元に戻す操作の実装

2015 年 8 月 7 日

@twyujiro15

プロフィール

加藤 裕次郎

1982.03.03 生まれ（うお座）

左利き（お箸は右）

twitter : @twyujiro15

プログラミング経験

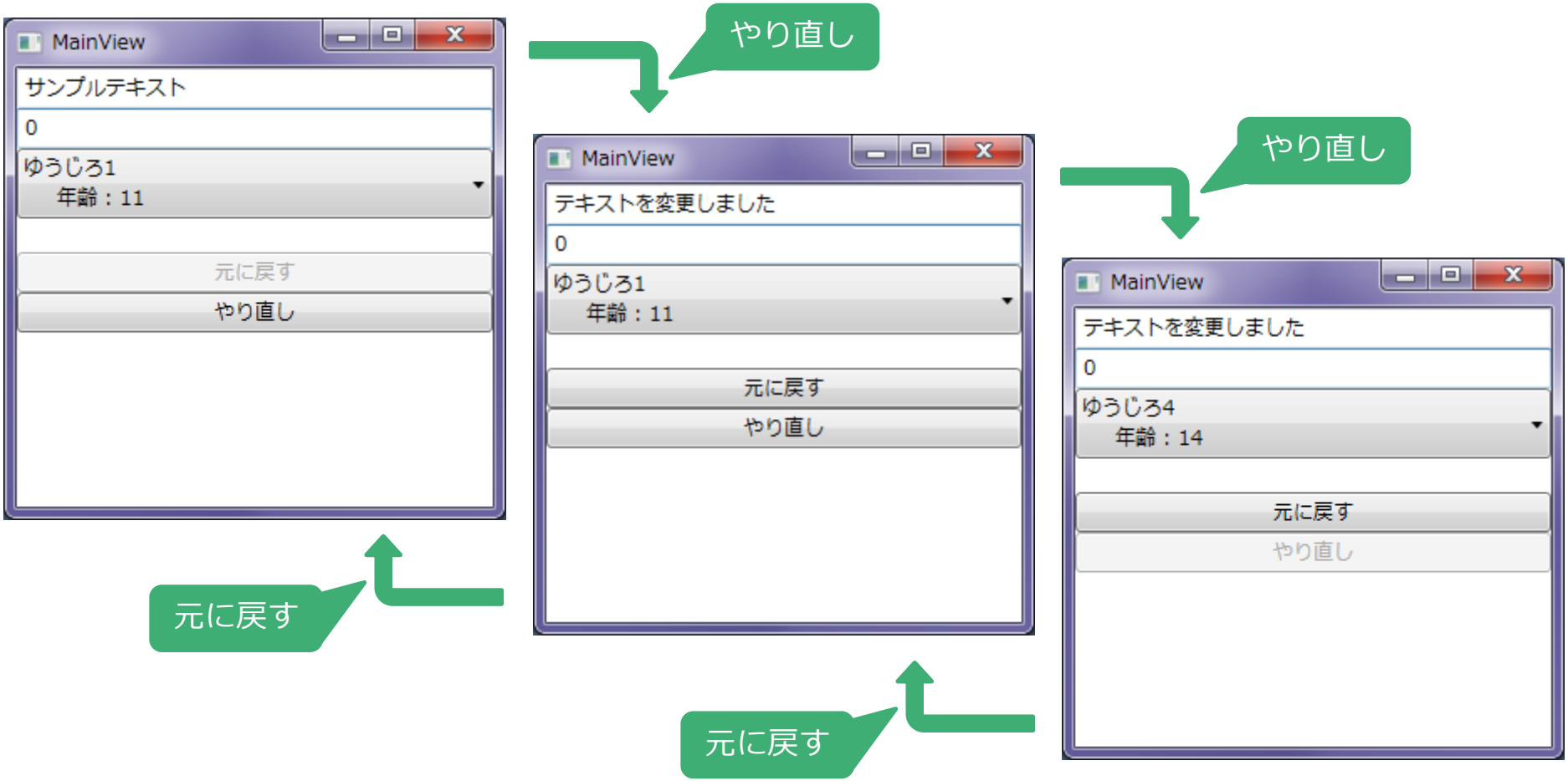
Excel VBA
MATLAB、MATX
C
VC++ (Windows SDK)
VC++ (MFC)
WPF + C#

本職は製造業の開発業務

- 2009 年 4 月に入社
  - ・組み込みソフトウェア開発で初めてまともに C 言語
  - ・デバッグソフトで Visual C++（MFC、Windows SDK）
  - ・Excel 大好きマンだったので VBA も使用
- 2013 年 10 月
  - ・あるサンプルが "WPF" なるものでできていることを知る
- 2015 年 8 月
  - ・これまでに1 個のライブラリと 20 個以上のソフトを作成

