The BSP Design Model

(Behaviour , Service, Presentation)

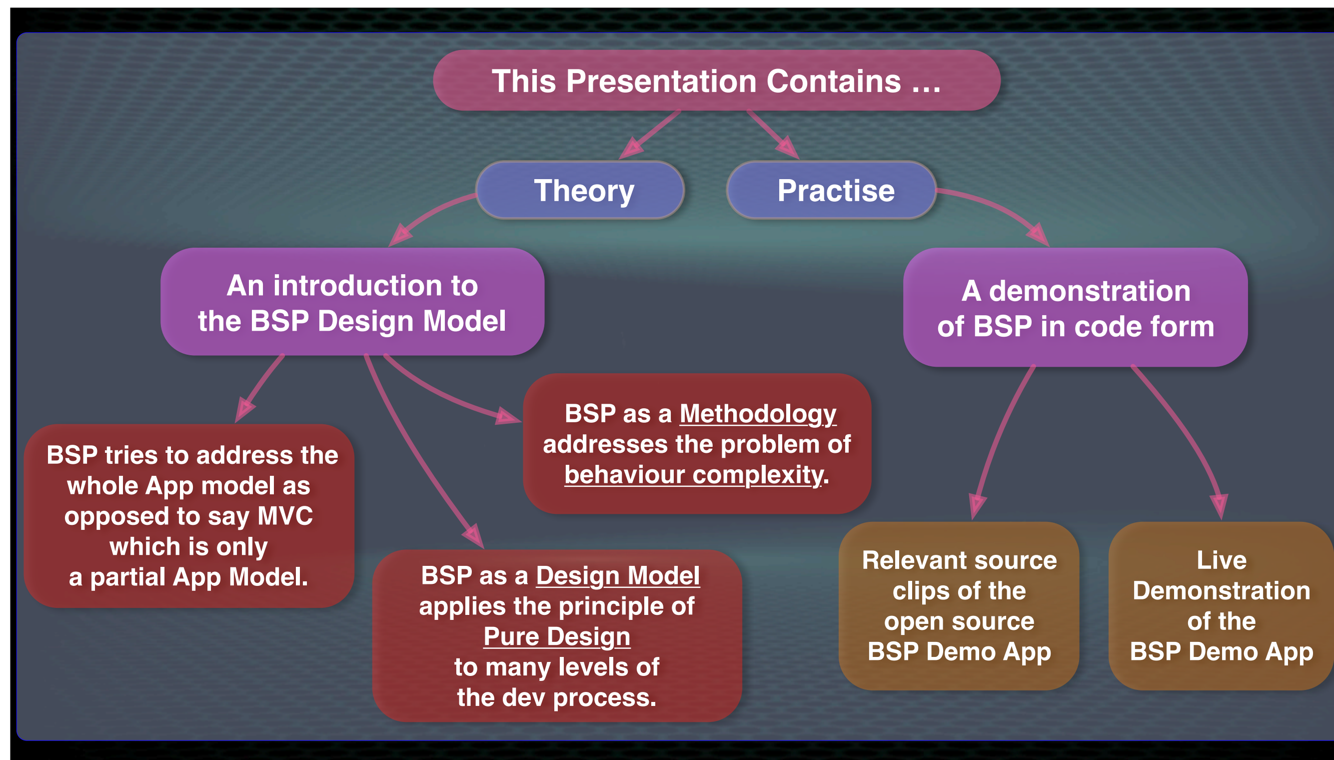
>>> See the presenter notes for the slide commentary <<<

Terry Stillone
Software Technology Research
terry@originware.com

I want to ask you to think about how much consideration you give to design, and what type of design constructs or design tools do you use.

I also want to differentiate between "whole design" that covers and supports the whole App as opposed to “partial design” where only a localised area is assigned a design (such design patterns or MVC/MVVM). These provide only a limited patched design.

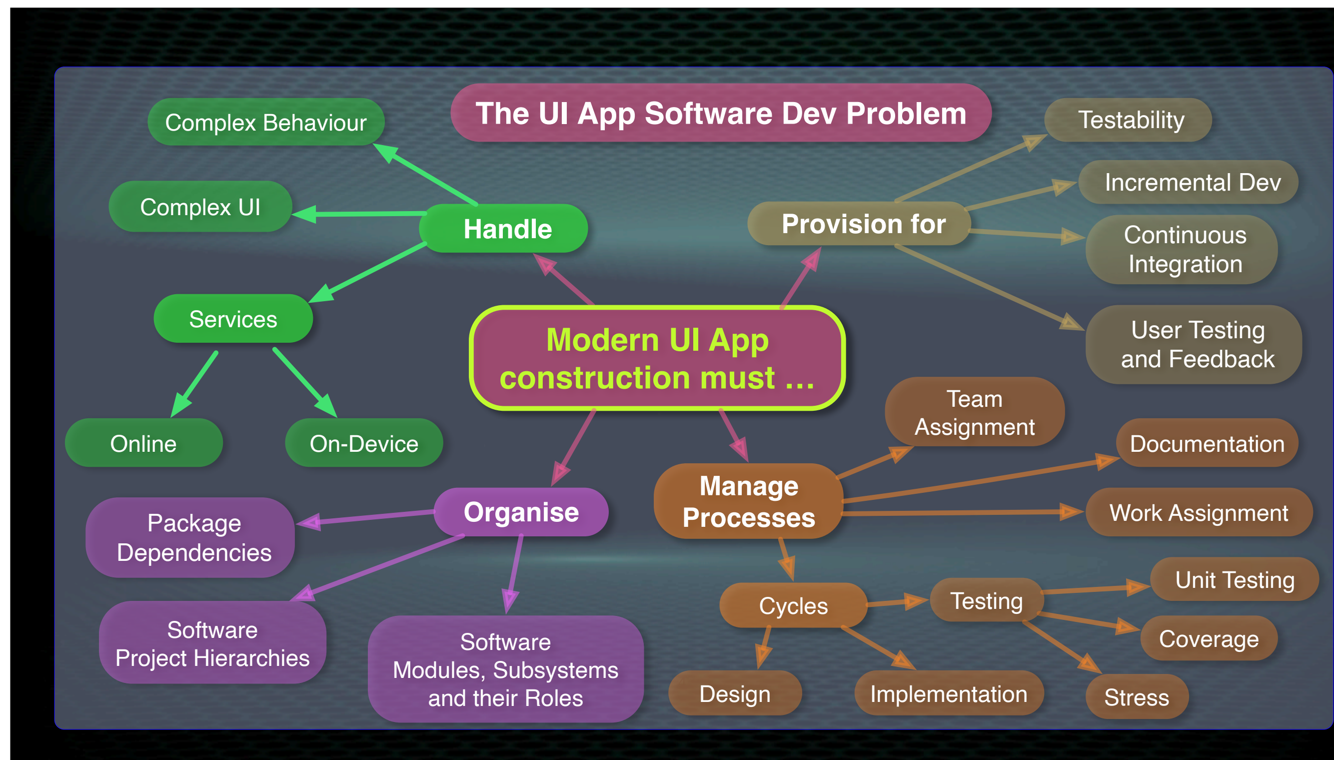
BSP is a whole design system, it supports not only the whole software system but the development process as well.



Of course the live Demo App demonstration is not available for the slide only content, but you can get the Demo App Xcode project source from: https://bitbucket.org/originware/bspdemoapp/src/master/ compile and run it for yourself.

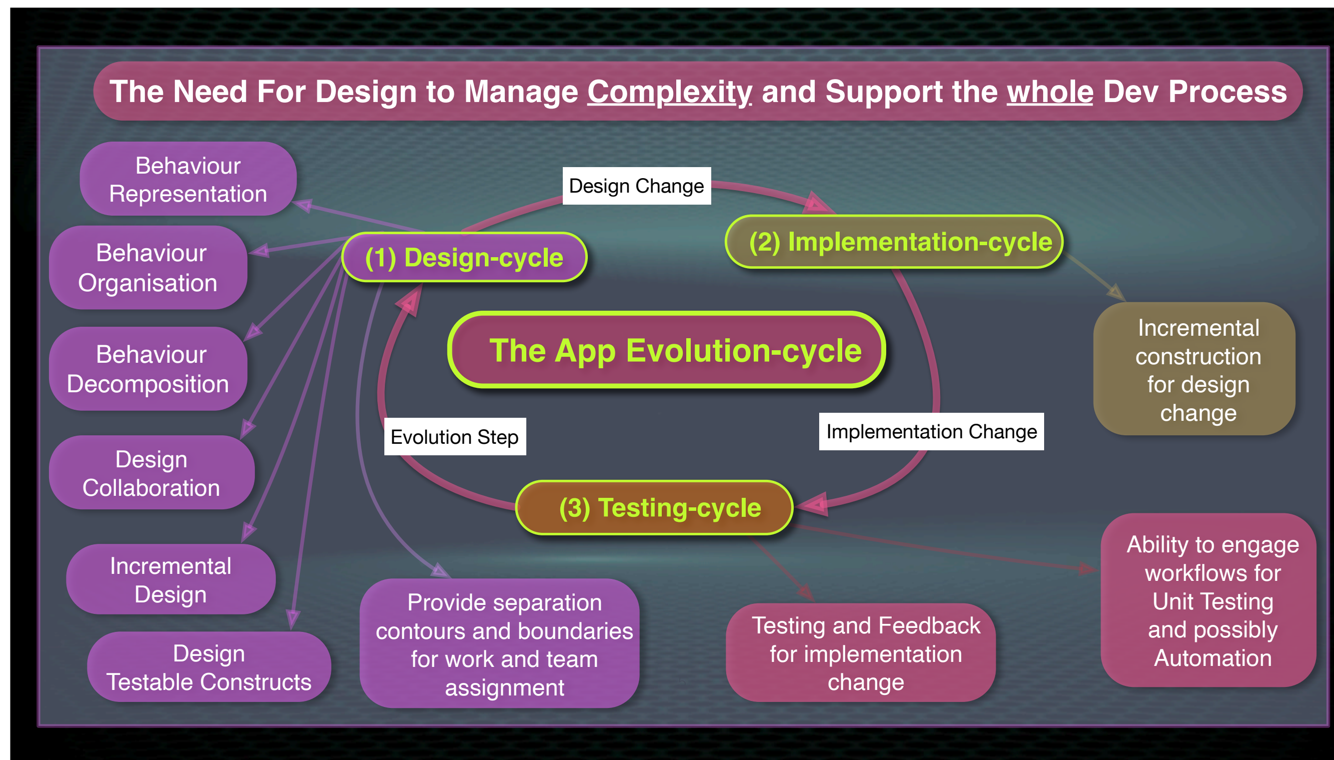
Theory Section 1 / 4

The Problems in Software Development



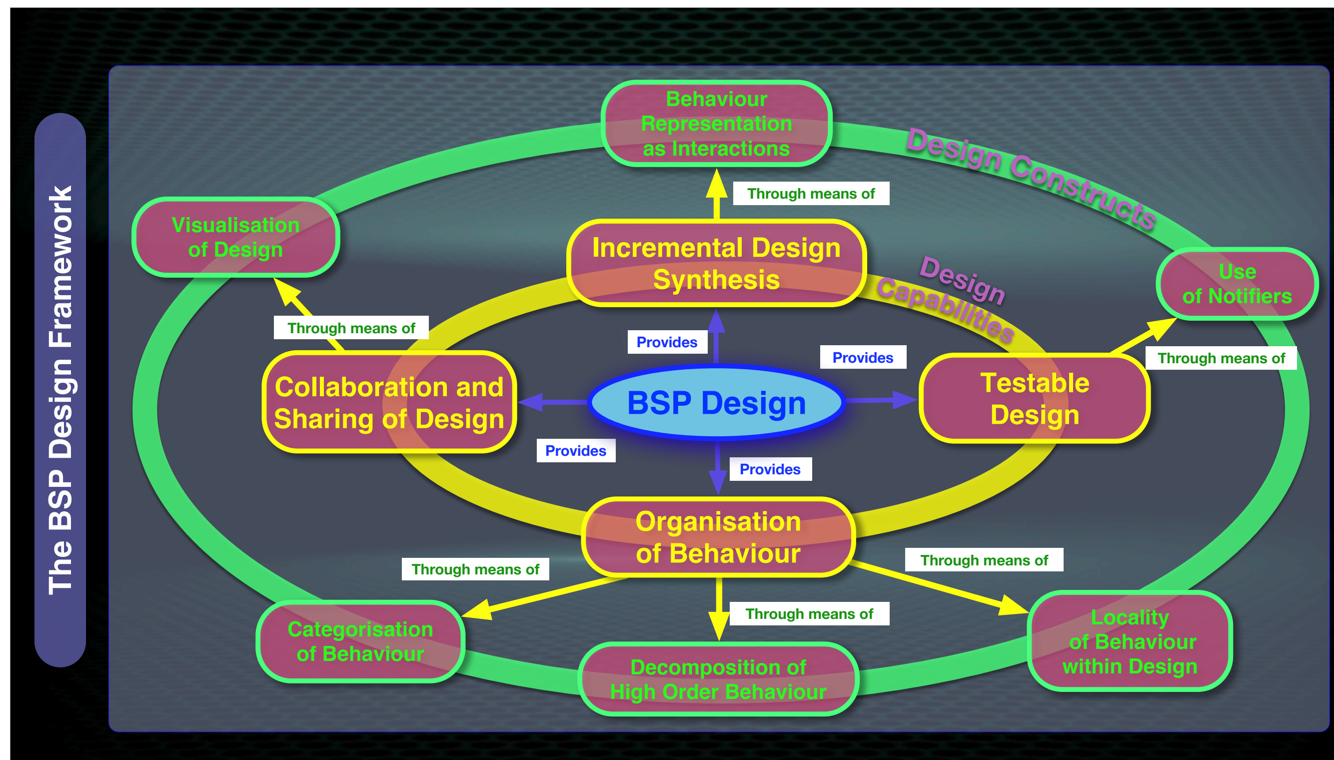
The essential problem is one of the management of complexity and complexity in all of the facets that make up modern UI App construction.

Every year, there is a greater expectation on Apps to do more, provide more, integrate with more services (both online and on-device), include more package dependencies and require more modules to operate the system. This ripples into all the various developmental processes and artefacts to incrementally increase their own individual complexity.



