ソフトウェア工学

オブジェクト指向プログラミング

4I No.4 aridai

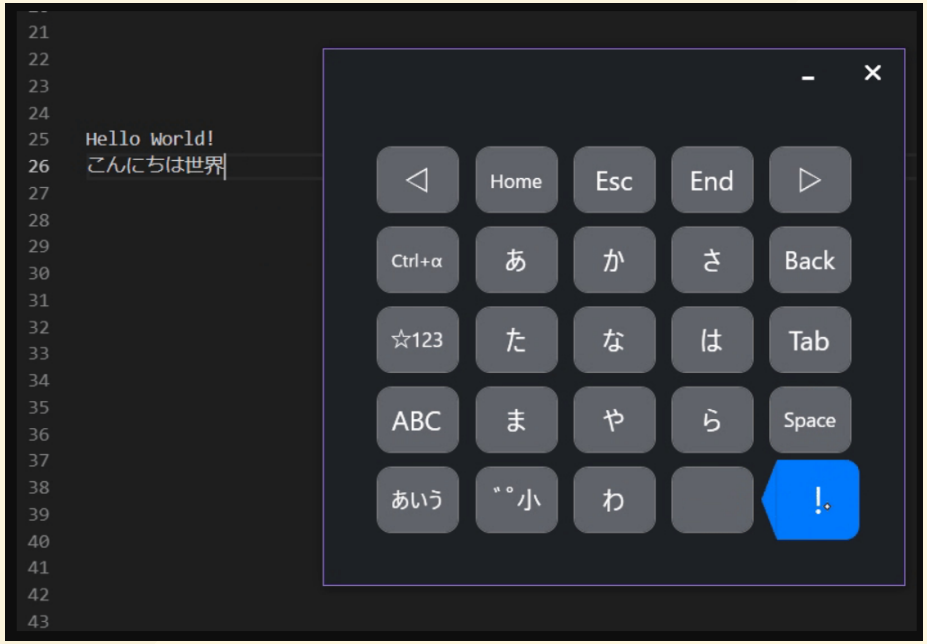
(GitHub: @aridai / Twitter: @aridai_net)

今回話すこと

- できたもの
- 使ったもの
- プロジェクトの構成
- WPFについて (フレームワーク/アーキテクチャ)
- Rxについて
- P/Invokeについて
- テストについて
- 感想