今回のゴール

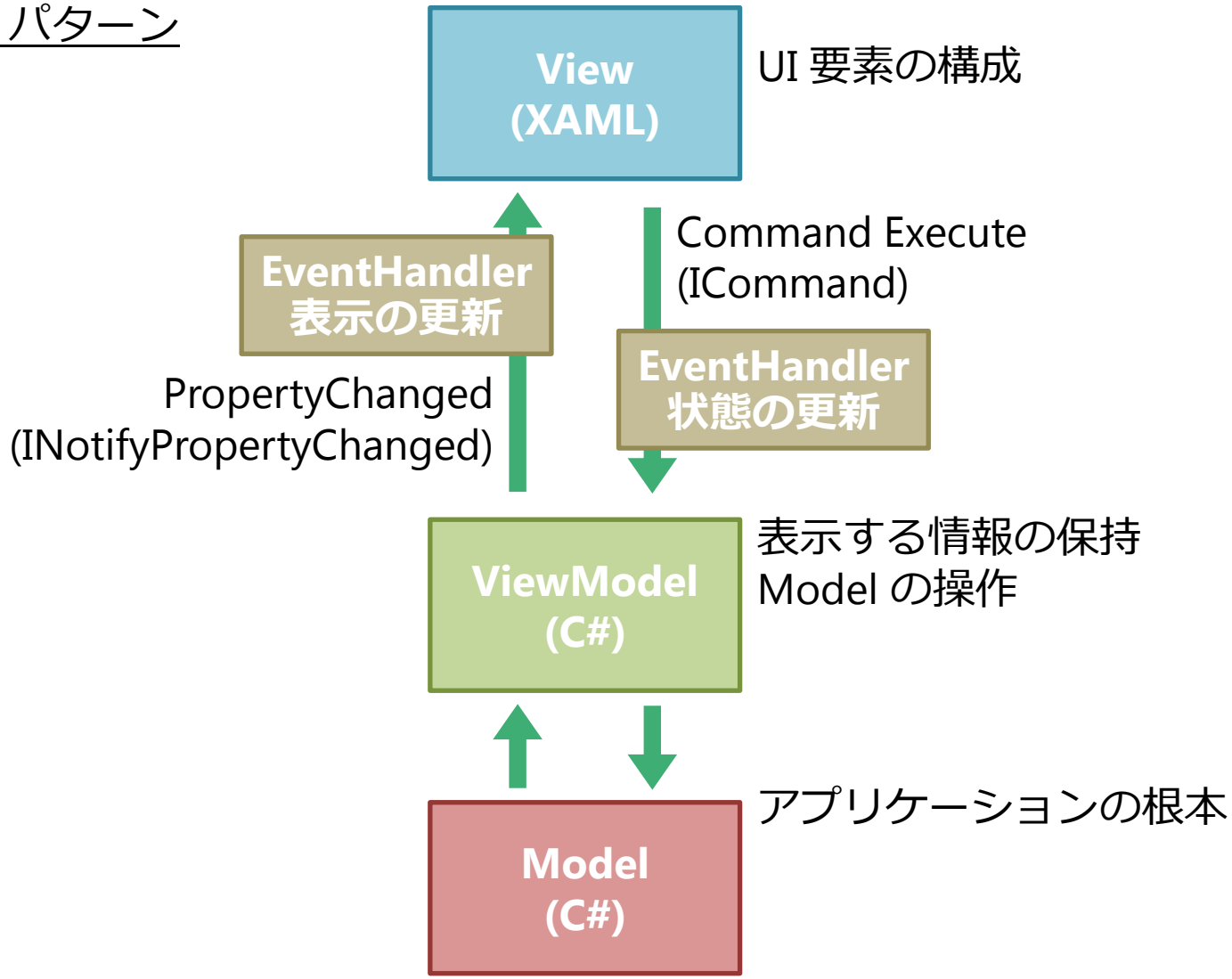
元に戻す/やり直し機能を実装する



・プロパティの変更を元に戻す / やり直す

せっかくなので基本から順に説明します

MVVM パターン



ICommand の実装 (1/2)

```
using System.Windows.Input;

public class DelegateCommand : ICommand
{
    /// <summary>
    /// 新しいインスタンスを生成します。
    /// </summary>
    /// <param name="execute">コマンドの処理内容を指定します。</param>
    public DelegateCommand(Action<object> execute)
        : this(execute, null)
    {
    }

    /// <summary>
    /// 新しいインスタンスを生成します。
    /// </summary>
    /// <param name="execute">コマンドの処理内容を指定します。</param>
    /// <param name="canExecute">コマンド実行可能判別の処理内容を指定します。</param>
    public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
    {
        this._execute = execute;
        this._canExecute = canExecute;
    }

    /// <summary>
    /// コマンドの処理内容を保持します。
    /// </summary>
    private Action<object> _execute;

    /// <summary>
    /// コマンド実行可能判別の処理内容を保持します。
    /// </summary>
    private Func<object, bool> _canExecute;
```

ICommand の実装 (2/2)

```
    #region ICommand のメンバ
    /// <summary>
    /// コマンド実行可能判別をおこないます。
    /// </summary>
    /// <param name="parameter">コマンドパラメータを指定します。</param>
    /// <returns>コマンドが実行可能な場合に true を返します。</returns>
    public bool CanExecute(object parameter)
    {
        return this._canExecute != null ? this._canExecute(parameter) : true;
    }

    /// <summary>
    /// コマンド実行可能判別に変更があった場合に発生します。
    /// </summary>
    public event System.EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    /// <summary>
    /// コマンドを実行します。
    /// </summary>
    /// <param name="parameter">コマンドパラメータを指定します。</param>
    public void Execute(object parameter)
    {
        if (this._execute != null)
            this._execute(parameter);
    }
    #endregion ICommand のメンバ
}
```

IPropertyChanged の実装 (1/2)

```
using System.ComponentModel;
using System.Runtime.CompilerServices;

/// <summary>
/// <c>System.ComponentModel.INotifyPropertyChanged</c> インターフェースを実装した抽象クラスです。
/// </summary>
public abstract class NotificationObject : INotifyPropertyChanged
{
    #region INotifyPropertyChanged のメンバ
    /// <summary>
    /// プロパティ値に変更があった場合に発生します。
    /// </summary>
    public event PropertyChangedEventHandler PropertyChanged;
    #endregion INotifyPropertyChanged のメンバ

    /// <summary>
    /// プロパティ値変更通知イベントを発行します。
    /// </summary>
    /// <param name="propertyName">プロパティ名を指定します。</param>
    protected virtual void RaisePropertyChanged([CallerMemberName]string propertyName = null)
    {
        var h = this.PropertyChanged;
        if (h != null) h(this, new PropertyChangedEventArgs(propertyName));
    }
```

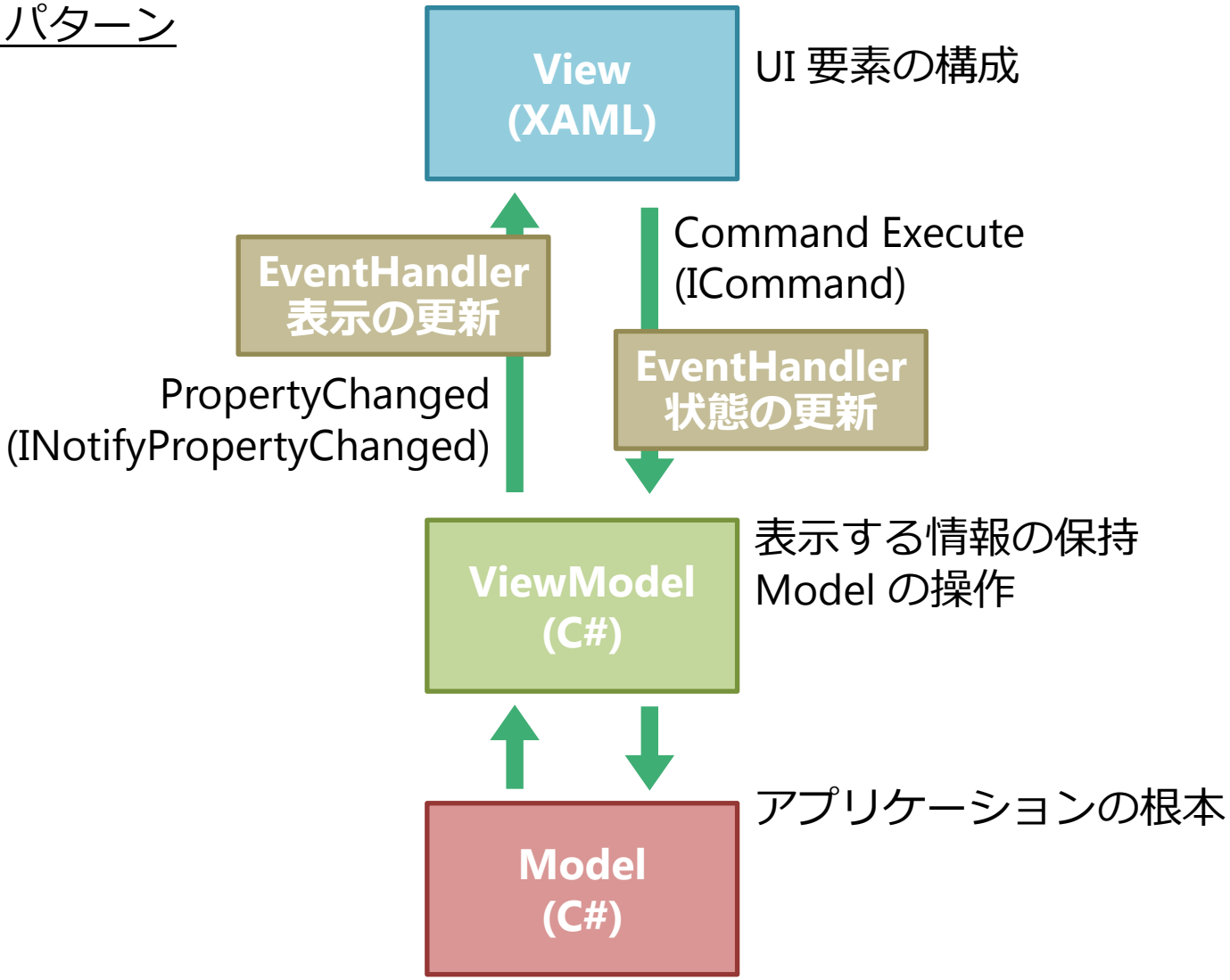
IPropertyChanged の実装 (2/2)

```
    /// <summary>
    /// プロパティ値変更のためのヘルパです。
    /// </summary>
    /// <typeparam name="T">プロパティの型</typeparam>
    /// <param name="target">プロパティの実体を指定します。</param>
    /// <param name="value">変更後の値を指定します。</param>
    /// <param name="propertyName">プロパティ名を指定します。</param>
    /// <returns>プロパティ値に変更があった場合に true を返します。</returns>
    protected virtual bool SetProperty<T>(ref T target, T value, [CallerMemberName]string propertyName = null)
    {
        if (!object.Equals(target, value))
        {
            System.Diagnostics.Debug.WriteLine(value.ToString() + " に変更されました。");
            target = value;
            RaisePropertyChanged(propertyName);
            return true;
        }
        return false;
    }
}
```

