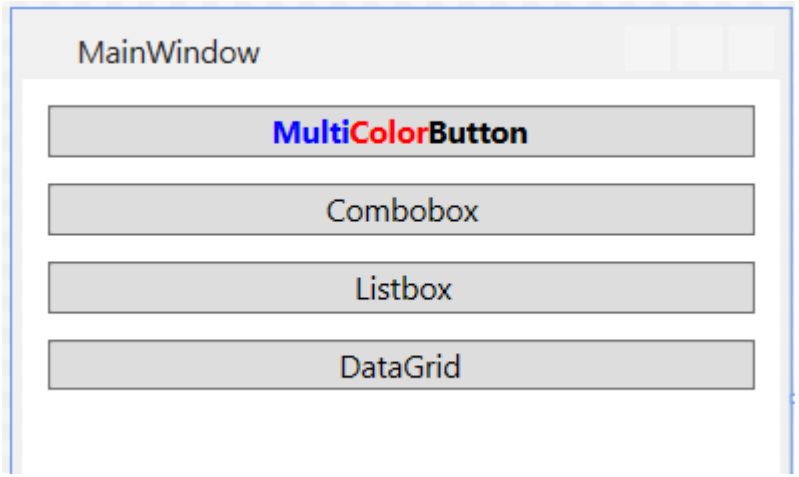
Button

includes all of the functionality required to generate a clickable button

The type inherits from ContentControl, meaning that a button can contain plain text or any other control, including layout controls with their own children.

Button

- Colored Button & Click
- Buttons with Commands

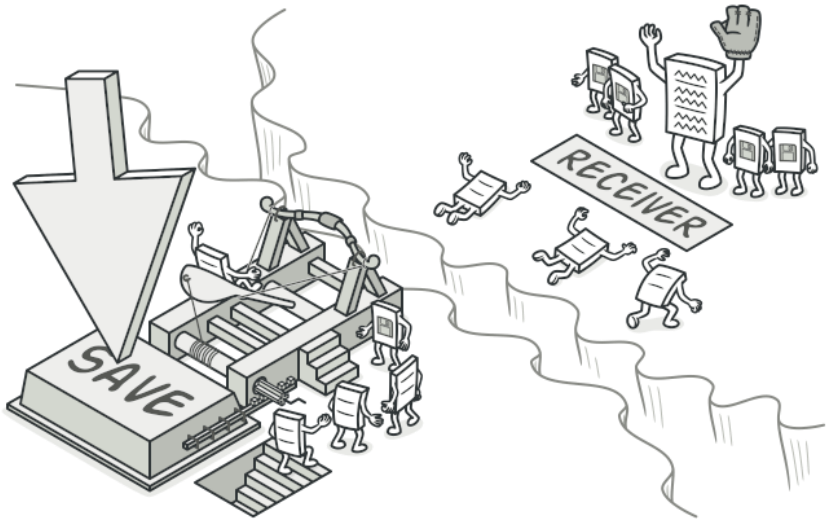



```
<Window.DataContext>
    <local:VM_MainWindowCommands/>
</Window.DataContext>
<StackPanel>
    <!-- Set a method behind a Click-Event in Buttons -->
    <Button Height="20" Margin="10,10,10,5" Click="Button_Click" >
        <Button.FontWeight>Bold</Button.FontWeight>
        <Button.Content>
            <WrapPanel>
                <TextBlock Foreground="Blue">Multi</TextBlock>
                <TextBlock Foreground="Red">Color</TextBlock>
                <TextBlock>Button</TextBlock>
            </WrapPanel>
        </Button.Content>
    </Button>
    <!-- Bind ICommand Properties to Buttons -->
    <Button Height="20" Content="Combobox"  Command="{Binding ShowComboboxCommand}" Margin="10,5,10,5"/>
    <Button Height="20" Content="Listbox"  Command="{Binding ShowListBoxCommand}" Margin="10,5,10,5"/>
    <Button Height="20" Content="DataGrid" Command="{Binding ShowDataGridCommand}" Margin="10,5,10,5"/>
</StackPanel>
```