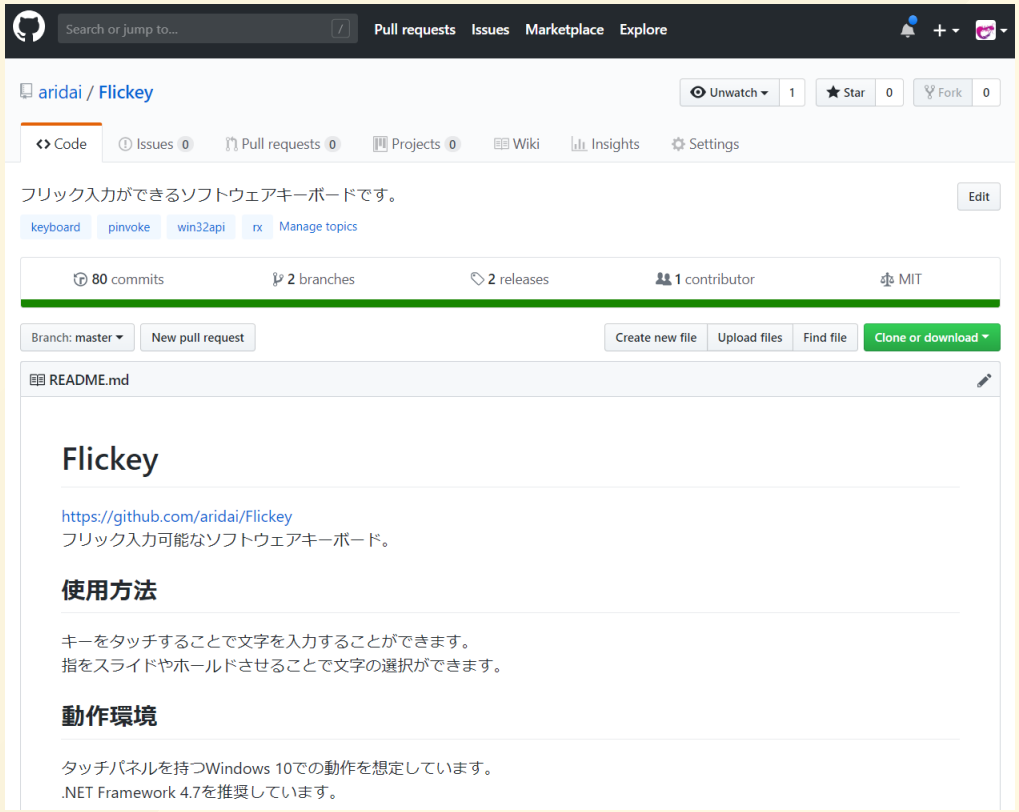
できたもの

- フリック入力可能なソフトウェアキーボード。
- Windows 10で動く。(タッチパネルが必要。)
- 指をスライドさせたり、ホールドさせたりして入力できる。



できたもの

設計図と配布物は設計図共有サイトでMITライセンスで公開します。
https://github.com/aridai/Flickey



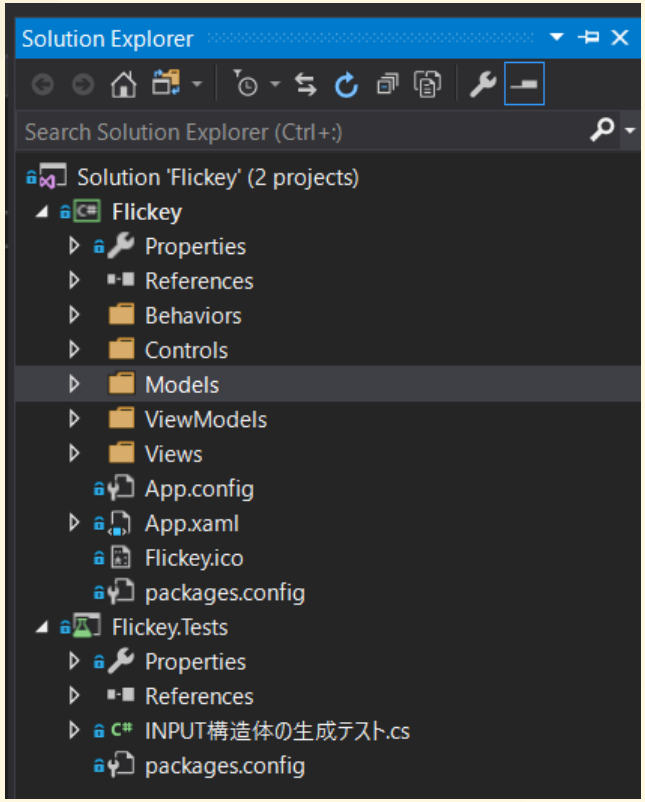
使ったもの

- 言語: C# 7.3
- GUIフレームワーク: WPF
- テストフレームワーク: MsUnit
- ライブラリ: Prism / ReactiveProperty
- IDE: Visual Studio Community 2017
- バージョン管理: Git + git-flow + 設計図共有サイト

