The BSP (Behaviour Service Presentation) Software System Design Model

Terry Stillone
January 2022
Originware.com

1. Introduction

While **MVC** has a fundamental relevance in App design, it does not address the complexities of modern UIs. It has not addressed the complexities of managing the complete system and that system is now required to interact with both online and on device services, handle navigational view stacks, employ the myriad of modern graphic controls, all the while provisioning for testability, organizing dependencies and supporting incremental development.

The **BSP** design model works on a fundamentally wider scope and scale than **MVC.** It does this by addressing the whole system, while respecting the fundamental principles of **MVC. BSP** is the product of the **"Design for interaction**" software design methodology and that technology recognizes **Behaviour** as a set of interactions. These interactions are idealized as a fabric of **Actors** connected though notifications pathways (called **Notifiers,** that emit from source to target **Actors)**. The propagation of notifications through the **Actor**-**Notifier**-**Fabric** defines the **Interactional Topology,** which in turn models the software system behaviour.

BSP recognizes that behaviour must be decomposed into a managed set of simpler sub-behaviours. Just as nature exhibits complex process through the interplay of a bounded set of simple processes, behaviour decomposition seeks to express complex behaviour as the interaction of simpler behaviours. It does this by first grading behaviour into levels and then recognizing the simpler sub-behaviours as pattern types.

These behaviour levels are normally categorized as: "**Macro", "Mid"** and "**Micro"** behaviour, Each of the levels have differing characteristics and requirements and so **BSP** assigns an appropriate design-model-architecture to each level.

Macro behaviour tends to deal with more wide ranging, abstract concepts, while effectively orchestrating and coordinating underlying interactions and workflow, On the other end of the scale, **Micro** behaviour being closer to the hosting OS, tends to operate with concrete OS constructs and terms and bears the weight of the actual direct "doing", i.e. directly invoking OS and UI services.

In terms of design-model-architectures, the **Macro** level representation is normally assigned to a design-architecture that supports abstract concepts and complex coordination and so in our case, that is **BSP**. The **Micro** level is typically designed as traditional sub-systems, as they are more suited to handling fine-detailed functional behaviour. **Mid** level is normally a mix, depending on trade offs between the directness and efficiency of subsystems and the systematization and consistency that interactional design brings to coordination.

The **BSP** model expresses **Macro** behaviour as three separate **Actor** models and employs the concept of a **Notifier** to transport their notifications.

To first give a loose set of definitions:

- The **Behaviour Actor Model:** *Models the application behaviour.*
- The **Service Actor Model:** *Models the thin interface to the hosting OS and comms services.*
- The **Presentation Actor Model:** *Models the thin interface to the hosting UI presentation system.*
- The **Notifier:** *Acts as the notification transport system between actors.*

2. Formal Definitions of BSP

The **Behaviour Actor Model** is formally defined as: "*The specialized behavioural logic that defines the unique characteristics of the application software system". This includes the traditional "business logic" but also covers* the application handles its responses to events and interactions *(i.e. interactions with the user, OS service events, etc*).