変更があると出力ウィンドウに表示される

とりあえず基本を実装しました

MVVM パターン



こんなモデルを用意して

Model (C#)

```
using System.Collections.Generic;

/// <summary>
/// 人物データリストを表します。
/// </summary>
public class People : List<Person>
{
}
```

```
/// <summary>
/// 人物データを表します。
/// </summary>
public class Person
{
    /// <summary>
    /// 新しいインスタンスを生成します。
    /// </summary>
    /// <param name="name">名前を指定します。</param>
    /// <param name="age">年齢を指定します。</param>
    public Person(string name, int age)
    {
        this.Name = name;
        this.Age = age;
    }

    /// <summary>
    /// 名前を取得または設定します。
    /// </summary>
    public string Name { get; set; }

    /// <summary>
    /// 年齢を取得または設定します。
    /// </summary>
    public int Age { get; set; }

    /// <summary>
    /// 人物データを文字列として表現します。
    /// </summary>
    /// <returns>文字列を返します。</returns>
    public override string ToString()
    {
        return string.Format("{0} ({1}才)", this.Name, this.Age);
    }
}
```

表示する情報を決める (1/2)

ViewModel (C#)

```
using UndoRedoApp.Models;

/// <summary>
/// MainView に対する ViewModel クラスです。
/// </summary>
public class MainViewModel : NotificationObject
{
    /// <summary>
    /// 新しいインスタンスを生成します。
    /// </summary>
    public MainViewModel()
    {
        this._people = new People()
        {
            new Person("ゆうじろ1", 11),
            new Person("ゆうじろ2", 12),
            new Person("ゆうじろ3", 13),
            new Person("ゆうじろ4", 14),
            new Person("ゆうじろ5", 15),
        };
        this._selectedPerson = People[0];
    }
```

```
    private People _people;
    /// <summary>
    /// 人物データリストを取得します。
    /// </summary>
    public People People
    {
        get { return _people; }
    }

    private Person _selectedPerson;
    /// <summary>
    /// 選択された人物データを取得または設定します。
    /// </summary>
    public Person SelectedPerson
    {
        get { return _selectedPerson; }
        set { SetProperty(ref _selectedPerson, value); }
    }
```

表示する情報を決める (2/2)

ViewModel (C#)

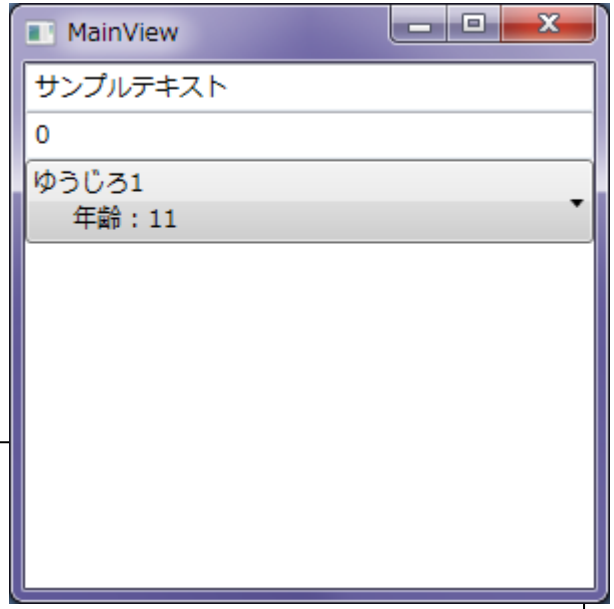
```
    private string _text = "サンプルテキスト";
    /// <summary>
    /// string 型の値を取得または設定します。
    /// </summary>
    public string Text
    {
        get { return _text; }
        set { SetProperty(ref _text, value); }
    }

    private int _value;
    /// <summary>
    /// int 型の値を取得または設定します。
    /// </summary>
    public int Value
    {
        get { return _value; }
        set { SetProperty(ref _value, value); }
    }
}
```

公開プロパティは

- Text プロパティ
- Value プロパティ
- People プロパティ
- SelectedPerson プロパティ

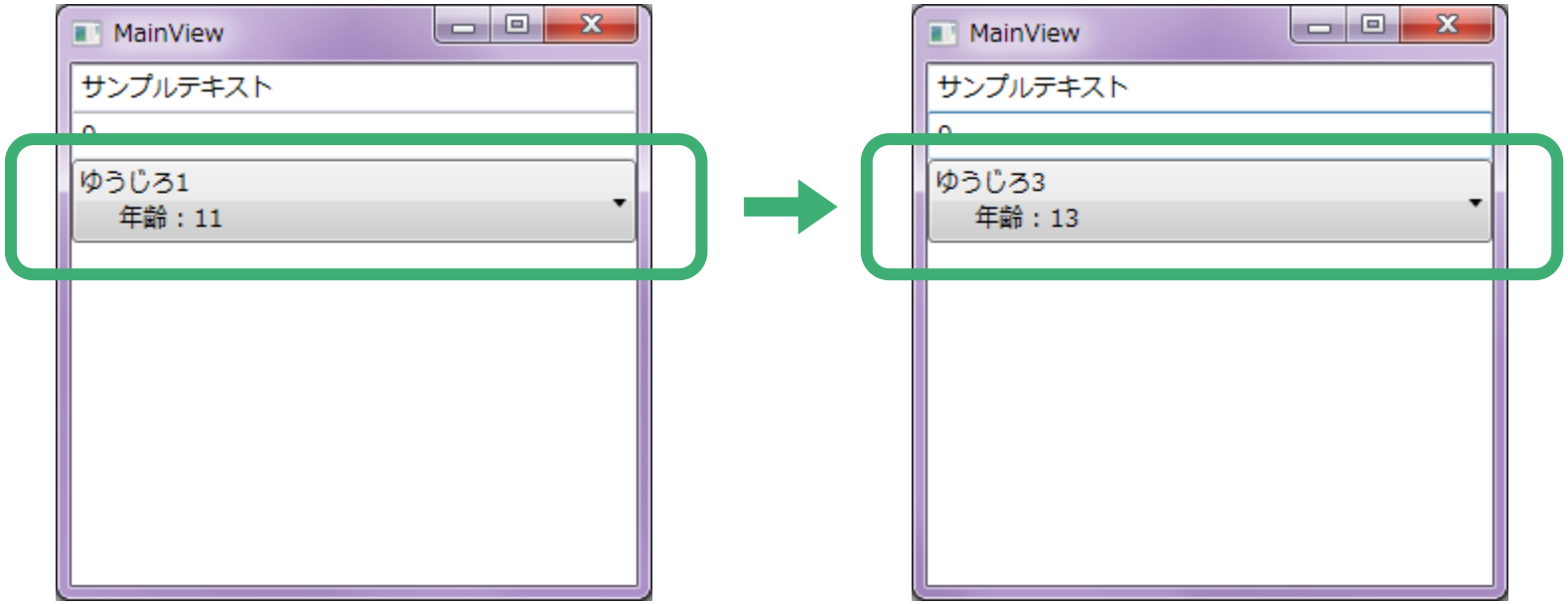
公開された情報を表示する

View (XAML)

