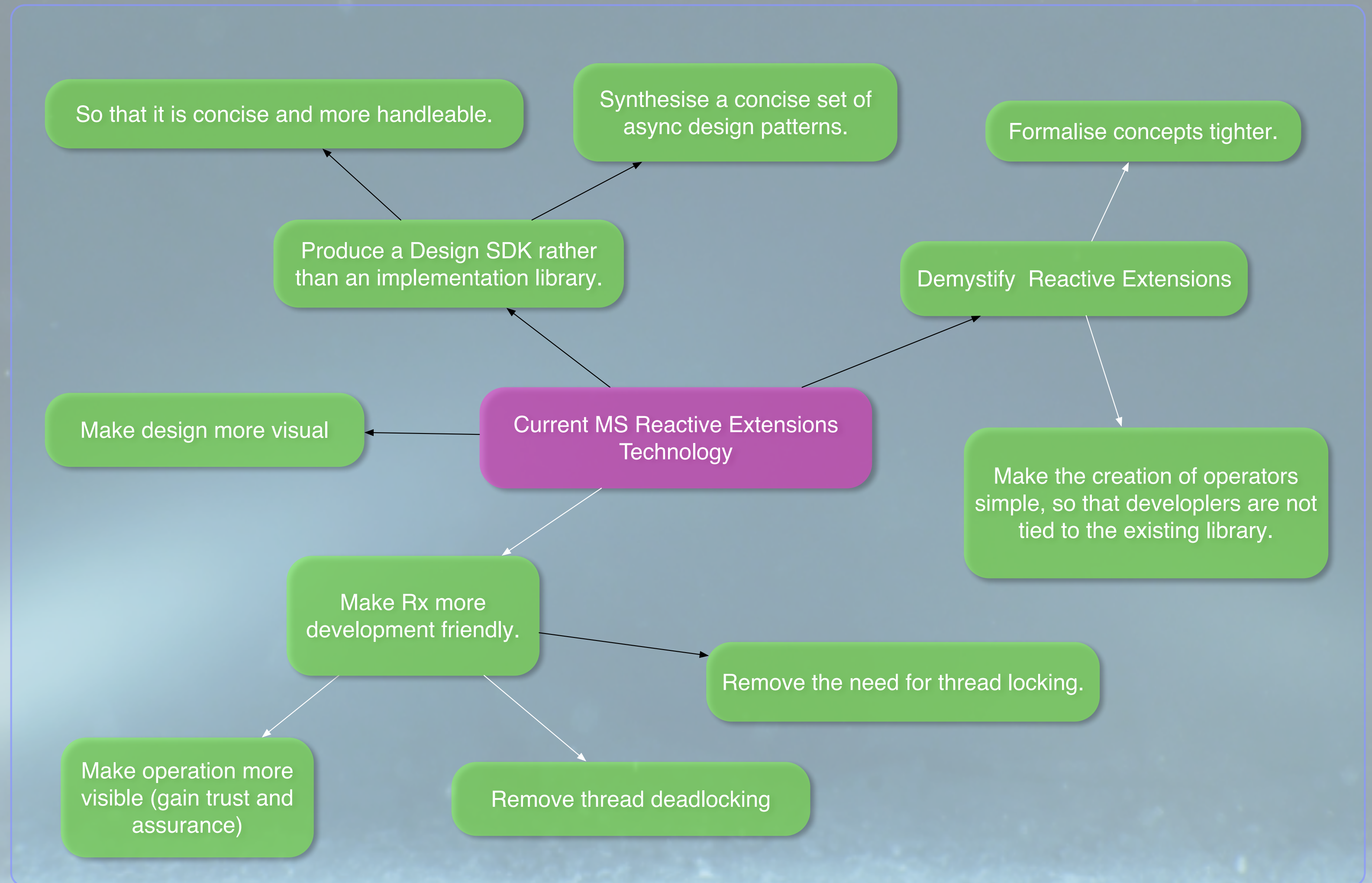
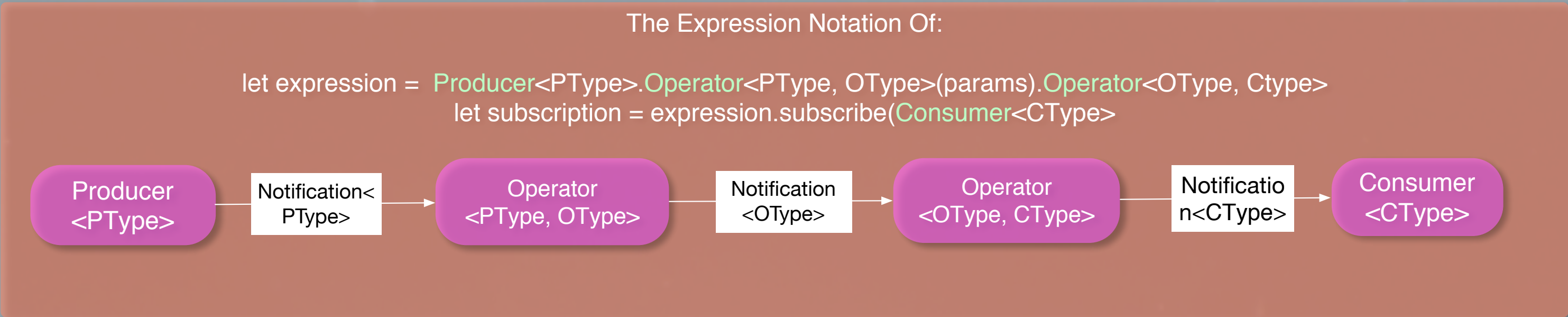