Button_Click with MessageBox

- Advanced Messagebox

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBoxResult result = MessageBox.Show("Would you like to greet the world with a " +
        "\"Hello, world\"?", "My App", MessageBoxButton.YesNoCancel);

    switch (result)
    {
        case MessageBoxResult.Yes:
            MessageBox.Show("Hello to you too!", "My App");
            break;
        case MessageBoxResult.No:
            MessageBox.Show("Oh well, too bad!", "My App");
            break;
        case MessageBoxResult.Cancel:
            MessageBox.Show("Nevermind then...", "My App");
            break;
    }
}
```

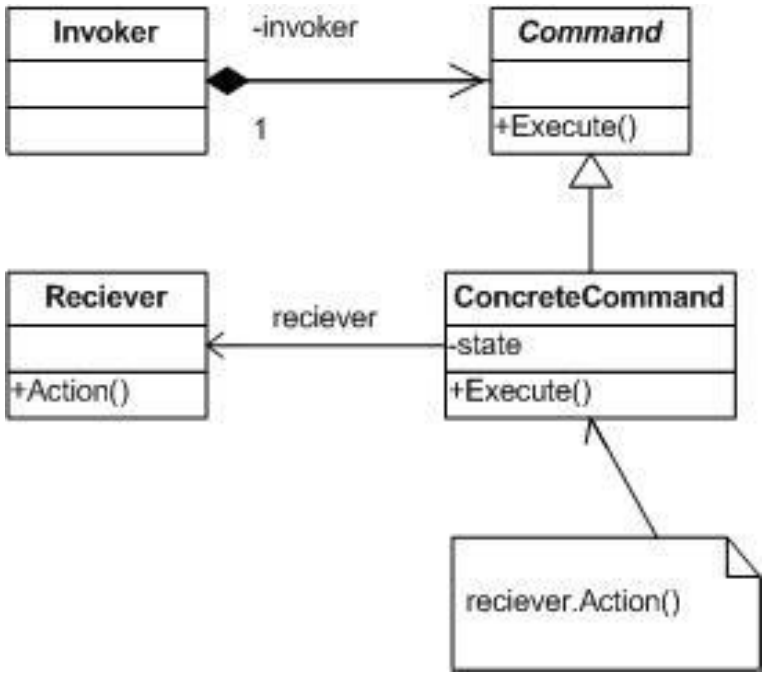


Commands

Not only property can be bound to a Control, Commands (Functions) can be bound to a Button using the ICommand Interface with Execute and CanExecute Methods

Command Pattern

- is a behavioral design pattern
- is used to turn a request into an object which contains all the information about the request



ICommand Interface explained

```
public interface ICommand
{
    //
    // Zusammenfassung:
    //     Tritt ein, wenn Änderungen auftreten, die sich auf die Ausführung des Befehls
    //     auswirken.
    event EventHandler CanExecuteChanged;

    //
    // Zusammenfassung:
    //     Definiert die Methode, die bestimmt, ob der Befehl im aktuellen Zustand ausgeführt
    //     werden kann.
    //
    // Parameter:
    //   parameter:
    //     Vom Befehl verwendete Daten. Wenn der Befehl keine Datenübergabe erfordert, kann
    //     das Objekt auf null festgelegt werden.
    //
    // Rückgabewerte:
    //     true, wenn der Befehl ausgeführt werden kann, andernfalls false.
    bool CanExecute(object parameter);
    //
    // Zusammenfassung:
    //     Definiert die Methode, die aufgerufen wird, wenn der Befehl aufgerufen wird.
    //
    // Parameter:
    //   parameter:
    //     Vom Befehl verwendete Daten. Wenn der Befehl keine Datenübergabe erfordert, kann
    //     das Objekt auf null festgelegt werden.
    void Execute(object parameter);
}
```

Bind Commands

- Set a DataContext to the Window or Control
- Create a Class SpecificCommand, which implements ICommand
- Create a Execute and a CanExcecute Method for the Button

- Bind the Command to the Button