So BSP proposes a cyclic iterative approach driven by design. An incremental design-cycle which adds more capability or refines existing capability, that leads to incremental development (including test-development) of the design change and the cycle is completed with a test-cycle and the process is re-executed.

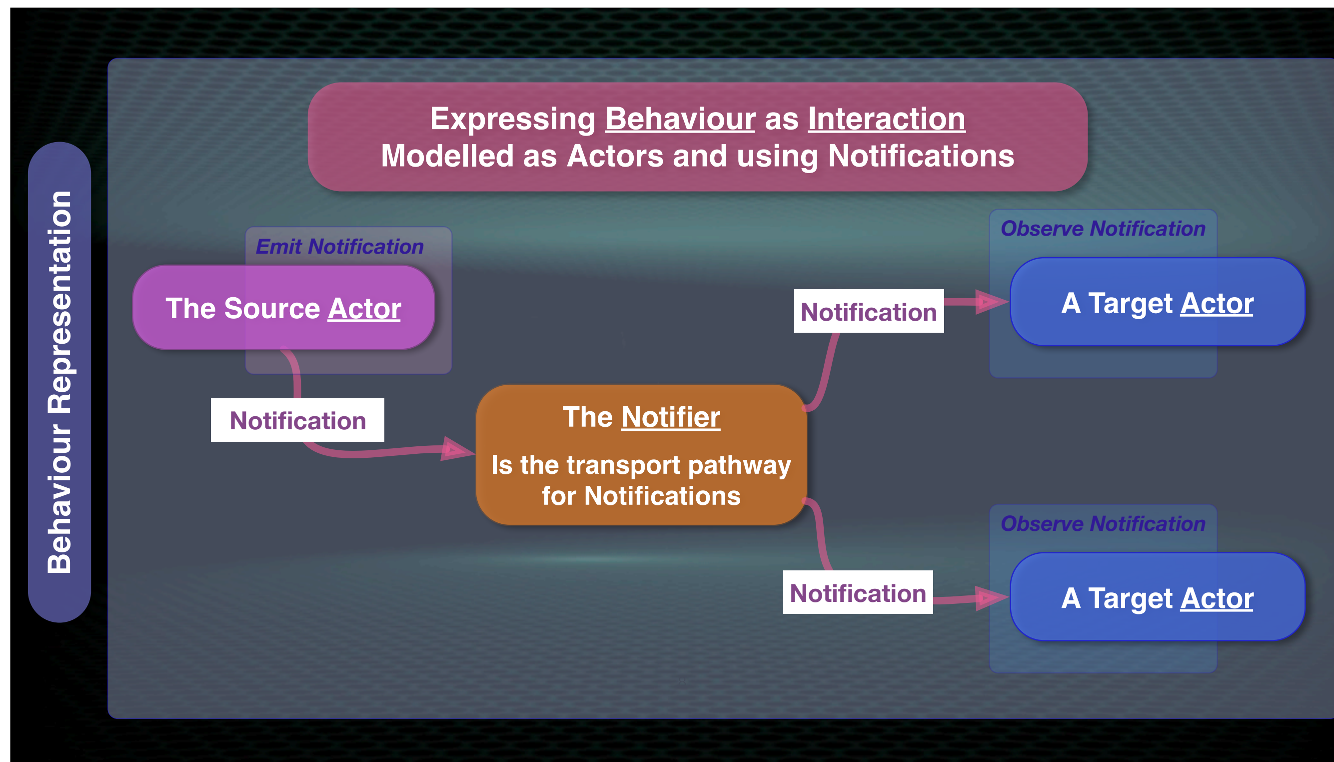
In order to do that, design must provide facilities to support that whole cycle and these are given in mauve on the left side.



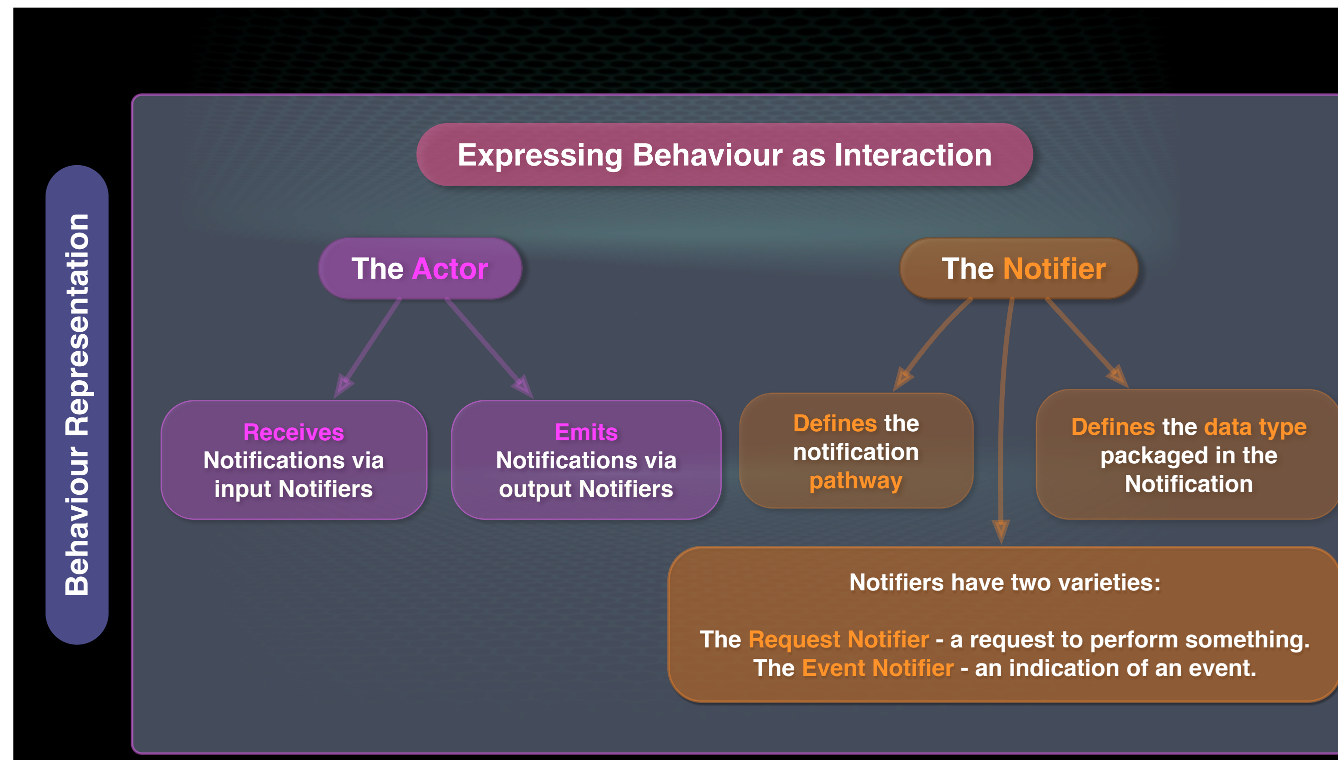
So BSP provides a set of Design Capabilities as given in the centre ring and these are supported through means of Design Constructs given in the outer ring.

Theory Section 2 / 4

Behaviour Representation

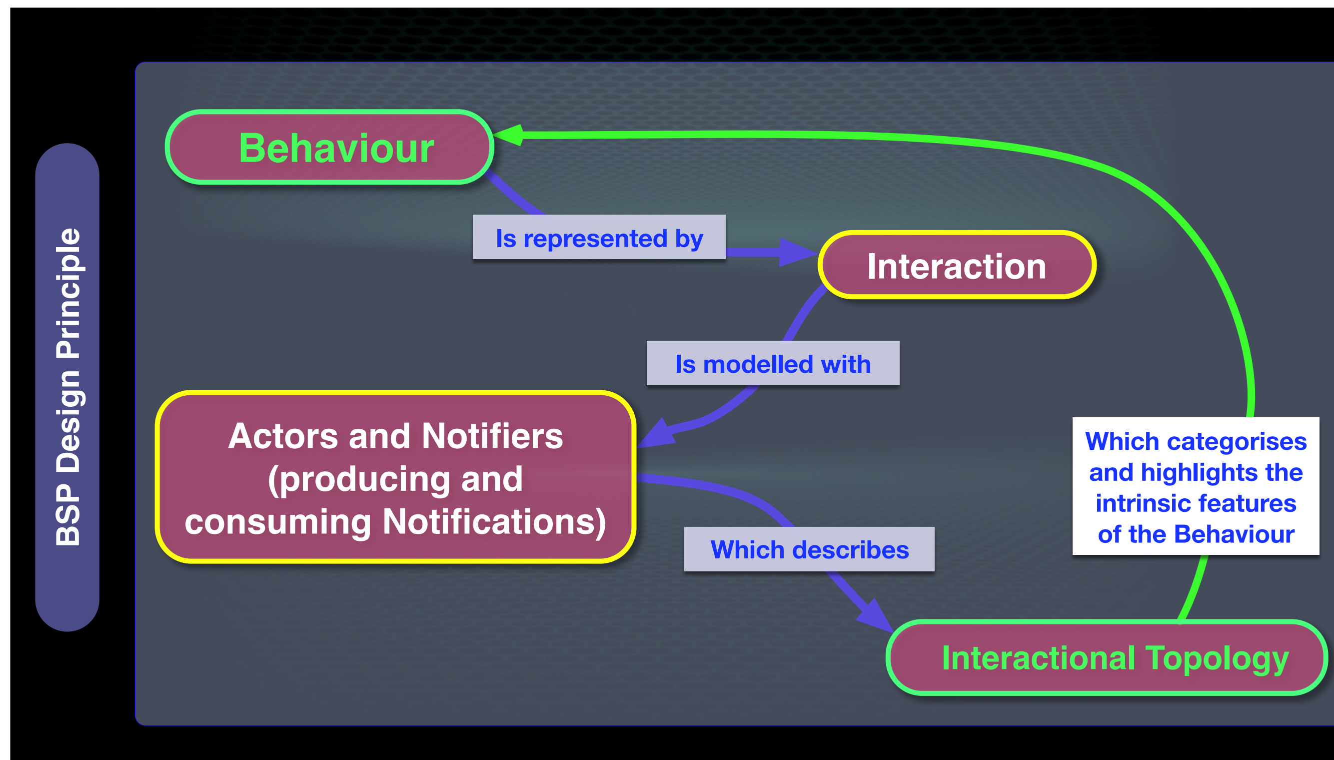


So BSP expresses Behaviour as Interactions and these interactions are modelled as Actors, interacting by means of notifications and the notifications are transported through pathways called “Notifiers”.



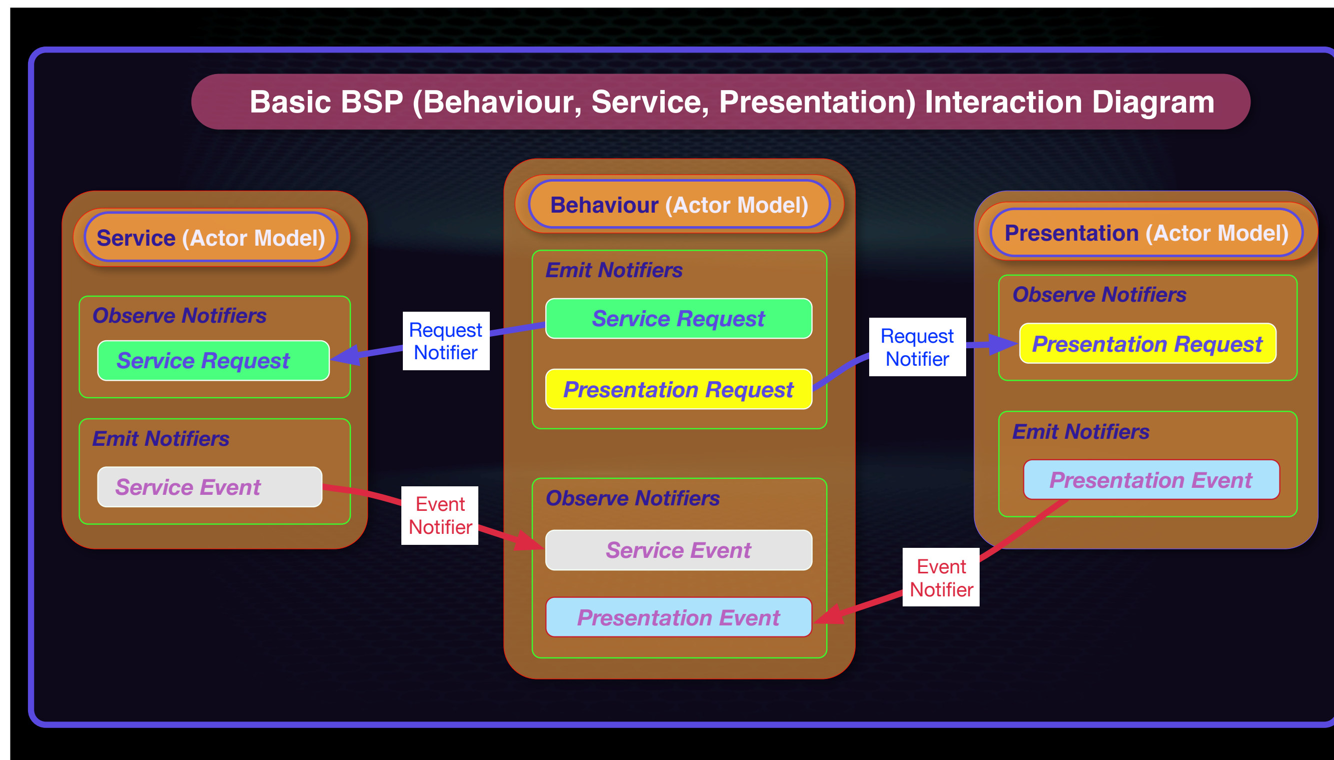
So Actors can receive notifications (i.e. they have inputs) and they can emit notifications (i.e. they have outputs). The Notifier defines the from and to pathway and they have two varieties, Request (where the notification is a request to do something) and Event (where the Actor has observed some event that it cannot act upon and so needs to indicate out the event occurrence so that it can be handled externally).

The data type associated with the notifier (i.e. its notifications) is indicative of the Request or Event that is issued.



In summary, the principle that BSP operates on is: Behaviour is represented as Interaction. Interaction is modelled with Actors and Notifiers and the connective arrangement forms an "Interactional Topology".

