棒球記分板

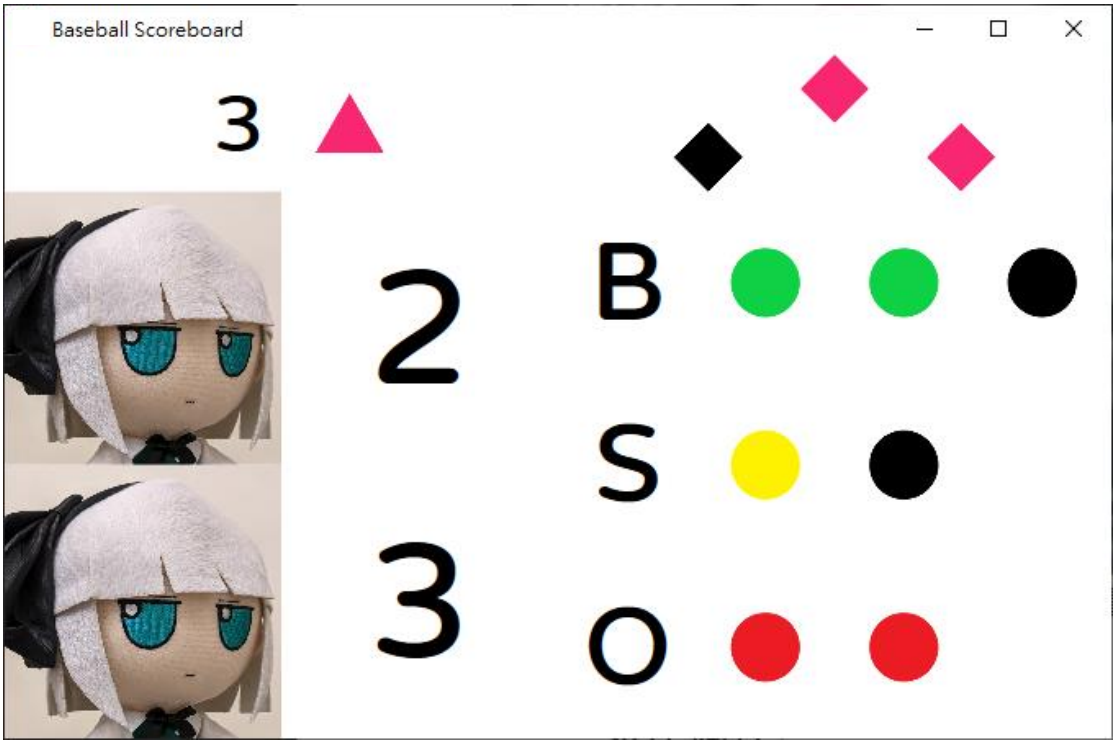
一、作品簡介

https://github.com/palapapa/BaseballScoreboard

此專案的製作動機是我的一個朋友託我所製作，他們的棒球隊在比賽時需要直播，因此就必須要有一個記分板的畫面，但是網路上並沒有符合他們需求的程式，因此就請我幫忙製作。此程式可以顯示局數、兩隊 Logo、比數、壘包、好球數、壞球數、以及出局數。在製作這個專案的過程中，我學到了如何使用 WPF、MVVM、Data Binding 以及 XAML 來製作一個可維護性高的 Windows 桌面應用程式。

下圖是我用來參考的圖片：

下圖是我的程式的畫面：



下圖為操作方法：

Key	Description
Up Arrow	Increase inning counter
Down Arrow	Decrease inning counter
1	Increase team 1 score
Q	Decrease team 1 score
2	Increase team 2 score
W	Decrease team 2 score
C	Toggle first base occupied
X	Toggle second base occupied
Z	Toggle third base occupied
B	Cycle ball counter
S	Cycle strike counter
O	Cycle out counter

Logos of both teams can be added by dragging and dropping their logos onto the scoreboard.

二、製作歷程與遭遇困難

在製作此專案前，我已經有撰寫過其他的 Winforms 專案，因

此，我一開始以為 WPF 只是有 XAML 的 Winforms，但是我在一次搜尋資料時發現了 MVVM 這個設計模式，我才知道 WPF 其實就是為了讓使用者能輕鬆建立 MVVM 應用程式而生的。