プロジェクトの構成

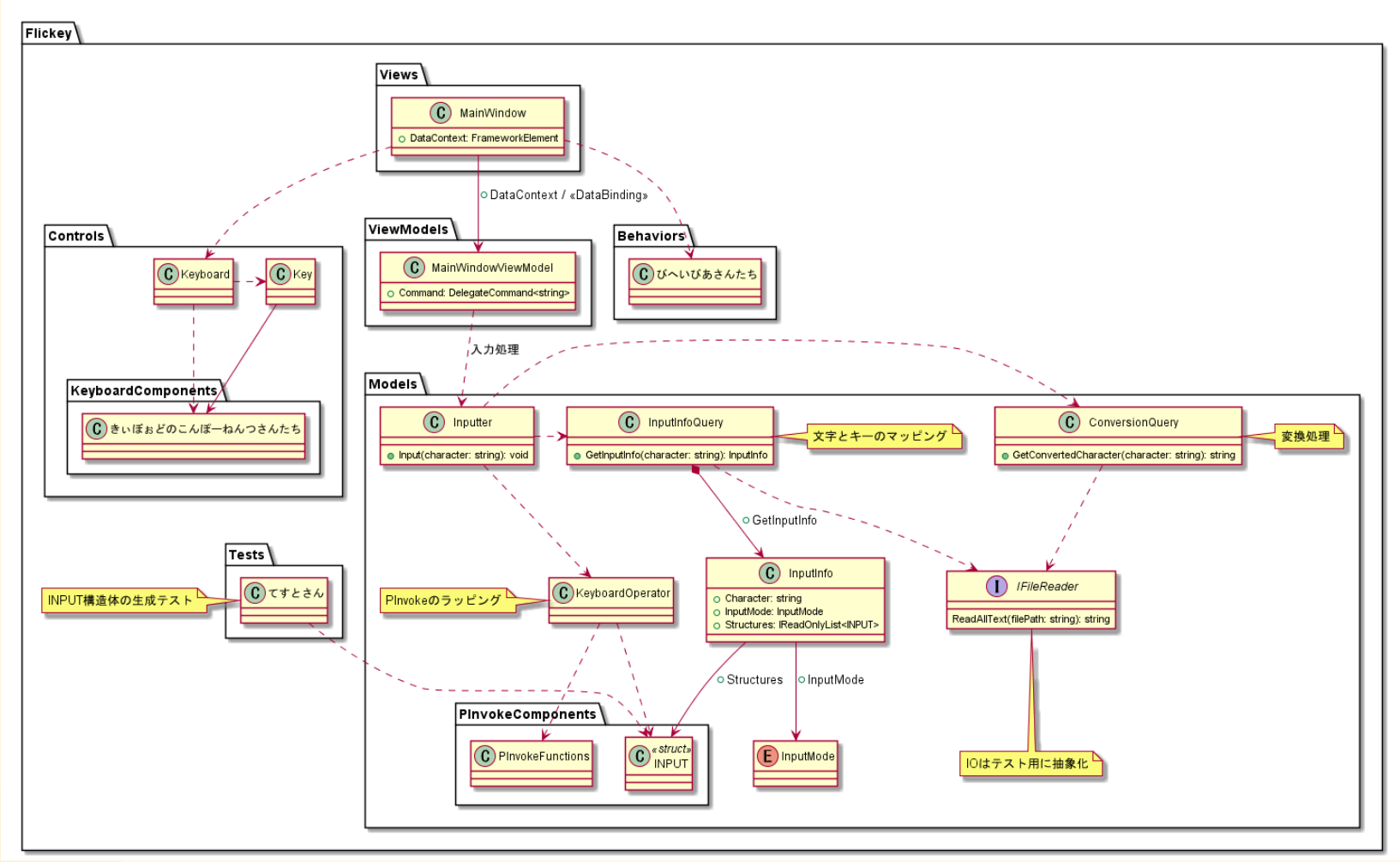
Visual Studioソリューションはこんな感じに。

- 2プロジェクト。
  (WPFApp + UnitTest)
- MVVMパターン +
  Behaviorパターン。
- UI部分はユーザコントロール
  として作成。