```
<Button Command="{Binding PersonAddCommand}" />
```

AddCommand

- Write an AddPerson Method
- Create a PersonAddCommand

PersonsVM

```
public void AddPerson(Person p)
{
    People.Add(new PersonVM(p));
}
public void RemovePerson(Person p)
{
    People.Add(new PersonVM(p));
}

public ICommand PersonAddCommand
{
    get
    {
        return new PersonAddCommand(this);
    }
}
```

```
class PersonAddCommand : ICommand
{
    private PersonsVM personsVM;
    public PersonAddCommand(PersonsVM personsVM)
        { this.personsVM = personsVM; }

    public event EventHandler CanExecuteChanged;
    public bool CanExecute(object parameter)
    {
        return true;
    }

    public void Execute(object parameter)
    {
        personsVM.AddPerson(new Person());
    }
}
```

```
<Button Command="{Binding PersonAddCommand}" />
```

Genereric Delegates

- Func<>
  - returns a value
- Action<>
  - no return value (void)
- Predicate<>
  - returns a bool

delegate

lamda =>

Func<>

Action<>

predicate<>

delegate

lamda =>

Func<>

Action<>

predicate<>

RelayCommand

ICommand with Generic Delegates

Func<> returns a value

Action<> no return value (void)

Predicate<> returns a bool

RelayCommand - simple

```
public class RelayCommand : ICommand
{
    private Predicate<object> canExecute;
    private Action<object> execute;

    public RelayCommand(Action<object> execute, Predicate<object> canExecute) {
        this.canExecute = canExecute;
        this.execute = execute;
    }
    public bool CanExecute(object parameter)  => canExecute(parameter);
    public void Execute(object parameter)     => execute(parameter);

    public event EventHandler CanExecuteChanged {
        add => CommandManager.RequerySuggested += value;
        remove => CommandManager.RequerySuggested -= value;
    }
}
```

Constructor

- with Action & Predicate

RelayCommand Advanced

```
public class RelayCommand : ICommand
{
    private readonly Action<object> _execute;
    private readonly Predicate<object> _canExecute;
    0 Verweise
    public RelayCommand(Action<object> execute) : this(execute, null) { }
    4 Verweise
    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        _execute = execute ?? throw new ArgumentNullException(nameof(execute));
        _canExecute = canExecute;
    }
    0 Verweise
    public bool CanExecute(object parameter) {
        return _canExecute?.Invoke(parameter) ?? true;
    }
    public event EventHandler CanExecuteChanged {
        add => CommandManager.RequerySuggested += value;
        remove => CommandManager.RequerySuggested -= value;
    }
    0 Verweise
    public void Execute(object parameter) {
        _execute(parameter);
    }
}
```

Verifys if execute != null

Verifys if canExecute != null returns true if it is null

https://csharp.today/c-6-features-null-conditional-and-and-null-coalescing-operators/

CanExecuteChanged

- Update the Button:
  - CanExecute returns a new value
  - Button gets activated or deactivated

  - Keep the EventHandler updated:
    Add to the CommandClass

```
public event EventHandler CanExecuteChanged {
    add { CommandManager.RequerySuggested += value; }
    remove { CommandManager.RequerySuggested -= value; }
}
```

Bind Commands with RelayCommand

- Set a DataContext to the Window or Control
- Use a BaseClass RelayCommand for all ICommand
- Create a Execute and a CanExcecute Method for the Button
- Create a new instance of `RelayCommand(Excecute,CanExecute)` for the Button

ICommand

- Bind a method to a control using the command attribute
  - Works the same way as Content and ItemsSource

    Except

    -> bind it to a *property* that returns an ICommand

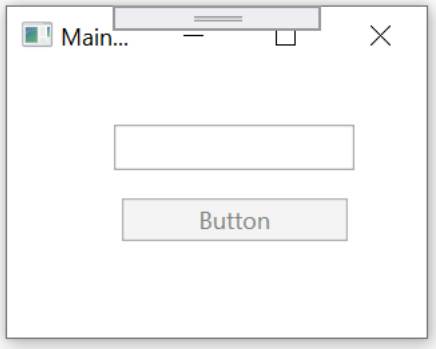
- implement a class 'RelayCommand' that implements ICommand then use it
  - ICommand requires two methods:
    - bool CanExecute
    - void Execute

Using RelayCommand

