# Composite UI Patterns

## Intro

* Previous experience building composite UI’s with Silverlight and WPF
* Before huddle didn’t have much experience with javascript, other than jquery and trying to use the module pattern to tidy up the code
* Learnt a lot about javascript in the last few months been fortunate to sit next to john and dom and discussions with james
* I enjoy working on the UI and I think there is a good separation of concerns, using knockout is fun
* There are some concepts that I think we could use in order to simplify some areas of our code base
* For example, we could simplify our event registration by using the event aggregator pattern, which I will talk about later
* Also, we could introduce the concept of view model life cycle
* At the moment we instantiate all views and viewmodels at startup time
* This means we have to keep them all in sync when data changes
* We use events to sync and it can be complicated to work out the chain of events and where the data originated from
* Also means that most of our view model properties need to be observables in order to keep the views in sync
* If we instantiate view models when they are needed, we can explicitly pass the data they require which would remove the need for syncing and we could use plain properties rather than observables
* A classic example of this is with modal dialogs, but it can also apply in URL navigation scenarios or tabbed interfaces
* Final thing is something we’ve started to do already, which is introduce the model part of MVVM, which we will discuss later

## Agenda

1. Why composite UI? – Advantages of this approach
2. Shell, Regions & Layouts - How we loosely couple the UI
3. Screens & Conductors - Managing lifecycle
4. URL Navigation – Compositional navigation
5. View Model Communication – different ways we can communicate across view models

* I’m going to show example implementations of these patterns using both WPF & HTML/JS
* Looked at other frameworks such as backbone, angular and ember. Some good points and some bad.
  + difficult to isolate your code from the framework
  + support for composition is lacking
  + poor separation of concerns around composition, mostly view-first
  + Angular and backbone marionette deserve looking at in more depth.
  + Angular: like the mechanism for creating reusable UI controls and dependency injection
  + Backbone marionette built around composition
* Finally, apologies if this is a little dry or incoherent. I’m throwing a lot of concepts around, hopefully you’ll find it interesting.

### Why composite UI?

* A way to enforce the SRP by breaking up application into components
* This helps us to manage complexity
* Promotes Loose coupling, more responsive to change

### UI Composition

* There are different types of composition in the UI
* They tend to fall out naturally by applying the single responsibility principle
* Slide1
* Here we have a static view
* First refactoring we could do if the view gets complicated would be to compose the view of other views
* Slide2
* All the views could potentially be composed using a layout template AKA masterpage
* Slide3
* Now, if we introduce rest api into the mix, we may decide to factor out a view model to deal with getting the data and providing a format that is easy for the view to bind to
* Slide 4
* When the viewmodel starts getting complicated we can factor that out:
  + Slide5
  + Services – logic/functionality reusable across view model
  + Factories – for when creating the viewmodel becomes complex
  + Compose with other viewmodels, which may have their own view
* If we find we want to isolate the code that deals with the rest api or we have client-side domain logic, we may choose to factor out a model
* Slide 6
* When the model becomes complex, we may need:
  + Slide 7
  + Services – logic/functionality reusable across model
  + Factories - for when creating the model becomes complex
  + The model can be built up using DDD concepts of aggregates entities and value object, although you might only use this if you needed client side domain logic.
* After that you might want to factor out a repository, slide 8
* Mappers
* Cache

The patterns I am now going to demonstrate are mainly around composition of the view model part.