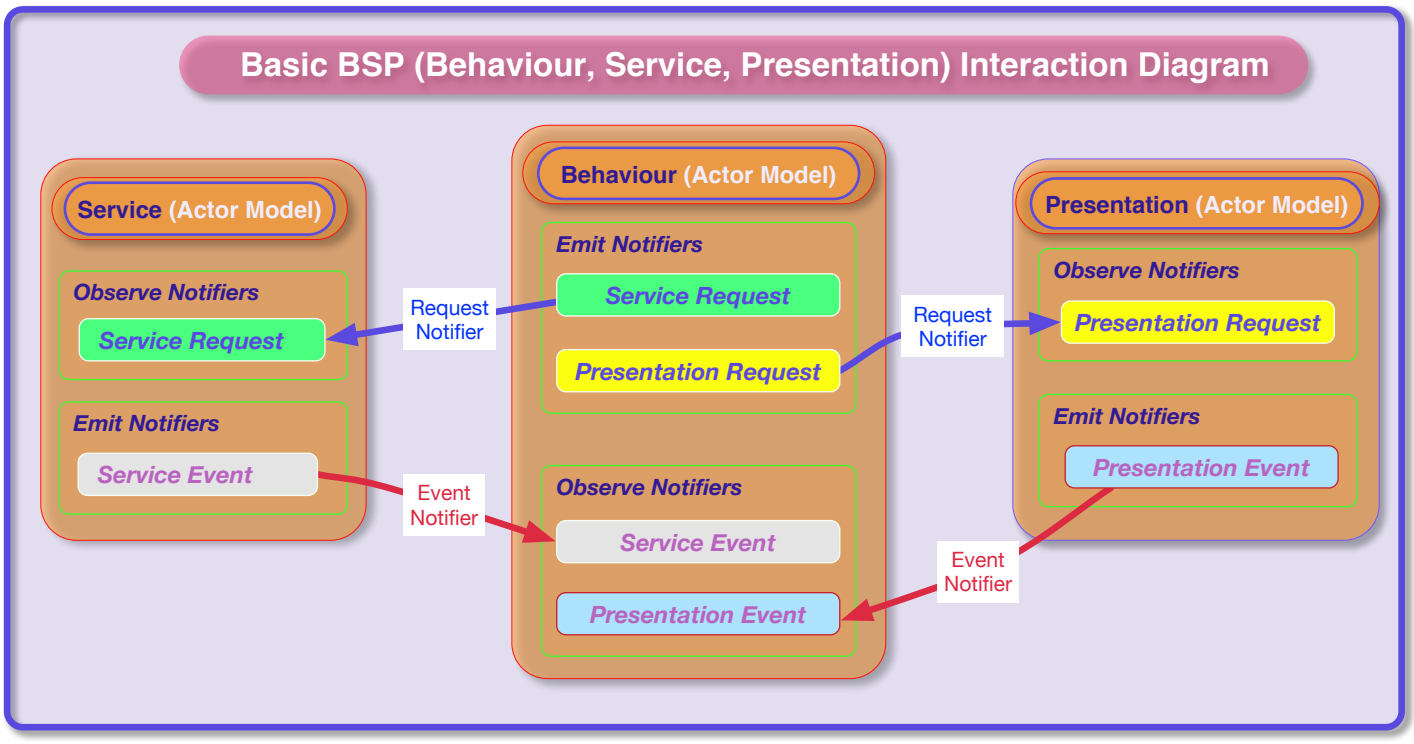
The Service Actor Model is: "*The pure interface logic that interacts with the host OS. More specifically, the logic that instructs host OS services and handles host OS service events". The **Service** actor model consumes notifier requests (to perform something) and emits host OS notifier events to the **Behaviour** actor model for event handling.*

The Presentation Actor Model is: "*The pure interface logic that interacts with the host UI Presentation system. The logic that directs and draws the graphic elements of the Presentation system and handles their associated events. It includes the traditional view models from **MVC** but the controllers only handle UI interaction and do not persist any behavioural state or imbue behavioural characteristics".* The presentation actor observes directed requests (*for it to present content*) and also emits UI related events (*in response to graphic events, but represents the event as intent*).

The Notifier Model is: *The notification transport mechanism which lays-out out the **Interaction Topology.** This is typically a simple synchronous push mechanism of a notification, packaged with specialized data, directed from a source to target **Actors**.*

Notifiers are responsible, for event routing. For example, **Presentation Actor Model** events, such as graphic element events are routed to the **Behaviour Actor Model** which in turn, triggers a cascade of notifications**,** with the ultimate emission of a **Notifier-**request to display the result (and of course, that result notification is directed back to the **Presentation Actor Model** to necessitate the presentation of the result).

Visually, the very basic **BSP** design looks like:



BSP shifts away from the traditional idealization of system behaviour as objects incorporating object properties and method calls, to a system of:

Interactions between Actor Models

which condenses **Macro** behaviour to an abstract description of:

The Propagation of Notifications through the Actor Fabric.

As a general side comment on the dynamics of "Behaviour", **Behaviour** in itself is not preemptive, but is instead, reactionary, reacting to events from sources such as presentation or service (e.g a facility authorization failure, a system timer, a UI button event). Behaviour does drive action, but only as a back reaction to an event that was originally sourced externally (i.e. not sourced internally with in the model).

BSP also supports more complex UI Navigational systems (sometimes called "UI Navigation Stacks"). The **Behaviour Model** can accommodate a **Behaviour Stack** married to the presentation navigation stack to provide more localised behaviour for each paired Navigation view (*See Appendix III for more details and examples*).