The Interactional Topology in turn describes boundaries and contours which reflects structure back onto Behaviour which was previously just an amorphous blob. These contours and boundaries can then be used in project management to separate out work items, team assignments and so forth.



This is the basic BSP "Interaction Model", there are three top level actors, Service (which performs OS Services), Presentation (which controls graphic elements and UI) and Behaviour.

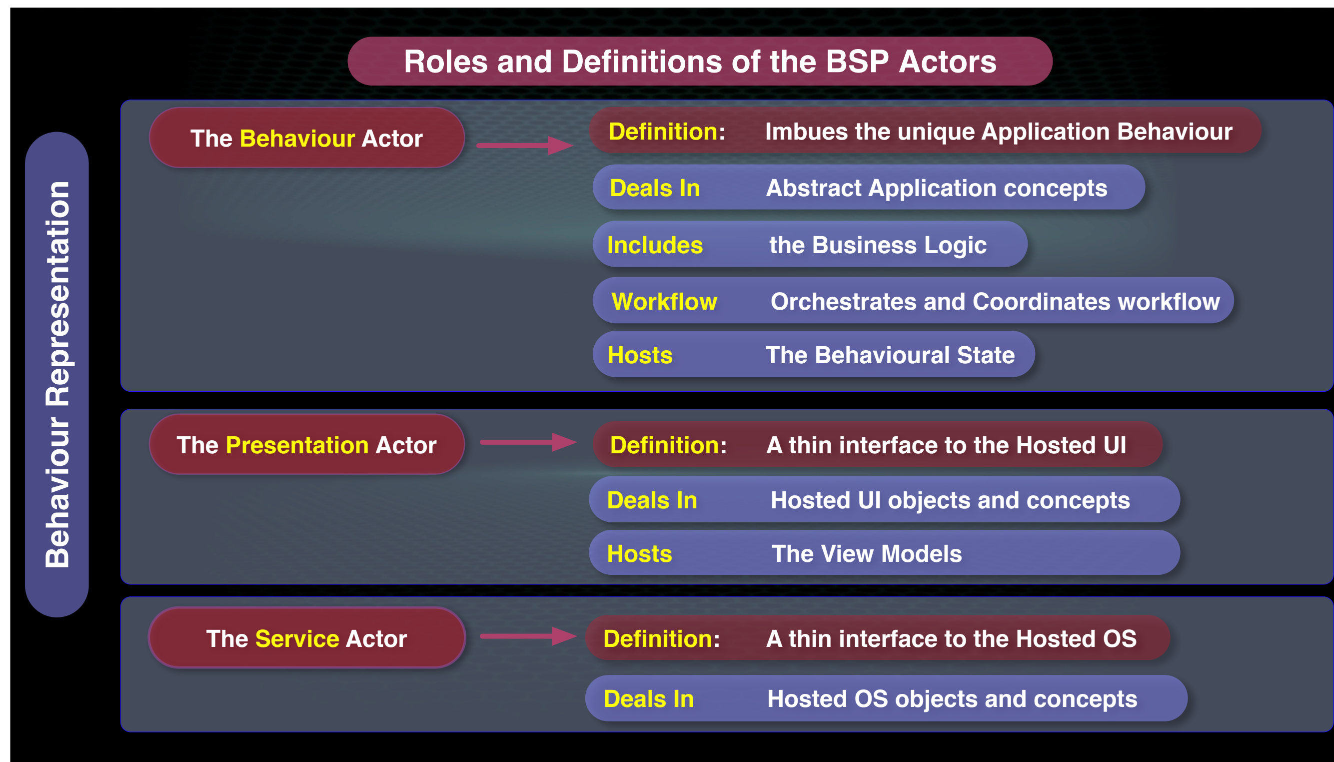
The Service and Presentation Actors both house their own Service and Event notifiers. The Behaviour Actor has no top level actors, but observes the other Actor Event notifiers and issues requests to their respective Request notifiers.

The Service and Presentation Actors are thin interfaces to the OS Services/Presentation, so they tend to have short call chains where as Behaviour is the fat workflow Actor and probably will have sub-Actors within it to handle its workflow complexity.

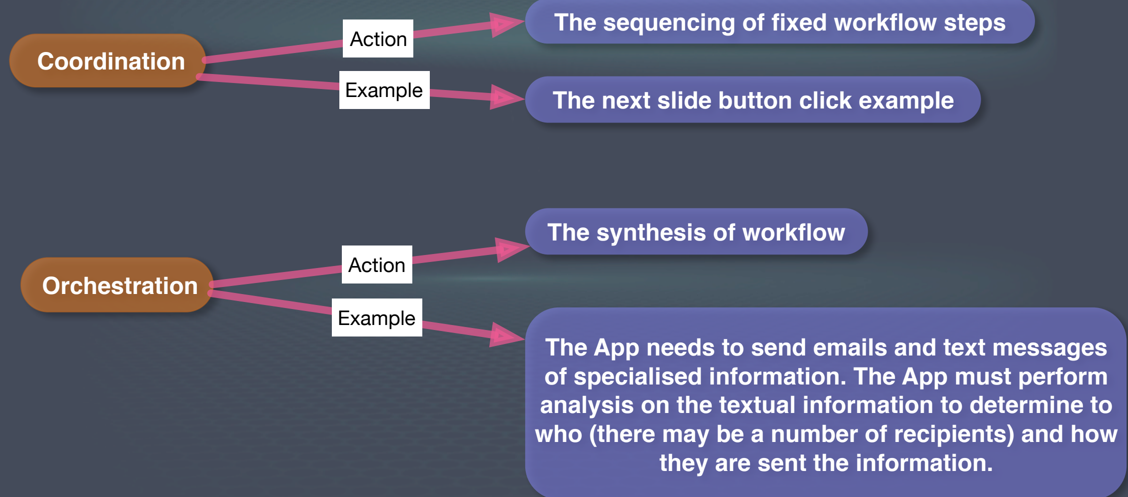
So Actors perform "Workflow" and Notifiers perform "Activity". Here activity can be engagement activity (that engages its respective workflow in the target Actor) and event activity in which an Actor is acknowledging out it has observed an event (typically being issued because it cannot handle the event). Notifiers are to go to point to instrument for monitoring/testing/automation. Where as, Actors are the go to point to determine what workflow is engaged by activity (e.g. for debugging workflow).

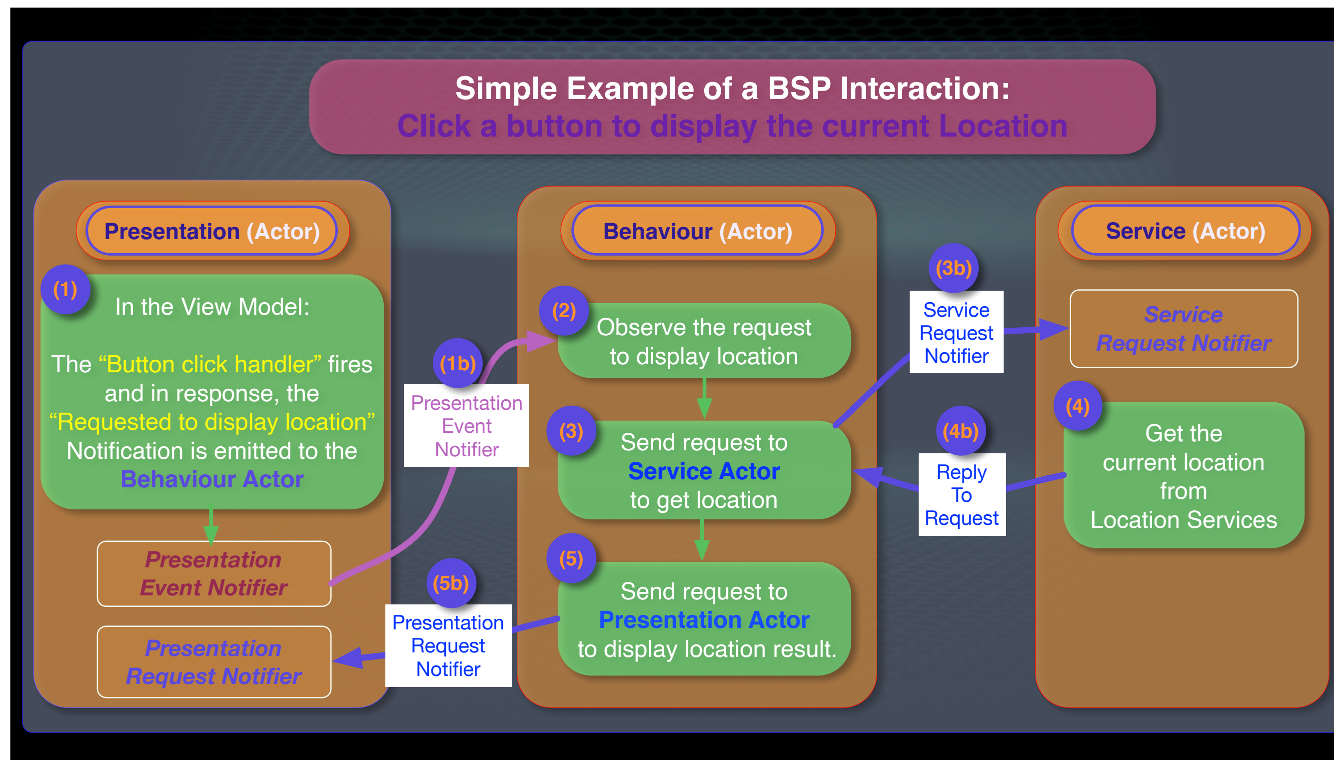
Notifiers decouple activity from workflow and so effectively decouple Actor dependencies. They also provide a natural avenue for testing Actors in isolation or grouped or together. They provide a "plugin" system that supports reusability (across separate Apps) for service and presentation workflows.

This model may seem simple, but the elegant structure exhibits natural boundaries and contours which provide a rich set of structures which can be used in work planning, work break-down and team assignment.



Here are formal definitions for the three Actors.
Note: the Presentation Actor honours MVC by hosting the view models.

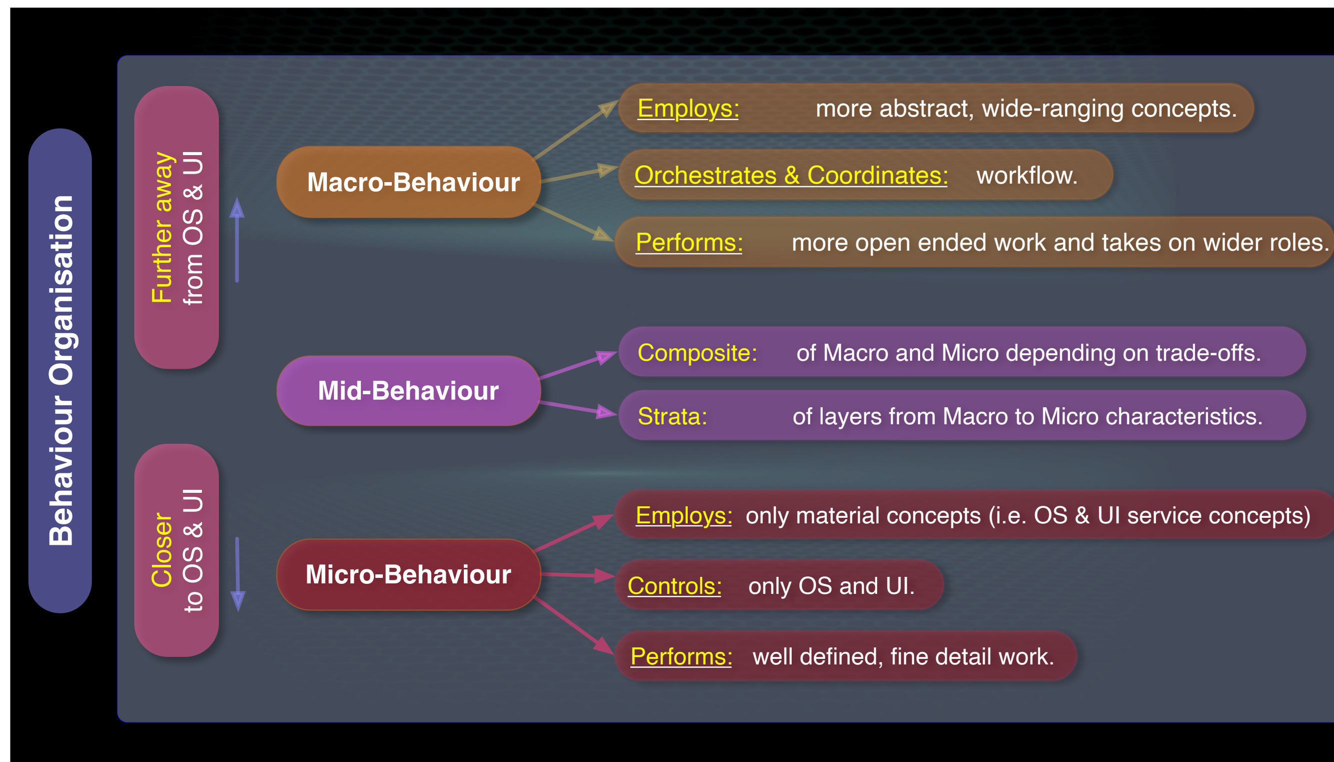

Difference Between Orchestration and Coordination
Coordination
Action
The sequencing of fixed workflow steps
Example
The next slide button click example
Orchestration
Action
The synthesis of workflow
Example
The App needs to send emails and text messages of specialised information. The App must perform analysis on the textual information to determine to who (there may be a number of recipients) and how they are sent the information.