```
class VM_MainWindowCommands
{
    0 Verweise
    public ICommand ShowComboboxCommand {
        get { return new RelayCommand(o => NewComboboxCommand(), o => true); }
    }
    1-Verweis
    private void NewComboboxCommand() {
        WindowComboBox w = new WindowComboBox();
        w.Show();
    }
    0 Verweise
    public ICommand ShowListBoxCommand {
        get { return new RelayCommand(o => NewListBoxWindow(), o => true); }
    }
    1-Verweis
    private void NewListBoxWindow() {
        WindowListbox w = new WindowListbox();
        w.Show();
    }
    0 Verweise
    public ICommand ShowDataGridCommand {
        get { return new RelayCommand(o => NewDataGridWindow(), o => true); }
    }
    1-Verweis
    private void NewDataGridWindow() {
        WindowDataGrid w = new WindowDataGrid();
        w.Show();
    }
}
```

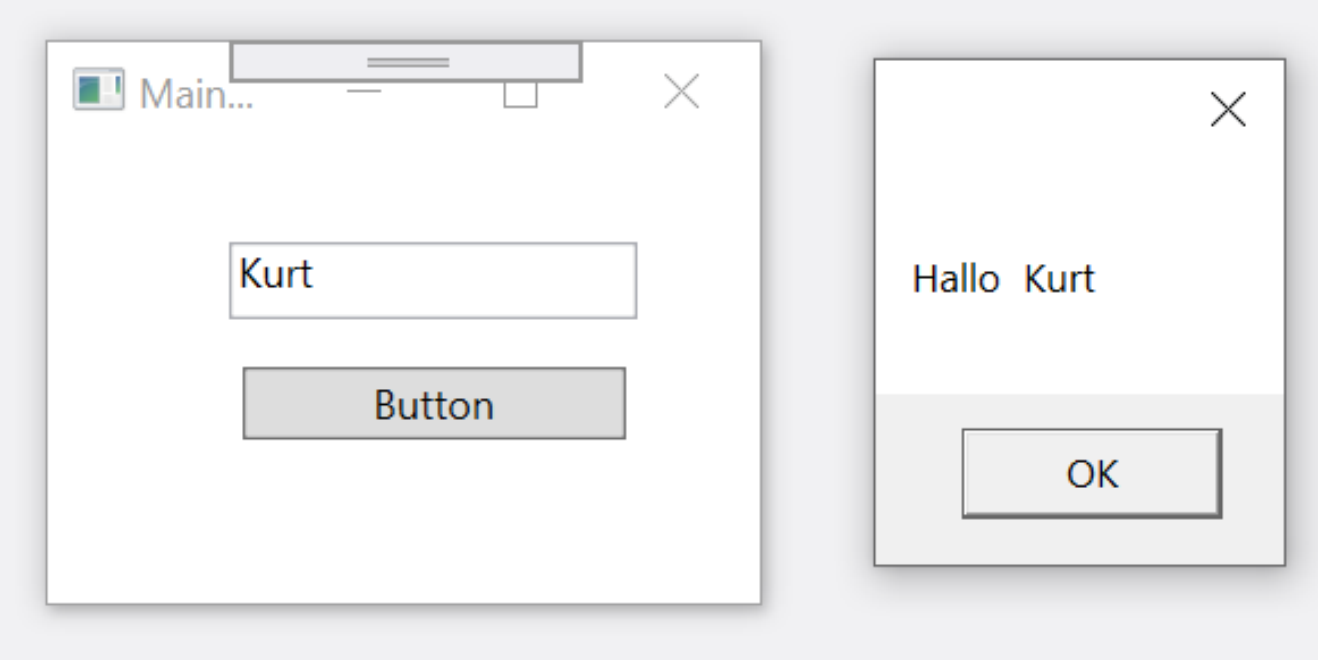
WPF Greet

WPF Command und Data Binding



Greet Example

- Add a Textbox for the Name
- Add a GreetCommand Button to activate a Greeting via Messagebox



RelayCommand to Copy