3. Notifiers In More Detail

The input and output mechanisms of **Actors** are expressed as **Notifiers**. The model inputs are assigned a **Request Notifier** variety and model outputs are assigned an **Event Notifier** type**.**

As such, **Notifiers** channel activity through the application system. They are the go to place to observe engagement, while the **Actor** is the place to observe how that engagement is handled. The functional power of **Notifiers** can be amplified**,** by internally augmenting them so that they can be traced, monitored and logged. The observed notifications can then be used in support of debugging, testing, metrics and auditing.

When projected into code, **Notifiers** are typically a code object (a class to be passed around or a struct). The act of a **Notifier** issuing a notification is implemented as a simple object method call with the **Notification** being passed as a parameter. Thus, debugging in a **BSP** model becomes one of following the notifier method call chain and observing their associated notification parameters.

There may be cases where more complex input-output interactions are required than a simple synchronous push system. For example the need for a asynchronous queued notification pipeline may arise. These more complex transport mechanisms or channels can be constructed within the source **Actor**, then deployed through synchronous **Notifiers** to the target **Actors**, and subsequently exercised on demand.

4. Decomposing Behaviour With Scopes

BSP recognises cyclic behaviour patterns and refers to them "**Scopes**" (because many of these exhibit a life-time-cycle pattern which can be organised into hierarchal scope set). These "**Scopes"** have a lifecycle pattern of:

i) An initiation cycle.

ii) An operational cycle (where a number of scope associated operations may be performed).

iii) A final termination cycle (which also demarcates the end of all the scope operations).

Some common **Scope** examples include:

- The **Application Cycle** (initiate with the construction of resources, operate the application terminate with resource destruction).
- The **Security Cycle** (initiate with obtaining authorization, operate secure services terminate with the relinquishment of authorization).
- The **Command Cycle** (initiate with the receipt of a command, operate the command, terminate with a result response).
- The **Modal Presentation Cycle** (initiate with the presentation of the modal, operate modal interaction and terminate with the removal of the modal and return of the result)
- The **Login-Session-Logout Cycle** (initiate with login, operate the session terminate with logout).
- The **Navigational Stack Cycle** (initiate with presentation of the nav view set, operate the view interaction, terminate with removal of the view set).

Many of the examples sited above are actually reversible processes, so their termination is the reverse of their initiation cycle. This is a common characteristic to take note of, when identifying potential **Scopes**.

Scoping is implemented using a "**Behaviour Stack**", housed in the **Behaviour Actor,** with each stack element operating as an **Actor** in its own right, specializing in the target **Scope** behaviour. The temporal window of the **Scope** on the stack marks the **Scopes** lifetime as well as the boundaries of child **Scopes** that it initiates.

To illustrate the hierarchal nature of **Scopes**, here is an example outline of the **scope** hierarchy for a generic net server based account App:

While **Scoping** is also useful for **Mid**-behaviour coordination, it may not cover all behaviour situations and needs. For a more complete design system, please follow the **Reactive Fabric Technology** link in section titled: "The Larger Design Picture".

5. Idealizing Behaviour in the Abstract "Design Space"

BSP decouples **Macro**-behaviour from implementation by representing the design in an abstract "**Design Space**" and so implementation now becomes a projection of the design onto "**Code Space**". This actually makes sense as **Macro**-behaviour deals in more abstract concepts and coordinates more abstract processes.

Design Space is more visualisable, more immediately changeable and so the perfect medium to prototype, to perform collaborative design, to share, to document and to incrementally design. Thus **Design Space** as a medium, is engineered to promote incremental design within a collaborative atmosphere.

Design Space also directly addresses the the cognitive load on developers through the use of decomposition. Decomposition reduces the size and complexity that developers "need to fit in their head". Putting it in stark terms: "A diagram is much easier to fit into the head than a thousand of lines of code"

6. Software Construction With BSP

Actors and **Scopes** provide a very "contour" rich set of aspects to the overall design and these "contours" can be very useful in the planning and analysis space.

The concept of the "**Aspect Contour**" describes the collective boundary layers for a particular aspect and these contours can be projected onto various spaces to give insight into their own natural contours. In the **BSP** context**, Scopes** identify contours with respect to the aspects of scope life-cycle, role and role-dependency. **Actors** identify the contours with respect to interaction, such as notifier pathways, notification types and notification content.