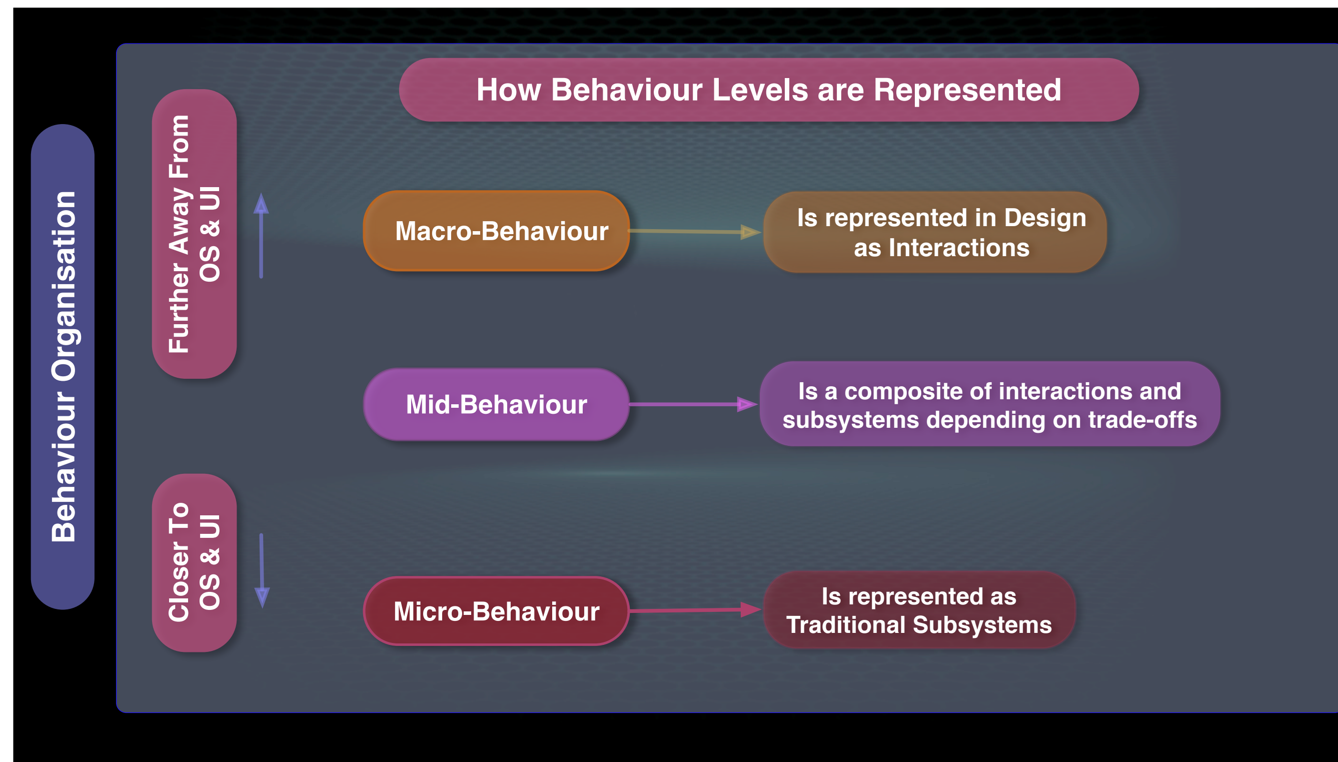
This is an interaction diagram example for a hypothetical situation where the view has a Button which when clicked is supposed to present the current location.
First, the View Model click handler for the button fires (on a button press) and the Presentation Actor workflow emits a Presentation Event Notifier event but it does not issue an event to indicate Button X was clicked, it issues an event related to the intent of the button click, so an event indicating that the Presentation Actor has received a request to show the location.
That is observed by the Behaviour Actor (that is listening to the Presentation Event Notifier), which begins its coordination workflow of requesting the location from the Service Actor and then issuing a request to present the resolved location to the Presentation Actor.

Theory Section 3 / 4

Behaviour Organisation



BSP categorises behaviour into three levels, Macro, Mid and Micro behaviour. Each has different characteristics. Mid behaviour is typically a stratified graduation from Macro characteristics to Micro.

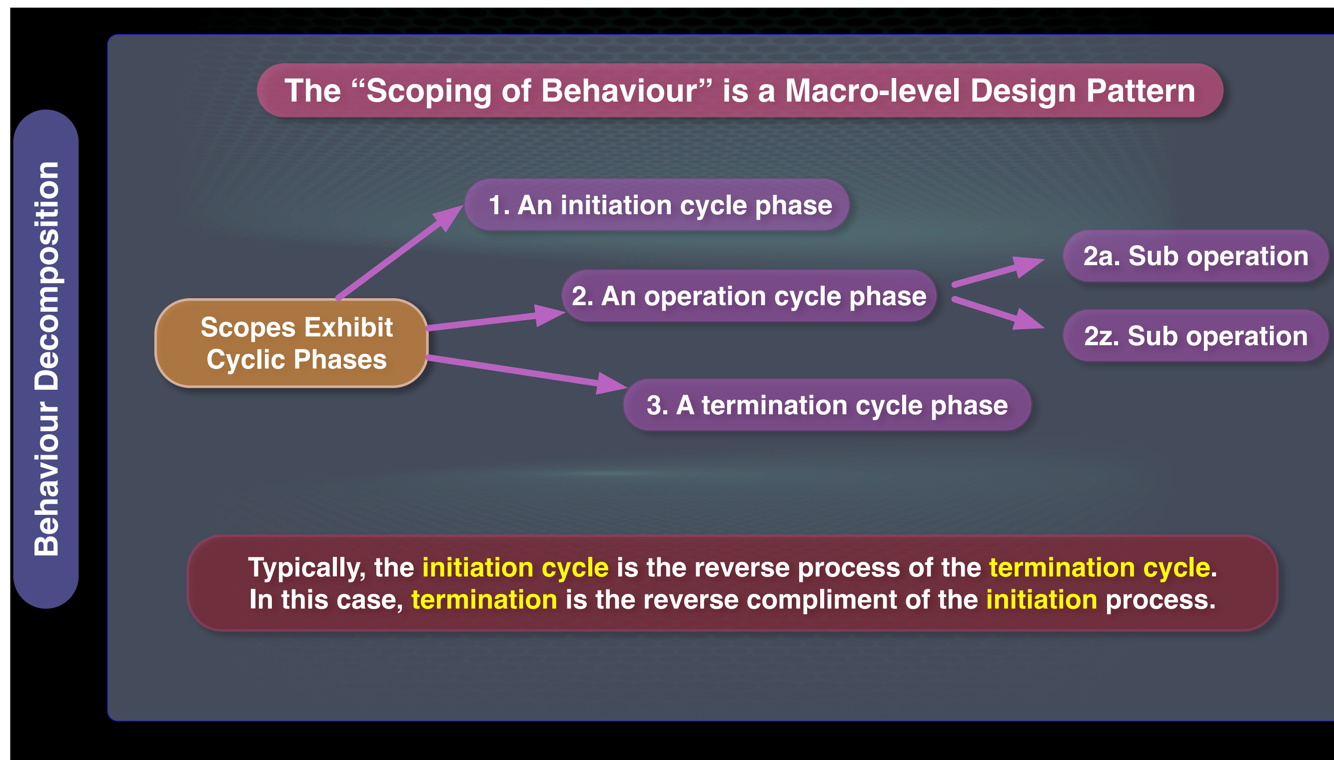


For each of these levels a Design system is assigned which matches the characteristics of the level. Macro level having more interactional characteristics is assigned an Interactional design model. Micro being more instructional in nature, and performing more fined grained, detailed work, it is assigned a traditional Subsystem design.

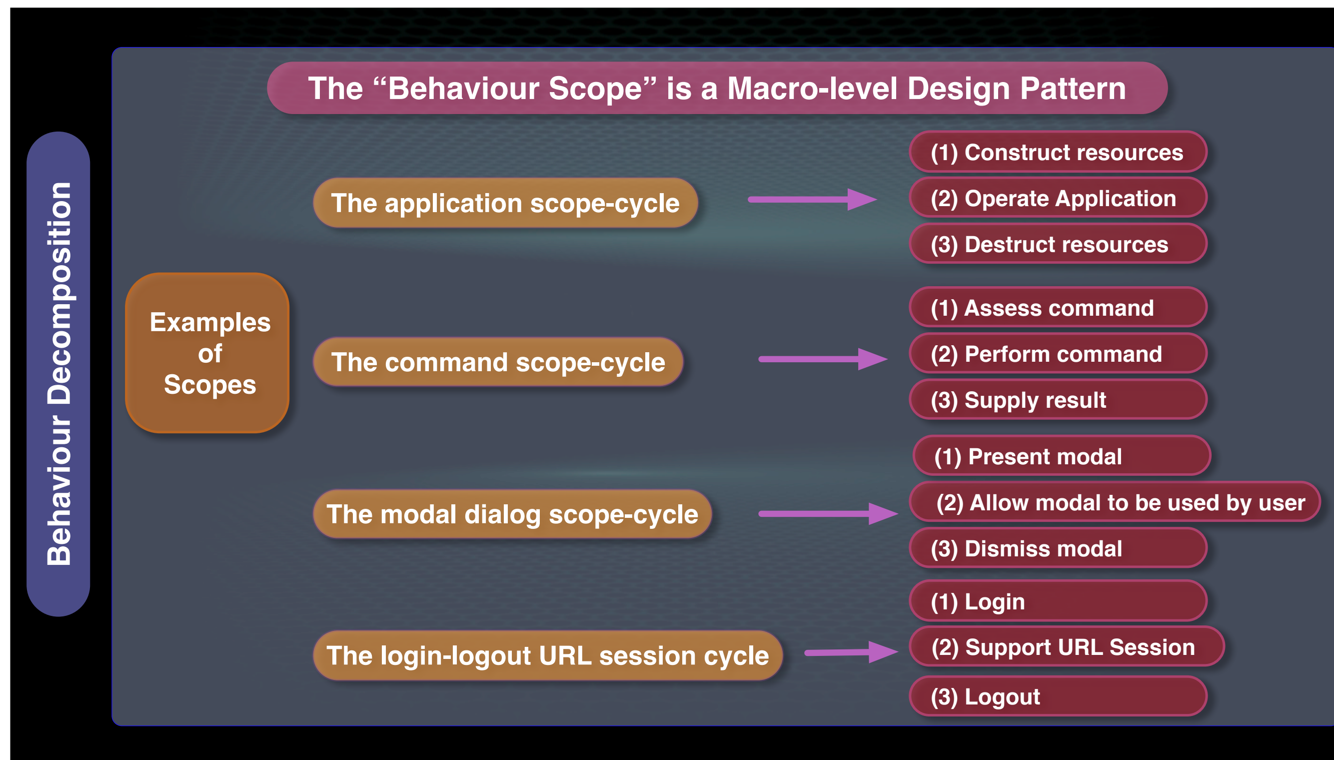
Mid-Behaviour is typically a strata of interactional layers to subsystem layers.

Theory Section 4 / 4

Behaviour Decomposition Into Scopes

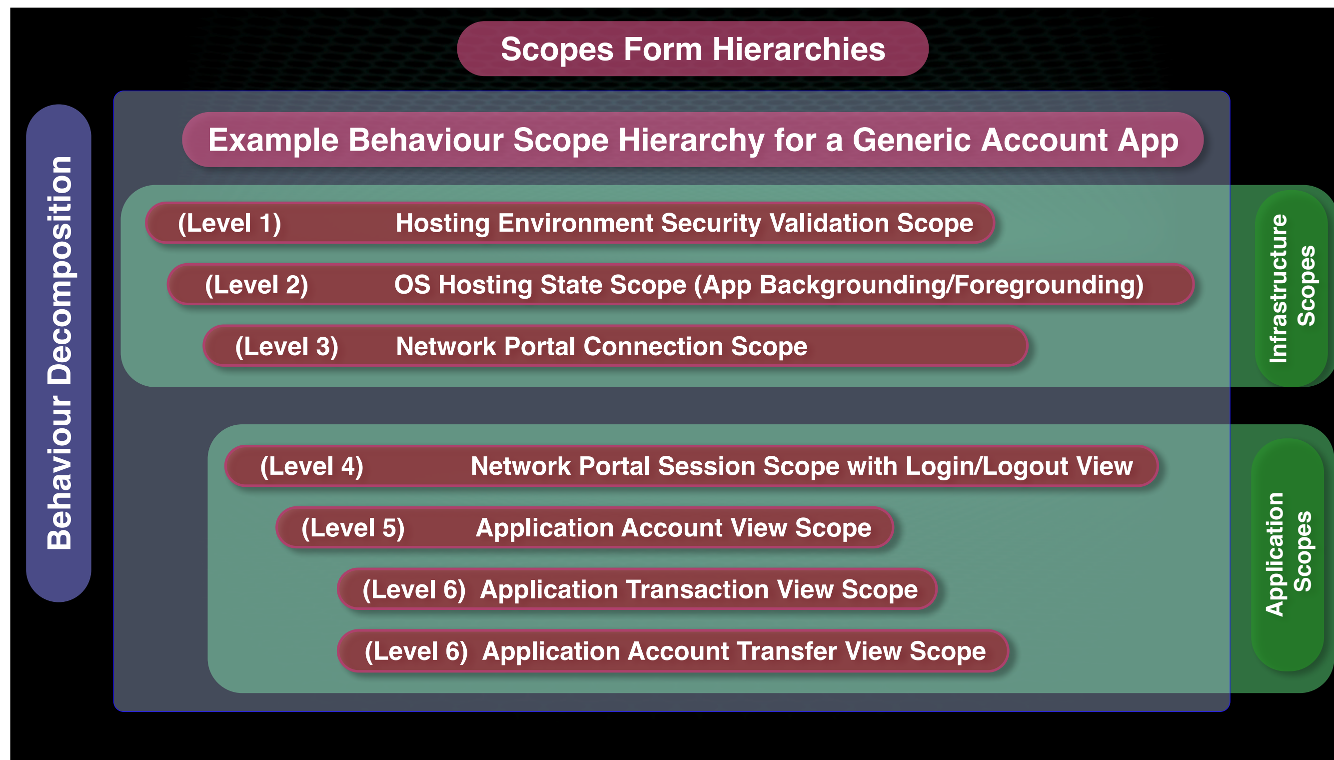


BSP decomposes high order behaviour into what is called "Scopes". These "Scopes" exhibit a three phase cycle: initiation, termination and in between an operational phase that may include a number of sub-operations.



Here are some typical examples of Scopes, that you should be familiar with. Scopes can be appropriate in application life cycle contexts, executional contexts, graphical contexts and network session contexts.