```
public class RelayCommand : ICommand    {
        private Predicate<object> canExecute;
        private Action<object> execute;
        public event EventHandler CanExecuteChanged {
            add => CommandManager.RequerySuggested += value;
            remove => CommandManager.RequerySuggested -= value;
        }
        public RelayCommand(Action<object> execute, Predicate<object> canExecute){
            this.canExecute = canExecute;
            this.execute = execute;
        }
        public bool CanExecute(object parameter) => canExecute(parameter);
        public void Execute(object parameter)    => execute(parameter);
    }
```

AViewModel to Copy

```
abstract class ANotifyPropertyChanged : INotifyPropertyChanged {
    public event PropertyChangedEventHandler PropertyChanged;

    public void OnPropertyChanged([CallerMemberName] string property = null)
        => PropertyChanged(this, new PropertyChangedEventArgs(property));
}
```

```
//To Copy
 abstract class ANotifyPropertyChanged : INotifyPropertyChanged {
        public event PropertyChangedEventHandler PropertyChanged;
        public void OnPropertyChanged([CallerMemberName] string property = null)
            => PropertyChanged(this, new PropertyChangedEventArgs(property));
    }
```

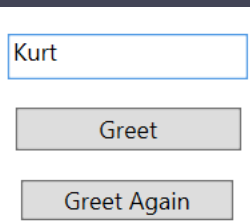
Greet View Model

```
class GreetVM :ANotifyPropertyChanged {

    string name;
    public string Name {
        get { return name; }
        set { name = value;  OnPropertyChanged(); }
    }

    public GreetVM() {
        greetCmd=new RelayCommand(
            //Action for Execute
            o => MessageBox.Show("Hallo  " + Name),
            //Predicate for CanExecute
            x => Name != null && Name != "");
    }

    private ICommand greetCmd;
    public ICommand GreetCmd { get { return greetCmd; } }

}
```

```
public partial class MainWindow : Window {
    public MainWindow() {
        InitializeComponent();
        DataContext = new GreetVM();
    }
}
```

```
<Button Command="{Binding GreetCmd}" Content="Greet"
<TextBox Text="{Binding Name, Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}"
```

Add a ConcreteCommand




```
class ConcreteGreetCommand : ICommand {
    private GreetVM vm;

    public event EventHandler CanExecuteChanged {
        add => CommandManager.RequerySuggested += value;
        remove => CommandManager.RequerySuggested -= value;
    }
    public ConcreteGreetCommand(GreetVM vm) {
        this.vm = vm;
    }

    public bool CanExecute(object parameter) {
        return (vm.Name != null && vm.Name != "");
    }

    public void Execute(object parameter) {
        MessageBox.Show("Hallo  " + vm.Name);
    }
}
```

```
<Button Command="{Binding GreetCmd}" Content="Greet"
<TextBox Text="{Binding Name, Mode=TwoWay,
        UpdateSourceTrigger=PropertyChanged}"
<Button Content="Greet Again" Command="{Binding GreetConcreteCmd}"
```

```
class GreetVM :ANotifyPropertyChanged {

    string name;
    public string Name ...

    public GreetVM() {
        greetCmd=new RelayCommand(
            //Action for Execute
            o => MessageBox.Show("Hallo  " + Name),
            //Predicate for CanExecute
            x => Name != null && Name != "");
        greetConcreteCmd = new ConcreteGreetCommand(this);
    }

    private ICommand greetCmd;
    public ICommand GreetCmd {
        get { return greetCmd; } }

    private ICommand greetConcreteCmd;
    public ICommand GreetConcreteCmd {
        get { return greetConcreteCmd; } }

}
```

Summary

Events:

https://www.wpf-tutorial.com/xaml/events-in-xaml/

Commands:

https://www.wpf-tutorial.com/commands/introduction/#aelm660