Objectives Of RxPatterns

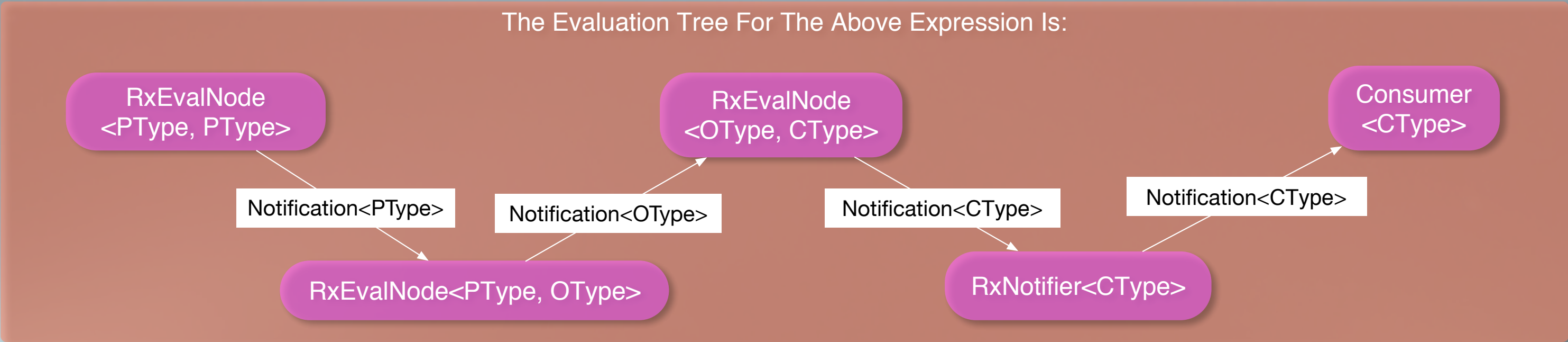


The RxPatterns Expression to Evaluation

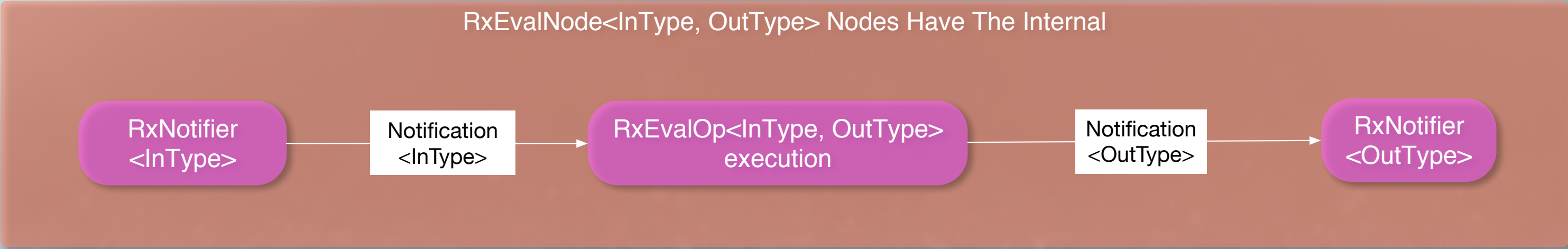
Expression



Evaluation



RxEvalNode Execution




```
public enum eRxEvalStateChange
{
    /// A new subscription is in progress. The state change is performed in
    /// the Subscription thread.
    case eNewSubscriptionInSubscriptionThread(RxSubscription)

    /// Subscription evaluation phase about to begin.
    case eSubscriptionBegin(RxSubscription)

    /// Evaluation about to begin.
    case eEvalBegin(RxSubscription)

    /// Evaluation about to end (occurs after a Completed notification).
    case eEvalEnd(RxSubscription)

    /// Subscription is about to end.
    case eSubscriptionEnd(RxSubscription)
}
```

```
public final class RxEvalNode<ItemInType, ItemOutType> : RxNotifier_Relay<ItemInType, ItemOutType>
{
    /// The item notification handler (delegate), by default emits the received item notification.
    public var itemDelegate = { (item : ItemInType, notifier : ARxNotifier<ItemOutType>) -> Void in

        // Default behaviour of itemDelegate. The delegate should really be over-written.
        // Attempt to convert item to ItemOutType.
        if let outItem = item as? ItemOutType
        {
            notifier.notifyItem(outItem)
        }
        else
        {
            assert(false, "Cannot convert \$(ItemInType.self) to \$(ItemOutType.self)\n")
        }
    }

    /// The completed/Error notification handler(delegate), by default emits the received completed notification.
    public var completedDelegate = { (error : IRxError?, notifier : ARxNotifier<ItemOutType>) -> Void in

        notifier.notifyCompleted(error)
    }

    /// The StateChange notification handler (delegate), by default does nothing.
    public var stateChangeDelegate = { (stateChange : eRxEvalStateChange, notifier : ARxNotifier<ItemOutType>)

                                            -> Void in

        // Do nothing, state changes are not passed along by evalNodes.
    }
}
```



```
class RxObservableMap<ItemInType, ItemOutType>
{
    . . .
}

extension RxObservableMap
{
    public final func map<TargetItemType>(
        mapAction : (ItemOutType) -> TargetItemType?
    ) -> RxObservableMap<ItemOutType, TargetItemType>
    {
        let evalOp = { (evalNode : RxEvalNode<ItemOutType, TargetItemType>) -> Void in
            evalNode.itemDelegate = { (item: ItemOutType, notifier : ARxNotifier<TargetItemType>) in
                if let mappedItem = mapAction(item)
                {
                    notifier.notifyItem(mappedItem)
                }
            }
        }
        return RxObservableMap<ItemOutType, TargetItemType>(tag: "map", evalOp: evalOp).chainObservable(self)
    }
}
```

```
extension RxObservableMap
{
    public final func take(count : UInt) -> RxObservable<ItemOutType>
    {
        let evalOp = { (evalNode : RxEvalNode<ItemOutType, ItemOutType>) -> Void in

            var currentItemCount : UInt = 0

            switch count
            {
                case 0:
                    // Complete the subscription immediately upon Eval Begin.

                    evalNode.stateChangeDelegate = { (stateChange: eRxEvalStateChange, notifier: ARxNotifier<ItemOutType>) in

                        switch stateChange
                        {
                            case eRxEvalStateChange.eEvalBegin:
                                notifier.notifyCompleted()

                                default:
                                    // do nothing.
                                    break
                        }
                    }

                default:
                    // Take count item notifications and then complete.