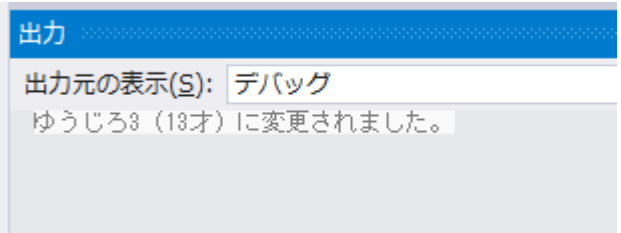



```xml
<Window x:Class="UndoRedoApp.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="300">
    <StackPanel>
        <TextBox Text="{Binding Text}" />
        <TextBox Text="{Binding Value}" />
        <ComboBox ItemsSource="{Binding People}" SelectedItem="{Binding SelectedPerson}">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel>
                        <TextBlock Text="{Binding Name}" />
                        <TextBlock Text="{Binding Age, StringFormat='{}年齢 : {0}'}" Margin="20,0,0,0" />
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
</StackPanel>
</Window>
```

ちょっと動かしてみましょう

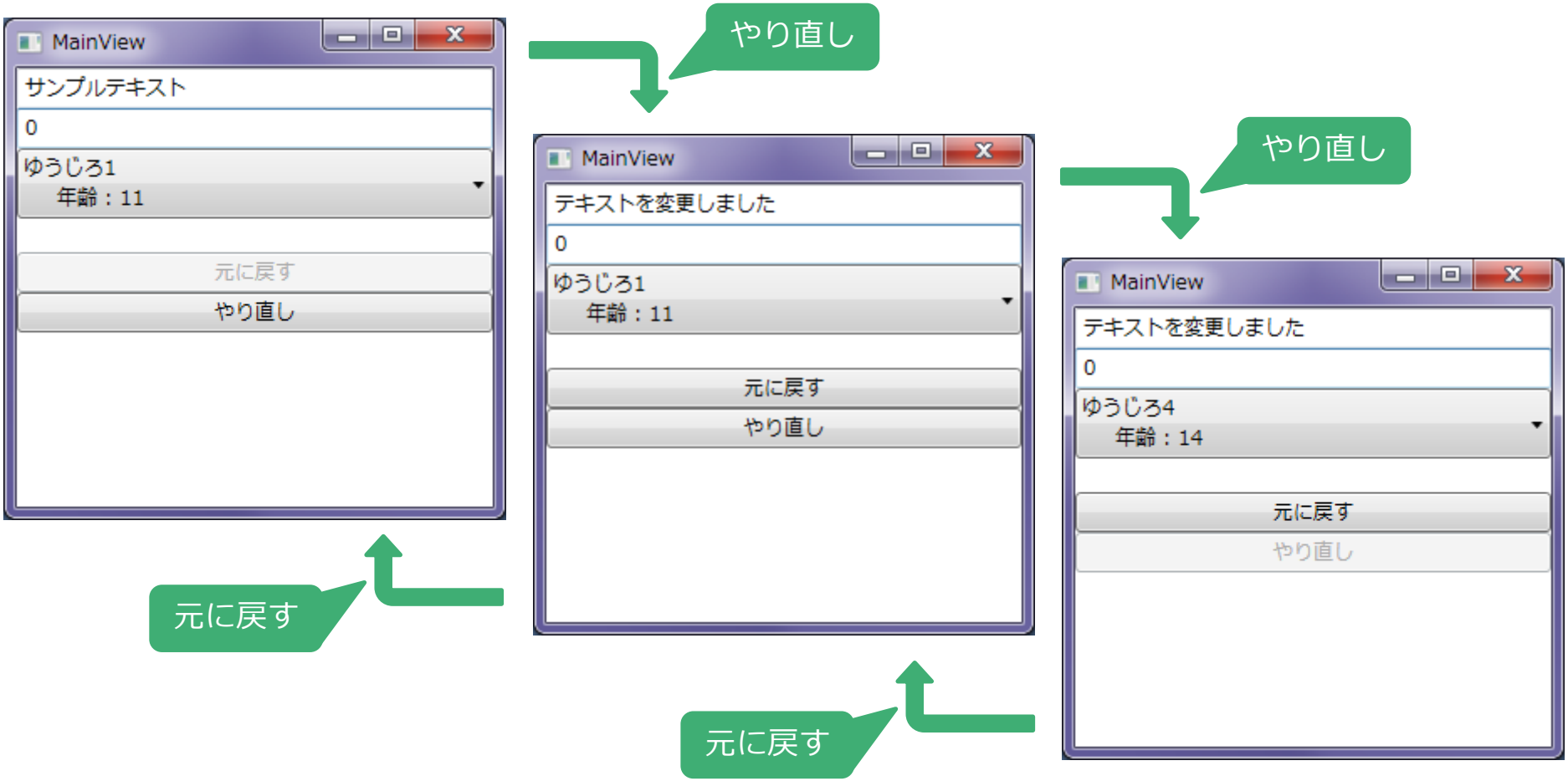


出力ウィンドウにも表示されている



前置きが長くなりましたが

今回のゴール



・プロパティの変更を元に戻す / やり直す

まずはプロパティ変更操作履歴用クラス (1/2)

```
using System;

/// <summary>
/// 特定のアクションを保持し、任意のタイミングで実行するためのクラスです。
/// </summary>
internal class SetPropertyHistory
{
    /// <summary>
    /// アンドゥアクションを保持します。
    /// </summary>
    private Action _undoAction;

    /// <summary>
    /// リドゥアクションを保持します。
    /// </summary>
    private Action _redoAction;

    /// <summary>
    /// 保持しているアンドゥアクションを実行します。
    /// </summary>
    public void UnDo()
    {
        this._undoAction();
    }

    /// <summary>
    /// 保持しているリドゥアクションを実行します。
    /// </summary>
    public void ReDo()
    {
        this._redoAction();
    }
```

まずはプロパティ変更操作履歴用クラス (2/2)

```
    /// <summary>
    /// 新しいインスタンスを生成します。
    /// </summary>
    /// <param name="undoAction">アンドゥアクションを指定します。</param>
    /// <param name="undoAction">リドゥアクションを指定します。</param>
    public SetPropertyHistory(Action undoAction, Action redoAction)
    {
        if (undoAction == null)
            throw new ArgumentNullException("必ずアンドゥアクションを指定してください。");
        if (redoAction == null)
            throw new ArgumentNullException("必ずリドゥアクションを指定してください。");
        this._undoAction = undoAction;
        this._redoAction = redoAction;
    }
}
```

NotificationObject 派生クラス (1/3)

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

/// <summary>
/// アンドゥ機能を備えた <c>NotificationObject</c> クラス派生のクラスです。
/// </summary>
public abstract class UndoRedoNotificationObject : NotificationObject
{
private Stack<SetPropertyHistory> _undoStack;
    /// <summary>
    /// アンドゥする操作を溜めておくスタックを取得します。
    /// </summary>
    private Stack<SetPropertyHistory> UndoStack
    {
        get { return _undoStack ?? (_undoStack = new Stack<SetPropertyHistory>()); }
    }

    private Stack<SetPropertyHistory> _redoStack;
    /// <summary>
    /// リドゥする操作を溜めておくスタックを取得します。
    /// </summary>
    private Stack<SetPropertyHistory> RedoStack
    {
        get { return _redoStack ?? (_redoStack = new Stack<SetPropertyHistory>()); }
    }
```

NotificationObject 派生クラス (2/3)

