# Composite UI Patterns

## Intro

* Demonstrating patterns I have used at previous jobs using XAML in WPF and Silverlight
* Wanted to show javascript examples, led me down a lot of research
* Managing complexity
  + Easy for UI code to get lost in spaghetti mess of data bindings, event bindings and callbacks
* Separation of concerns, loose coupling and encapsulation
* Maintainability and flexibility
  + Shared mental model
  + Create ubiquitous language similar to DDD

## MVVM vs MVP vs MVC

<http://wiki.ecmascript.org/doku.php?id=harmony:observe>

## Shell, Regions & Layouts

### Regions

* Container for dynamic content
* Content is injected in at runtime
* Allows loose coupling of UI components
* <http://i.msdn.microsoft.com/dynimg/IC448657.png>
* <http://i.msdn.microsoft.com/dynimg/IC448646.png>

### Layouts

* A region that contains other regions
* Similar to MasterPages

### Shell

* Application root layout that holds the primary UI content
* Contains “chrome”, eg background, main menu, widgets, toolbars, etc
* <http://i.msdn.microsoft.com/dynimg/IC448656.png>
* WPF: Window object
* Silverlight: RootVisualUserControl
* Html/JS: Application container element

### DEMO

* Regions
* Skim over how XAML apps work
* App.xaml
* Bootstrapper
* Shell view model -> defines regions
* Shell view -> defines app layout, containers for regions, conventions
* Components -> Search, WhatsNew and workspaces
* Could be in separate assemblies
* Add components to regions, could use registry or IOC

## Screens

### Screen

* A stateful unit of work existing within the presentation tier of an application
* Contains one or more related views
* Examples include dialogs, a page in a browser or a visual studio code editor window

### 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, you should remember that these 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

* Manages transitions between screens - activation and deactivation

### DEMO

* Screens
* Shell view model -> changed to a conductor
* Components -> TwitterFeed, WidgetPicker -> conductor
* WidgetPicker activates screens
* Button binds to method by conventions
* Twitterfeed 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

* Maintains list of currently opened screens
* Typically works together with conductor:
  + Opening a new document – add to collection, switch to active screen
  + Closing a document – deactivate, remove from collection (as long as the CanClose method on the screen returns true)
* Databind to a UI component such as a TabControl

### 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

* Conductor for handling window management
* Provide a consistent mechanism for activating screens in:
  + Modal dialogs
  + Pop ups
  + Notifications (toast)

### DEMO

* WindowManager
* FileViewModelBase
* DialogViewModel

## MVVM Navigation

1. Shell view model is a simple conductor
2. Front Controller listening to URI changes
3. URI is translated into a view model instance by the Front Controller
4. URI parameters passed to view model constructor
5. View model passed to shell conductor for activation

* 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

* Different options, each with trade-offs
* Typically trading off coupling with complexity

### Direct References

* Maintain direct references to other view models and invoke methods on them
* Eg, conductor asking children if they can close
* Pros
  + Simple to follow chain of execution
* Cons
  + Increased coupling
* Usage: Request/response scenarios for logically coupled view models
* 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

* Depend on another view model and subscribe to events on it directly
* Pros
  + Minimized afferent coupling
  + Open/Closed principle – easier to add new features without modifying existing code
* Cons
  + Efferent coupling - must have a direct reference to source view model
  + Observer pattern obscures chain of execution
  + Memory management – must explicitly unsubscribe
  + Complex to use for request/response
* Usage: Prefer event aggregator for its reduction in coupling

### DEMO

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

### Event Aggregator

* <http://martinfowler.com/eaaDev/EventAggregator.html>
* Variant of the mediator pattern
* Channel events from multiple objects into a single object to simplify registration for clients.
* Similar to command processor
* Subscribe to topics on the aggregator rather than having to reference the publisher directly
* Show example -
* Pros
  + Minimized afferent and efferent coupling
  + Open/Closed principle – easier to add new features without modifying existing code
  + Simplifies event registration
  + Makes unsubscribing of events easier
* Cons
  + Observer pattern obscures chain of execution
  + Memory management – must explicitly unsubscribe
  + Complex to use for request/response
  + Can be difficult to trace usages in un-typed languages
* 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

* MVVM
  + Model – Business data, logic and rules
  + View – Application user interface
  + ViewModel – Layer above model to reduce view coupling and enable data-binding
* View and ViewModel live in client
* Model data lives in server and client
* Using a model prevents need to pass shared data between view models
* 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 \*\*