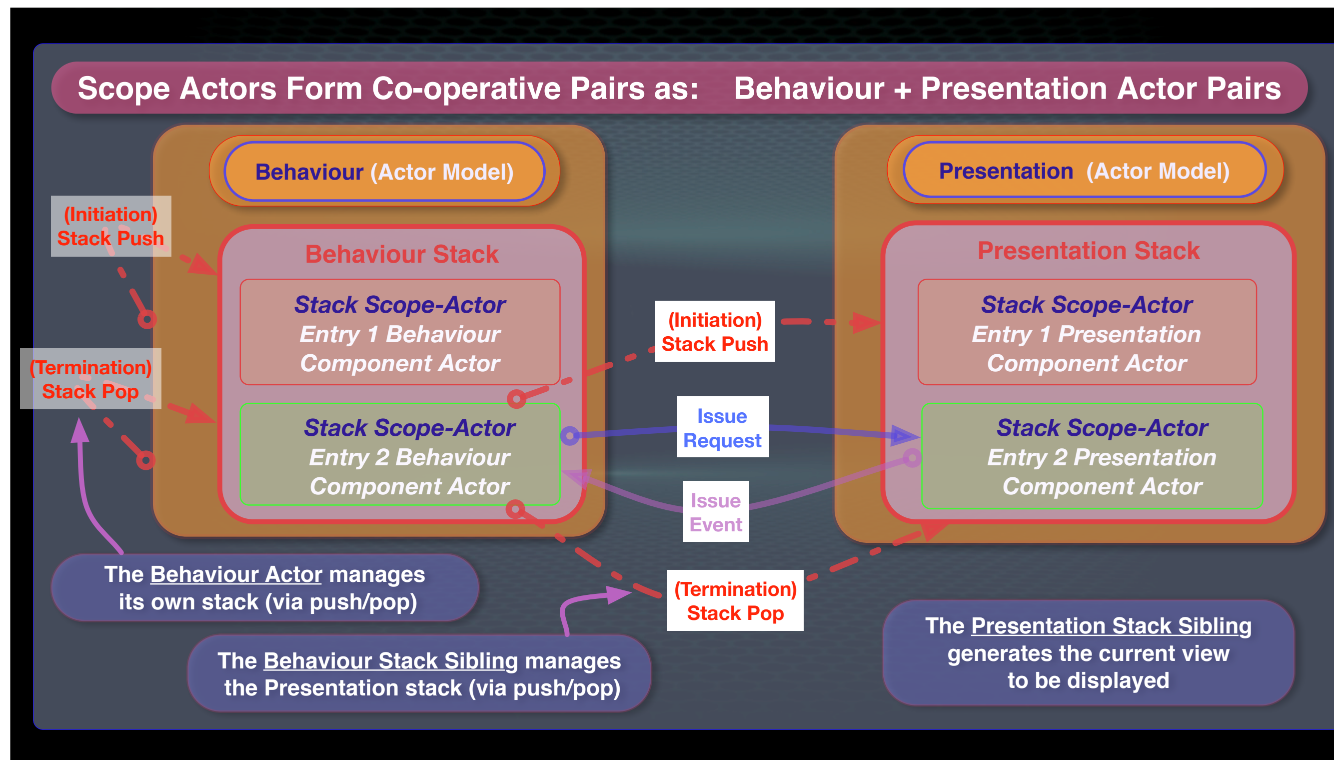
In many times, the termination cycle is the reverse process of the initiation cycle.



This is an example of a Generic Account App that connects to a portal service and the diagram describes the scopes that support it.

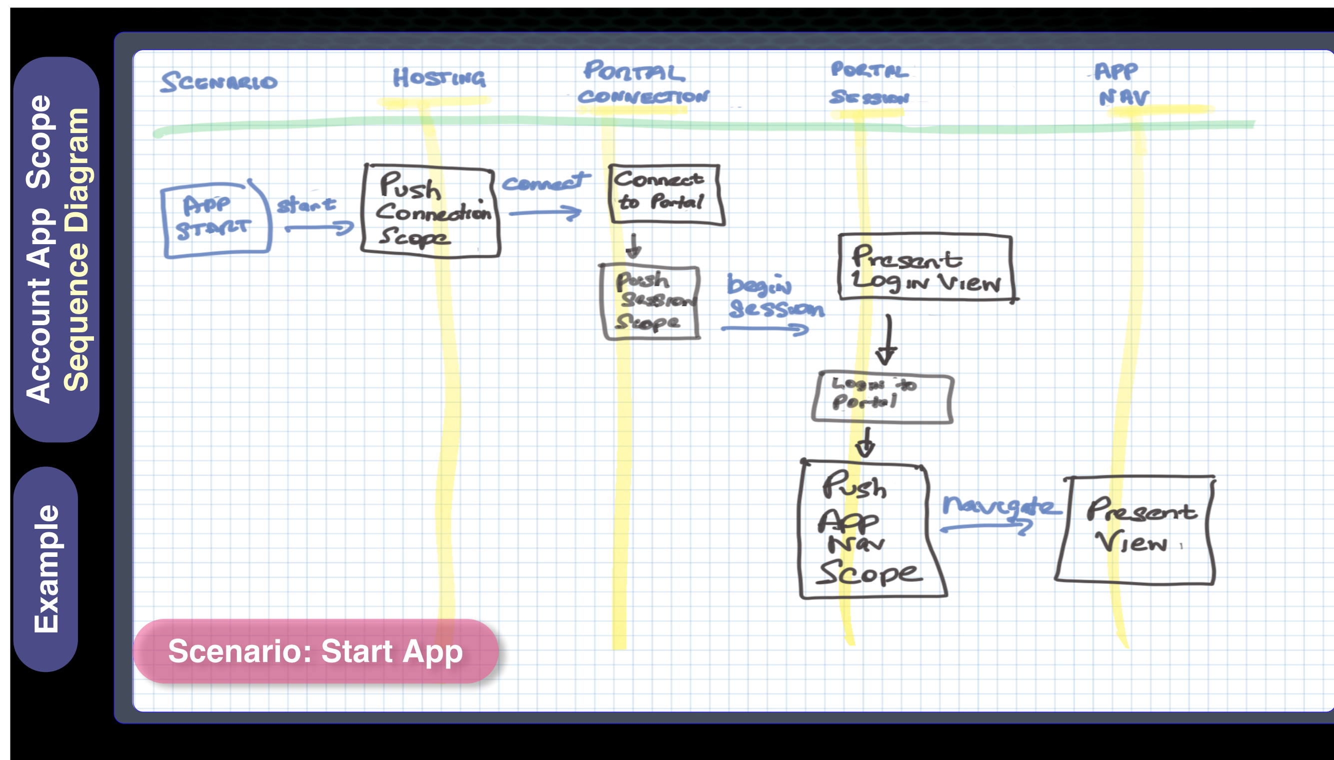
The first scope layers are infrastructural and have minimal view support while the second set are for the Application workflow and have full associated views.

The Hosting environment scope checks for the existence of certificates, keys and the integrity of the host device.
The Network Portal Scope performs discovery on the portal, checks for service availability and then forms a secure encrypted connection.
The Network Portal Session Scope displays the Login/Logout view and allows the user to authorise a valid user portal session.



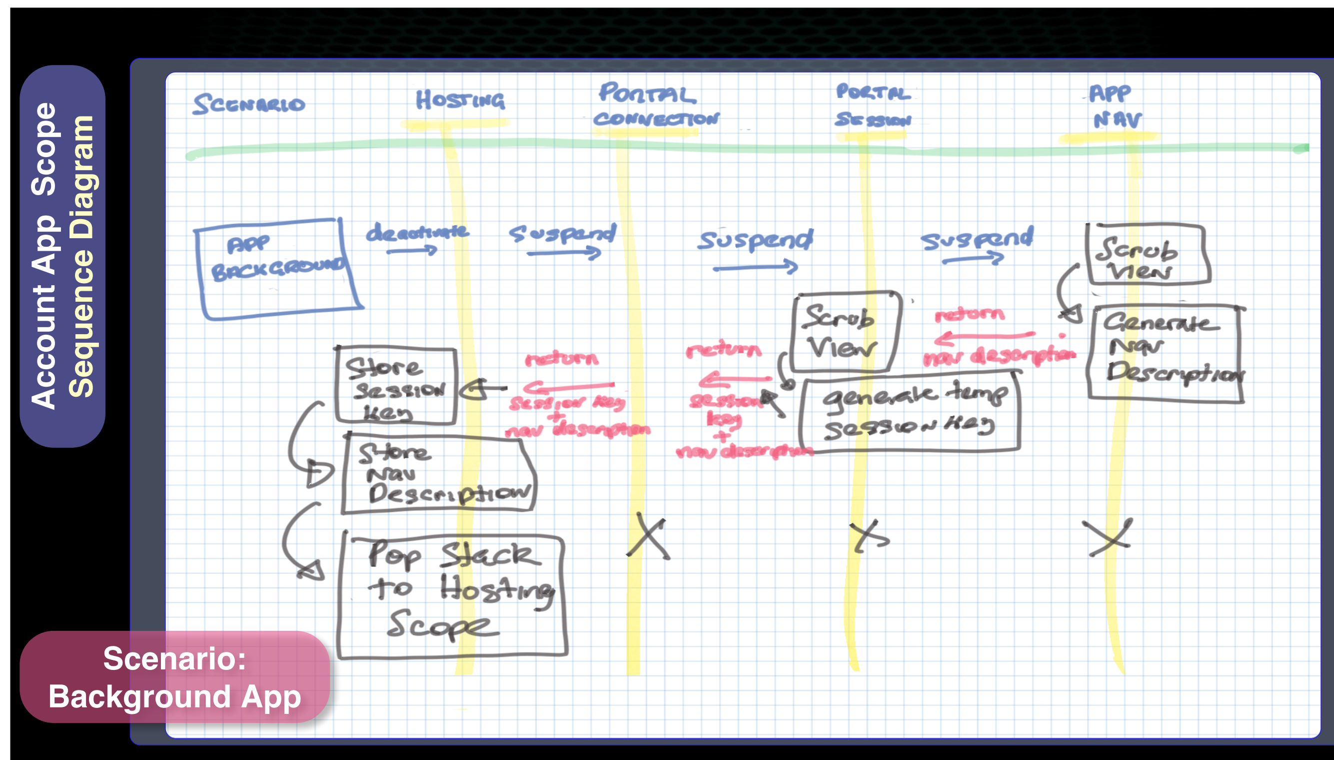
Scope are implemented with scope stacks as given in the diagram, there is a Behaviour Scope Actor (that handles the unique behaviour of the scope) and a Presentation Scope Actor that handles the View set associated with the Scope. The scope companion Actors interact through the Presentation Scope Actor Request and Event Notifiers.

Note: The Behaviour Scope Actor may need to interact externally and so may need to receive external requests and may require its own request notifier. This depends on how the Scope Behaviour Actor is engaged, if it needs external engagement or just be event triggered by its companion Presentation Scope Actor.



Here is a sequence diagram for the previous example Generic Account App. It is written by hand to demonstrate how the interaction sequence can be sketched out. There are three slides in this sequence diagram set, one for the the scenario of App startup, App backgrounding and then App foregrounding.

The sequence diagrams start at the Hosting scope and end on the first Application scope for brevity.

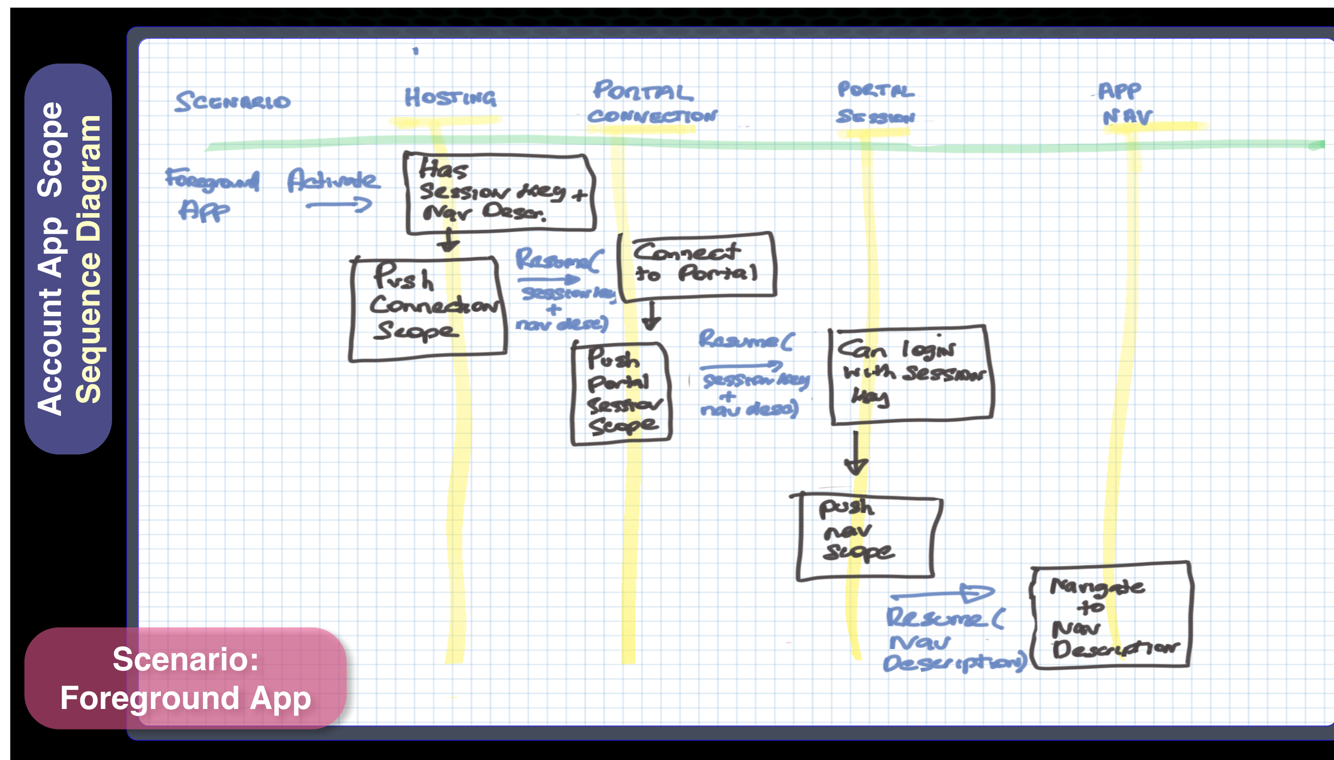


This is the sequence for App backgrounding.