## Shell, Regions & Layouts

### Shell

* The “composition root” of the application

### Regions

* Image 2 -> Regions contain components that may call out to multiple back-end systems, services, and data stores
* Especially good fit for dashboard-type applications
* Also, can be useful when there are independently evolving UI components that heavily integrate with each other and that are often maintained by separate teams

### Layouts

### DEMO: Regions

* Skim over how XAML apps work
* App.xaml
  + Root of a WPF application
  + XAML is a declarative markup language for creating UI components
  + XAML directly represents the instantiation of objects, basically a serialization format that’s easy for humans and tools like blend to read
  + XAML enables a workflow where separate parties can work on the UI and the logic of an application, using potentially different tools (designers/devs)
  + XAML typically has an associated code behind file for handling events (like web or winforms)
  + Has a concept of DataContext that allows databinding of the XAML to a CLR object, this can be set programmatically in the code behind or declaratively in the XAML itself
  + In App.xaml we define a resource that maps directly to a AppBootstrapper class in our application. This will be instantiated when the application starts.
* Bootstrapper
  + We are using a compositional framework called Caliburn.Micro, which provides some plumbing code to enable make MVVM and composition in general easier
  + Bootstrapper takes a type which will be instantiated and used to show the application
  + Caliburn.Micro prefers a ViewModel first approach. The alternative is View first, where the responsibility to create the viewmodel is in the view.
* Shell view model -> defines regions
  + We are adding components to regions using poor mans IoC here, but we could use a registry or IOC or MEF.
  + Prism has concept of RegionManager and IRegion interface, but caliburn is a lot more lightweight
* Shell view model – find view
  + Bootstrapper applies conventions in order to determine what view should be loaded
  + Loads view and sets the shellviewmodel as its datacontext
  + Content controls bind via convention to properties on the view model
  + These properties are in turn view models that have associated views
  + Composition flows from there
* Shell view -> defines app layout, containers for regions, apply conventions
* Regions are loosely coupled to the shell, we could package them up separately

## Screens

### Screen

Examples: a page in a browser – “although this could contain multiple screens”

### Screen Activator

* Screens often have an associated lifecycle which allows custom activation and deactivation
* Ex: Visual studio code editor toolbar icons, activation/deactivation does setup/teardown of the application toolbars
* Screen Activator is responsible for this activation and deactivation
* In simple scenarios, the ScreenActivator is often the same class as the Screen. However, they are two separate roles. If a particular screen has complex activation logic, it may be necessary to factor the ScreenActivator into its own class in order to reduce the complexity of the Screen. This is particularly important if you have an application with many different screens, but all with the same activation/deactivation logic

### Screen Conductor

### DEMO: Screens

* Shell view model -> changed to a conductor
* Adding items to the conductor activates them
* WidgetPicker is a nested conductor. Parent conductors are responsible for the activation of child conductors. As we navigate around our application and perform various functions, activation and deactivation of components becomes important to manage. Nesting conductors in this way allows us to perform complex activation and deactivation in a loosely coupled fashion.
* Conducts twitterfeedviewmodel and whatsnewviewmodel
* View -> Button binds to method by conventions
* Twitterfeedviewmodel -> IActivate -> activation refreshes tweets
* Also asks the screen if it can close
* eg unsaved data
* might be triggered by application closing down or conductor itself
* Closing is semantically different to deactivation, eg switching between tabs may cause deactivation, but not close
* Conductors can also be screens, and therefore may be conducted by other conductors, creating a composite structure. Although it is generally not good UI design to have deep levels of nesting. Win 8 semantic zoom.

### Screen Collection

### DEMO: Conductor

* FileEditor is document editor, similar to visual studio
* Holds different types of files
  + Editable
  + dirty tracking
  + cannot close if dirty (also app)
  + save removes dirty
  + toolbar changes
* FileEditorViewModel -> Is a conductor and collection of IFileViewModel screens, with one active
* Initialising with a toolbar and a set of files
* ToolbarViewModel holds collection of buttons
* Buttons
* Button views configured by convention
* Each file takes toolbar as a dependency, because it will need to manipulate it
* FileEditorView
* FileViewModelBase -> Common file behaviour

## Window manager

### DEMO: WindowManager

* FileViewModelBase
* DialogViewModel
* Window manager is good example of creating an abstraction in our view model layer to model user interaction

## MVVM Navigation

* Can maintain history by storing a stack of view models
* Need to be careful of memory management of history
* Html5 push state, pass state or possibly serialize to local storage

### DEMO: Navigation

* Added AddressBar widget that publishes UrlChanged events
* Shell handles event and creates a view model based on the Url
* View model is instance that has lifecycle
* NotFoundViewModel

## Communication between view models

### Direct References

* Maintain direct references to other view models and invoke methods on them
* Eg, conductor asking children if they can close
* *Applicable where the view models are logically tightly coupled anyway*
* *Minimise coupling by using interfaces*

### DEMO

* ParentChild
* User – DetailsViewModel
* Shell is setting directly
* User – EditViewModel, pass DetailsViewModel to it
* Can reduce coupling by passing an interface (see SaveButtonViewModel in Conductor)

### Direct View Model Event Subscription

* Pros
  + Minimized afferent coupling – Parent doesn’t need a reference to the child
* Cons
  + Efferent coupling - must have a direct reference to source view model
* Usage: Provides more loose coupling than direct references, but prefer event aggregator for its reduction in coupling

### DEMO: DirectEvents

* TweetTrackerViewModel, takes dependency on TwitterFeedViewModel so it can subscribe to change events

### Event Aggregator

* Usage: When you have independent view models that are potential event sources

### DEMO

* EventAggregator
* TweetTrackerViewModel, takes dependency on EventAggregator
* Events now raised from TwitterFeedViewModel -> SelectedTweet setter

### Shared Context – AKA the Model in MVVM

* Using a model prevents need to pass shared data between view models – we can simply ask the model for the current state
* Server is source of truth – do not normally want to invoke business logic or rules on client (unless we support offline mode)
* Offline mode – domain model in client, event sourcing – merge event streams
* Some model data is client specific, eg browser history
* Usage:
  + View model depends on repository to get model data
  + Mapper populates view model properties from model
  + Model could publish change notifications when using client-server events, eg signalR
  + Optionally implement a cache in repository

### DEMO: Model

* We now have more moving parts as we have greater separation of concerns
* AppBootstrapper - Added dependency injection to help manage this
* Traded complexity for maintainability
* Extracted navigation logic into INavigationService
* Added factories to decouple view model creation, as we rely on runtime values
  + Factories usage
    - Complex object creation
    - Runtime values
    - Lifecycle needs to be managed by consuming object
* Added a user repository
* DetailsViewModel and EditViewModel no longer pass data between them
* UserModel can raise events
* Could use observables for more fine grained notification

## Composite views

## Packaging

## Views

* Reusable view functionality
* Eg, tabcontrol, listbox, autocomplete
* HTML not a good mechanism for custom controls
  + Defining your own tags (XAML approach) not? good
  + jQuery controls often mix databinding and behaviour
  + KO binding handlers
  + Encapsulate using server side control
* View specific javacript
* Controls must support databinding
* XAML -> Not necessary to use separate view -> datatemplates
* Views can be reused by convention (attributes, base classes, meta data, etc) or composition

## General Points

* These patterns helped clarify my thinking around how to decompose a UI
* Patterns all based around separation of concerns
* About identifying roles within
* In a small system a lot of these roles can be combined
* Useful to have a common language of pattern so we reason
* Loose coupling
* Testability
* Encapsulation
* Extract reusable or complex code into “services”
* Split complex view models up
* Memory leaks
* ViewModel vs View First -> views should not be the ones composition the application (mixing concerns), isolates you from any framework, should be able to run the application headless, durandal and caliburn both “plug” into abstractions (eg windowmanager)
* Simple code -> boilerplate code removed via conventions -> split complex viewmodel using composition
* Roles -> Views, VM, Model (repository, mapper, service)
* View code can include javascript – eg KO bindings

\*\* translations \*\*

<http://codebetter.com/jeremymiller/2007/07/26/the-build-your-own-cab-series-table-of-contents/>