1. ViewModels hierarchy should reflect UI hierarchy

- Every level of nesting of UI should correspond to a level of ViewModels nesting

- Single responsibility principle – don’t be afraid of very, very short classes!
- Value composition over inheritance

How do we achieve that?

Create simple, hierarchical xaml:

Example: `ConceptsViewModel.cs`, and its DataTemplate

How to test it:

- Check if an instance can be created successfully – sometimes it's enough ☺

Initialise all controls that support ItemsSource via collections in ViewModel (*that’s new to learn*)

- That way we keep xaml simple, xaml and code is easier to read
- And we can easily e.g. change order of tabs, or switch off a feature without commenting out half of a xaml file

Example: `PlotsViewModel`,and its DataTemplate
// we will talk about the `ExportFactory` later

Detect ‘Cross Cutting Concepts’

(local singletons, or pieces of code and data that you share among multiple components)

- This can be as small as an enumerable, and an event that is fired on a change

- Communication - Parent → Child
  - In code - directly
  - In xaml - nested properties see `PlotViewModel` DataTemplate

- Communication – Child → Parent, or between not connected ViewModels
  - With events and Cross Cutting Concepts
    - Cross Cutting Concepts example: `SliderViewModel`,and its usages
    - Events example: `On_GenerateNewData_event_Measurements_should_be_changed()`
    - I’d recommend creating custom events rather than listening to property changed event:
      - Better performance
      - Easier to read
  - Don’t use static properties and fields! (Sam asked for an example: EvilStaticPropertyTests.cs!)

2. Memory management:

Why? - Not to have memory leaks!
Example: `ConceptsViewModelIntegrationTests`

How do we achieve that?

- Injecting everything
- Using factories – `ExportFactory<T>` - (*that's new to learn*)
  - Example: `PlotsViewModel`

- Implementing IDisposable to:
  - Unsubscribe from C# events, EventAggregator events, RX data sources
  - To dispose 3rd party components
  - Example: `PlotViewModelTests.Dispose_should_be_successful()`

3. Testing

What should be tested?

Three basic rules of High Coverage Master:
1. Test ViewModel’s initialization
2. Test Dispose()
3. Check if all user actions were successful

It can also be very useful (especially when you refactor a spaghetti code), to check if the change you expect happened only once! // No example right now!

Example: `PlotViewModelTests.`

4. Best practices:

- Don’t use user controls. Use:
    - EventToCommand
    - Behaviors
- Avoid Is*PropertyName*Visible pattern, replace it with ContentPresenter pattern
- Don’t raise property changed event when unnecessary (use `Mode=OneTime` in bindings to prevent memory leaks)
- Use AutoFixture ☺