在我原本的打算中，我本來要像是寫 Winforms 一樣，在 UI 發生事件的時候，就用程式碼裡的 EventHandler 去做我想做的事情，但是在 MVVM 的概念中，這樣做就是把 View、View Model 和 Model 全都纏在一起了，這樣會使得程式的可維護性降低，因為抽換掉其中任何一個都可能弄壞程式。在 MVVM 中，View 只負責顯示，在 WPF 中，這部分是用 XAML 來完成；View Model 負責把 Model 裡的資料轉換成方便在 View 上呈現的形式，並且負責處理如按鈕按下等等的 UI 邏輯，而 View 則會使用 Data Binding 來連接到 View Model，View Model 反而不知道 View 的存在，這就代表 View 可以被任意抽換，也不會影響程式的運作，這就大大提升了可維護性；Model 則是用來儲存資料，或是處理商業邏輯，例如查詢資料庫等，相同的，Model 不會知道 View Model 的存在，是 View Model 主動去連接 Model。只要完成 MVVM 的架構，一個應用程式就可以被拆分成這三個部分，且這三個部分都是「單向認知」，增加可維護性之餘，也可以讓一個團隊的 UX 設計師和程式設計師的工作分開，增加效率。

知道了 MVVM 的重要性後，我查詢了如何將我目前類似 Winforms 的寫法改成符合 MVVM 的寫法，我發現這個關鍵就是 WPF 的 ICommand，就是因為有它，才可以成功把 UI 邏輯移動到 View Model。MVVM 的關鍵就是 View 和 View Model 分離，而如果只使用 EventHandler，就一定要把 UI 邏輯寫在 View 裡面，這是因為 EventHandler 不支援 Data Binding。然而，使用 ICommand 的話，就可以把所有處理 UI 邏輯的方法全部移到 View Model 以變數的形式儲存，而 View 只要 Data Bind 到 View Model 裡的 ICommand 就可以做到符合 MVVM 的架構。

ICommand 是一個介面，但是如果直接把它實作在某個 class 上，就代表每次想做一個新的 UI 邏輯，都要再寫一個 class，為了

避免這種麻煩，我發現標準的作法是宣告一個 class 名為 RelayCommand，如下圖：

```
internal class RelayCommand : ICommand
{
    private readonly Action<object> execute;

    private readonly Func<object, bool> canExecute;

    10 references
    public RelayCommand(Action<object> execute)
    {
        this.execute = execute;
    }

    3 references
    public RelayCommand(Action<object> execute, Func<object, bool> canExecute)
    {
        this.execute = execute;
        this.canExecute = canExecute;
    }

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

    0 references
    public bool CanExecute(object parameter)
    {
        return canExecute is null || canExecute(parameter);
    }

    0 references
    public void Execute(object parameter)
    {
        execute(parameter);
    }
}
```

這就代表，你只要把想要執行的方法傳進 RelayCommand 裡，它就會幫你產生一個有實作 ICommand 的物件，就可以只用這一個 class 打天下了，如下圖：

```
IncreaseInningCounter = new RelayCommand(
    (parameter) =>
    {
        if (!IsTopInning)
        {
            Inning++;
        }
        IsTopInning = !IsTopInning;
    }
);
DecreaseInningCounter = new RelayCommand(
    (parameter) =>
    {
        if (IsTopInning)
        {
            Inning--;
        }
        IsTopInning = !IsTopInning;
    },
    (parameter) => Inning > 1 || (Inning is 1 && !IsTopInning)
);
SetTeamLogoSource = new RelayCommand(
    (parameter) =>
    {
        EventHandlerArgPack<DragEventArgs> args = (EventHandlerArgPack<DragEventArgs>)parameter;
        Image image = (Image)args.Sender;
        DragEventArgs e = args.EventArgs;
        if (e.Data.GetDataPresent(DataFormats.FileDrop))
        {
            string path = ((string[])e.Data.GetData(DataFormats.FileDrop))[0];
            try
            {
                image.Source = new BitmapImage(new(path, UriKind.Absolute));
            }
            catch (NotSupportedException)
            {

            }
        }
    }
);
```

至於為什麼我在 CanExecuteChanged 要做那些 add 和 remove，是因為 WPF 裡的很多的內建控制項都會在他們的 Command 屬性裡存的 ICommand 的 CanExecute 回傳 false 時做出某種反應，例如按鈕會變成灰色並且不能點，但是 WPF 總不能一直不停地去檢查 CanExecute 的回傳值，這樣會嚴重影響效率，所以它會去訂閱 CanExecuteChanged 這個事件，當我們通知它 CanExecute 改變時，那個控制項才會去檢查 CanExecute 的值。但是很多時候其實很難，或是很麻煩才能知道什麼時候 CanExecute 會改變，所以這裡的作法是，當有控制項訂閱 CanExecuteChanged 時，「偷偷的」也幫它訂閱 CommandManager.RequerySuggested，訂閱這個事件的 EventHandler，會在 WPF 認為有任何 CanExecute 可能已經改變時被呼叫，包括光是移動滑鼠都會導致這個事件被觸發，所以我們