Each story or feature exercises a set of pathways that traverse a series of **Scope** and **Actor** contours, some of those contours are perquisites and will need to be scheduled for construction prior to the story/feature and some of the contours will be new and will need to be included as work items for the story/feature.

Contour awareness can be a very useful tool in planning, in the determination of team assignments, in the establishment of both temporal and behaviour dependencies, in the sequencing of construction, in the formulation of work-sizing and much more. The terms described by **Contour Aspects** can become part of the project terminology, to describe the process-pathways and the more vague aspects of the design.

Please, also be mindful that **Scopes** do not rigidly imply that they need to be developed in their hierarchy order (if there is no active dependency). Mock replacement pass-through scopes can be used instead as placeholders until implementation is required. So in our previous example, UI related scopes can be constructed with hosting and environment security scopes implemented pass-throughs and the Server Session scope can simply mocked (while being developed), so as not block other parallel development with active session dependancies.

7. Testing Scenarios.

BSP testing procedure is typically a three step process:

1. Exercise target input notifiers with generated testing notifications.
2. Observe the notifications of the resultant output notifiers.
3. Match the observed notifications with expected results.

Notification observation can be implemented by placing auditing hooks within **Notifiers** to log notifications to a centralized auditing store and then later be used to match against expected results. These stored observations can be rendered as strings (together with their notifier identification) and compared by sub-string matching.

Testing scenarios fall into:

- Testing individual models: **B, S, P**
- Testing model composites: **B + S** and **B + P**
- Whole system testing: **BSP**

For those unit tests that need to be driven by presentation events, you can either exercise the graphic elements directly (using some available framework), or you can engage the graphic element action by exercisng the associated **Presentation Model Event Notifier**.

7. Pros and Cons of BSP

(i). Benefits of BSP

BSP code tends to be more compact, more performant and definitive (i.e. the code indicates the "what" of what is being done and less of the "how" it is being done). Models tend to have much less state, in fact the actual state is encoded in the notifier call chain so in a sense much of the implied state is moved to a functional implied state. Debugging becomes more of an immediately accessible experience by following notifier call chains in a debugger.

BSP also tends to be naturally testable, as **Notifiers** are the natural conduit for interaction and interaction drives the behaviour.

(ii). Issues Related to BSP

The upper most concern for **BSP** is that it provides a perfect opportunity to create true **Notifier** spaghetti. In fact, it is quite easy to gravitate to an entangled mass of **Notifiers** over time. The countering process is to diagram the **Models**, the **Notifiers** together with their inter-connections and so highlight areas of concern. If your **BSP** model is hard to draw with many cross-over connections then it is a prime candidate for connection refactoring.

There is also the post implementation redesign syndrome, where a completely functional **BSP** model is written in code only to then diagram it and realize how silly and overly complicated it really is. Iterative, on-going diagramming should be a natural part of your design-implement-redesign cycle. Diagramming provides the operational high level view of the system and working in **Design-Space** gives much more immediate insights at the macro-scale than coding.

8. The Larger Design Picture.

Looking at the bigger scheme of things, it is important to appreciate that the **BSP** design system is actually a limited application of **Originware's** much wider "**Design for Interaction"** technology, called:

"**Reactive Fabric Design Methodology"**

It provides a complete framework for iterative design and visualization. For more information, please see the website link:

Reactive Fabric Technology from Originware

9. More Information.

Demonstration source is also available**.** This is a fully functional iPad **Xcode** project, written in **Swift 5.5 and SwiftUI 3**. See the git repro:

The BSP Demo App @BitBucket

Address queries and comments to: **Terry Stillone** (terry@originware.com)

and see the **Originware**.com site for additional technologies.

Appendix I. What is Design for Interaction ?

To give you a wider perspective, there are many design principles around. The more notable ones include:

- Design for function calls (e.g assembly, fortran etc).
- Design for objects (traditionally called “object oriented design”).
- Design for process (e.g. micro services).
- Design for domain (bounded contexts) and design for aspect.

Each design principle recognizes the boundaries and contours of particular aspects of the system. Some of these principles are more mature and cover a number of separate aspects but they tend to focus on a primary aspect. **Design for Interaction** recognizes the **Data** aspect (represented as notifications), the **Process**, the **Actors** in the interaction and the I**nteractional Topology,** but does make **interaction** the foreground focus.