Lets say there was a set of requirements placed on the App:

1. To scrub all data (including views) when the App backgrounds, for security reasons.
2. An allowance for the App to perform a one-time quick session re-login on App foregrounding if the portal allows it, based on disconnection duration.
3. The View based scopes must be able to re-navigate back to the original Nav View but do not need to reconstruct all the view-data that was in the original Nav View at backgrounding time.

So at backgrounding time, the hosting scope issues a suspend request which is propagated down stream. The view based scopes scrub their data (on the suspend request), record their Nav View position (which they hand back in the notifier result) and they all pop off their scope Actors. The Portal Session Scope also obtains the one time session key and hands it in the request result to the Hosting scope. The App formally backgrounds on the Hosting scope being active.



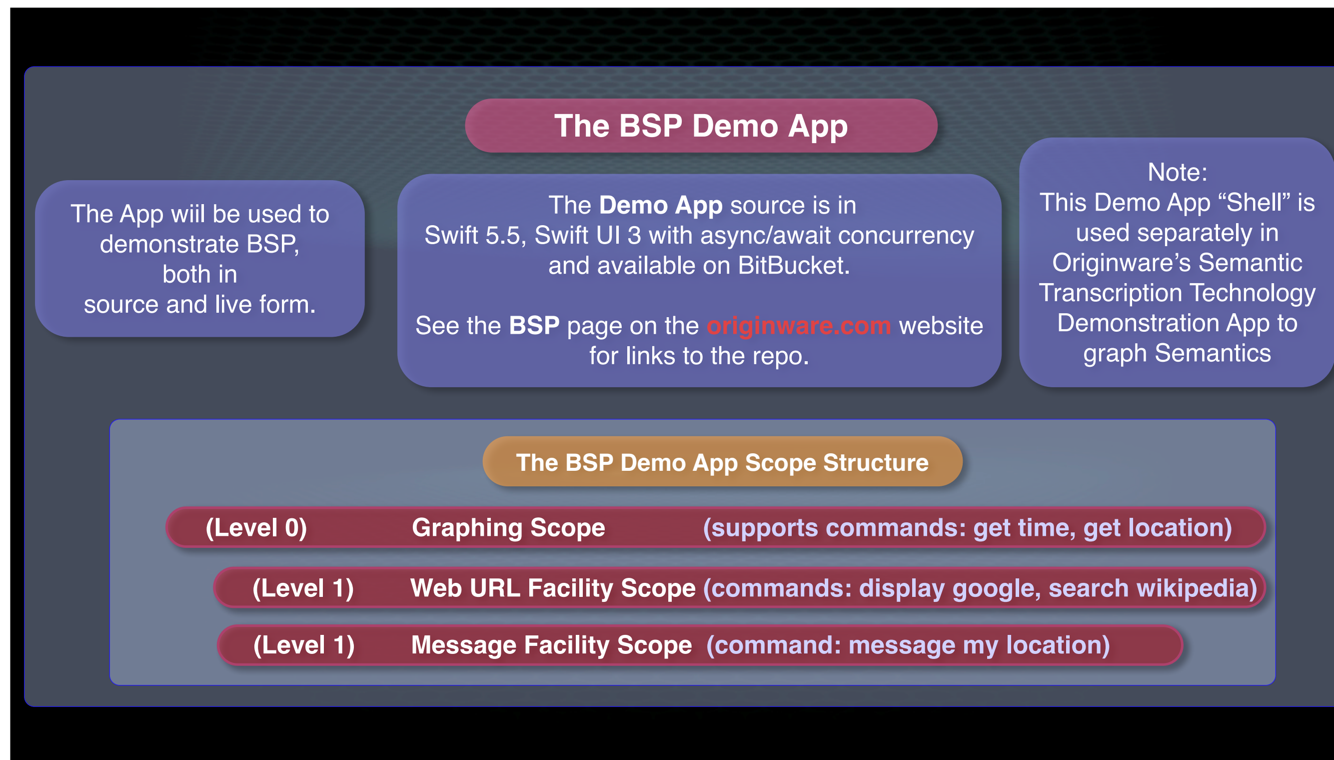
This is the sequence for App re-foregrounding.

The App resumes in the Hosting Scope and the scope issues a resume request with the session key and nav description. The resume request is propagated down stream.

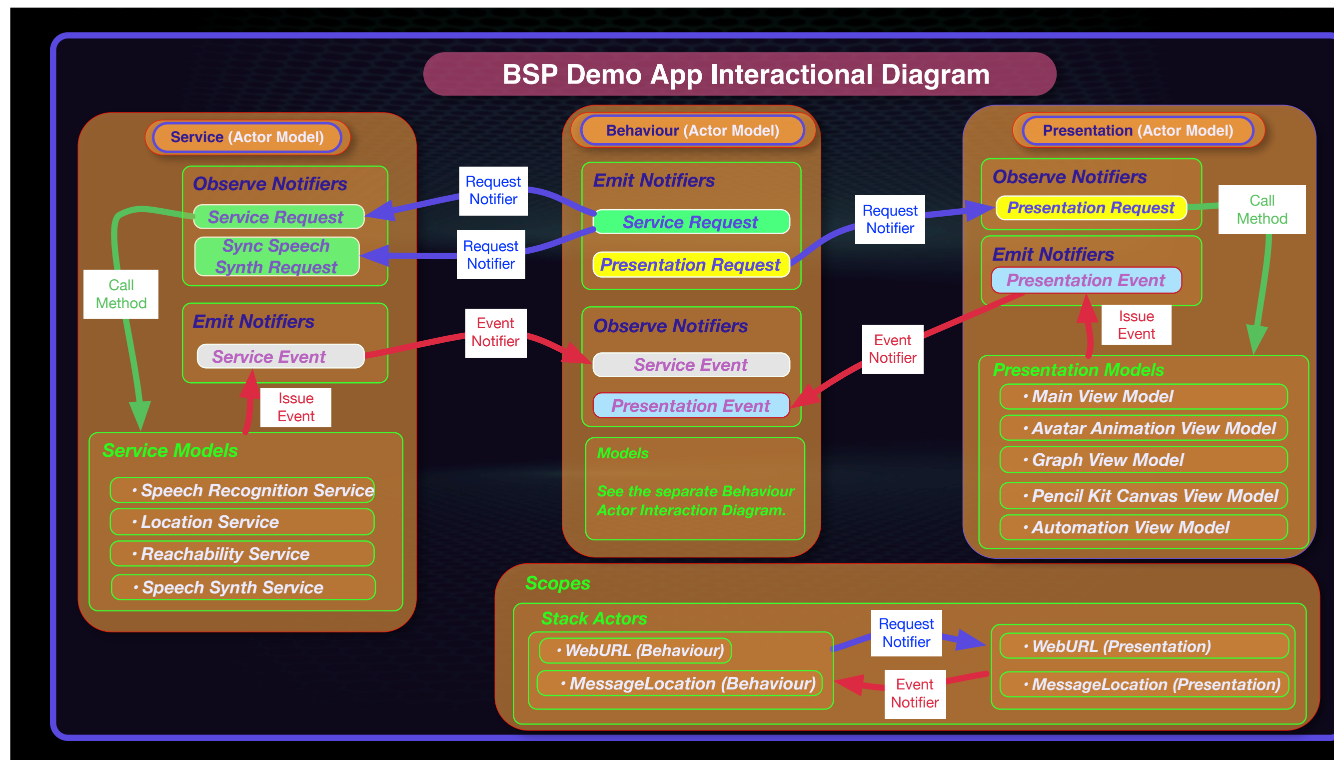
The App is supposed to try to perform a quick re-login using the stored one-time session key (if the portal allows it) and then go to the recorded Nav View position.

If there is no session key or the portal does not accept the session key, then the Scope progression stops at the Portal Session Scope with the Login/Logout view presented.

Practical Section

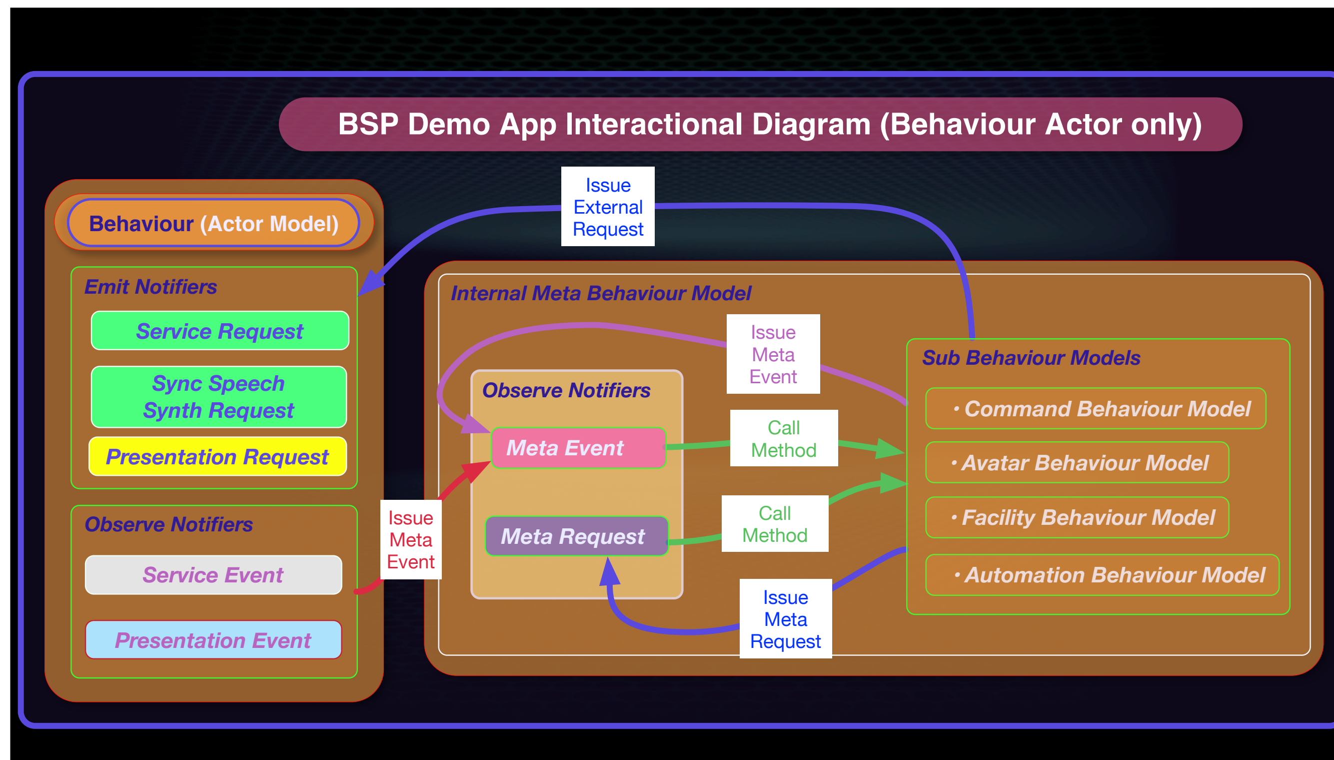


The Demo App Xcode project source is in: https://bitbucket.org/originware/bspdemoapp/src/master/ you can compile and run it for yourself.

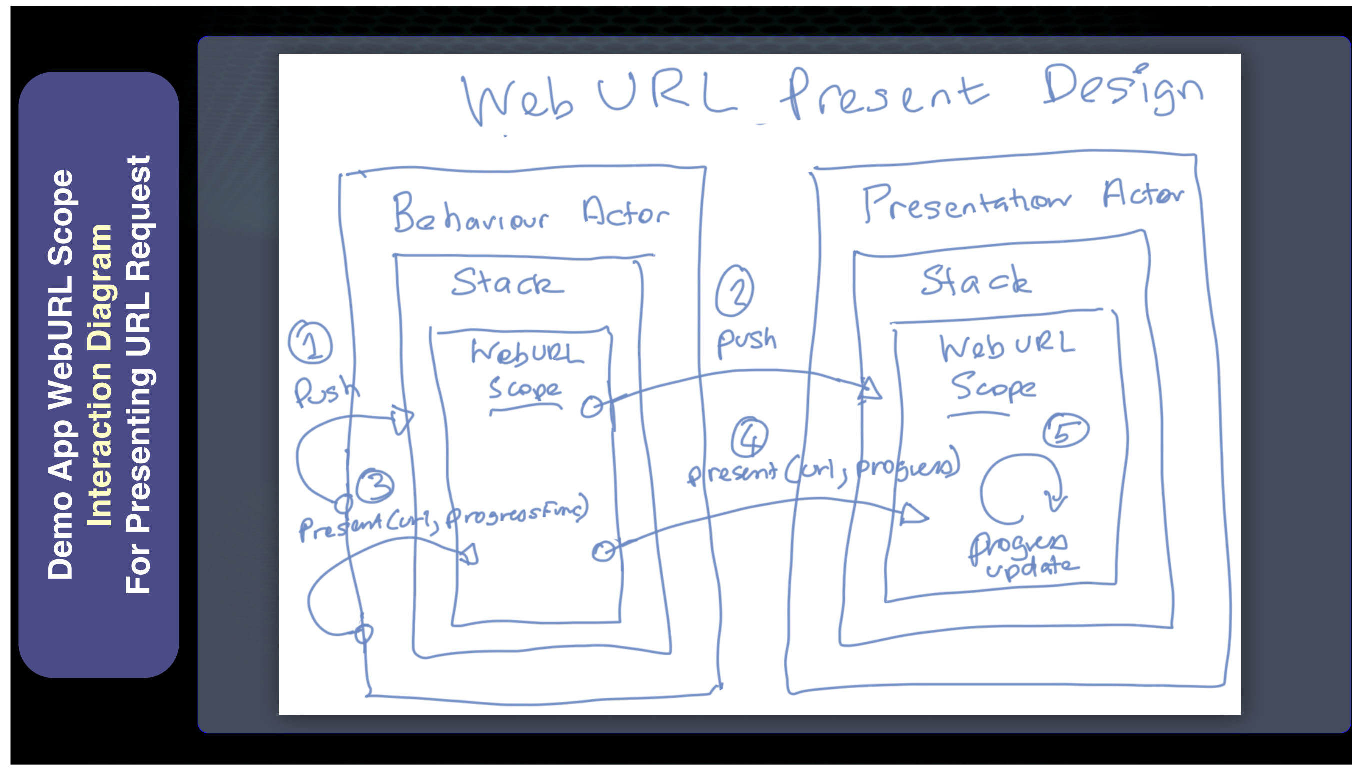


There are two separate diagrams for the BSP Demo App (see the next slide).

This slide details the internal models of the Service and Presentation Actors. The next slide details the Behaviour Actor.



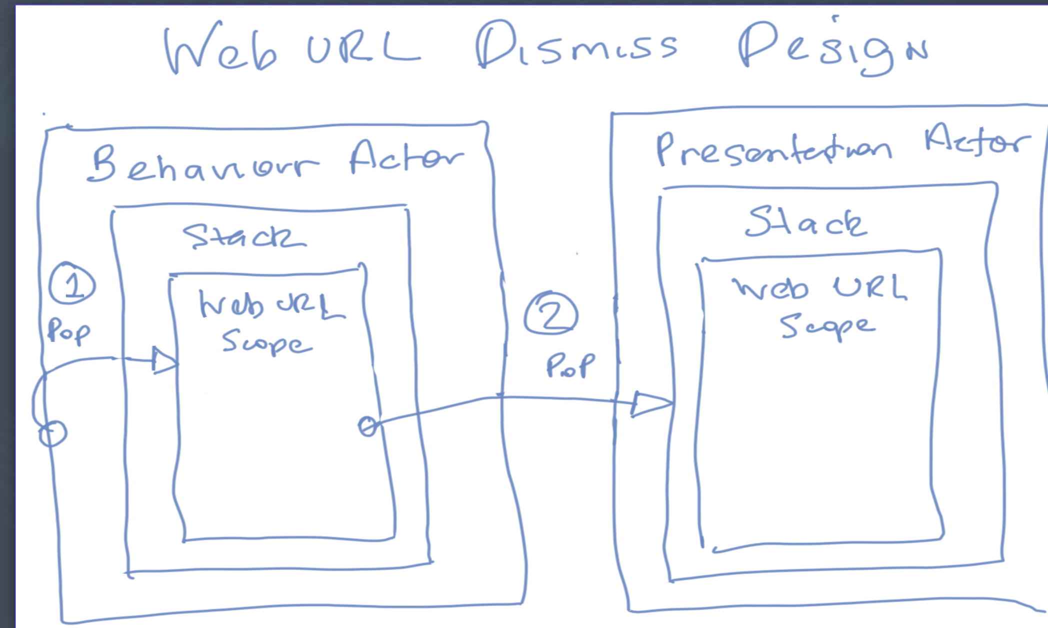
The Behaviour Actor employs an internal “Meta Actor” so that its internal workflow has access to all the capabilities encapsulated within the Behaviour Actor, such as requesting the Avatar to voice a vocal string.



Again, another hand version of an Interaction Diagram for presenting URLs with the WebURL scope. This diagram includes both the scope stack push and requesting for a URL to be presented.

Note: The Behaviour Actor workflow is responsible for pushing the WebURL scope Behaviour Actor onto the Behaviour Stack and then when this scope actor is activated, it pushes the companion Presentation Scope Actor onto its Presentation Stack.

The WebURL scope only has one request, that is to present a given URL with a progressFunc callback closure to be exercised on the fetch, download and completion events.



The hand version of an Interaction Diagram for dismissing the WebURL scope and view.

There are no requests associated with this sequence. This sequence is engaged when the WebURL Scope Behaviour Actor is deactivated by popping it off the behaviour stack, it in turn deactivates the Presentation Scope Actor and pops it off the presentation stack.

As a side comment, while this WebURL capability was implemented as a scope and it was implemented this way for demonstration purposes, in practical terms, it could have easily been implemented as an un-scoped facility.

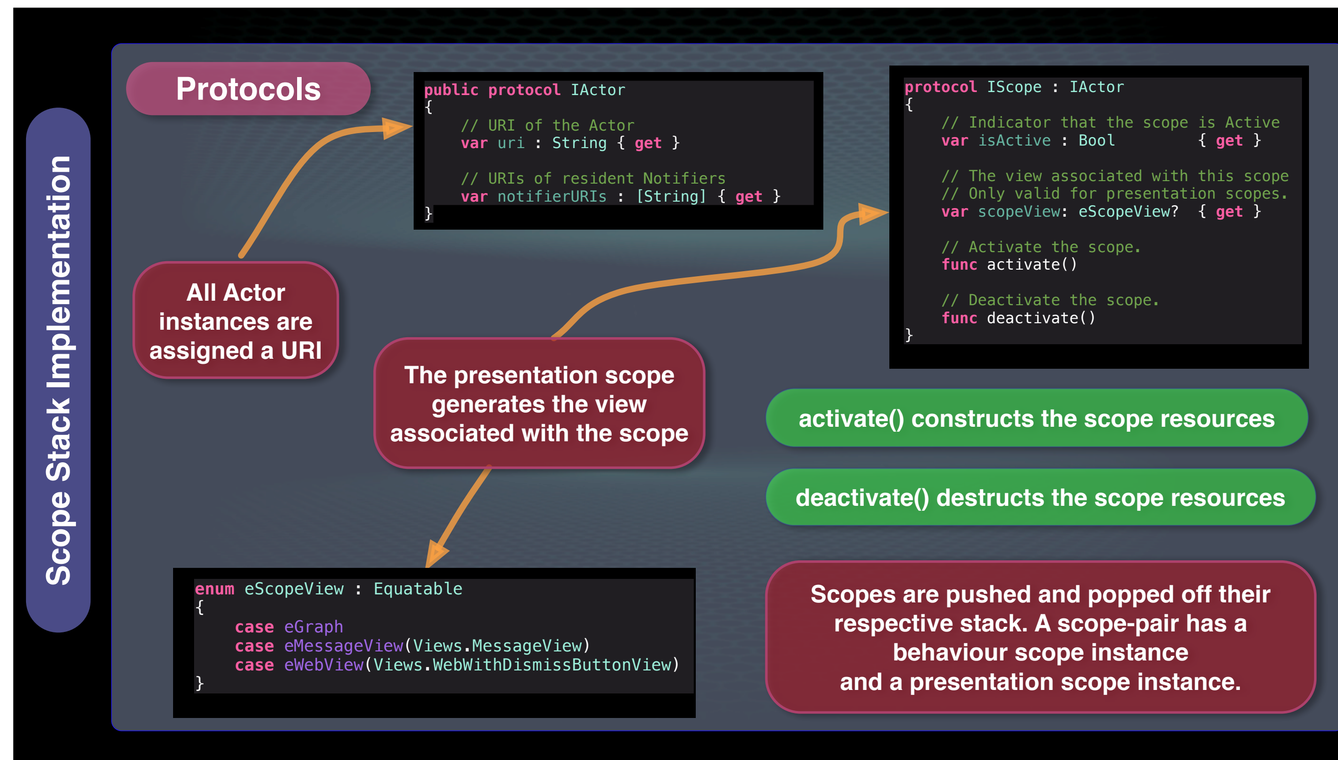
The BSP Demo App Live Demonstration Of Scopes

The **BSP Demo App** source (Swift 5.6) is available on BitBucket.

See the **BSP** page on the **originware.com** website

for links to the BSP Demo App repo.

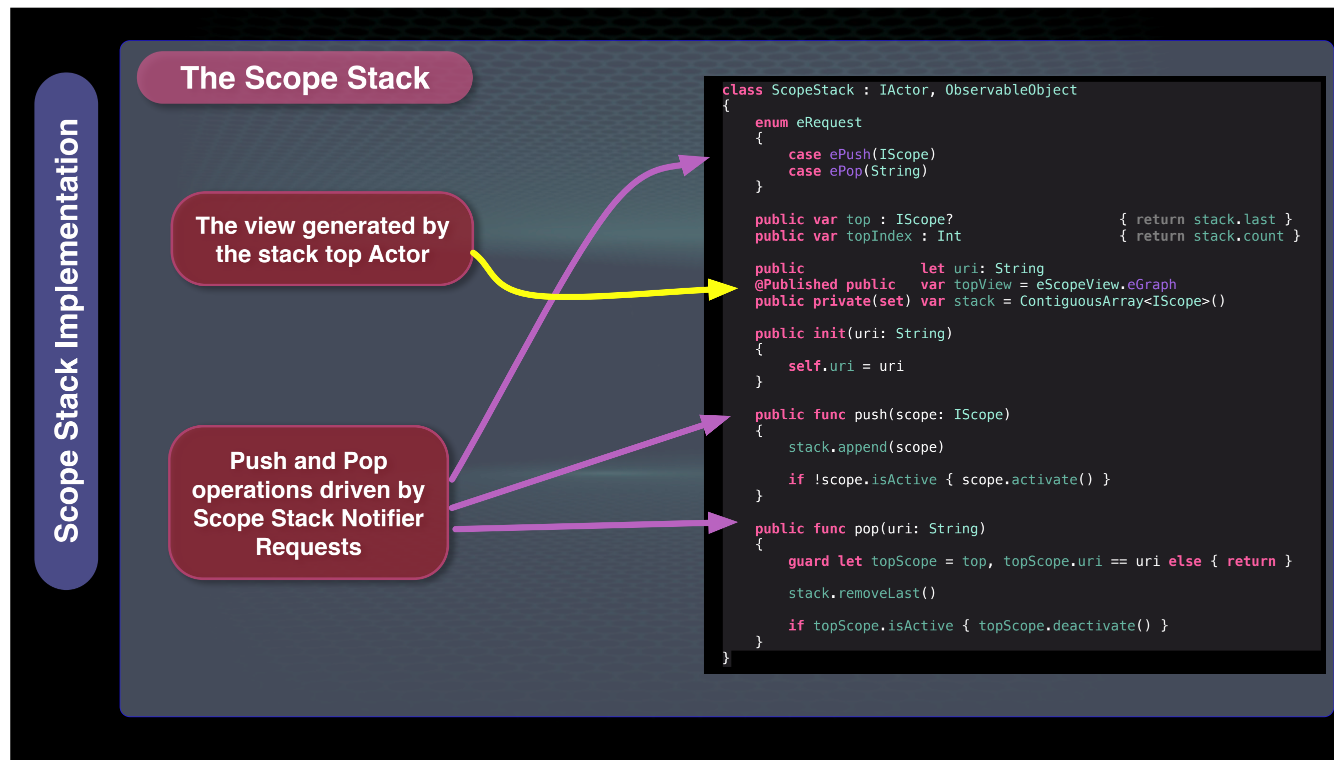
See: https://bitbucket.org/originware/bspdemoapp/src/master/



Now going onto some of the significant code snippets from the Demo App source.

Actors have an associated URI for identification, such as “/actor/service” and “/actor/presentation”. The IActor protocol also defines the URIs for the associated Actor notifiers (e.g: “/actor/presentation/notifier/request")

Scopes are Actors, so there is a protocol inheritance on IScope for IActor. Scopes have an activation and deactivation func that are exercised after being pushed onto stack and just before being popped off. The scopeView property indicates the view associated with the scope. This property is only valid for the Presentation companion scope.



Hers is the ScopeStack with its push and pop methods. These methods are driven by a request stack notifier with the associated request data type: ScopeStack.eRequest.

The published property “topView” indicates the view to be presented. The “topView” property maps to the ScopeStack “scopeView” property of the current top presentation scope and so represents the View to be injected into the View hierarchy.

Scope View Implementation

Scope View Injection

```
enum eScopeView : Equatable
{
    case eGraph
    case eMessageView(Views.MessageView)
    case eWebView(Views.WebWithDismissButtonView)
}
```

The Scope View is injected into View Content by using the Published topView property in the Presentation Scope Stack

```
// Stack Views: Graph (with Pen Canvas), Message and WebKit

ZStack(alignment: .bottomTrailing) {

    switch presentationScopeStack.topView
    {
        case .eGraph:

            <Graph View code>
            <Graph View legend label code>

        case .eMessageView(let messageView):

             <Message View code>
             <Message View legend label code>

        case .eWebView(let webViewWithDismiss):

             <Web View code>
             <Web View legend label code>
    }
}
```

This is how the presentation scope view is injected into the View hierarchy (through the topView property).

Web Scope Implementation

WebURL Scope Pair Example

This is the WebURL skeleton code for:

The Behaviour Scope and The Presentation Scope

Each scope has an:

func activate() and a func deactivate()

Here is the code to create the WebURL behaviour scope and push it onto the behaviour scope stack

```
let webScope = WebURLScope.Behaviour(notifierDirectory: m_notifiers)

await m_notifiers.behaviour.stack.sendAsync(.ePush(webScope))
```

```
public struct WebURLScope
{
    public class Behaviour : IScope
    {
        private let m_presentationScope : Presentation
        private let m_notifiers : ActorNotifierDirectory
        private var m_subscribers = Set<AnyCancellable>()

        init(notifierStore: NotifierStore)
        {
           m_notifiers = ActorNotifierDirectory(notifierStore: notifierStore)
           m_presentationScope = Presentation(notifierStore: notifierStore)
        }

        public func activate()
        {
             <activate behaviour code> (see following slides)
        }

        public func deactivate()
        {
             <deactivate behaviour code> (see following slides)
        }
    }

    public class Presentation : IScope, ObservableObject
    {
        public func activate()
        {
            <activate presentation code> (see following slides)
        }

        public func deactivate()
        {
           <deactivate presentation code> (see following slides)
        }
    }
```

This is the code skeleton for the WebURL Scope with the Behaviour and Presentation companion scopes and their activate and deactivate funcs.

WebURL Behaviour Scope

The WebURL Behaviour Scope activate() and deactivate() funcs

The Behaviour Scope listens to notifier requests and emits Presentation Scope notifier requests

```
public func activate()
{
    m_notifiers.presentation.stack.send(.ePush(m_presentationScope))

    m_notifiers.behaviour.webURL.request.sinkAsync(receiveValue: { [weak self] (notification) in

       guard let strongSelf = self else { return }

       switch notification.value
       {
           case .ePresentURL(let url, let progressFunc):

               strongSelf.m_notifiers.presentation.webURL.request.send(.ePresentURL(url, progressFunc))
       }

    }).store(in: &m_subscribers)
}
```

The Behaviour Scope also manages the Presentation Scope Stack

```
public func deactivate()
{
    m_notifiers.presentation.stack.send(.ePop(m_presentationScope.uri))
}
```

Describing the activate func: the WebURL Behaviour Scope on activation, pushes its presentation companion and begins listening for request notifications. When it observes a ePresentationURL request it forwards it to the companion Presentation Scope.