等於是請 WPF 幫我們「猜」什麼時候 CanExecute 已經改變了，這樣我們就可以不用自己實作什麼時候要觸發 CanExecuteChanged 的功能，犧牲的是一點點執行速度。

在我這次的專案中，使用 ICommand 時有遇到一個困難，那就是我在實作拖拉隊伍圖片的功能時，我發現如果我要在使用者把一個檔案拖到我的圖片控制項上時，呼叫 View Model 裡的 ICommand 的話，似乎一定要經由 View 的程式碼來間接呼叫 ICommand，這是因為大多數控制項的 Command 屬性都只對應到某一個事件，不是所有事件都支援 Data Bind 到 ICommand，例如按鈕，如果你把它的 Command 屬性 Data Bind 到某個 ICommand 的話，這個 ICommand 只會在這個按鈕被點擊時呼叫，而忽略其他任何事件。這就代表，如果我想要在某個檔案被拉到圖片上時，呼叫我用來改變圖片的 ICommand，就一定要先用 EventHandler 接下這個事件，然後在這個 EventHandler 裡再轉交給 View Model 裡的 ICommand 處理。我在網路上搜尋這個問題的解決辦法時，發現可以用 System.Windows.Interactivity 裡面的某個 attached property 來解決，可是這樣還是不能把事件的 EventArgs 傳給我的 ICommand，這就代表我不知道被拉進來的圖片的檔案路徑是什麼，自然也就不能更新隊伍圖片。雖然也有人提供一個函式庫可以讓你可以把 EventArgs 傳給 ICommand，但是我認為這樣太麻煩了，而且我也查到有人說我先用 EventHandler 接下來的方法是可以的，因為眞正的邏輯還是在 View Model 裡面，而且使用 MVVM 不代表 View 完全不可以有程式碼。所以，我最後的解決方法是，建立一個 class 名為 EventHandlerArgPack，如下圖：

```
internal class EventHandlerArgPack<TEventArgs> where TEventArgs : EventArgs
{
    2 references
    public object Sender { get; set; }

    2 references
    public TEventArgs EventArgs { get; set; }

    1 reference
    public EventHandlerArgPack(object sender, TEventArgs eventArgs)
    {
        Sender = sender;
        EventArgs = eventArgs;
    }
}
```

它可以幫我把 EventHandler 的兩個參數包到一個物件裡，這是因為 ICommand 只能使用一個參數。之後，我就可以把這個事件從 EventHandler 轉傳到 ICommand 處理，如下圖：

```
private void TeamLogo_Drop(object sender, DragEventArgs e)
{
    EventHandlerArgPack<DragEventArgs> args = new(sender, e);
    if (viewModel.SetTeamLogoSource.CanExecute(args))
    {
        viewModel.SetTeamLogoSource.Execute(args);
    }
}
```

為了實現 View 和 View Model 完全分離，我在這次的專案中也使用了 Style Trigger，它是 XAML 裡可以為控制項設定的一個屬性，它可以在 View Model 裡的資料符合條件時，自動更新某個控制項的屬性，我就是利用這個來做到壘包和好壞球的顯示等等的功能，我在 View Model 裡有幾個屬性是代表壘包是否有人，或是目前有幾顆壞球等等的，而在 View 裡面，我就使用 Style Trigger 來檢查這些屬性的值，並自動更新它們的顏色，這樣可以做到一行程式碼都不用寫就可以完成 UI 邏輯。

但是我在使用 Style Trigger 時有遇到一個問題，就是它預設只支援比較是否等於，無法比較大於小於之類的，這就在我的好壞球顯示上造成了問題，因為我原本打算的實作方法是：第一個燈在壞球數大於等於 1 時，就需要點亮。幸好這個問題可以透過 IValueConverter 解決，它是一個用在 Data Binding 的介面，可以在 View Model 裡的資料傳到控制項身上之前，先經過某種處理，

這個功能原本是方便使用者對資料做某種格式化，讓資料在 View 上好看一點，但這個功能也可以被用在 Style Trigger 身上，這是因為 Style Trigger 在檢查 View Model 裡的值時，它也是使用 Data Binding 去取得那個值，所以我可以利用 IValueConverter 來在 View Model 裡的壞球數傳到 Style Trigger 裡時先「攔截」它，在我的 Style Trigger 裡做大於等於的比較，之後才把比較結果的 true 或 false 傳給 Style Trigger，而 Style Trigger 只要檢查結果是不是 true，就完成使用 Style Trigger 檢查大於等於的功能了。

三、心得與反思

在完成這次專案後，我學到了如何正確使用 WPF 和套用 MVVM 設計架構，我認為這不但讓我更熟悉桌面開發，也可以在我將來學習製作 Web App 時，更無縫接軌 MVC 設計模式，讓我成為一個更有能力的程式設計師。