The aspect of “**Interactional topology**” also touches on the manner of interaction between **Actors.** The characteristics of interaction can be categorized into the following aspect matrix:

Aspect	**Interactional Aspect**	
	Single-way	Bi-way
Temporal Aspect Synchronous	Synchronous single-way push from the Producer Actor to the Consumer Actor	Synchronous bi-way. A push from the Producer Actor to the Consume Actor followed by a synchronous reply back to the request (from Consumer back to the Producer)
Asynchronous	A single-way push from the Producer Actor that is received asynchronously by the Consumer Actor.	Asynchronous Bi-way Interaction. A push from the Producer Actor to the Consumer Actor followed by an asynchronous reply back in relation to the request, from Consumer back to the Producer.

Actors can be real (such as a user or a device), material (related to the host OS), they can be purely abstract, representing abstract concepts (these are termed abstract-**Actors**) and they can also represent aggregations of notifications from other resident **Actors (**referred to as meta**-Actors)**.

Interaction Topology also recognizes the connection patterns that an **Actor** employs:

- The **Source** pattern: The Actor is single-way producer to output with no input.
- The **Collector** pattern: The Actor is a single-way consumer of input with no output.
- The **Operator** pattern: The Actor is a single-way consumer of input and producer to output.
- The **Service** pattern: The Actor is a bi-way consumer of the input with no output.
- The **Socket Source** pattern: The Actor is a bi-way producer to output with no input.
- The **Socket** pattern: The Actor is both a bi-way consumer of input and bi-way producer to output.

Appendix II. Why Use Design For Interaction ?

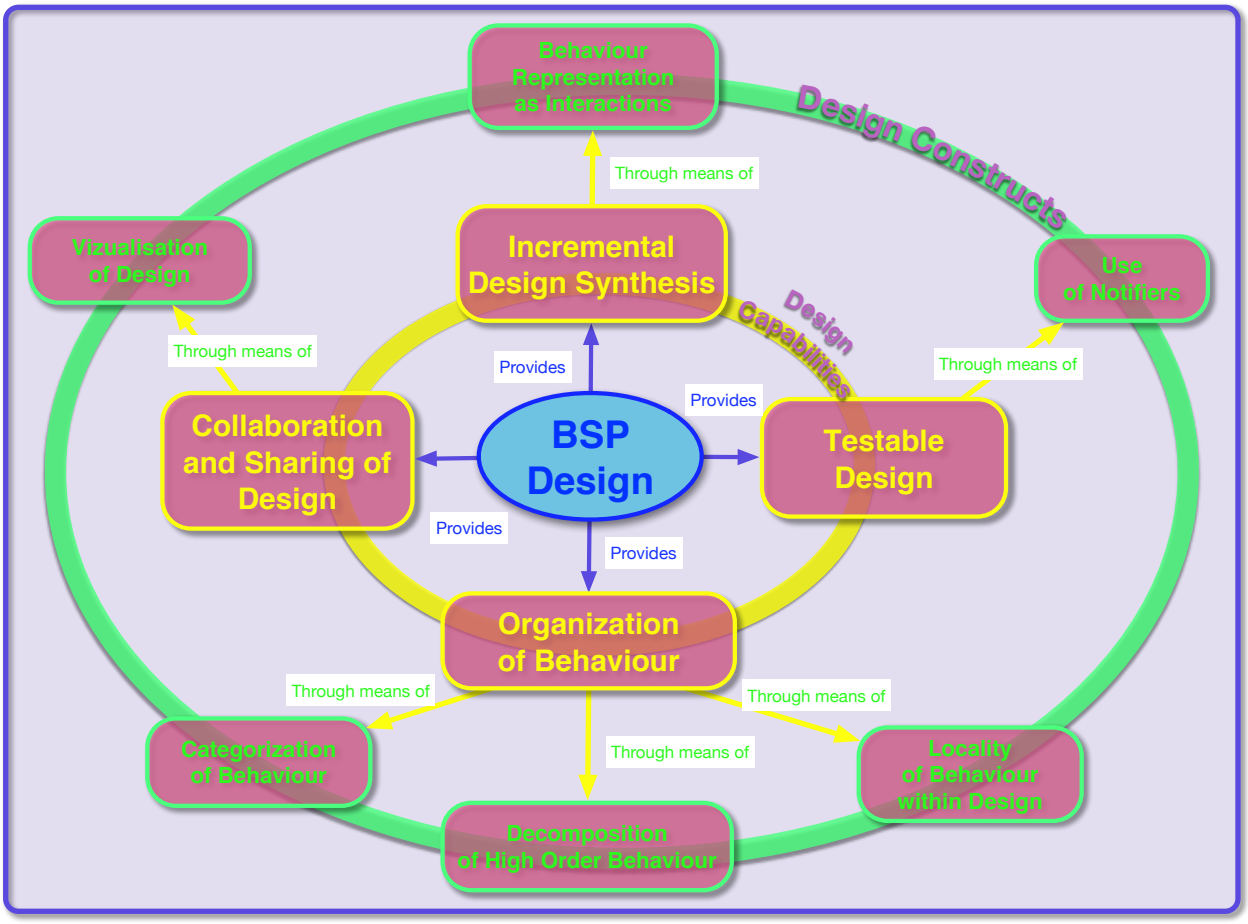
Humans naturally envision interactions, party because their foundational biology employs interactions for operation (e.g. the nervous system with electrical impulses, chemical signalling of metabolic control, etc) but more, humans have evolved to recognize and craft interactions, especially between themselves and their world. So, basically, what the "**Design for Interaction**" principle does, is to represent complex process into something that humans naturally relate to. Contrast this with the "**Design for function calls**" principle that is not a natural human process, that takes time to become literate in and even more time to be able to compose. If you take a school child with an appropriate level of reading comprehension and give them a (simple) interaction diagram, they will begin to grasp at a basic level the basic features of the process, such as its implied sequencing.