プロジェクトの構成

クラス図はこう。一応、頭の片隅に**SOLID原則**を。

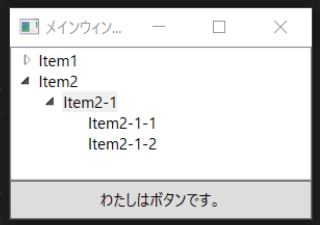


WPFについて

Windowsアプリケーションのフレームワーク。
UIを**XAML**を使って記述する。

```xaml
<Window
    x:Class="WpfApp.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    Title="メインウィンドウ" Width="500" Height="400">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
        <Button Grid.Row="1" Content="わたしはボタンです。" />
        <TreeView Grid.Row="0">
            <TreeViewItem Header="Item1">
                <TreeViewItem Header="Item1-1">
                    <TreeViewItem Header="Item1-1-1" />
                    <TreeViewItem Header="Item1-1-2" />
                </TreeViewItem>
                <TreeViewItem Header="Item1-2">
                    <TreeViewItem Header="Item1-2-1" />
                    <TreeViewItem Header="Item1-2-2" />
                </TreeViewItem>
            </TreeViewItem>
            <TreeViewItem Header="Item2" IsExpanded="True">
                <TreeViewItem Header="Item2-1" IsExpanded="True" IsSelected="True">
                    <TreeViewItem Header="Item2-1-1" />
                    <TreeViewItem Header="Item2-1-2" />
                </TreeViewItem>
            </TreeViewItem>
        </TreeView>
    </Grid>
</Window>
```

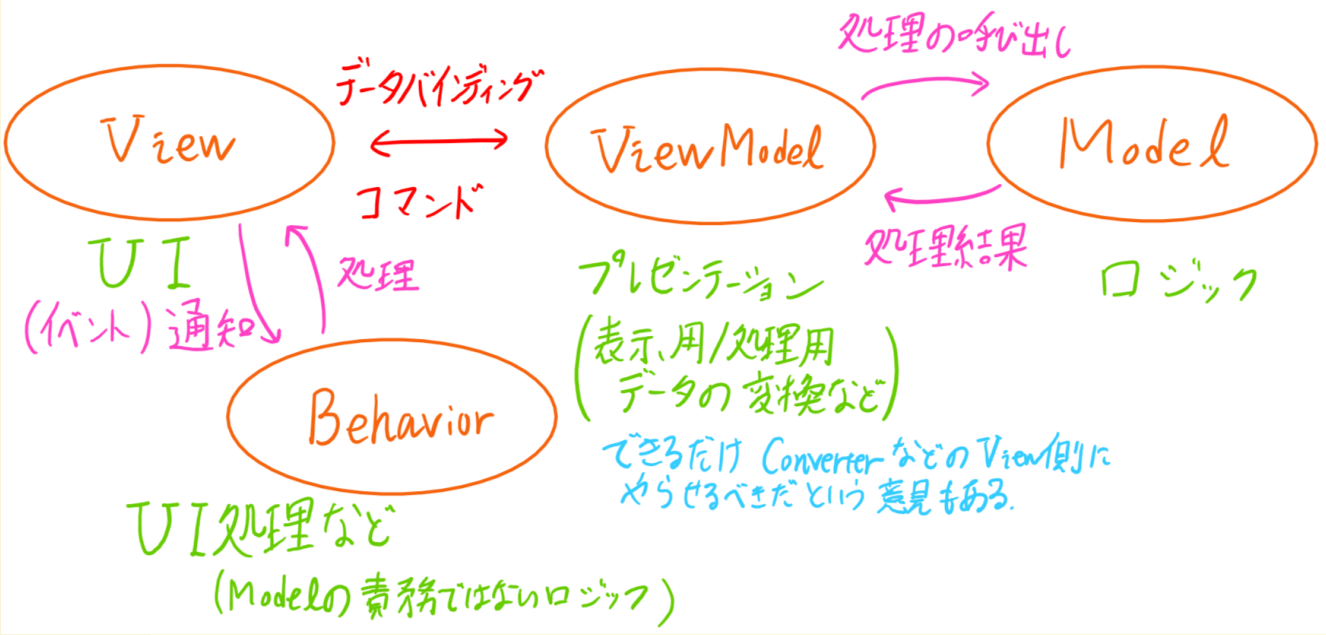


WPFについて

- **MVVMパターン**
  Model・View・ViewModelの3つの層に分けて設計・実装を行うアーキテクチャパターン。
- **Behaviorパターン**
  ViewのUI処理を**Behavior**の層で行う。(語弊あり)
- **データバインディング**
  一方のデータが変更されると他方のデータも更新される仕組み。
- **コマンド**
  操作とそのパラメータ (引数/利用可能状態) をまとめて1つのものとして扱ったもの。

WPFについて

今回の開発も**MVVMパターン**に従って、画面部分と処理部分と分けて作るようにした。
具体的には、キーボードの制御 (表示の切り替えなど) に関しては、キーの入力処理が知らなくてもいいようになっている。(疎結合でみんな幸せ!)



WPFについて

ウィンドウのイベント処理は、**コードビハインド** に書かずに **Behavior** で処理する。
再利用ができ、コードビハインドの複雑性が減る。
(機能ごとにBehavior単位で分割できる。)

```
8   public sealed class MainWindowDragAndMoveBehavior : Behavior<Window>
9   {
10      protected override void OnAttached()
11      {
12          base.OnAttached();
13          this.AssociatedObject.MouseLeftButtonDown += this.OnMouseLeftButtonDo
14      }
15
16      protected override void OnDetaching()
17      {
18          base.OnDetaching();
19      }
20
21      private void OnMouseLeftButtonDown(object sender, MouseButtonEventArgs e)
22      {
23          if (e.LeftButton == MouseButtonState.Pressed)
24          {
25              this.AssociatedObject.DragMove();
26          }
```

Rx (Reactive Extensions) について

LINQのobserver-pattern版みたいなやつ。
キーボードのタイミング制御などが複雑になりそうだったため、これを使うと、時間やスレッドをまたぐ制御が楽になった。

```
//  長押し判定の例
var holdDetection = operationTarget
    .Select(target => target.deviceId)  //  データの選択
    //  IDが有効で、
    .Where(id => id != null)            //  通知のフィルタ
    //  一定時間経過したら、
    .Delay(timeOut)                     //  時間差通知
    .ObserveOnDispatcher()              //  スレッド変更
    .Where(id => id == deviceId)
    //  長押し判定にする。
    .Select(_ => OperationType.Hold);
```

Rx (Reactive Extensions) について

Rxはいいぞ!

```
//  入力操作の種類を通知
operationType = Observable.Merge(
    //  タップ判定とスライド判定と長押し判定を合成
    isTapping, isSlideing, isHolding)
//  過去のデータと結合
.Pairwise()
.Where(pair =>
  //  現在スライド中で過去がタップじゃない
  !((pair.NewItem == Slide && pair.OldItem != Tap)
  //  または現在長押し中で過去がタップじゃない
  ||(pair.NewItem == Hold && pair.OldItem != Tap)))
.Select(pair => pair.NewItem)
//  ReactiveProperty化
.ToReadOnlyReactiveProperty()
.AddTo(this.disposable);
```

Rx (Reactive Extensions) について

今回は主に **ReactiveProperty** を利用した。

- `IObservable<T>` に加え、値をキャッシュする仕組みと、双方向バインディングをサポートした。
- `IObservable<T>` から `.ToReactiveProperty()` で生成。
- `ToReactivePropertyAsSynchronized()` で生成すると、双方向バインディング可能。
- オープンソースライブラリで、GitHubでコードが読める。
  https://github.com/runceel/ReactiveProperty

P/Invokeについて

- **P/Invoke**
  C# から **C++** のコードを呼び出す機能。(語弊あり)

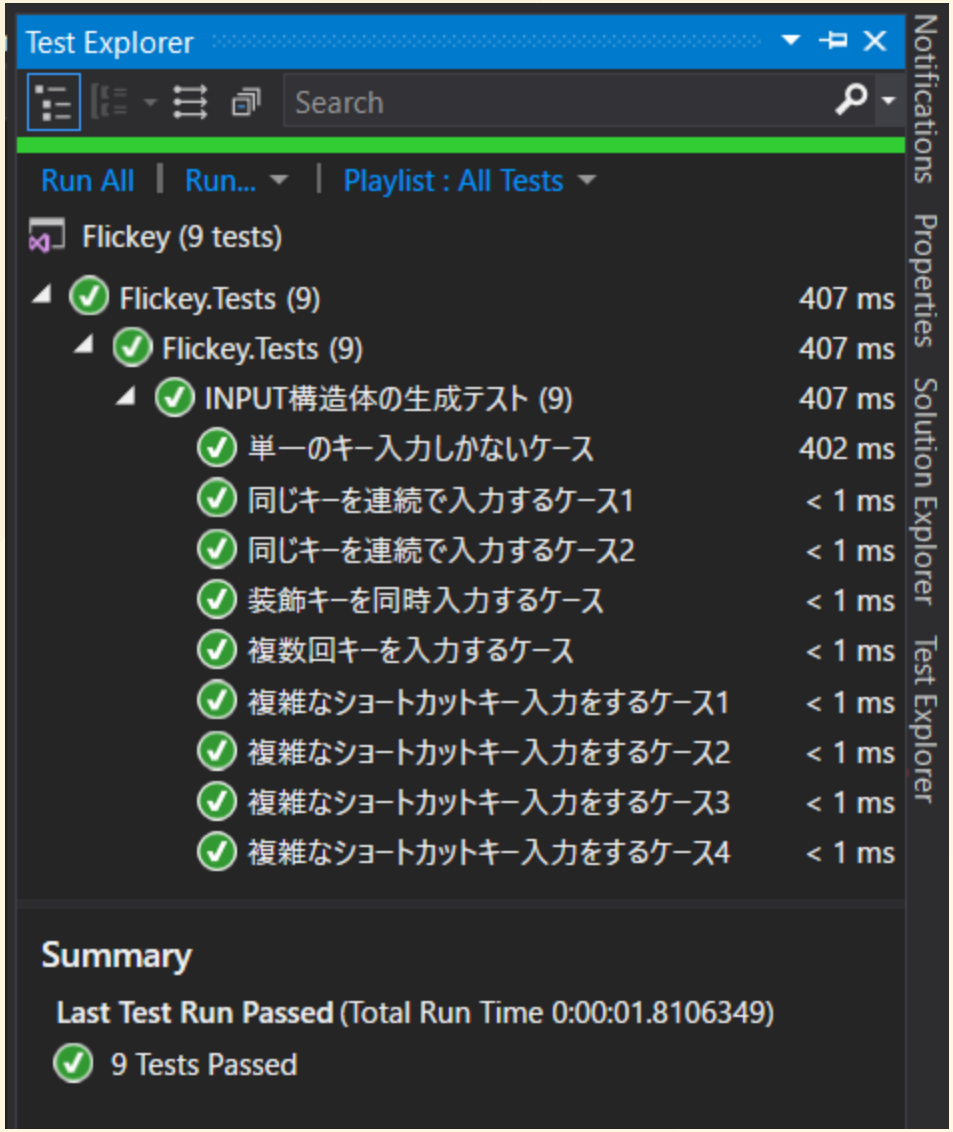
IMEの処理をするには、**Win32 API** を利用しなくてはならなく、それは **C++** で書かれているため、**P/Invoke** を利用して呼び出した。(語弊あり)

```
[DllImport("user32.dll")]
public static extern int SendMessage(
    IntPtr hWnd, uint Msg, uint wParam, int lParam);

[DllImport("user32.dll")]
public static extern int SendInput(
    int nInputs, INPUT[] pInputs, int cbSize);
```

テストについて

- MsUnitを利用した。(xUnitと迷った。)
- 入力データの生成が正しくできているかをテストした。
- 先にテストケースを書いて実装した。
- テストが通るとめっちゃ気持ちいい!



感想

前期に座学でやったことを意識しながら開発ができてよかったと思いまぁす。

今回は規模的/時間的にできなかったけれども、**TDD**や**DIコンテナ**、**CIサーバ**なども試してみたいと思った。

タッチパネル付きのWindowsPCを持っている人は是非使ってみてね。

issues / **pull requests** 大歓迎でぇす。
リポジトリ: https://github.com/aridai/Flickey