                    evalNode.itemDelegate = { (item: ItemOutType, notifier : ARxNotifier<ItemOutType>) in

                        assert(currentItemCount < count, "take: Expected item count to be less than \("\(count)")")

                        notifier.notifyItem(item)

                        currentItemCount += 1 // PS. Please put auto-increment/decrement back in swift!

                        if currentItemCount >= count
                        {
                            notifier.notifyCompleted()
                        }
                    }
            }

            return RxObservable<ItemOutType>(tag: "take", evalOp: evalOp).chainObservable(self)
        }
    }
}
```



```
extension RxObservableMap
{
    public final func takeStrict(count : UInt) -> RxObservable<ItemOutType>
    {
        let evalOp = { (evalNode : RxEvalNode<ItemOutType, ItemOutType>) -> Void in

            var currentItemCount : UInt = 0

            // Strictness check, that we did have count item notifications previously emitted.
            evalNode.completedDelegate = { (error : IRxError? = nil, notifier : ARxNotifier<ItemOutType>) in

                assert(currentItemCount <= count, "takeStrict: Expected items emitted to be less than \count)")

                let errorToEmit : IRxError? = error ?? (currentItemCount >= count)
                                                            ? nil
                                                            : RxError(.eCustom("takeStrict: Expected at least \count) item
notifications"))

                notifier.notifyCompleted(errorToEmit)
            }

            // The same delegate setting code from the non-strict take operator would follow here:

            . . .

        }

        return RxObservable<ItemOutType>(tag: "takeStrict", evalOp: evalOp).chainObservable(self)
    }
}
```

Async Design Patterns in RxPatterns

Core Pattern

RelayPat - receive notifications and optionally emit a notification. Also handle state eval state changes.

Source Sync Patterns

SyncGenPat - generate synchronous notifications

Source Aync Patterns

AsyncGenPat - generate asynchronous (timed) notifications.

RedirectPat - redirect notifications from a notifier into the current expression.

TickPat - use an external notifier triggering a given notification generator func to generate notifications when the notifier sends a tick notification.

Observable Patterns - single stream handling

ExtractPat - extract notifications to delegate functions.

Observable Patterns - multiple streams handling

Switch Pattern - switch multiple streams into the current expression.

Gate Pattern - gate multiple streams into queues and handle mapping of items into current expression.

Benefits Of RxPatterns

Concept Abstraction

Concepts of: Consumers, Producers, Notifiers, EvalQueue, Notification

Abstraction aides in design visualisation and design terminology.

RxPatterns is a Design SDK (not an implementation library)

You develop your own operators (if required). Handle your specific error conditions and optimise for your specific conditions.

The Toolkit has the source implementation for practically all the standard MS Operators for use or as a source reference for your custom operators.

You can develop your own notifiers.

Use of EvalQueues rather than threading

No locks required - cleaner code (possibly faster).

RxEvalQueue has thread deadlock detection - no lock up (unless threads are directly blocked)

Internal operations are monitorable

Operations are instrumented, they can be viewed or logged. You can write your own monitor handler

Multi Platform

On swift platforms which also support libdispatch (currently OS/X, iOS and linux coming)