```
private DelegateCommand _undoCommand;
/// <summary>
/// 元に戻すコマンドを取得します。
/// </summary>
public DelegateCommand UndoCommand
{
    get
    {
        return _undoCommand ?? (_undoCommand = new DelegateCommand(
        _ =>
        {
            var action = this.UndoStack.Pop();
            action.UnDo();
            this.RedoStack.Push(action);
        },
        _ => this.UndoStack.Count > 0));
    }
}

private DelegateCommand _redoCommand;
/// <summary>
/// やり直しコマンドを取得します。
/// </summary>
public DelegateCommand RedoCommand
{
    get
    {
        return _redoCommand ?? (_redoCommand = new DelegateCommand(
        _ =>
        {
            var action = this.RedoStack.Pop();
            action.ReDo();
            this.UndoStack.Push(action);
        },
        _ => this.RedoStack.Count > 0));
    }
}
```

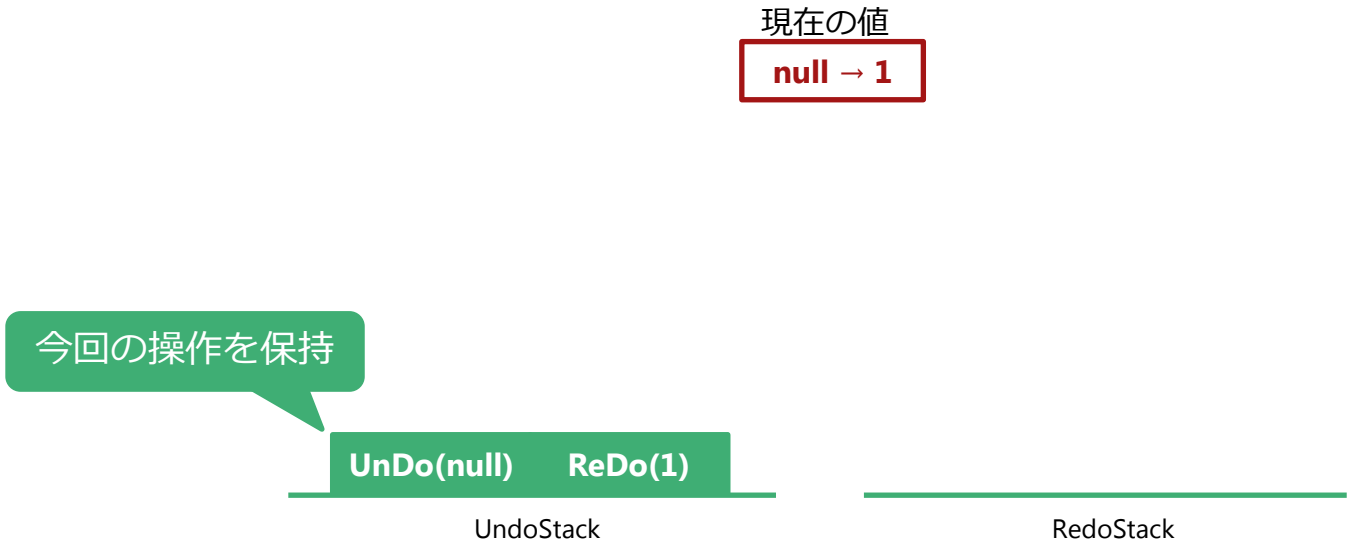
NotificationObject 派生クラス (3/3)