As a methodology, **Design for Interaction** asks:

"What are the external incoming interactions ?",

... and for each of those interactions, "What are the end targets they need to get to ?". These targets are normally material boundaries (such the Hosting OS Services or Presentation services which are represented as material Actors). Then, intermediate **Actors** between these sources and targets are synthesized using iterative design analysis. Synthesis is performed by grouping the incoming interactions into behaviour-cycles and ordering those cycles into scopes (and with scopes ideally forming a collective hierarchy). The scopes define the behaviour scope stack elements and each stack element becomes an **Actor** to handle the individual interactions for that scope.

This iterative design methodology as a process, can take on a quite amazing quality of unfoldment, as it progresses and evolves from the grosser, basic foundational scopes, to the more fine grained behavioural nuance that the richer behaviour **Scopes** imbue. This evolve-to principle brings a more organic, more evolutional characteristic to the construction process. It brings a more mature progression as opposed to the limited dimensionality of the traditional "write code and integrate" standard cycle.



Appendix III. The BSP Model In More Detail

The table below, details the aspects of the BSP Actors:

	Behaviour Model	Service Model	Presentation Model
Functional Role	Encompasses the traditional business logic. In more abstract terms, models the control and coordination of the cycles and activities of the application.	Engages and controls OS services Emits OS Service events.	Engages and controls the traditional View Models and View controllers. Emits Presentation events (e.g. button click events).
Accepts Request Types	Does not accept external requests.	Exercise Host OS services (e.g. comms, data services, location services, speech services, logging services).	Draws graphical elements. Push and Pop Navigation Stack Element.
Observes and Emits Event Types	Does not externally emit events.	Authorisation events, OS service failure events. Hosting state changes.	Graphical element events (e.g. mouse clicks, text entry events).
Houses	The Behaviour Stack, if required.	OS control interface instances.	View model instances and graphical element instances. Navigational Stack and elements.
Persists State	The application state and behaviour state.	Security and authorisation state.	The view navigation state and view model state.

In terms of state persistence, there are guidelines as to what the **Actors** persist. The **BSP Service** and **Presentation Actors** are oblivious to the behavioural state, only the **Behaviour Actor** persists behavioural state. Of course the presentation system may give the appearance of responding to behavioural state (such as, undo my last action) but this is a result of presentation being driven by the **Behaviour Model** (which can adapt to its own state).