WebURL Presentation Scope activate() and deactivate() funcs

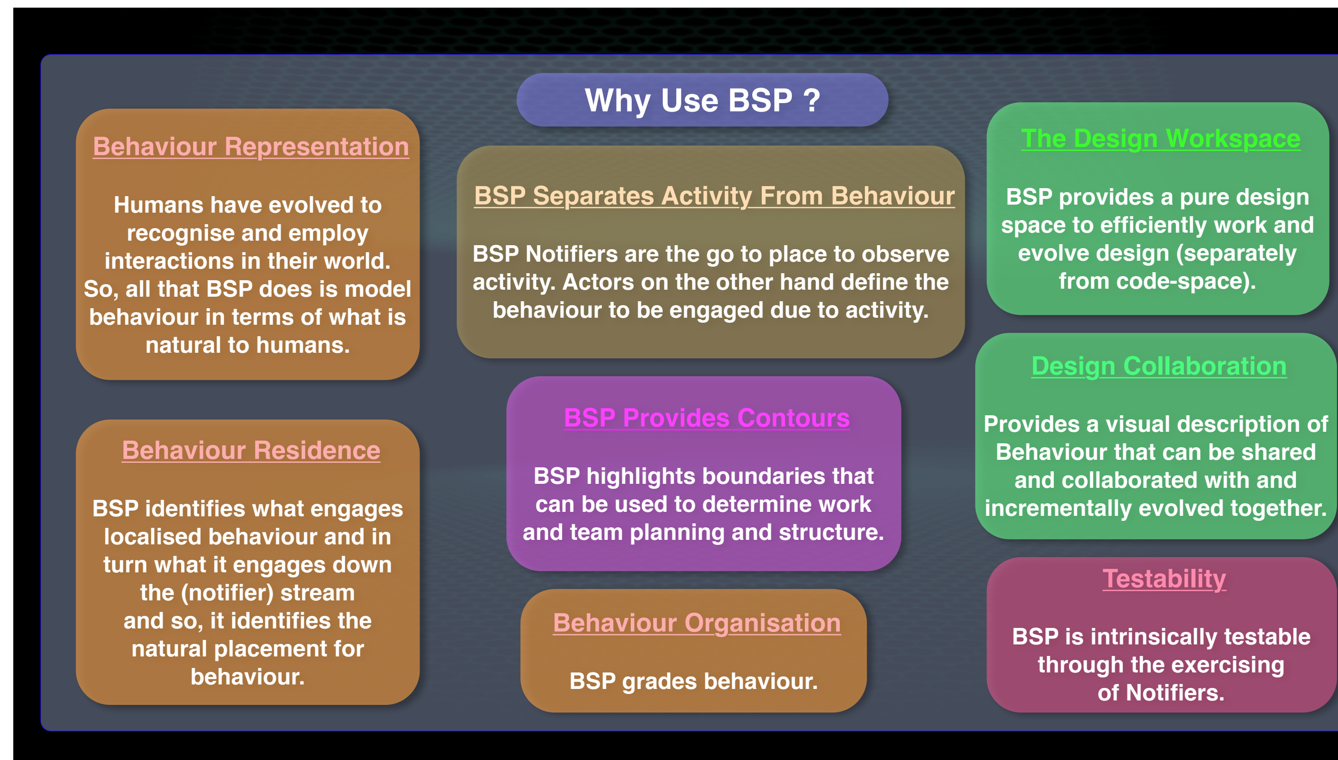
WebURL Presentation Scope

```
public func activate()
{
    m_notifiers.presentation.webURL.request.sinkAsync(receiveValue: { [weak self] (notification) in

        guard let strongSelf = self else { return }

        switch notification.value
        {
            case .ePresentURL(let url, let progressEventFunc):

                for await webEvent in await strongSelf.m_webViewModel.load(url: url)
                {
                    switch webEvent
                    {
                        case .eFetchingURL(let url):           progressEvenFunc(.eOnFetch(url))
                        case .eDownloadingURL:                 progressEvenFunc(.eDownloadingURL(url))
                        case .eDownloadProgress(let progress): progressEvenFunc(.eDownloadProgress(progress))
                        case .eLoadedURL:                      progressEvenFunc(.eOnPresent(url))
                        case .eError(let error):               progressEvenFunc(.eOnError("Alert: \(error.localizedDescription)"))
                    }
                }
        }

    }).store(in: &m_subscribers)

    let view = Views.WebWithDismissButtonView(webViewModel: m_webViewModel, notifierDirectory: m_notifiers, webContentHeight: 0)

    scopeView = .eWebView(view)
}
```

```
public func deactivate()
{
    scopeView = nil
}
```

Describing the activate func: the WebURL Presentation Scope on activation, listens for request notifications and sets the scopeView property for it to be injected into the View hierarchy.

When it observes an ePresentationURL request it calls the view-model load func to load, fetch and present the URL content. The load func instruments Apples SessionURL delegate funcs that denote the various events which are mapped to an AsyncStream event stream and then enumerated in this code and then passed to the given progressEventFunc. The progressEventFunc comes from the original behaviour Actor code that initiated the WebURL scope. So that Behaviour Actor code gets those events and issues requests for the events to be voiced.



So BSP is designed for Apps with more complex requirements but it still will provide benefit for simplistic Apps, especially if they are to grow into more complex systems over future releases. You will probably find it much easier to perform additive capability with BSP. The design system supports you in determining the natural residence for the individual parts of additive capability.

All that BSP is really doing, is representing Behaviour/Design in a form that is quite natural to you. It is natural to you, because you have been using and sculpting interactions your whole life. Your socialisation depends on recognising interactions between people and shaping your own interactions to inject into the social fabric. Even you as a biological system, your body uses signalling-interactions to perform biological processes such as nerve impulses and chemical messaging for metabolic process control. In fact biological systems came across this “design” problem a long time ago and followed an evolutionary thread of separating design principles into a three level hierarchy of:

1. Top-level: A neuronal design principle for cognition, assessment, decision making and so forth (I will leave aspects of consciousness out).
2. Mid-level: Interactional signalling for sensory and metabolic control.
3. Bottom-level: Instructional system (encoding into DNA/RNA) for stepped processes such as protein construction, deployment of proteins and so forth.

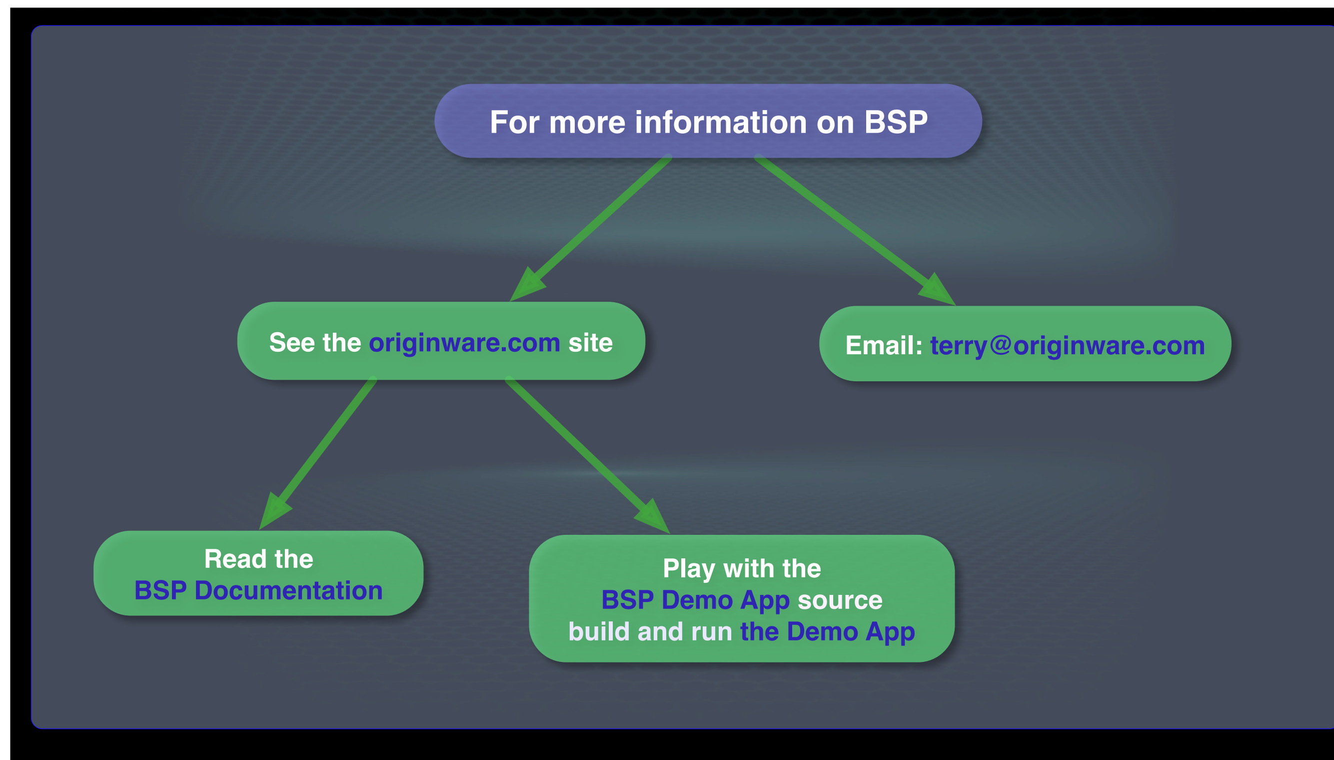
Computing has chosen to use an instructional design principle (code-languages) and only recently started to employ AI (neuronal design principle) but it still needs to use interactional for the intermediate space between instruction and AI.

BSP promotes:

You to prioritise on asking: “What are the interactions going on in this software system ?”,
rather than first asking what are the components needed to support the system.

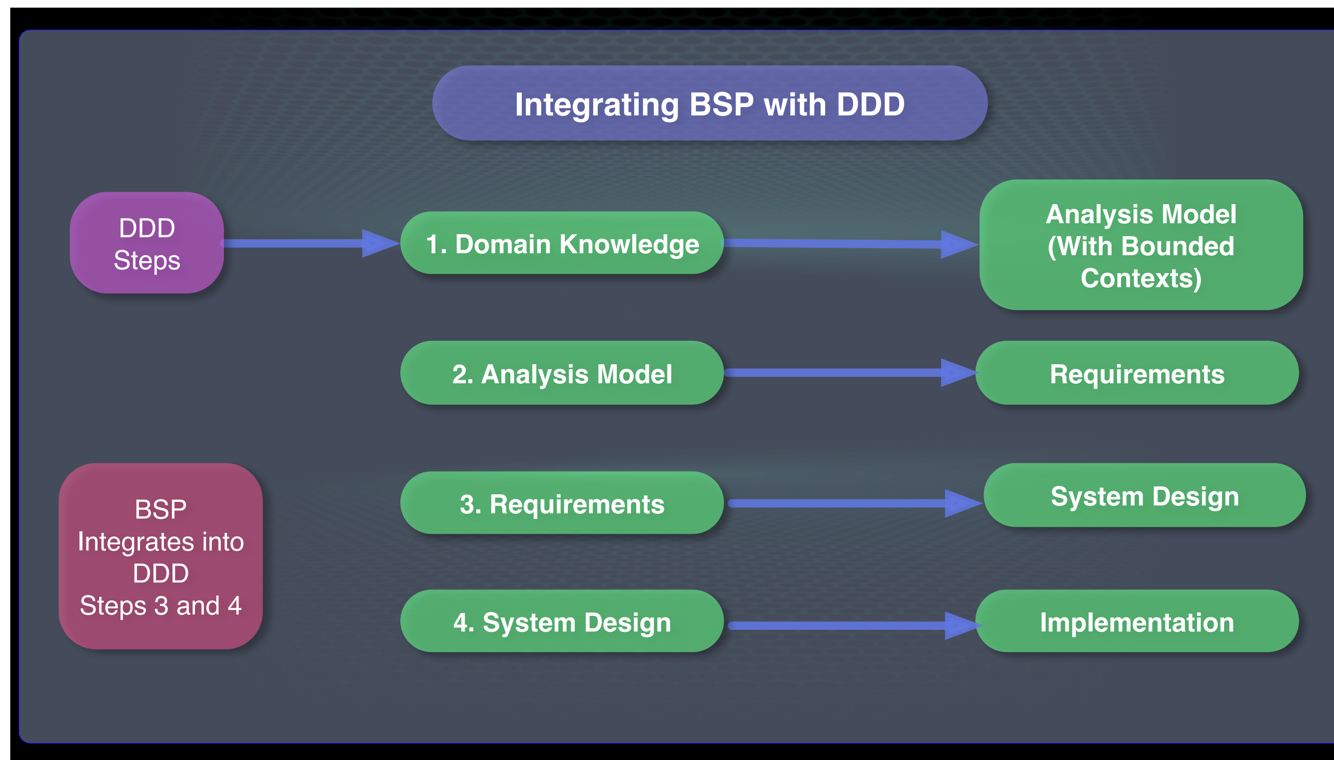
That includes the interactions with the User, the User is typically the originator of the interaction and many design systems, for all the benefits they provide, do not recognise or honour that.

I want to also acknowledge that the concept of separating design into “Behaviour-Service-Presentation” may not be totally unique to BSP. There are other similar models which tend to separate into roles of: “Business Logic-Infrastructure-Presentation”. BSP is more unique in its formalisation and methodology.



On the topic of the origin of BSP.

BSP is actually a specialisation of Originware's "Reactive Fabric" technology. It is specialised for Presentation Apps and you may want to read more on Reactive Fabric (on the Originware site) if your Behaviour Actor is going to be extremely complex. Reactive Fabric is a full stack solution, meaning it is appropriate from Server, to Application, to App to embedded/SOC platforms.



On an end note, I wanted to briefly comment on DDD (Domain Driven Design).

BSP and DDD complement each other symmetrically. DDD puts a great deal of emphasis onto Domain analysis which BSP does not address. But, BSP does complement DDD quite well in the software design and dev steps 3 and 4. The synthesised Bounded Contexts (from the analysis step) that have interactional relationships, can be expressed as Actors in the design.

BSP Question Time

BSP - Presentation End

Originware.com

Terry Stillone
Software Technology Research
terry@originware.com

Technologies:

- Semantic Transcription
- Reactive Fabric
- The BSP (Behaviour Service Presentation) Design Model