```
    /// <summary>
    /// プロパティ値変更のためのヘルパです。
    /// </summary>
    /// <typeparam name="T">プロパティの型</typeparam>
    /// <param name="target">プロパティの実体を指定します。</param>
    /// <param name="value">変更後の値を指定します。</param>
    /// <param name="action">変更前の値を引数として、変更を元に戻すためのアクションを指定します。</param>
    /// <param name="propertyName">プロパティ名を指定します。</param>
    /// <returns>プロパティ値に変更があった場合に true を返します。</returns>
    protected virtual bool SetProperty<T>(ref T target, T value, Action<T> action, [CallerMemberName]string propertyName = null)
    {
        T oldValue = target;
        bool ret = SetProperty(ref target, value, propertyName);
        if (ret)
        {
            this.UndoStack.Push(new SetPropertyHistory(
            () =>
            {
                // 元に戻す操作
                action(oldValue);
                RaisePropertyChanged(propertyName);
            },
            () =>
            {
                // やり直す操作
                action(value);
                RaisePropertyChanged(propertyName);
            }));
            this.RedoStack.Clear();
        }
        return ret;
    }
}
```

操作履歴のスタックのイメージ (1/8)

例えばこんな操作をしたら…

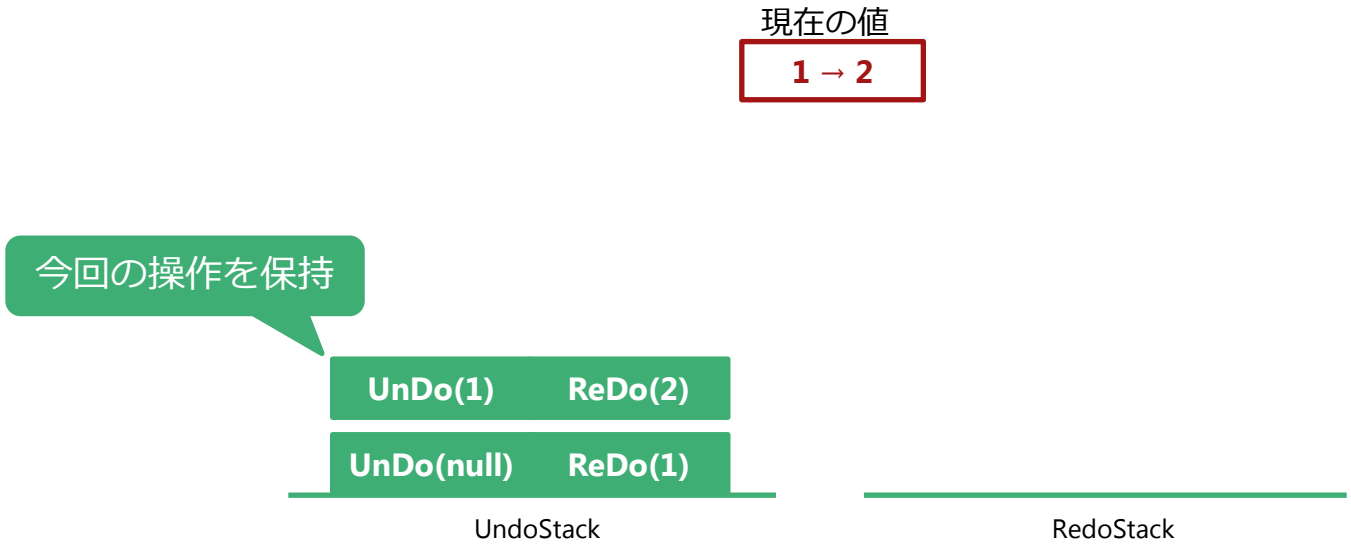
1. null から "1" に変更



操作履歴のスタックのイメージ (2/8)

例えばこんな操作をしたら…

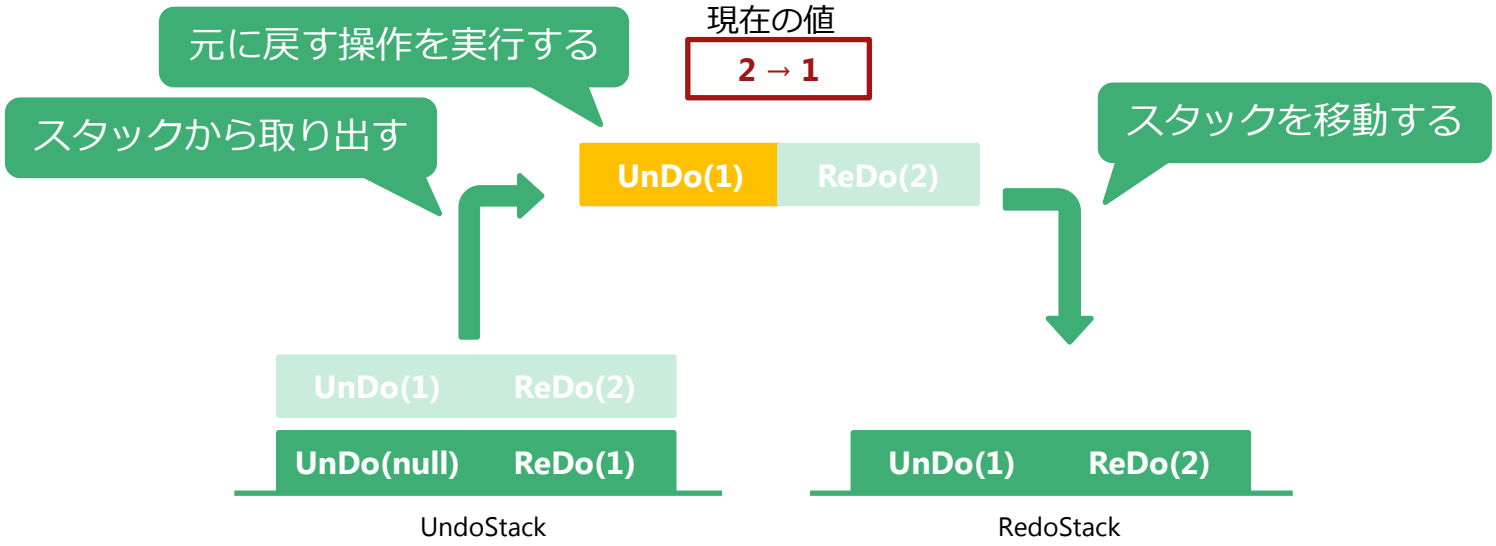
1. null から "1" に変更
2. "1" から "2" に変更



操作履歴のスタックのイメージ (3/8)

例えばこんな操作をしたら...

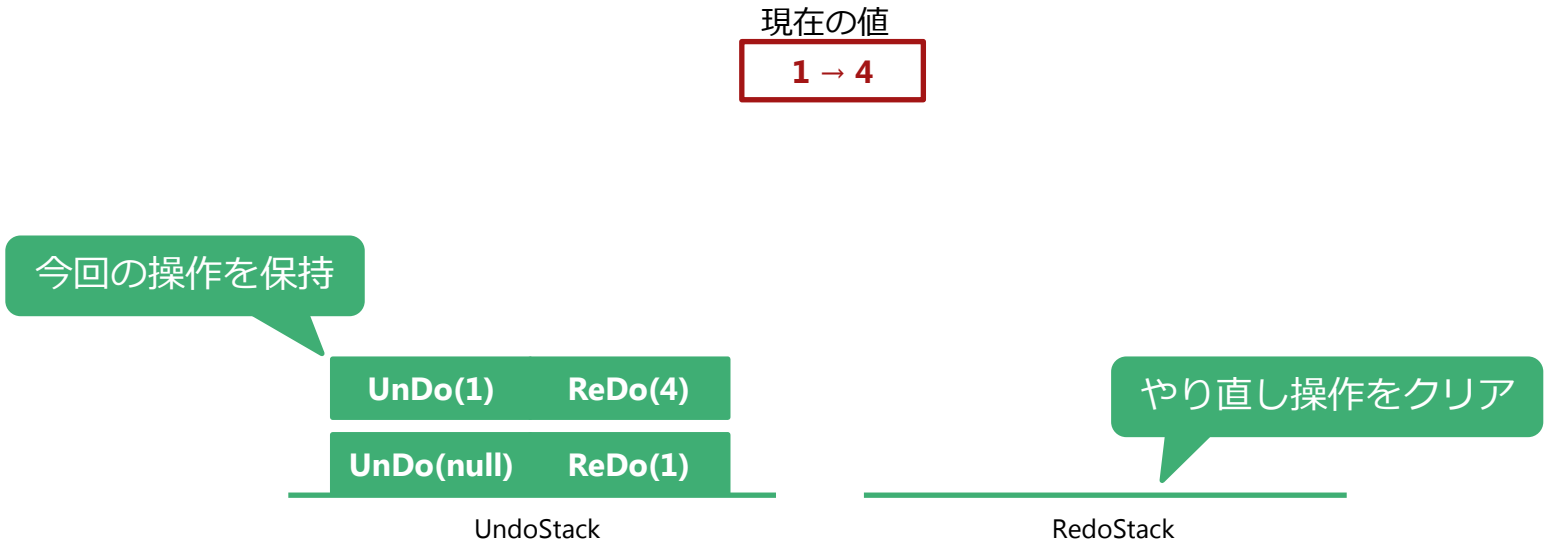
1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）



操作履歴のスタックのイメージ (4/8)

例えばこんな操作をしたら…

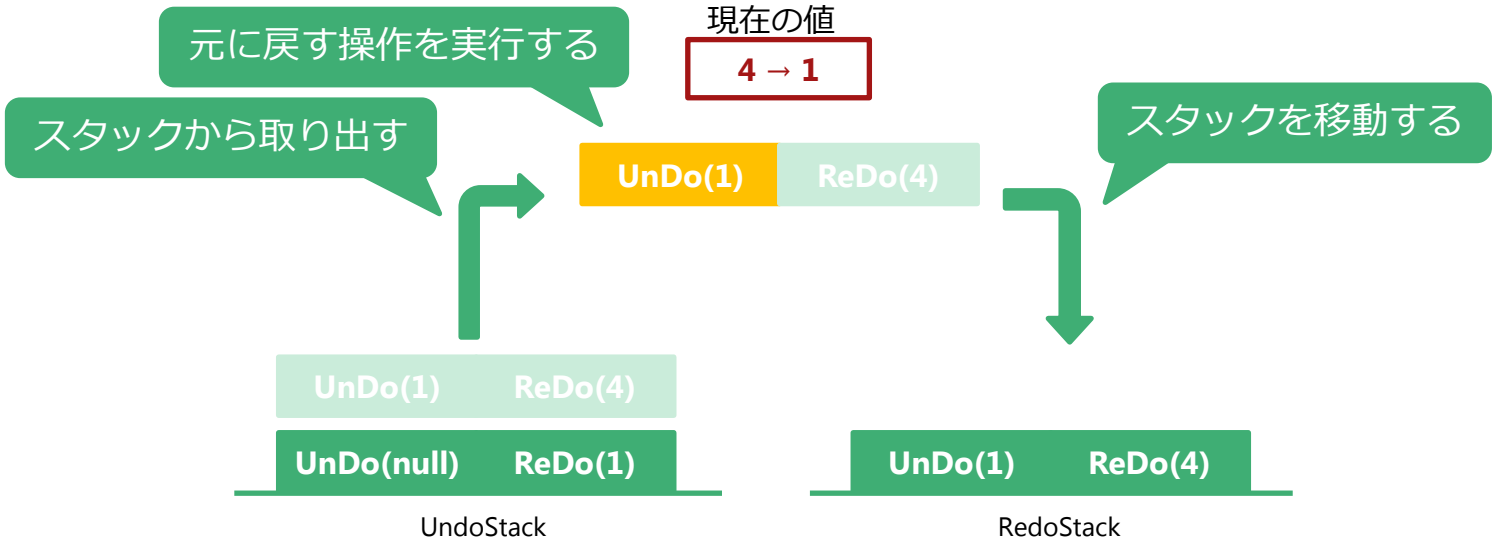
1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）
4. "1" から "4" に変更



操作履歴のスタックのイメージ (5/8)

例えばこんな操作をしたら...

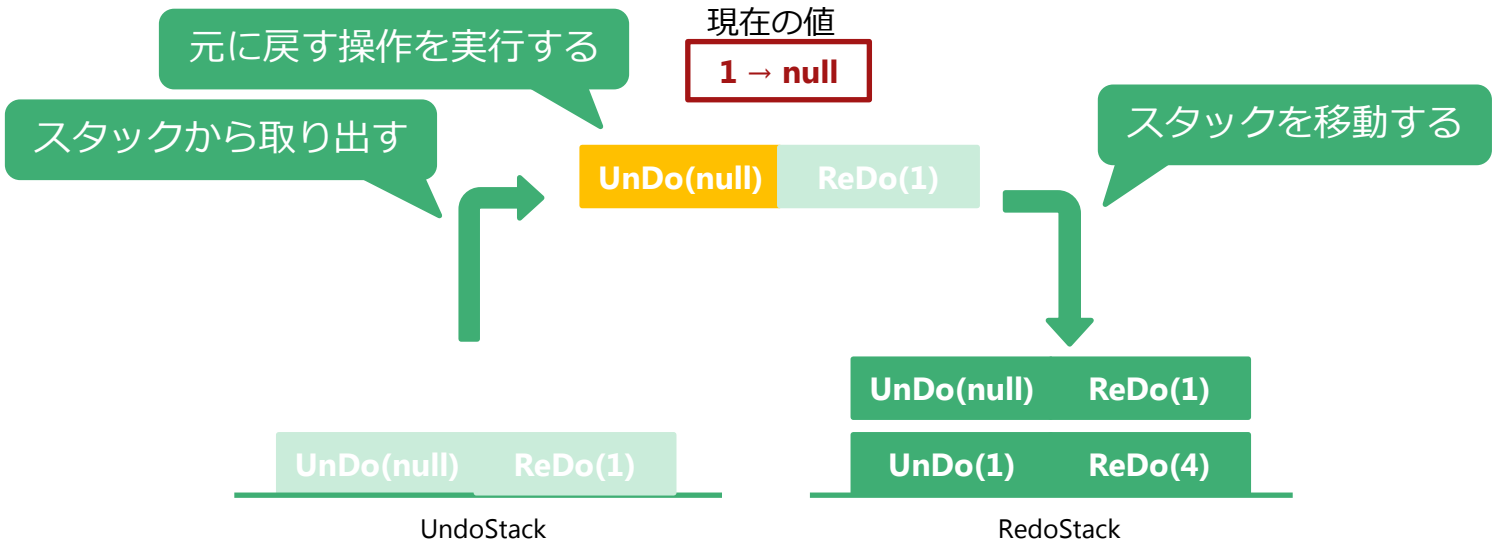
1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）
4. "1" から "4" に変更
5. 元に戻す（"4" から "1" に戻る）



操作履歴のスタックのイメージ (6/8)

例えばこんな操作をしたら…

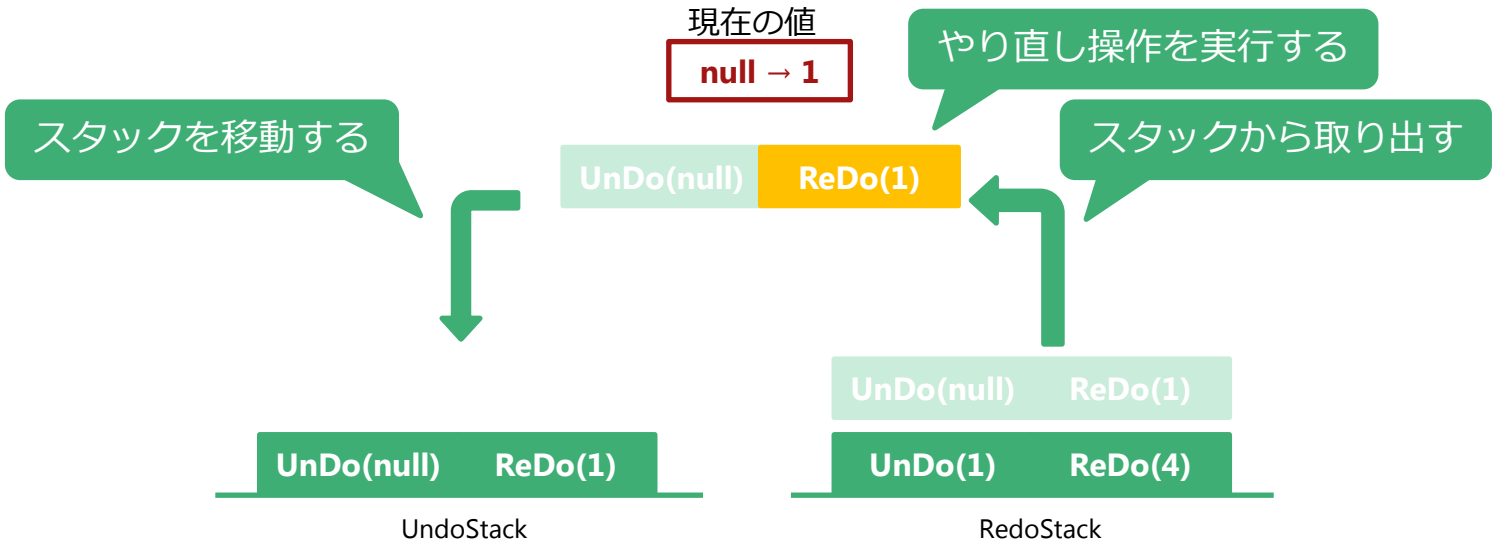
1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）
4. "1" から "4" に変更
5. 元に戻す（"4" から "1" に戻る）
6. 元に戻す（"1" から null に戻る）



操作履歴のスタックのイメージ (7/8)

例えばこんな操作をしたら…

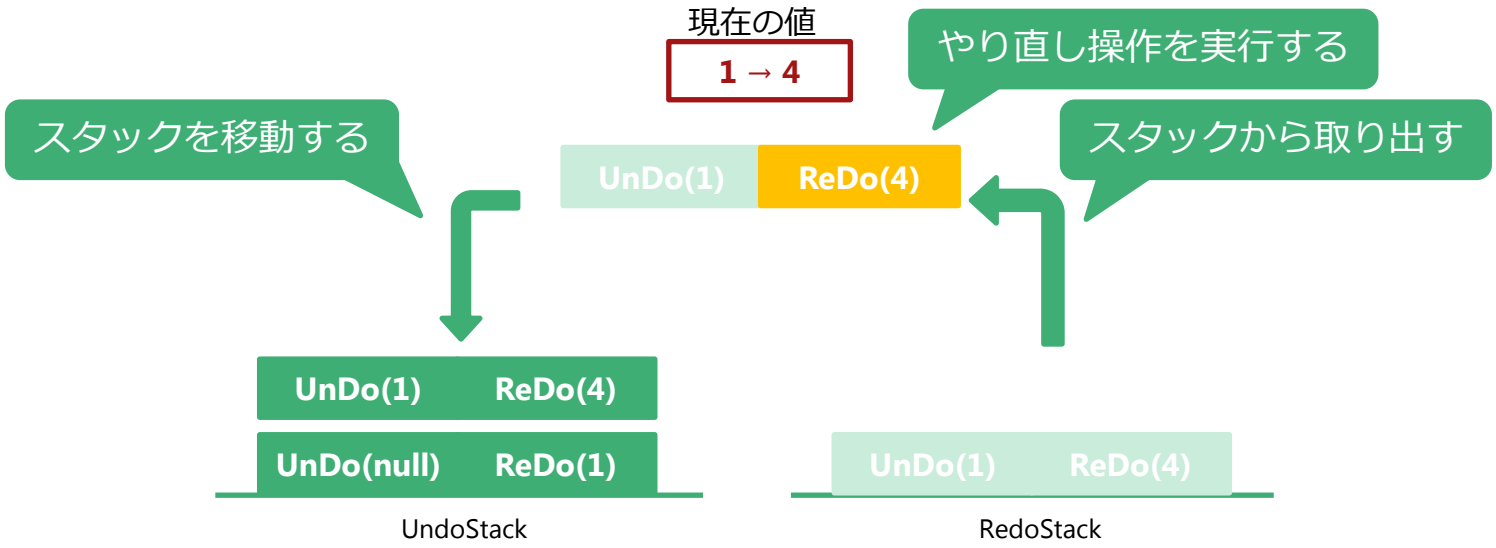
1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）
4. "1" から "4" に変更
5. 元に戻す（"4" から "1" に戻る）
6. 元に戻す（"1" から null に戻る）
7. やり直す（null から "1" に変更）



操作履歴のスタックのイメージ (8/8)

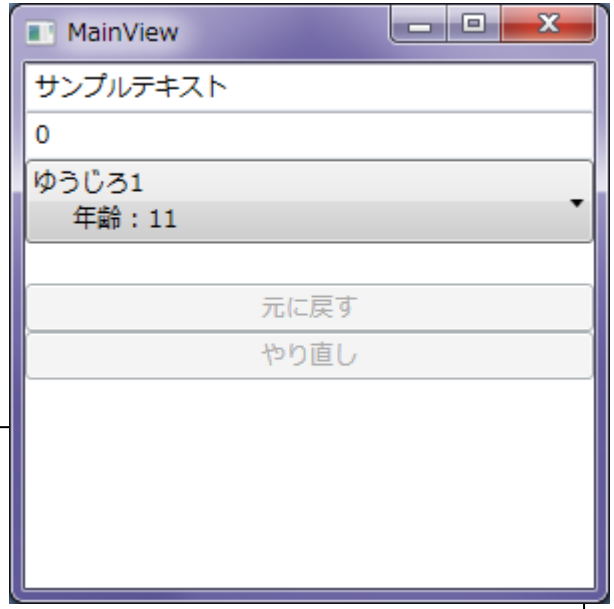
例えばこんな操作をしたら…

1. null から "1" に変更
2. "1" から "2" に変更
3. 元に戻す（"2" から "1" に戻る）
4. "1" から "4" に変更
5. 元に戻す（"4" から "1" に戻る）
6. 元に戻す（"1" から null に戻る）
7. やり直す（null から "1" に変更）
8. やり直す（"1" から "4" に変更）



元に戻す / やり直しボタンを追加する

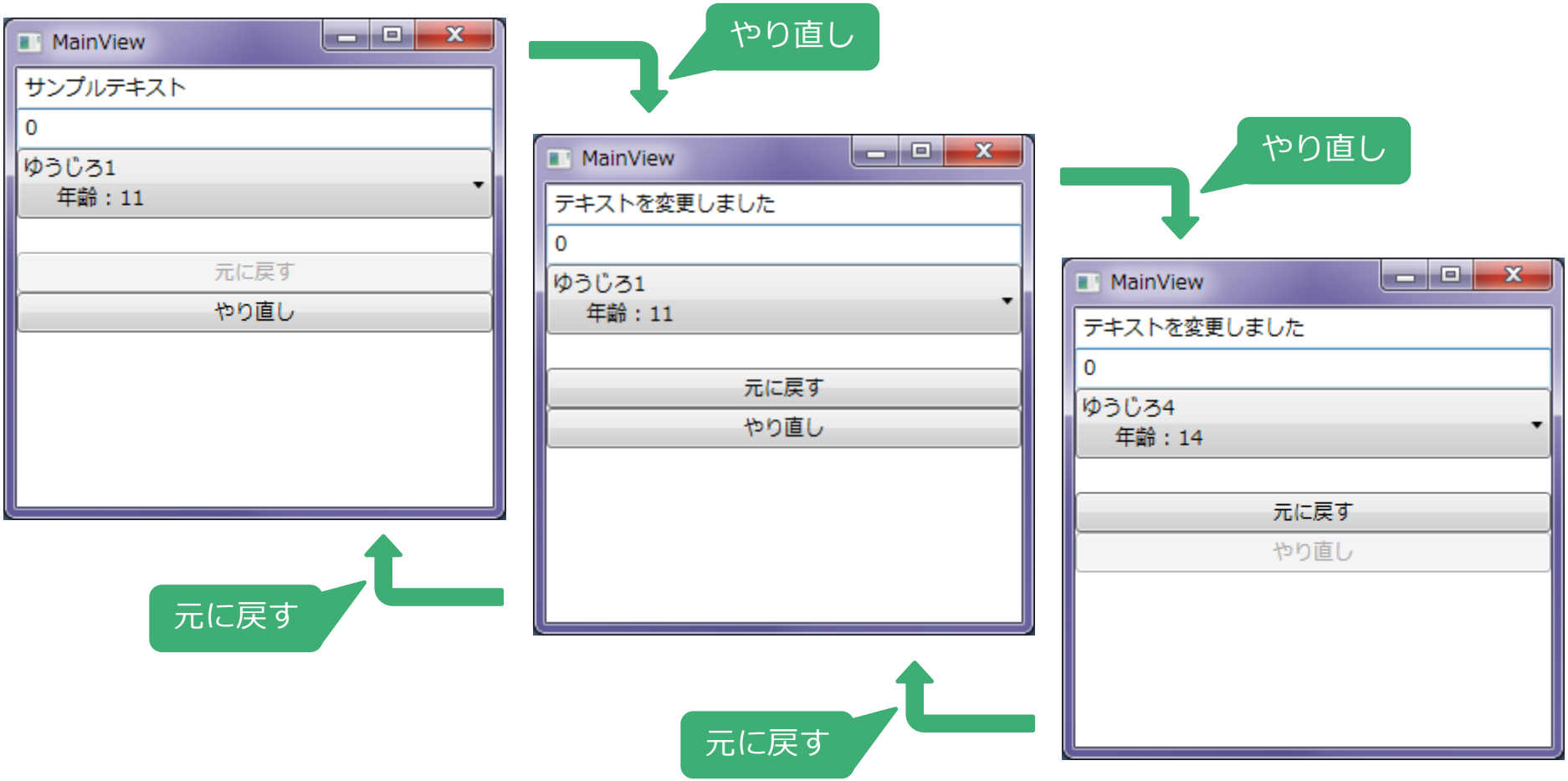
View (XAML)




```xaml
<Window x:Class="UndoRedoApp.Views.MainView"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainView" Height="300" Width="300">
    <StackPanel>
        <TextBox Text="{Binding Text}" />
        <TextBox Text="{Binding Value}" />
        <ComboBox ItemsSource="{Binding People}" SelectedItem="{Binding SelectedPerson}">
            <ComboBox.ItemTemplate>
                <DataTemplate>
                    <StackPanel>
                        <TextBlock Text="{Binding Name}" />
                        <TextBlock Text="{Binding Age, StringFormat='{}年齢 : {0}'}" Margin="20,0,0,0" />
                    </StackPanel>
                </DataTemplate>
            </ComboBox.ItemTemplate>
        </ComboBox>
        <Button Content="元に戻す" Command="{Binding UndoCommand}" Margin="0,20,0,0" />
        <Button Content="やり直し" Command="{Binding RedoCommand}" />
</StackPanel>
</Window>
```

実際に操作してみよう

元に戻す/やり直し機能が実現できましたか？



気になること

- プロパティ変更にしか対応していない
  - コレクションの要素変更を考慮する必要あり
- イベントハンドラの登録/解除に対応していない
  - Person クラスにイベントがあって MainViewModel が購読していたら…
  - リストから消去したけど裏でイベントを拾ってしまうこともあり得る

ご清聴ありがとうございました