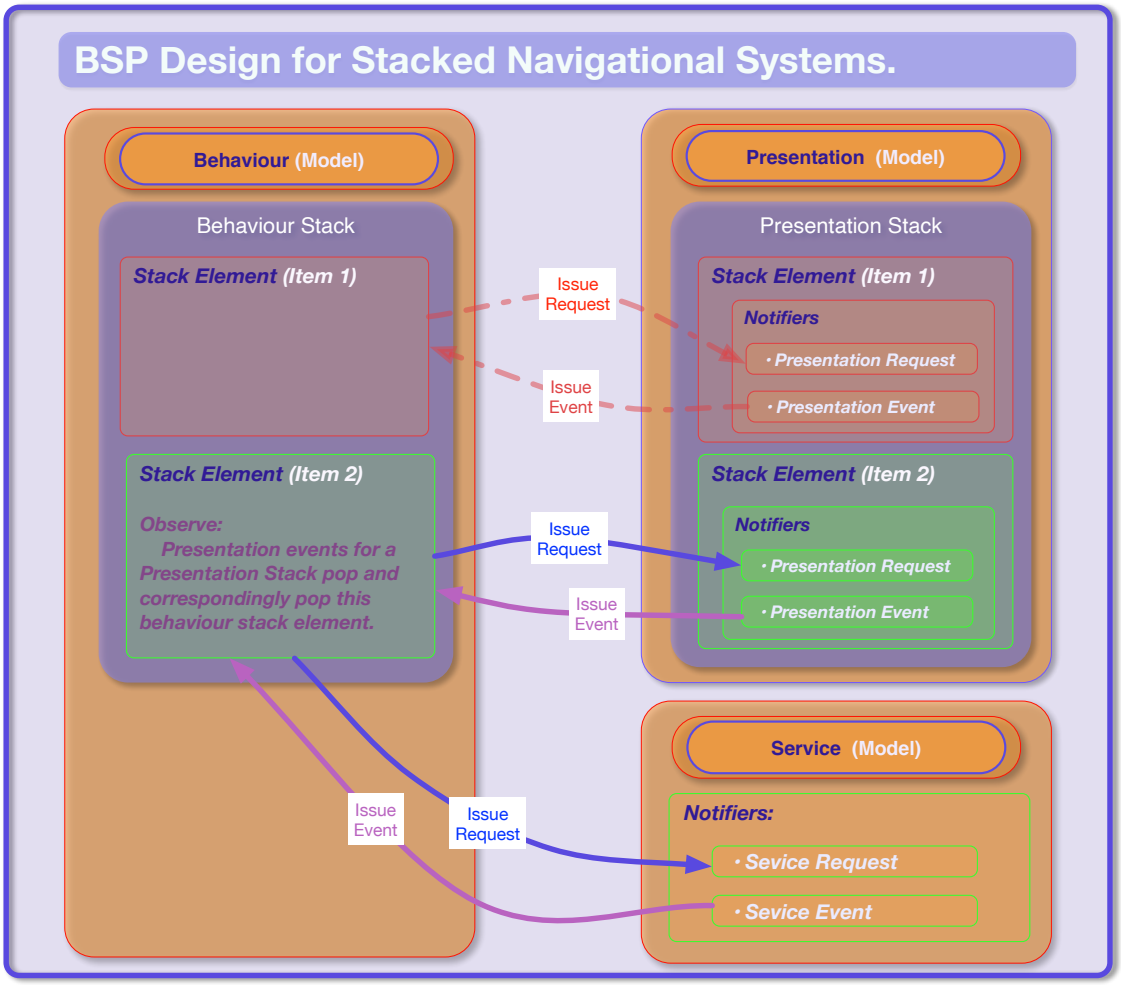
As you work with **BSP**, you may find:

- The **Behaviour Actor** is the most complex **Actor** of the three models. The model requires much more attention, much more support (e.g. a **Behaviour Stack)** and may require the use of other sub-**Actors**.

- It is worth synthesizing internal **Behaviour Actor** meta-events which are an aggregation of actual presentation and service events into the **Behaviour Model** so as to simplify and localize in-behaviour-model handling as one complete grouping.

- The separation of **Actor** roles pulls in more interactions between the individual **BSP Actors** as notifications may need to be cross routed to their target model, rather than being handled locally in the source **Actor**. This is a trade-off to bring in consistency to the system and which ultimately reduces the key concern of complexity.

BSP supports UI Navigation Stacks, below is an example notification diagram supporting a behaviour stack.



To illustrate the interaction between a behaviour stack and presentation here is a sequence diagram for pushing a modal confirmation dialog, to a stacked presentation model: