Quantum MVVM

https://github.com/xlasne/Quantwm

Cocoaheads Paris

January 2018

Manage complexity

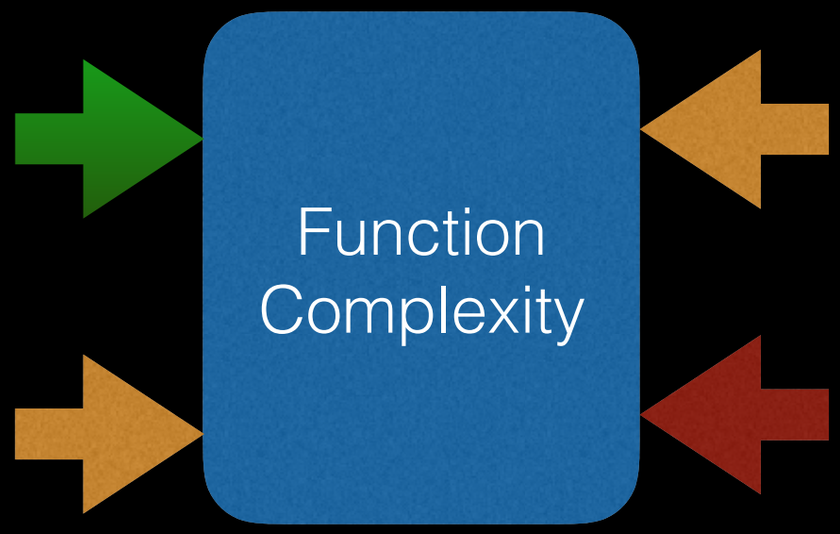
As Software Engineers, we must use architecture, concepts and technology, to rule the chaos of a program whose complexity grows at each new feature into a clean, decoupled, bug-free, testable software.

What is functional complexity ?

Inner Complexity

Internal function processing complexity

function parameters variability



Contextual Complexity

fan-in: # of contexts from which this function is called

impure function depending on local/external variables:
# of possible value set
-> analysis of each variable state & lifecycle

Functional complexity depends on:

- Inner Complexity, which is visible hence easier to manage
- Contextual Complexity, which is a mental abstraction and evolving during development.

Contextual complexity may be formalized via a Contract establishing the constraint and rules expected from the context.

Quantwm Architecture

Quantwm comes from this "simple" idea:

I want to register to a set of data model properties, and be called at most once during the event loop, synchronously, when all these properties are stable.

The naïve original goal was to eliminate redundant notifications, but this is just the tip of the iceberg, and drive a clean architecture design.

Quantwm Architecture

In order to achieve this goal, Quantwm architecture refines MVVM by introducing a mediation layer between the Model and the View Model, and defines an architecture capable of reducing the contextual complexity of applications:

Data Centric: Control application state from model only (Undo, Version browser)

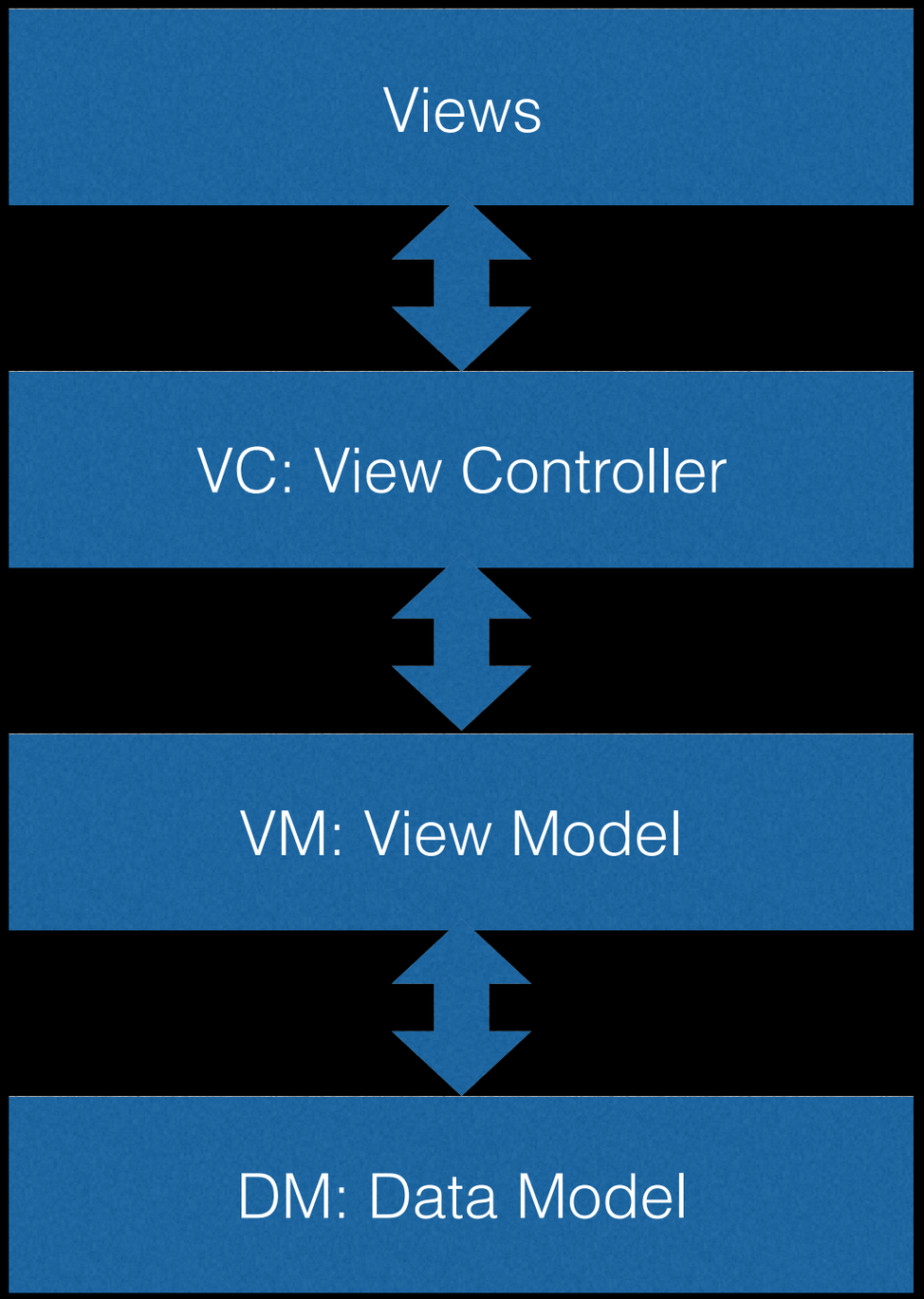
Flux-like: Defined, Ordered model mutation rules

Synchronous: Views are consistent and synchronized with the Model

Contract based: Contextual complexity management

Fully decoupled: Code factorization, testability, contextual complexity.

From MVVM Architecture ...



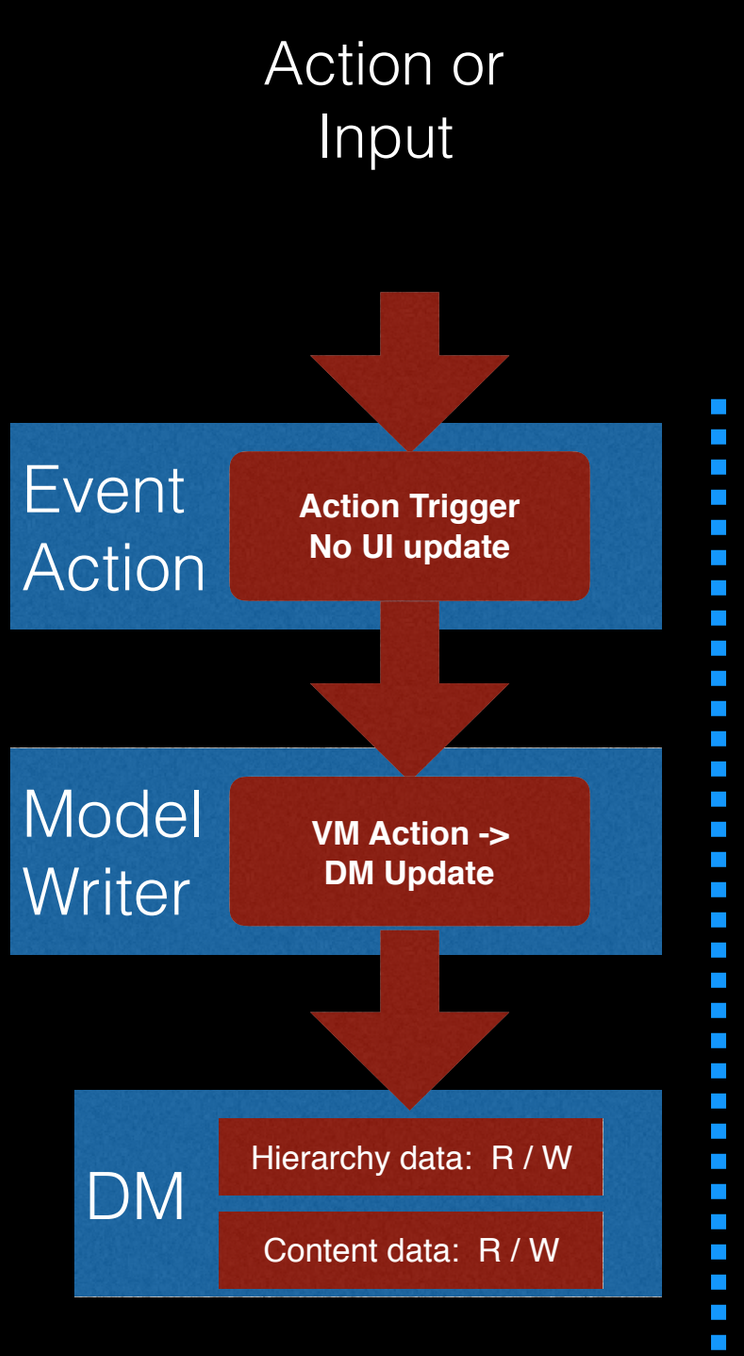
View hierarchy
Action triggers VM action
Content filled on VM content notification

VC manages view hierarchy
VC binds view action with VM action
VC binds view content with VM content.
Owns view hierarchy and view models.

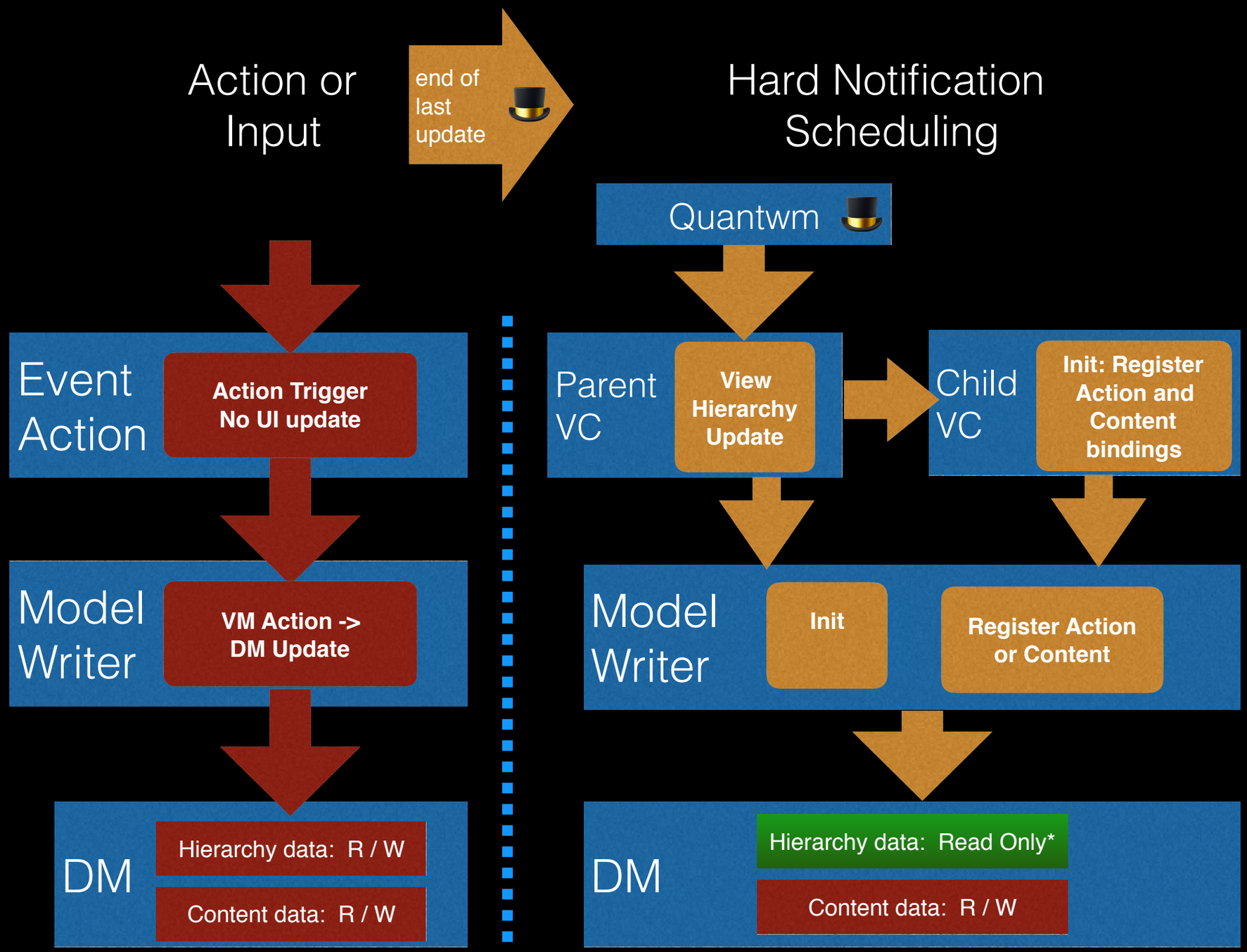
VM manages view-model interaction
VM action triggers DM update.
VM content reads DM and format it for the view

DM manages data persistency and consistency.

… to Quantwm Architecture

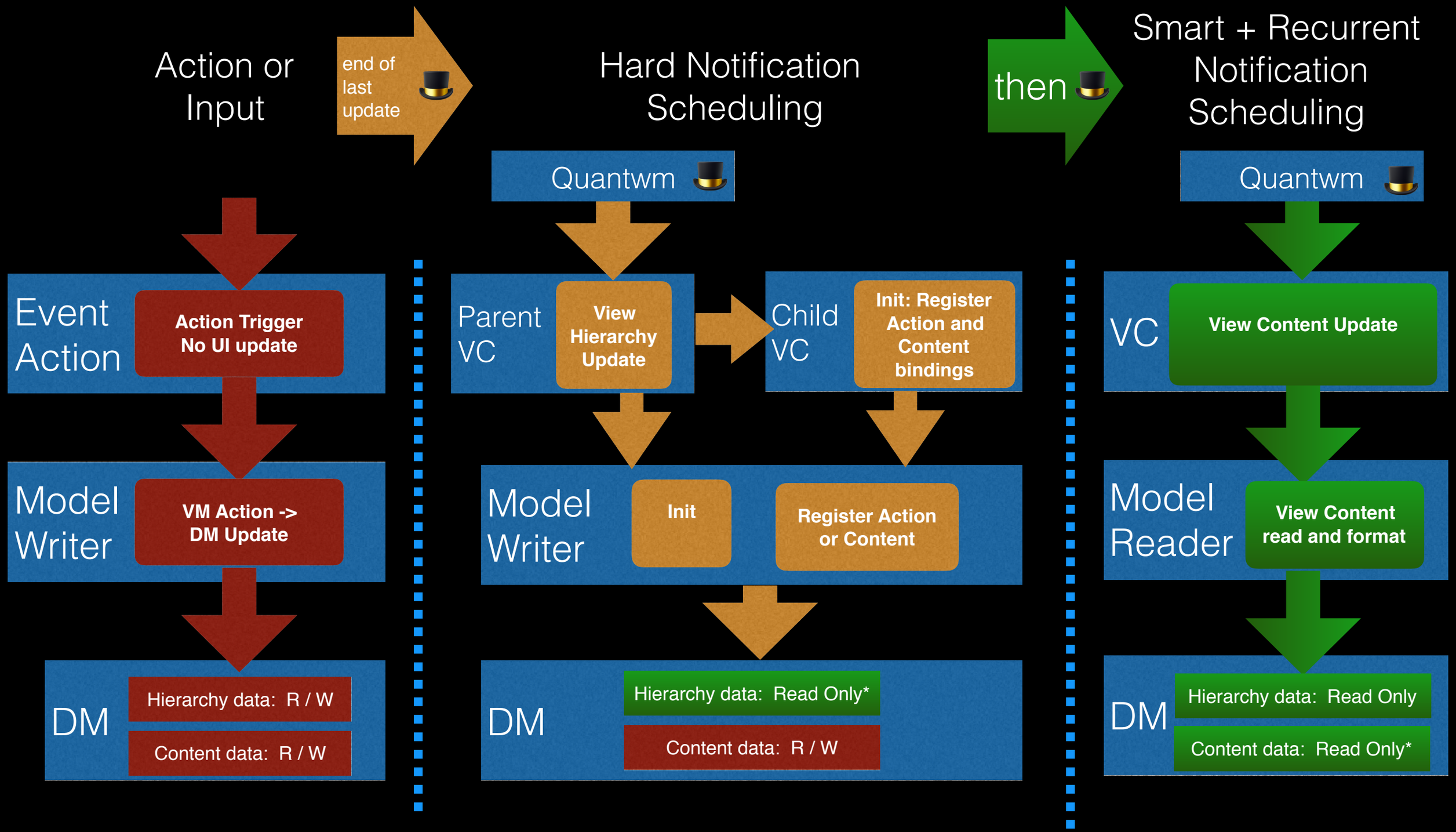


… to Quantwm Architecture



Each VC / VM code is associated to a specific context.
DM access is conditioned by data type and context.

… to Quantwm Architecture



Each VC / VM code is associated to a specific context.
DM access is conditioned by data type and context.

Quantwm Concepts

- Data Centric
- Activity
- Update Transaction
- Registration: Read and Write Set
- Holistic architecture
- Model Scheme

Data Centric

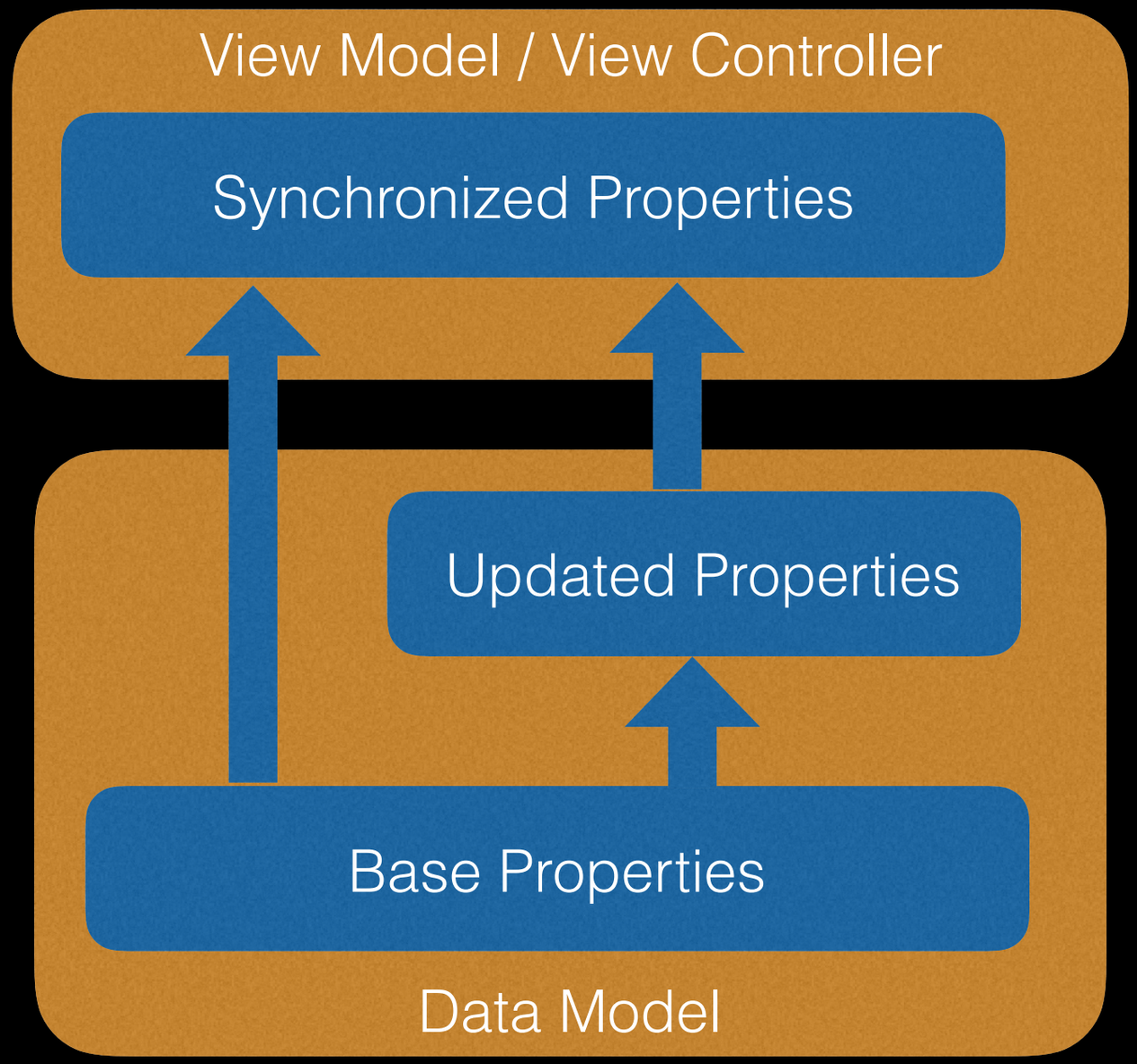
Data Model

Data Model is the owner of the state of the application.

Data Model is responsible of Storing and Archiving the all the data state of the application.

Objective: Control the Model, and you control the whole application: Undo / Save / Load / Version Browser / State Restoration / ...

Property update flow

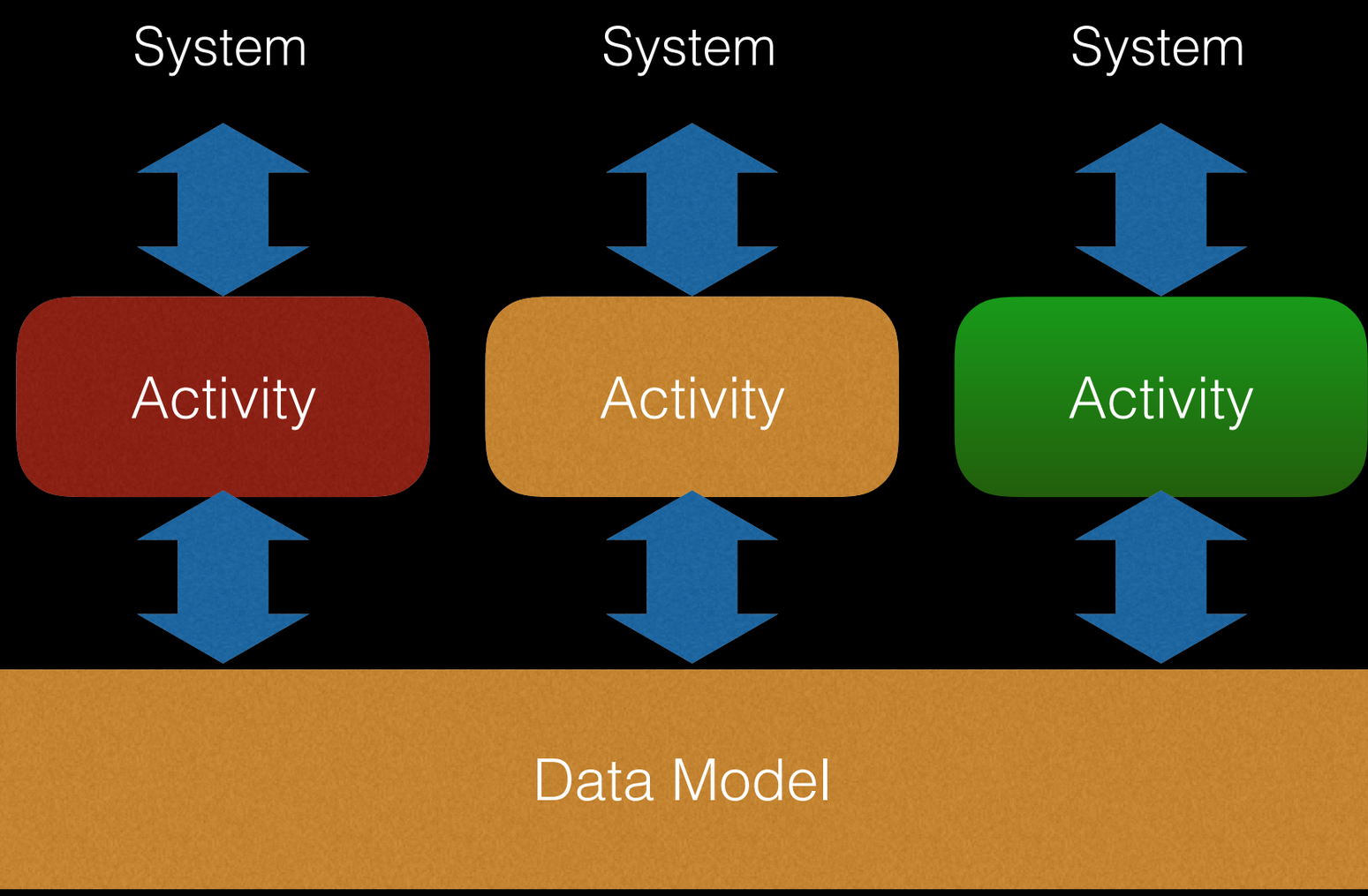


3: Set by Smart Scheduling Notification

2: Set by Hard Scheduling Notification

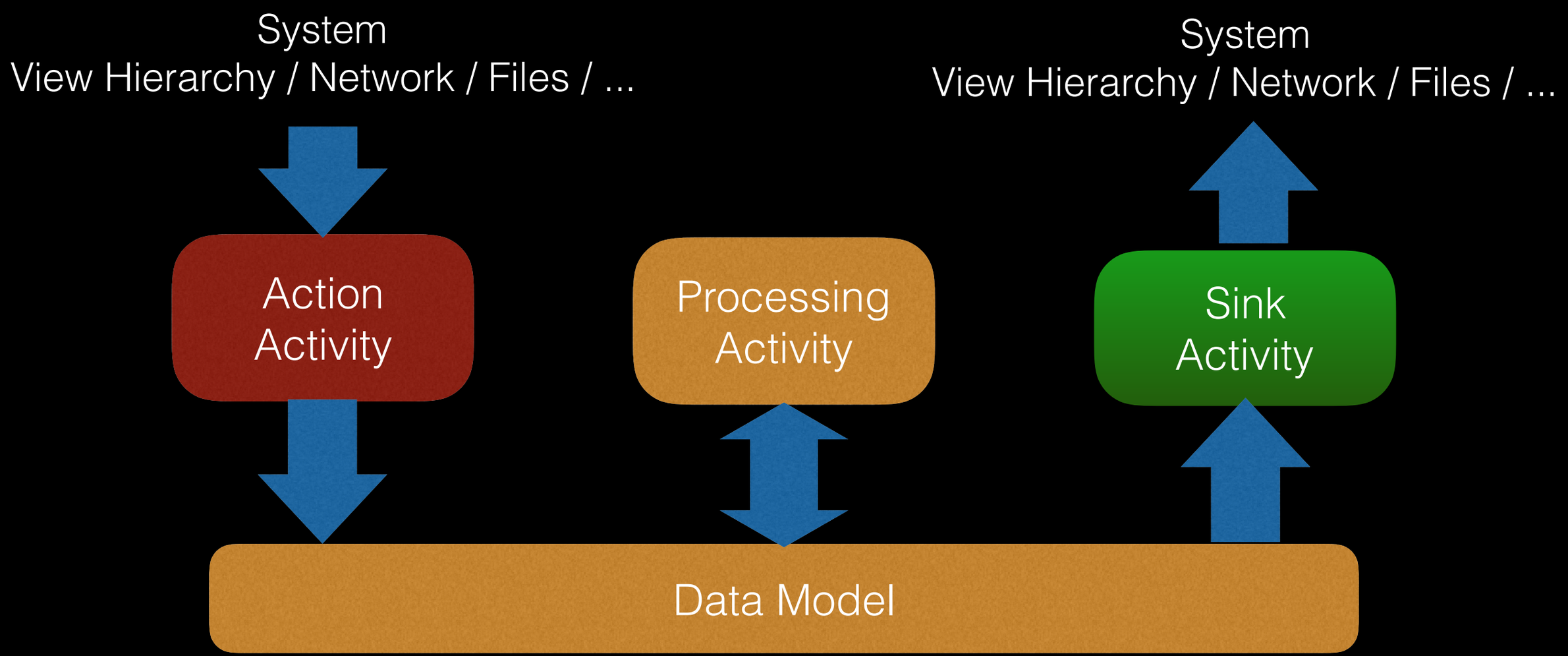
1: Set by Action

Activity



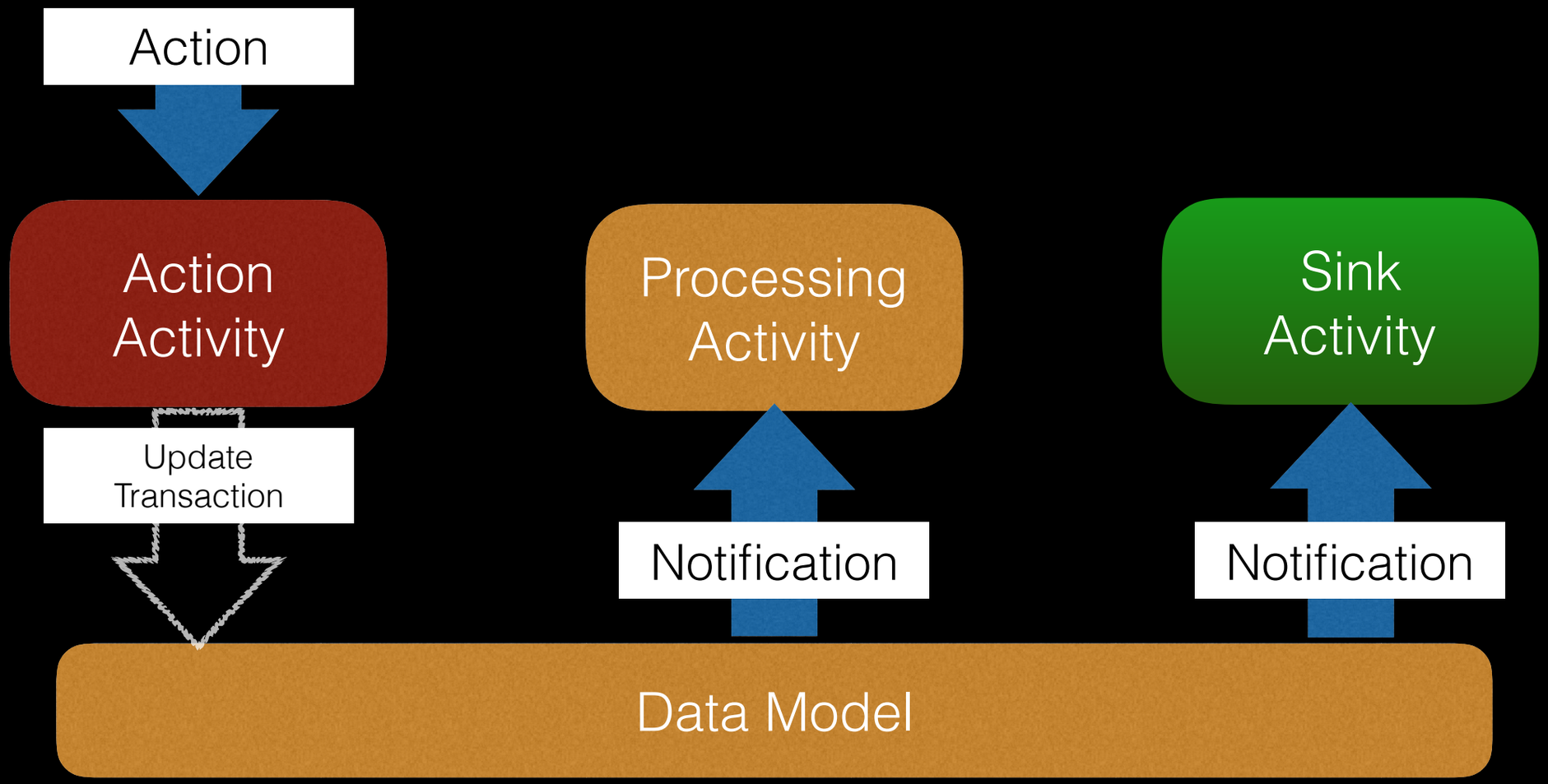
Activities are interactions between the Data Model and the system.

3 Types of Activities



In order to define a clear execution context,
each Activity shall belong to one of these 3 types:
Action Activity - Processing Activity - Sink Activity

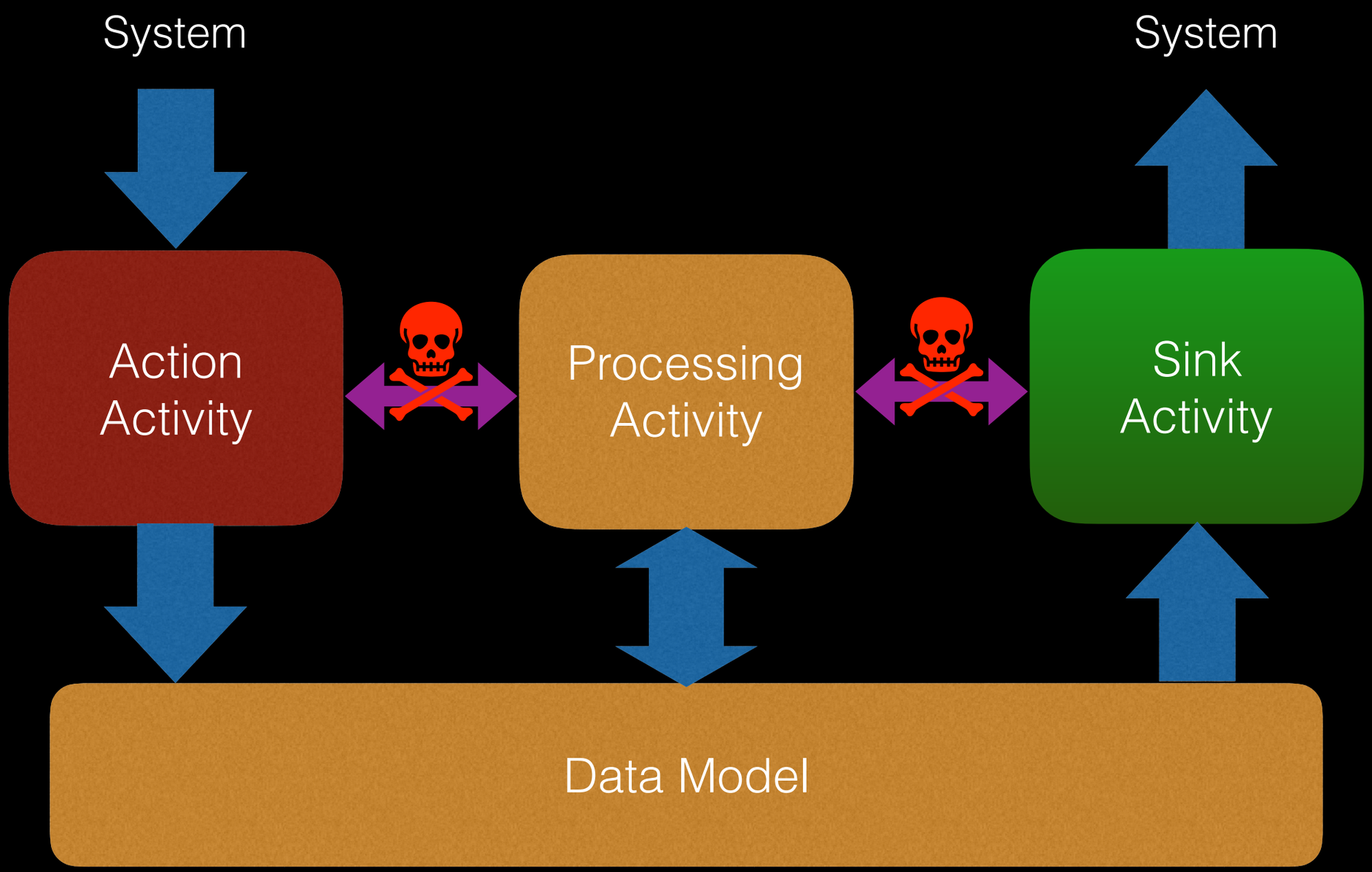
Activity trigger



Action Activity are triggered by Action.
Data and Sink Activities are triggered by notifications.

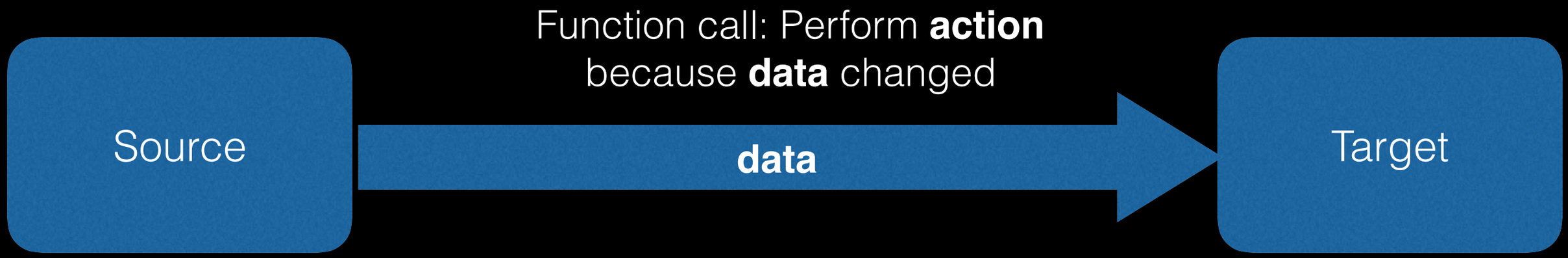
An activity includes the set of all functions executed and variables accessed during a synchronous interaction
i.e the call graph between the system and the data model.

Fully decoupled



Ideal activities have no shared data or common functions,
and are thus **fully decoupled, with a single execution context.**
All shared data shall be owned by the Data Model.

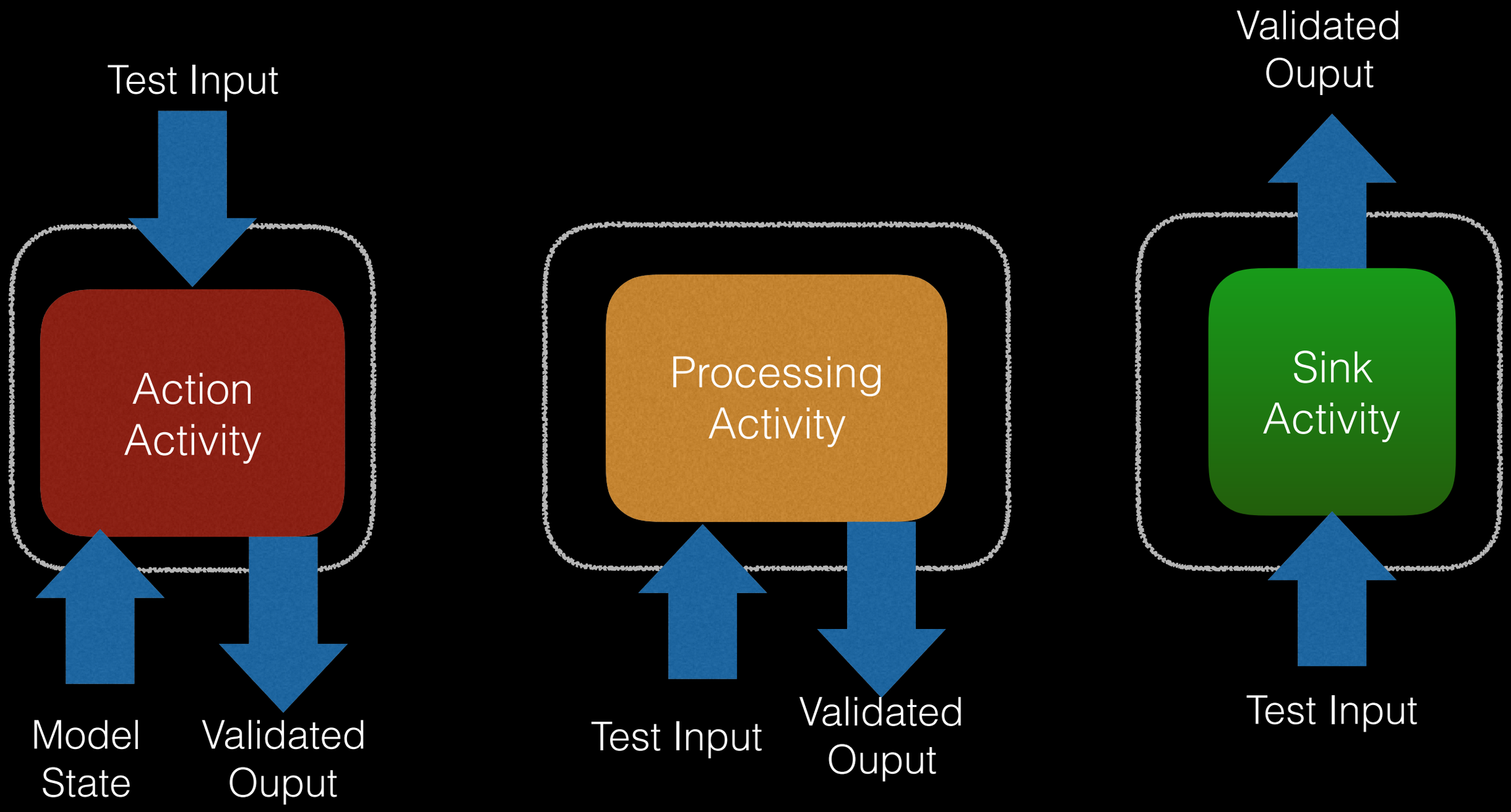
Why decoupling is the key ?



The function call carries several dimensions :

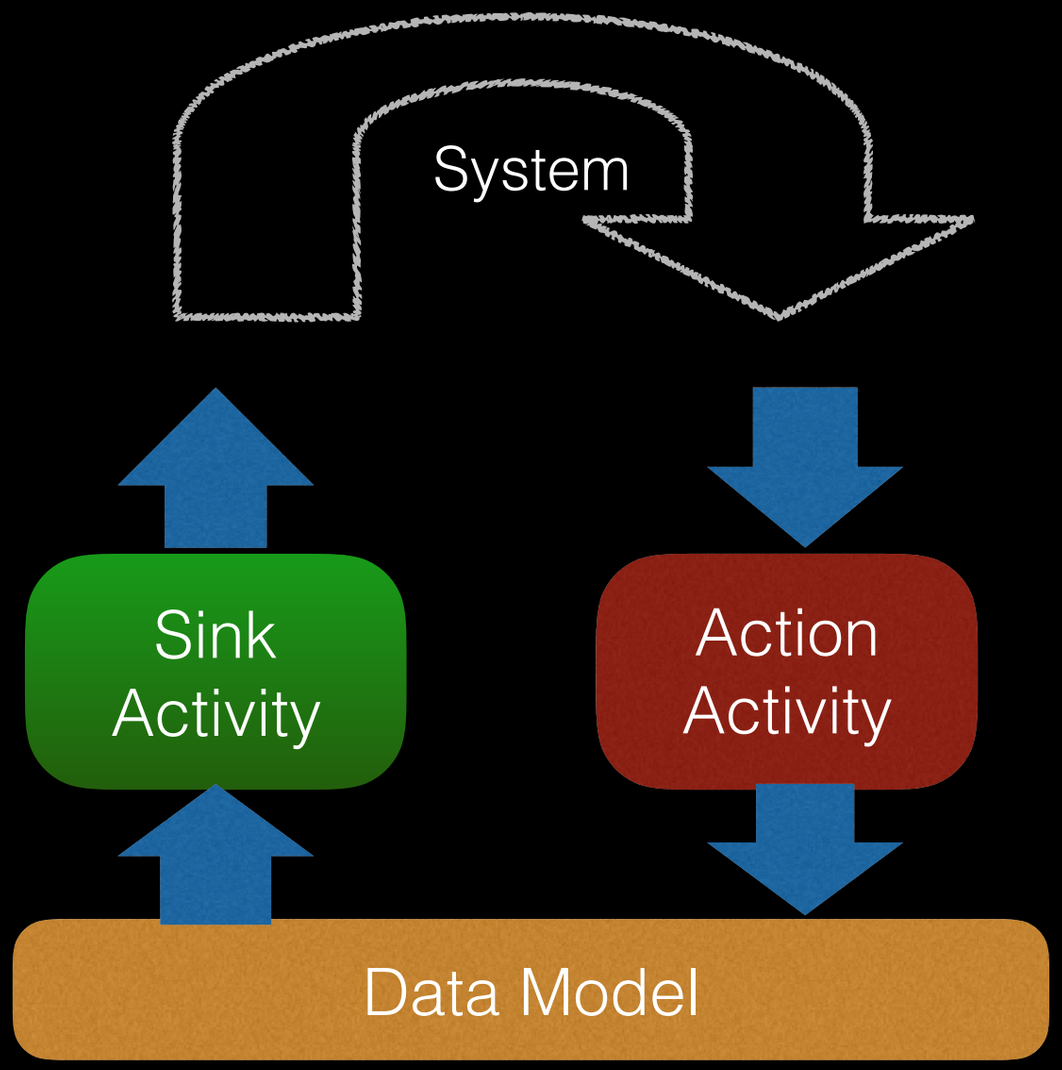
- **Data dimension**: Is the source the only owner of changed data ? Or shall target register to the change of this data ?
- **Time dimension**: Is "now" the right time to perform this action verus the rest of the system ? Is action dependening on other data originating from other sources ?
- **Reversibility dimension**: On Undo, how is this action reversed ?
- **Ownership dimension**: Is this call associated to a transfer of ownership of the data ? Is the data pure or is it a system element (like UI view) ? And undo again ?

Performing all the function calls inside a fully decoupled activity with defined context manages all these dimensions.



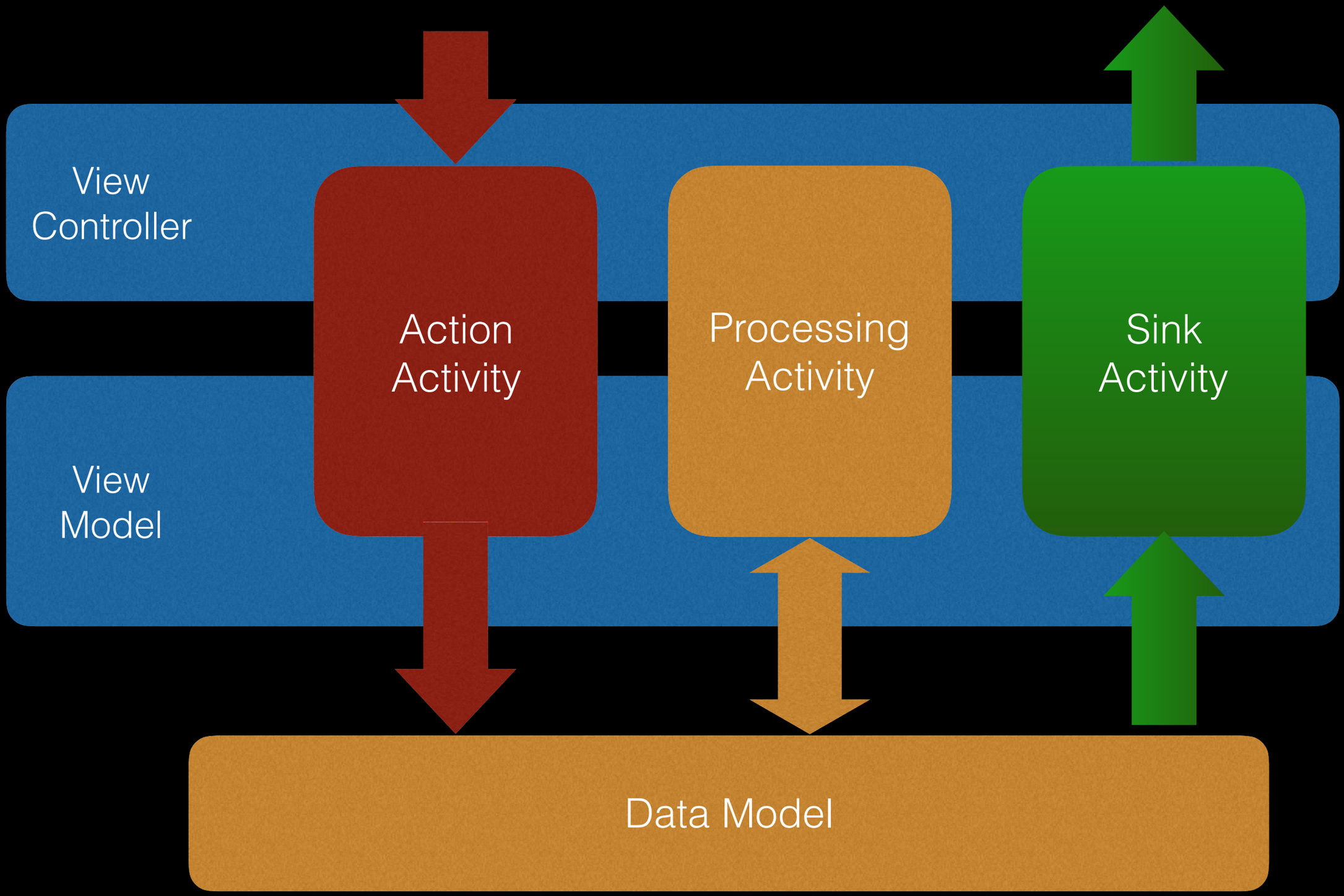
No dependency between activities
Context Reduction -> Less cases -> Easier Testing
Execution context is defined by the input.
Ideal Activity is a pure functional call graph

Synchronous

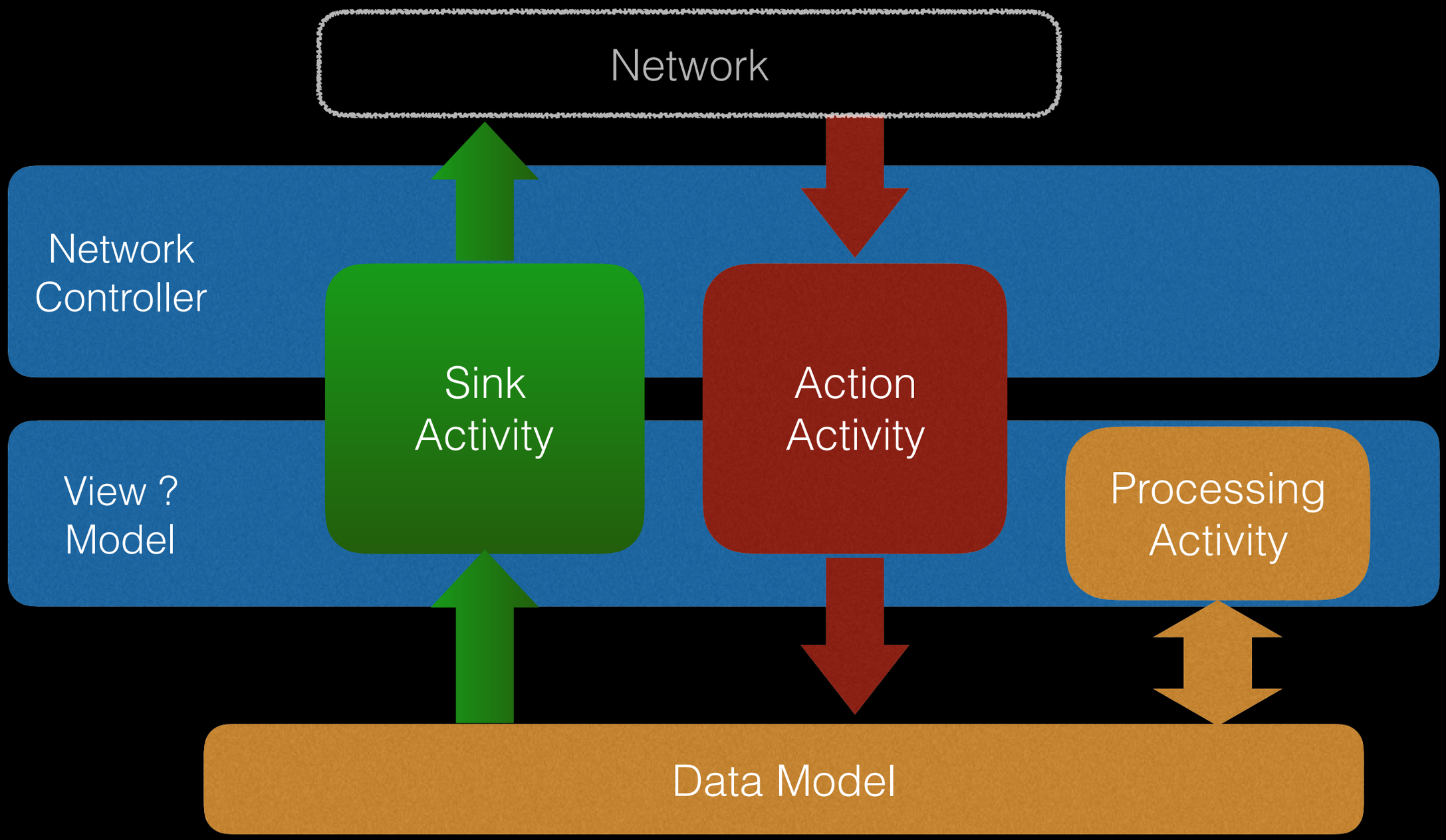


As an activity is synchronous, any asynchronous processing is decomposed into a synchronous processing/sink activity and a synchronous action activity.

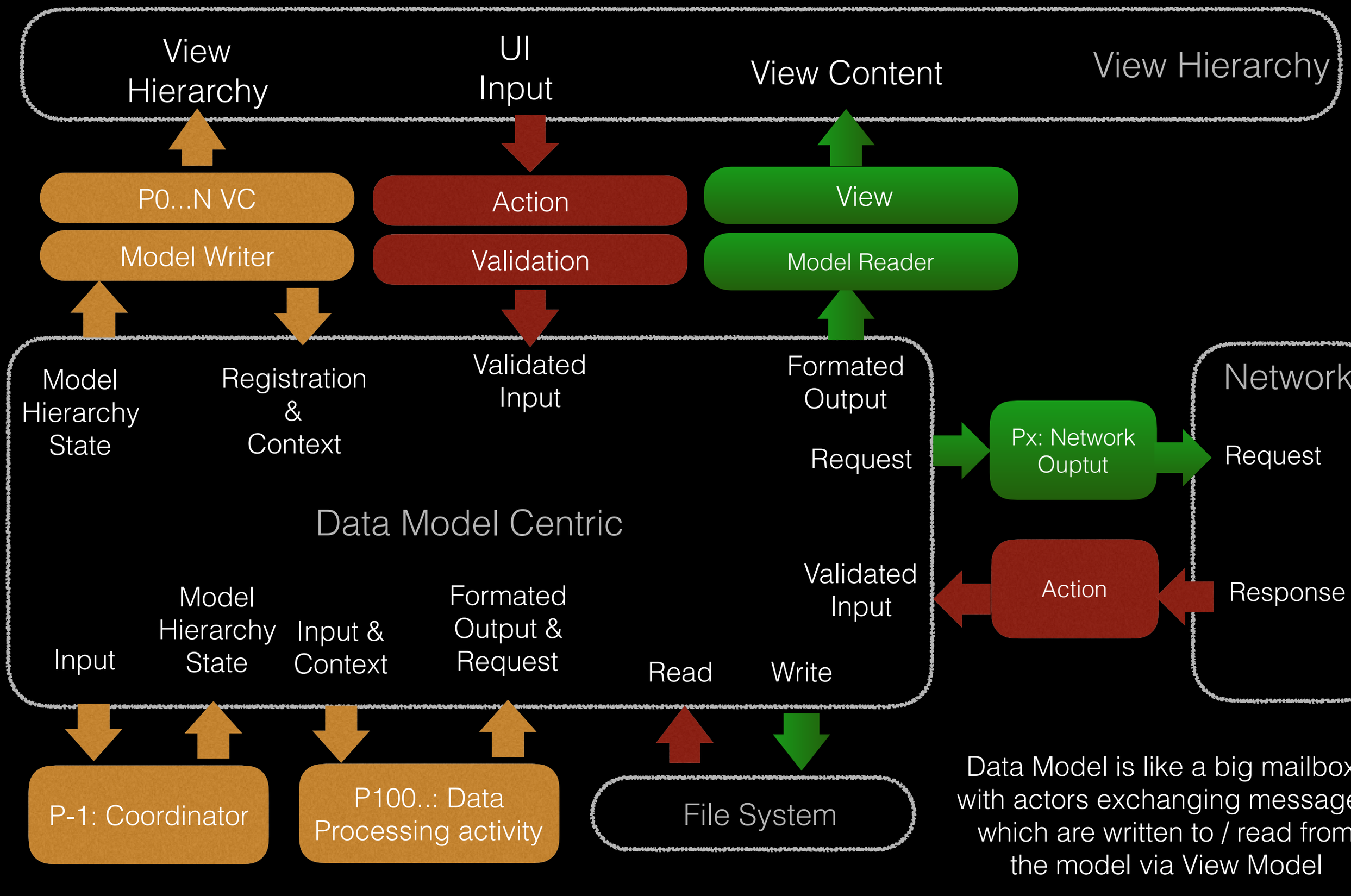
Application to MVVM



Application to Network

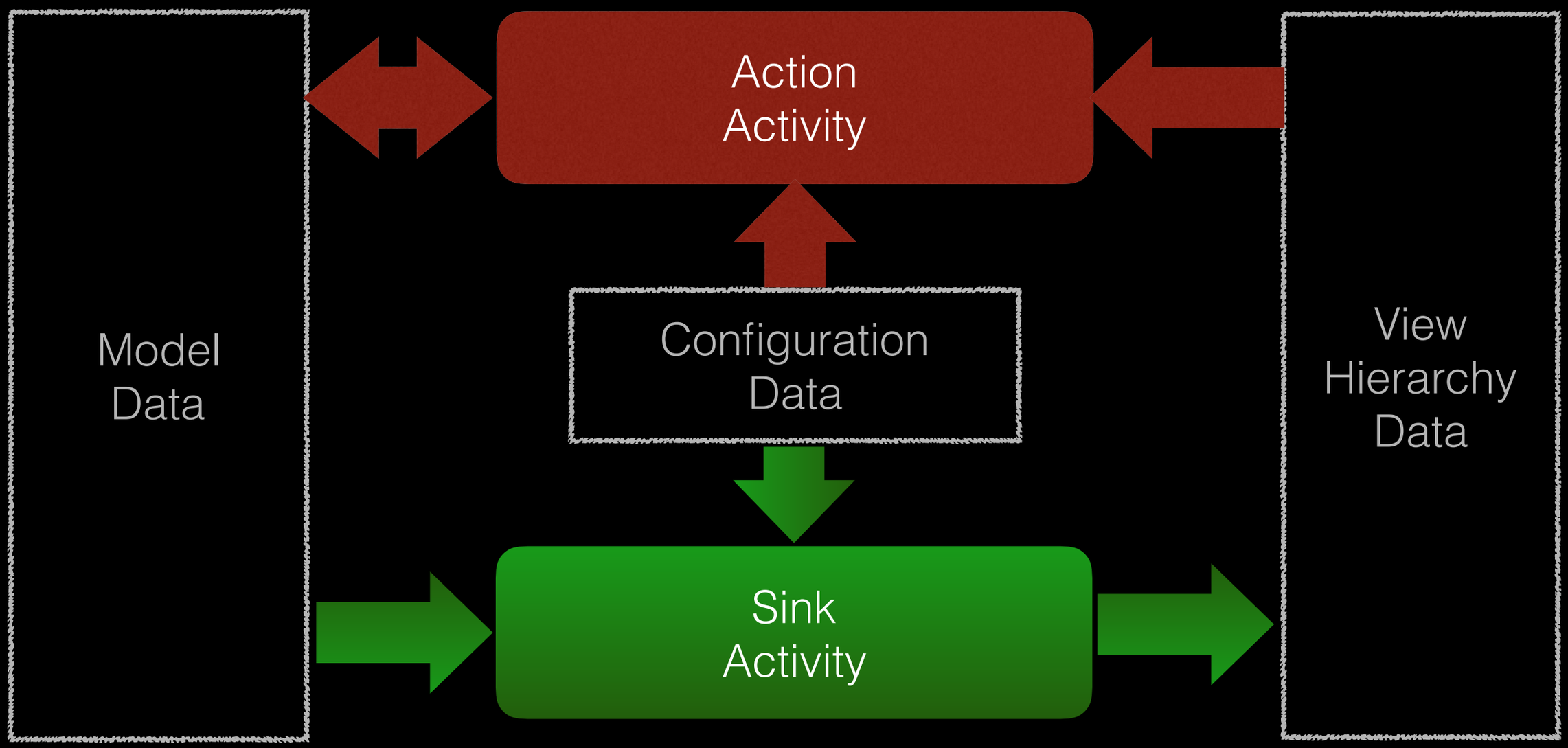


Architecture Overview



Data Model is like a big mailbox, with actors exchanging messages which are written to / read from the model via View Model

VC / VM Variables classification



Clearly identify the type of each variable ...

Data Storage

Configuration Data
Init() / State Restoration.

Configuration Data shall be saved and configured by parent controller, and/or saved via State Restoration

View Hierarchy Data
viewDidLoad()

Owned by View Hierarchy
Configured and Managed from Model Data

Model Data
viewWillAppear()
Smart Notif

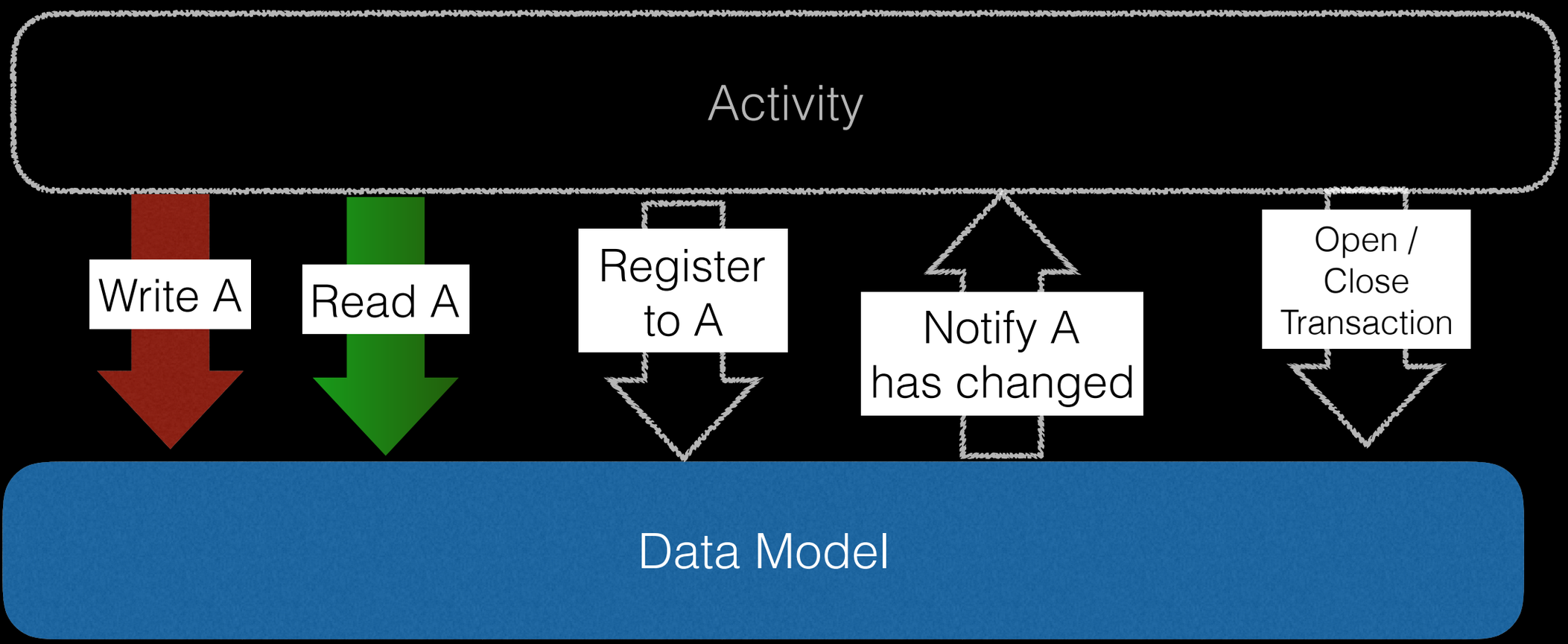
Shared data is managed by Data Model
Persistent data is managed by Data Model
All data affected by Undo or Version Browser is managed by Data Model

Other ?

Consider refactor to pure functional

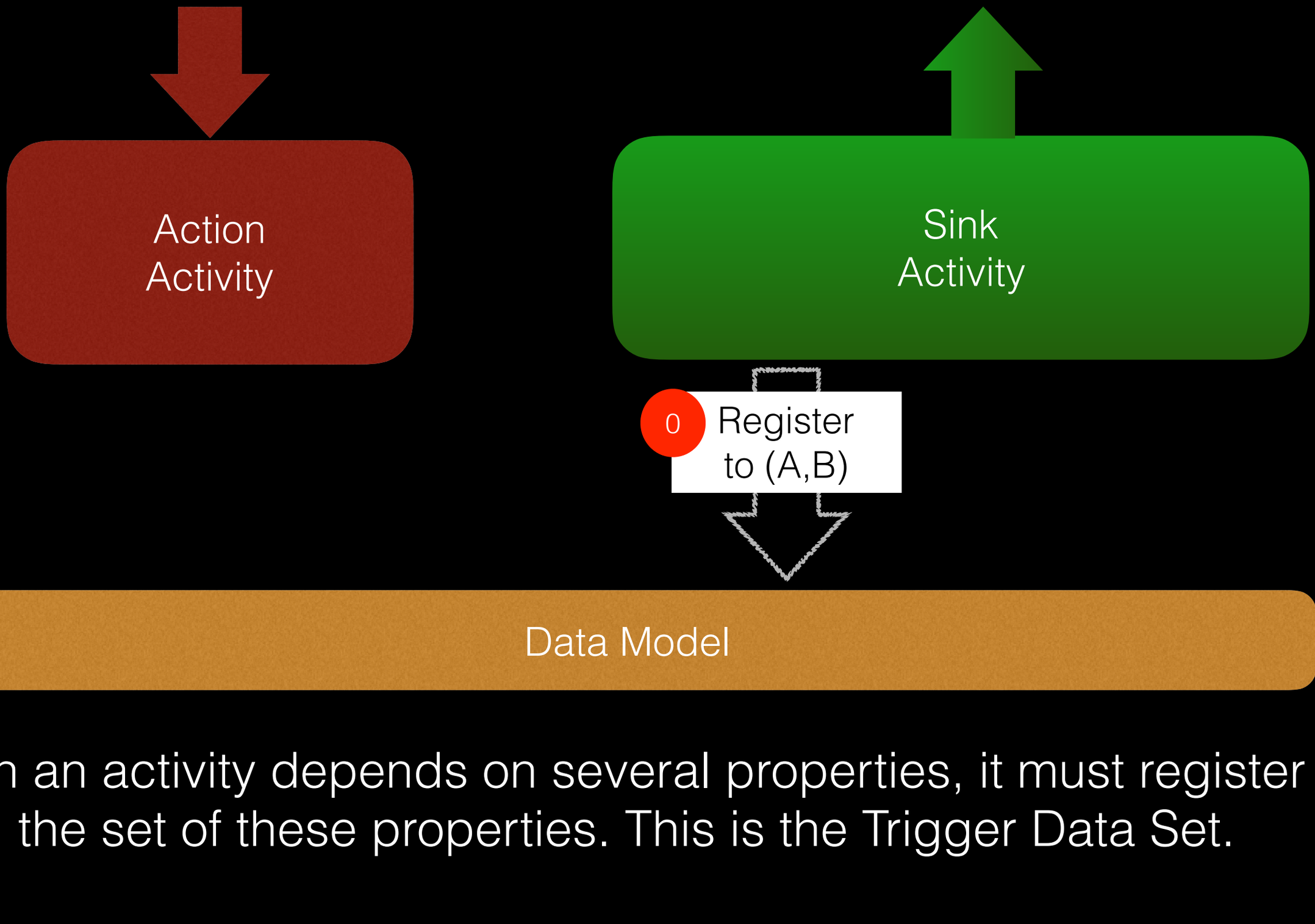
... and manage its persistency accordingly.

Activity - Model communication



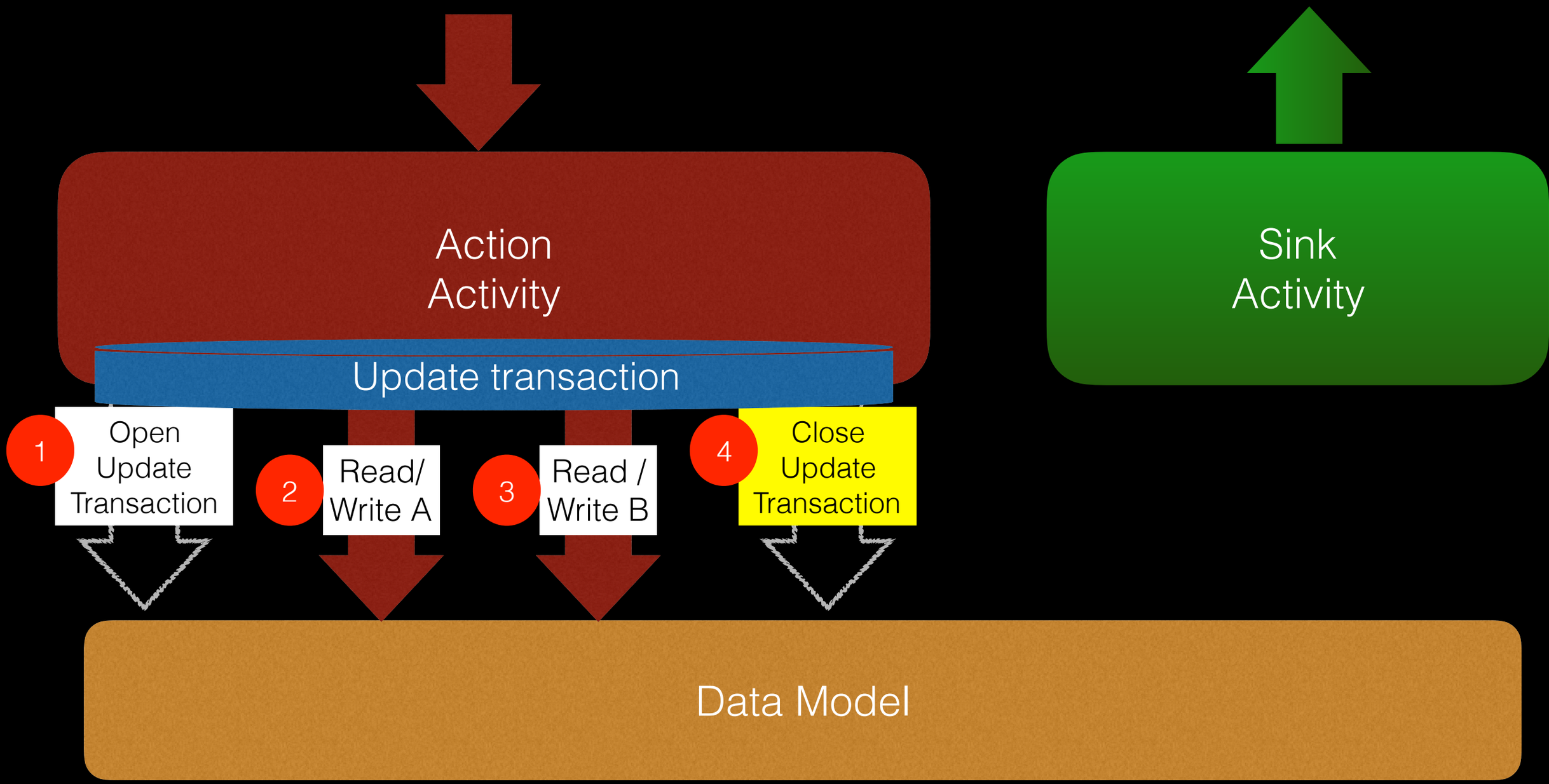
The model is passive, but provide **synchronous** notification service. All communications between Processors and Model shall use this service.

Example: Write (A,B), Read (A,B) - 1/3



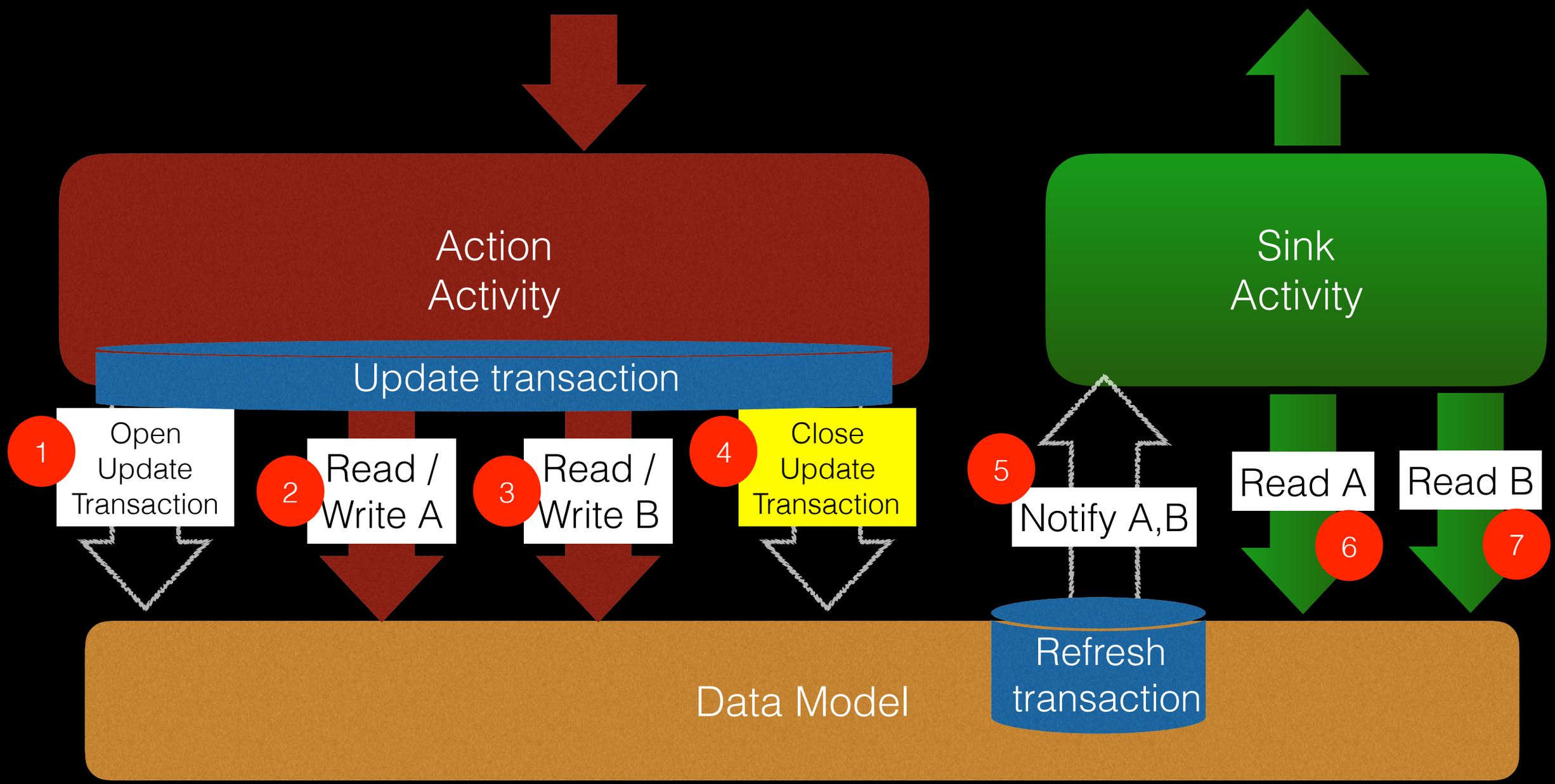
When an activity depends on several properties, it must register to the set of these properties. This is the Trigger Data Set.

Example: Write (A,B), Read (A,B) - 2/3



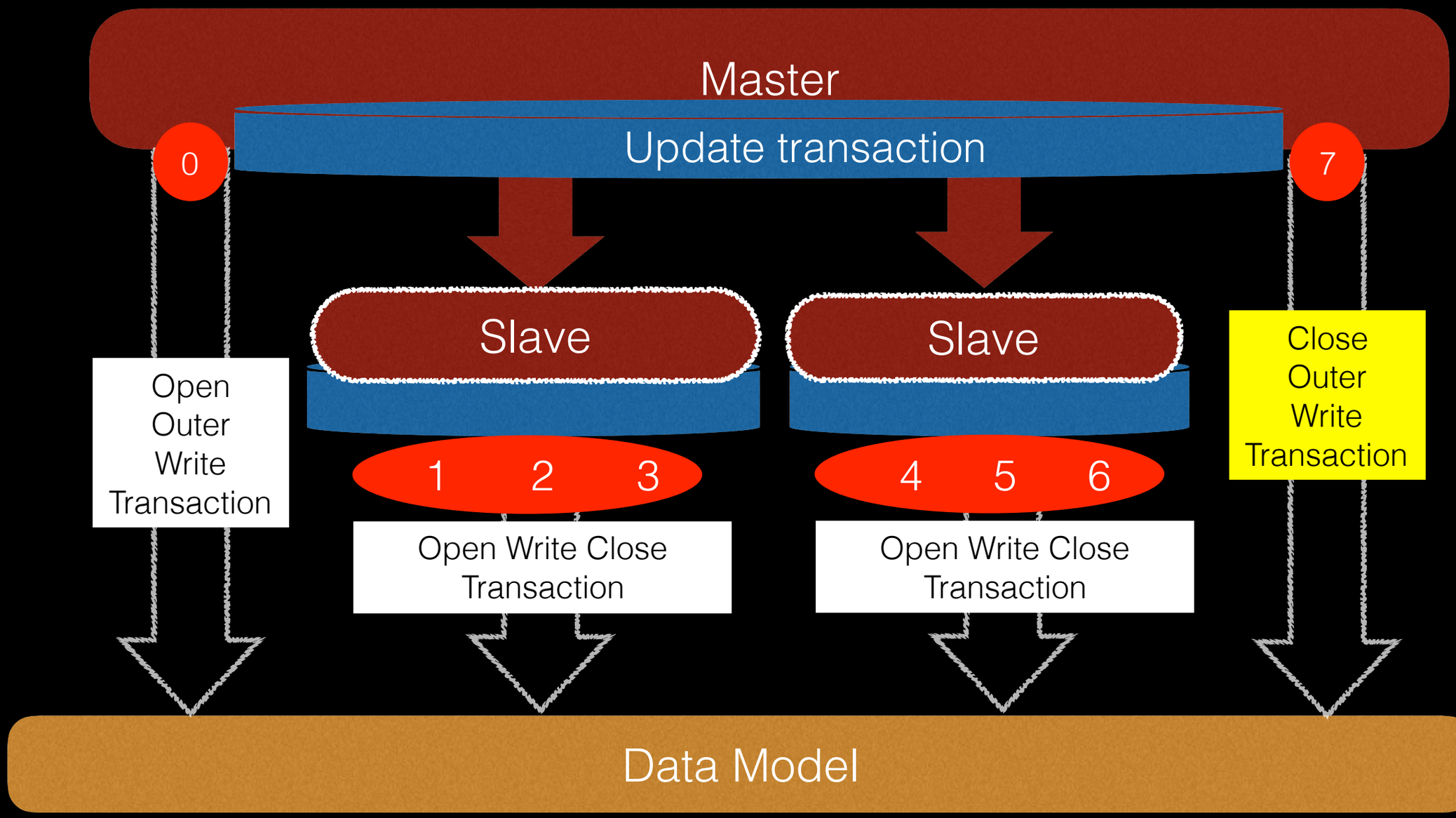
On the reception of a system event, an Action processor shall synchronously open a **update transaction**, perform all its read/ write with the model, and close the update transaction.

Example: Write (A,B), Read (A,B) - 3/3



The Close Write Transaction triggers the notification.

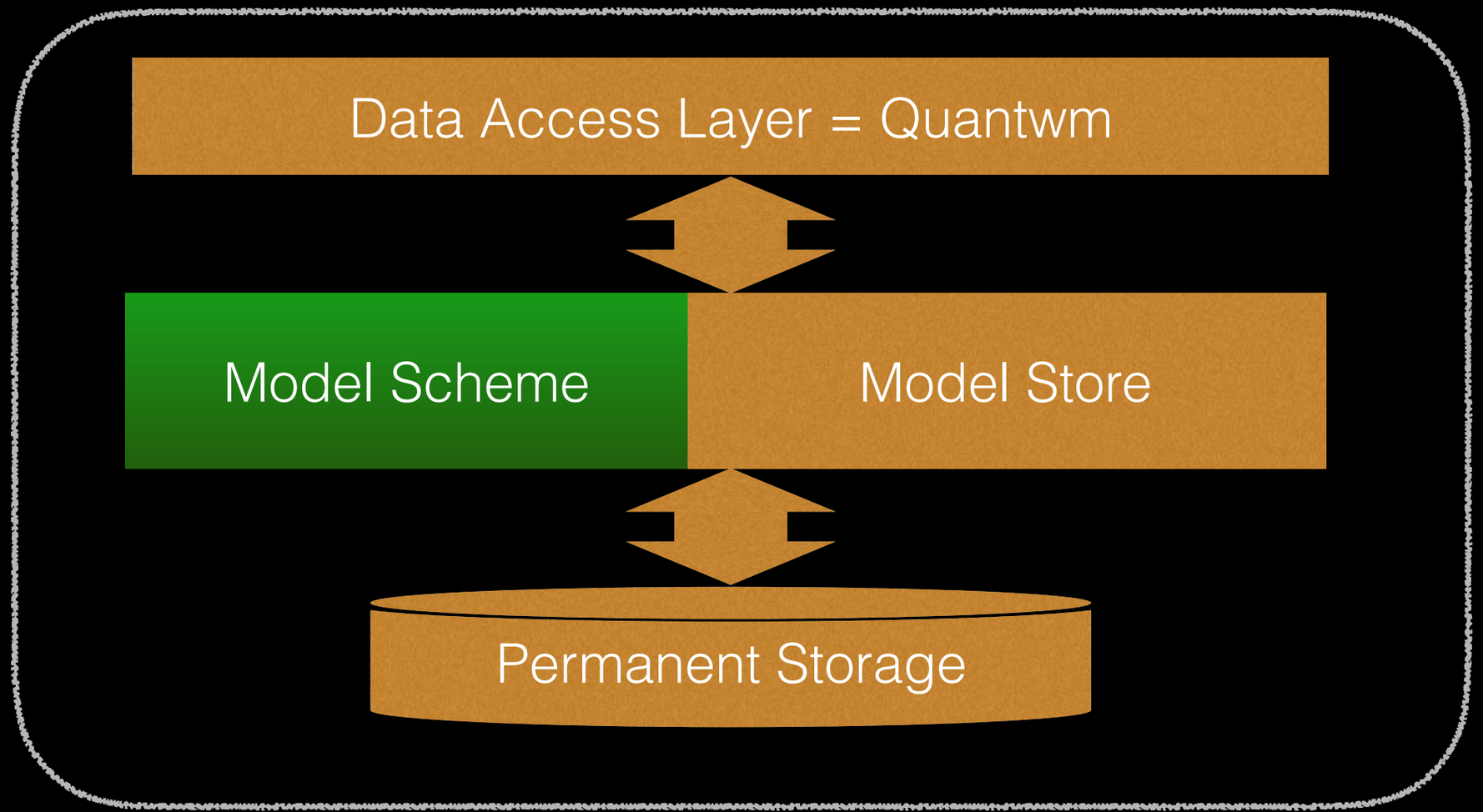
Nested update transaction



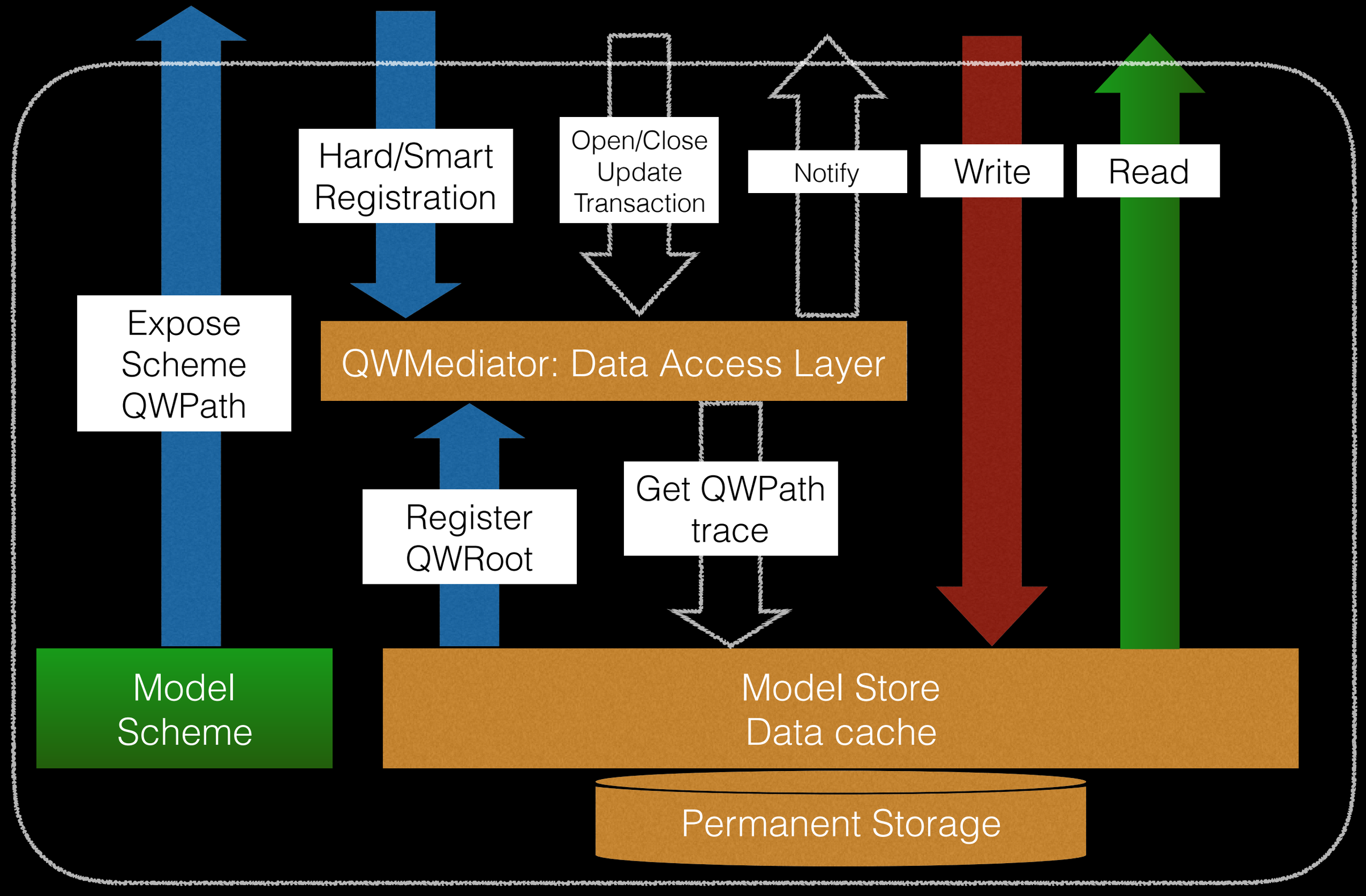
Update transactions can be nested.
Only the outer update transaction will trigger the notifications.

Data Model

Data Model is decomposed into 4 components:



Data Model


Hard/Smart
Registration
Open/Close
Update
Transaction
Notify
Write
Read
Expose
Scheme
QWPath
QWMediator: Data Access Layer
Register
QWRoot
Get QWPath
trace
Model
Scheme
Model Store
Data cache
Permanent Storage

Registration Concept

Static definition

QWRegistration

QWMap =
Set<QWPath>

Model
Scheme

Activity

```
func registerObserver(
    registration: QWRegistration,
    target: AnyObject,
    notificationClosure: @escaping () -> ())
```

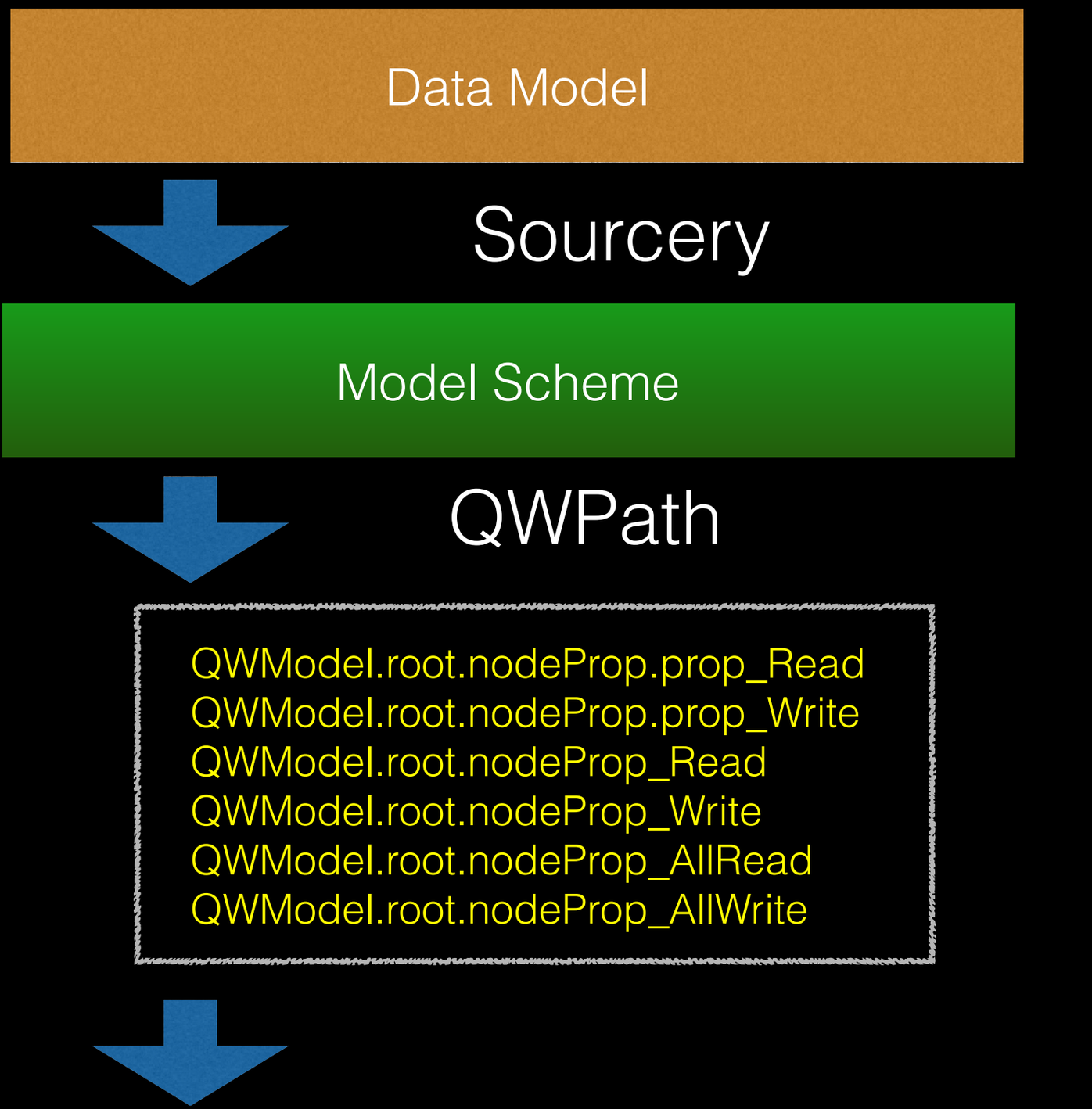
QWMediator
Data Access Layer

QWRegistration are defined as static immutable objects, defining registration type, and properties read and written. Activity can register to nodes / properties which are not yet created, and are notified on creation / update / deletion.

Model Scheme and QWPath

Model Scheme is generated automatically via Sourcery (thanks !!!!!!!)

Static QWPath are easily generated from Model Scheme with auto-completion, for registration.

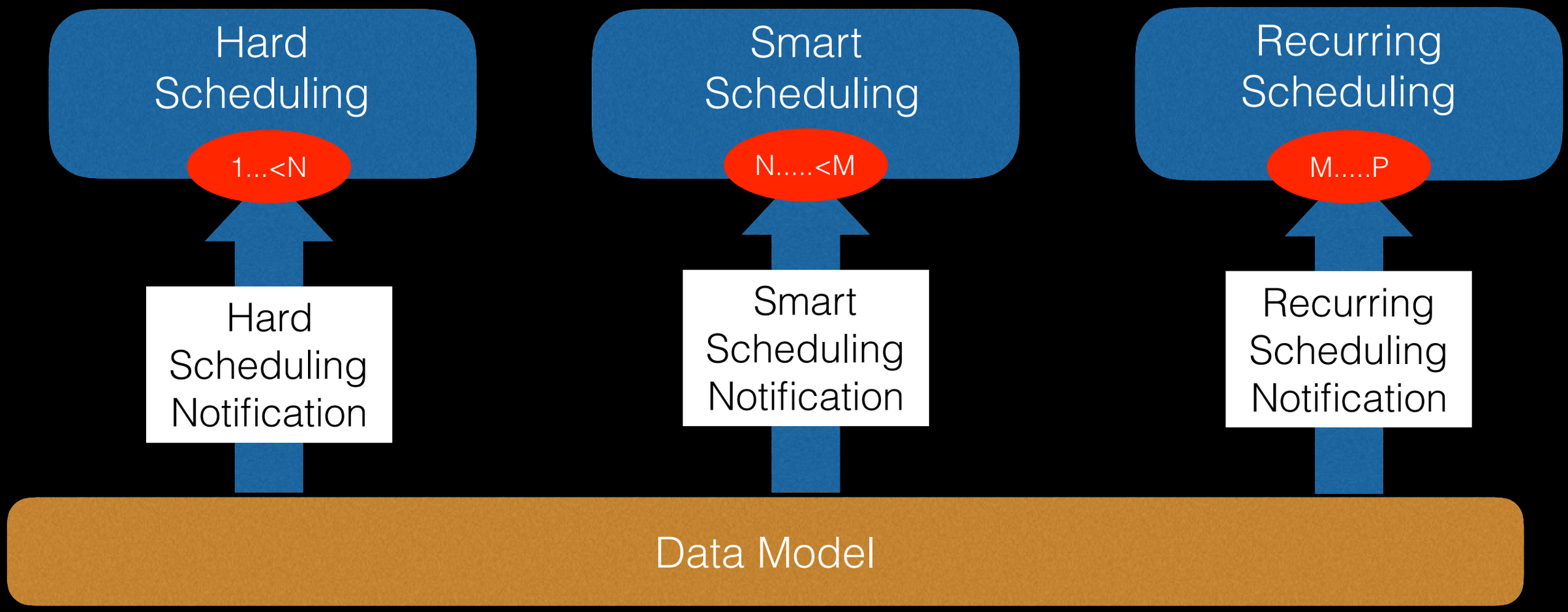


Register Observer

```
override  func viewWillAppear(_ animated: Bool) {
     super.viewWillAppear(animated)
    let viewModel = PlaylistsCollectionViewModel(
         mediator: qwMediator,
         owner: "PlaylistsCollectionViewController",
         playlistCollectionModel: model)
    self.viewModel = viewModel
    viewModel.updateActionAndRefresh {
         viewModel.registerObserver(
             registration: <<QWRegistration>>,
             target: self,
             selector: #selector(PlaylistsCollectionViewController.titleUpdated))
     }
 }
override func viewDidDisappear(_ animated: Bool) {
    super.viewDidDisappear(animated)
     viewModel?.unregisterDataSet(target: self)
     viewModel = nil
}
```

Exist in target/selection or target/closure
Weak capture of the target, and unregister on target release
QWRegistration is defined independently

QWRegistration types



Notifications are specialized into Hard and Smart and Reccuring notifications, delivered in this order.

Hard Registration

```
class QWRegistration
...
public convenience init(hardWithReadMap readMap: QWMap,
                        name: String,
                        schedulingPriority: Int,
                        collectors: [QWCollector] = [])
```

- Hard registration are triggered if any readMap property has changed since last trigger, or if a collector is triggered.
- Hard notification trigger order is defined by their increasing scheduling priority.
- The processing shall be enclosed in an update transaction.
- May perform any read, write and registration of lower priority.
- Same code can be called either by an action or a hard notification.
- Triggered too by the collector activation

Smart Registration

```
class QWRegistration
...
public convenience init(smartWithReadMap readMap: QWMap,
            name: String,
            writtenMap: QWMap? = nil,
            collectors: [QWCollector] = [])
```

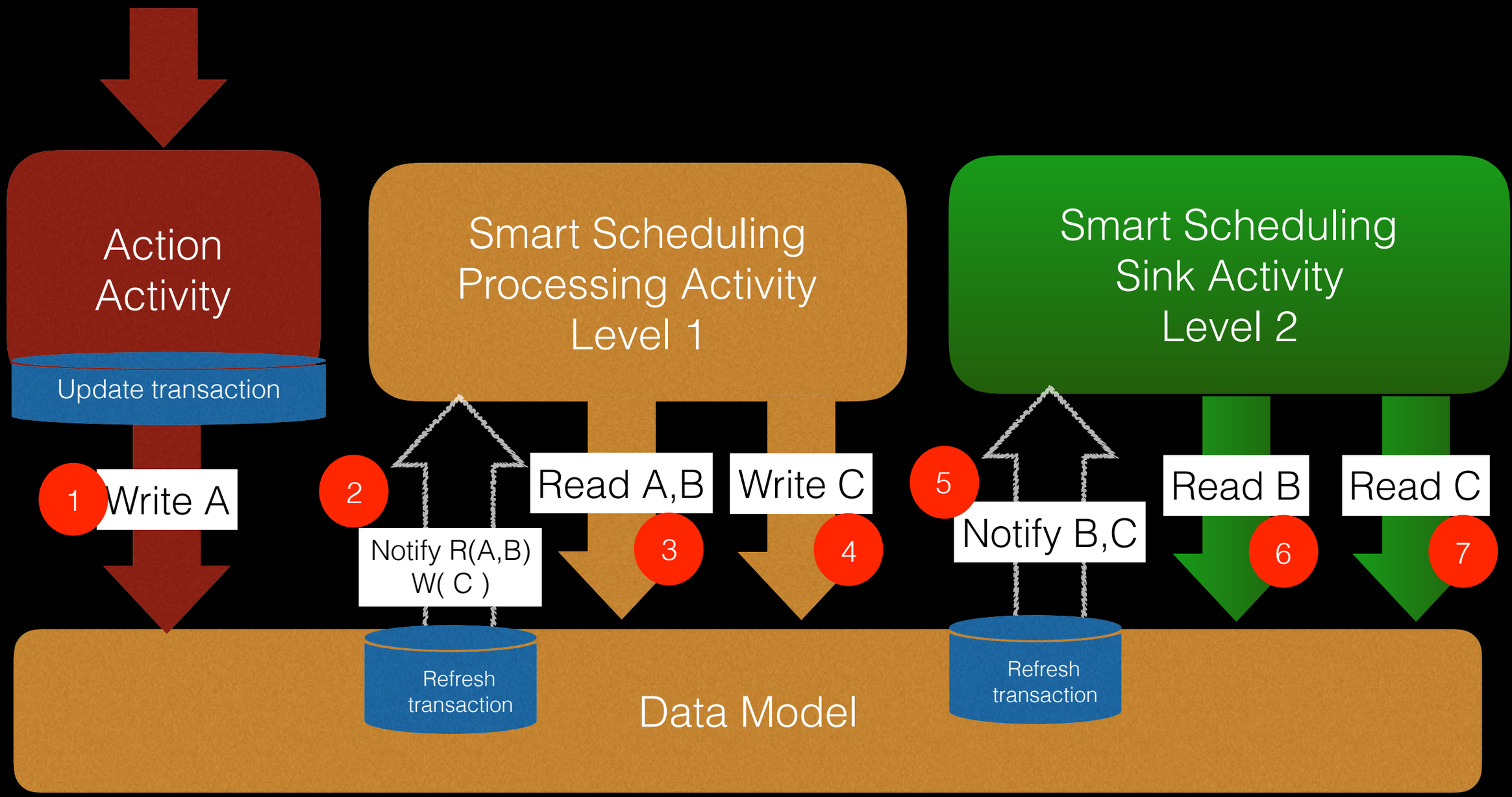
- Smart registration are triggered if any readMap property has changed since last trigger, or if a collector is triggered.
- Can only read properties from readMap and collectors read and write map.
- Can only write the writtenMap properties.
- No transaction needed.
- Smart notifications trigger order is defined by their level, which is computed based on property dependency

Smart scheduling: Property Level

Smart scheduling is ordered depending on property level.
Each processing increase the level of a property.

A Smart Notification is sent only once during the event loop, once all the registered properties are stable.

Smart order Example



Recurring Registration

```
class QWRegistration
...
public convenience init(recurringWithName name: String)
```

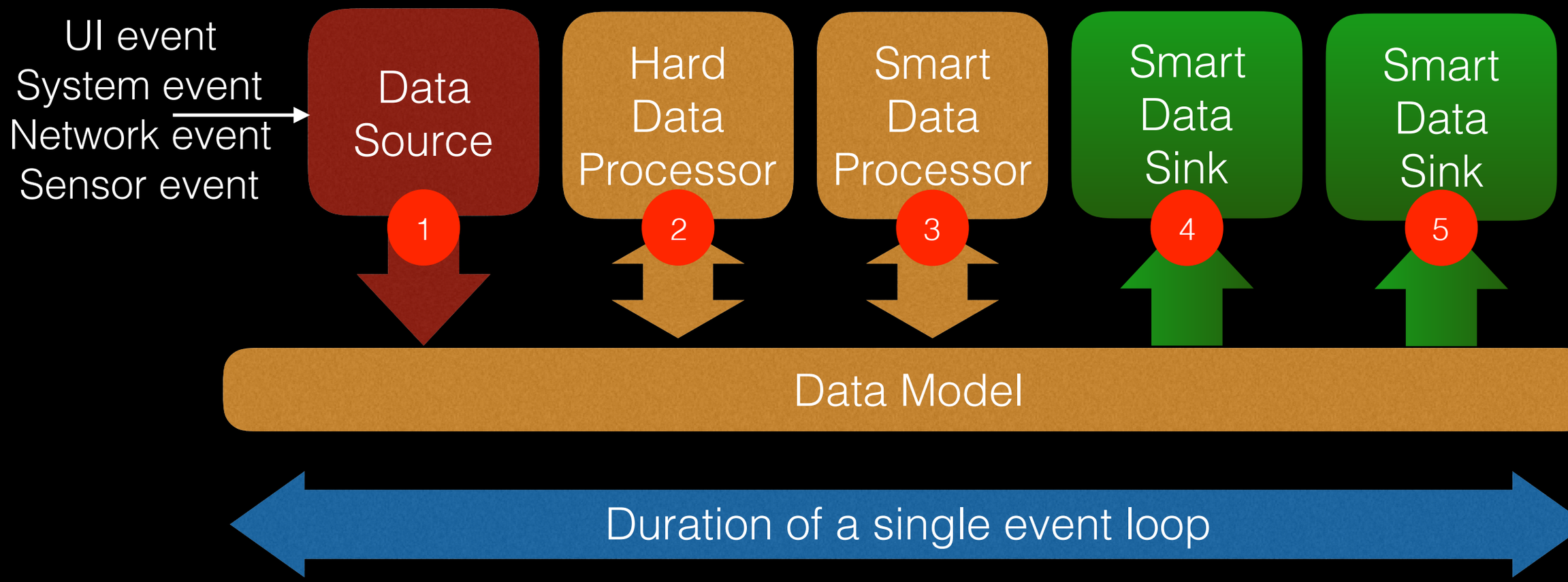
- Recurring registration are triggered at each update transaction
- Trigger order is after all other notifications
- Can read any properties
- Can not write any property.
- No transaction needed.

Collector Registration

```
class QWCollector: QWRegistration
...
public convenience init(collectorWithReadMap collectedMap: QWMap,
                        name: String,
                        writtenMap: QWMap?,
                        collectors: [QWCollector] = [],
                        schedulingPriority: Int? = nil)
```

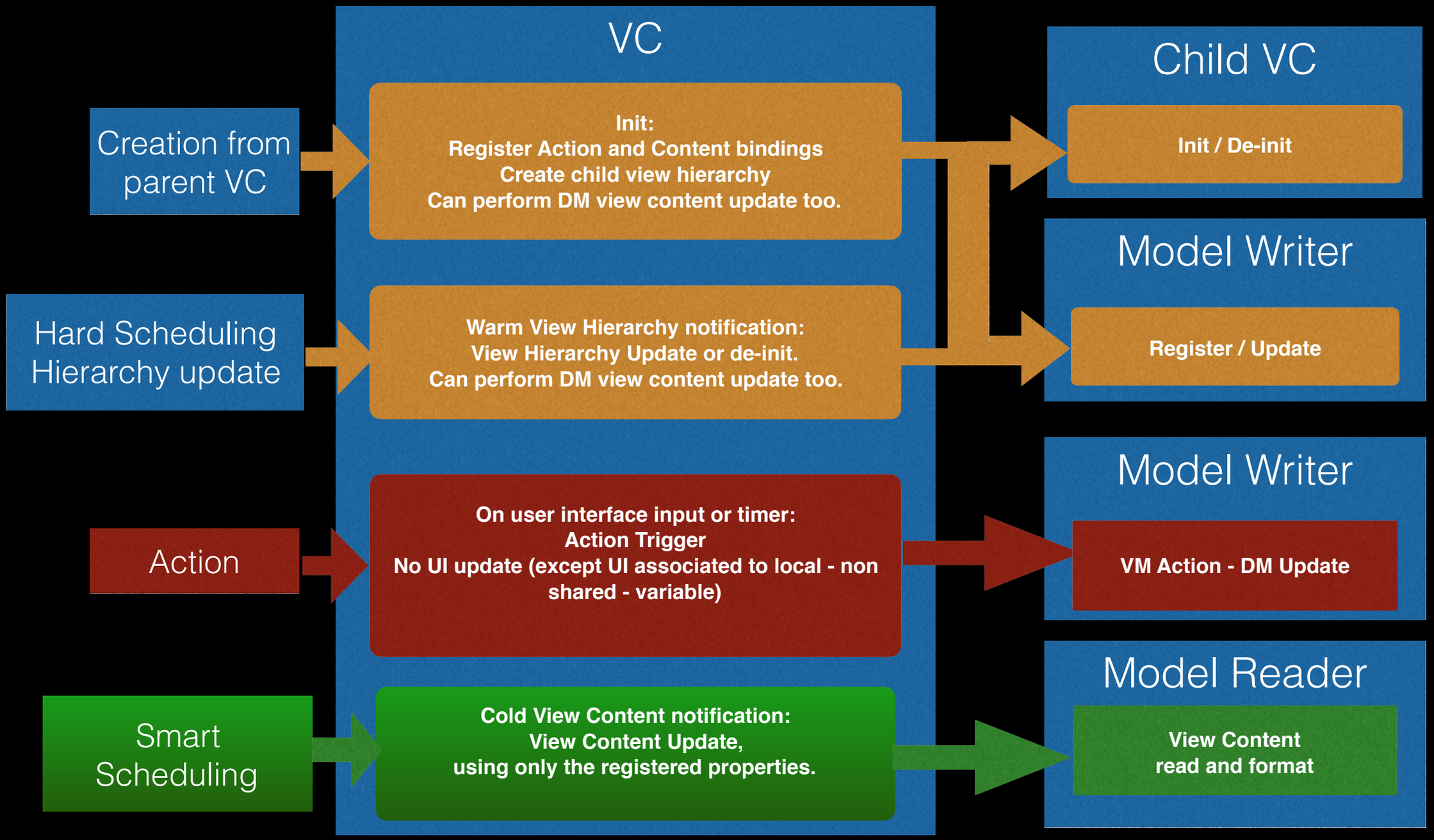
- Collector registration behave like a hard or smart registration, depending on schedulingPriority defined or not.
- Hard Collector can write any property, the writtenMap is only here for collector dependency.
- Smart Collector can only write properties from their writtenMap.

Time Control



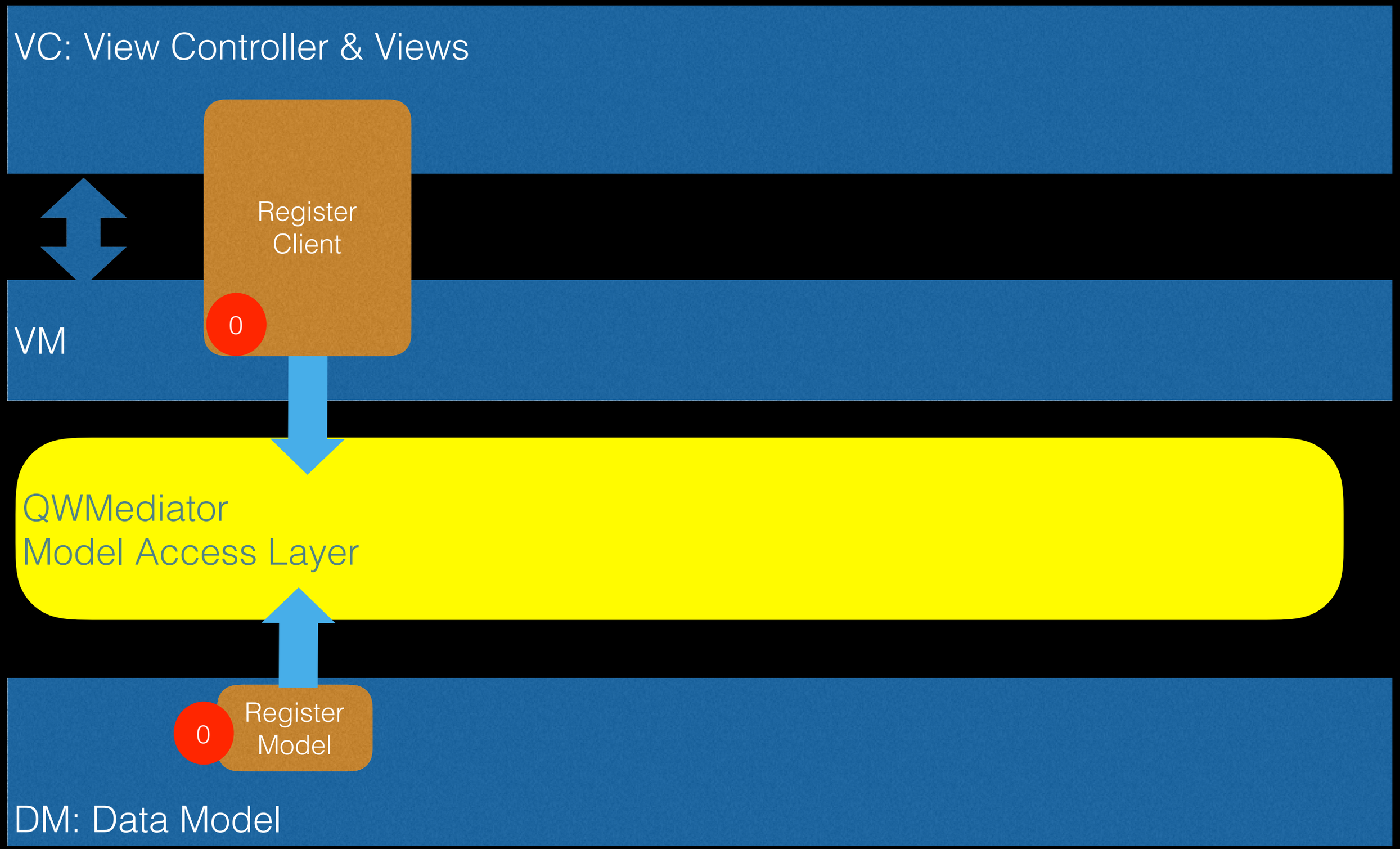
Determinist and synchronous event loop intra-scheduling eliminate a large part of software execution variability

MVVM Example

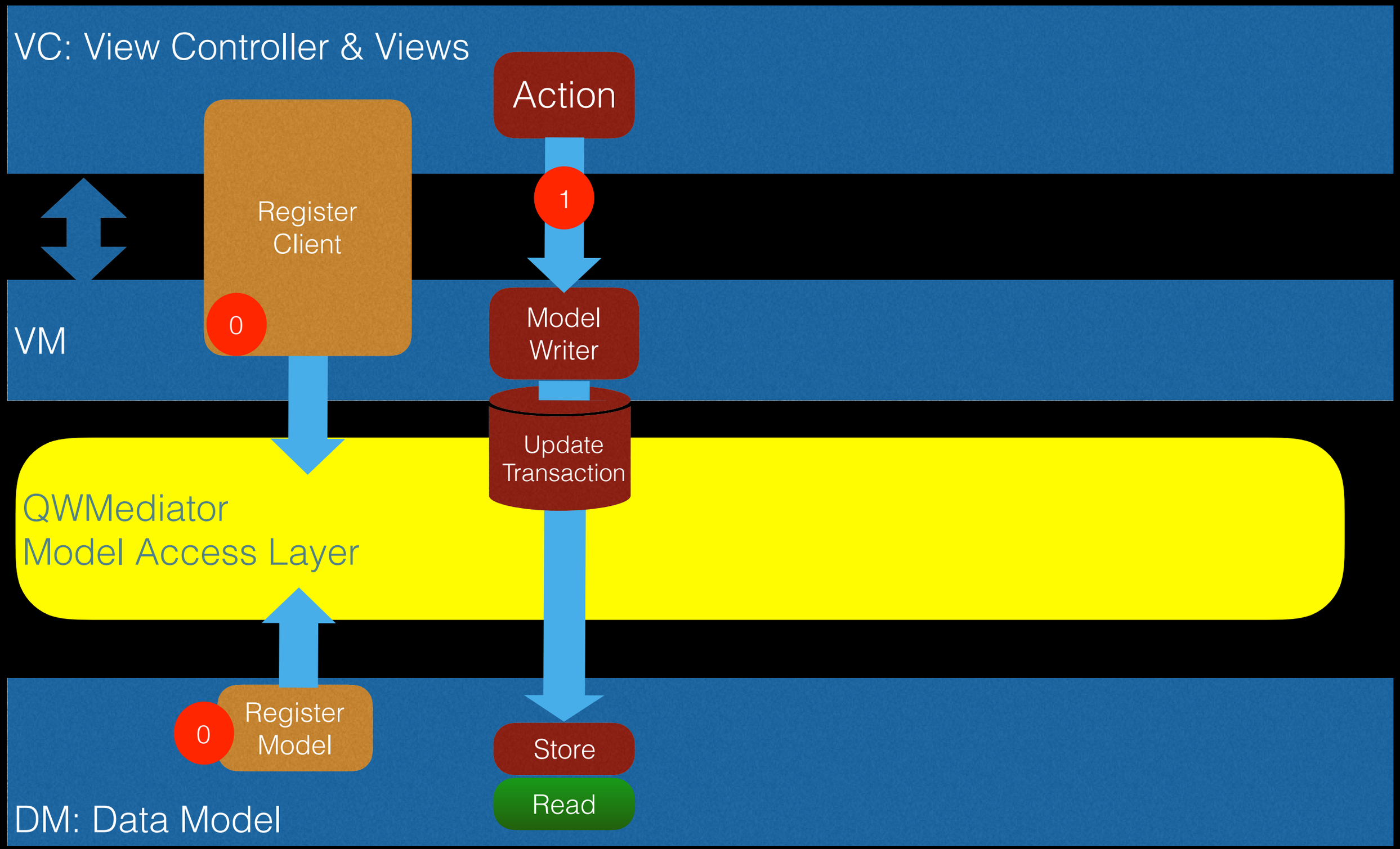


Each method shall fit into his defined role and context .
VC contains decoupled methods with clear context

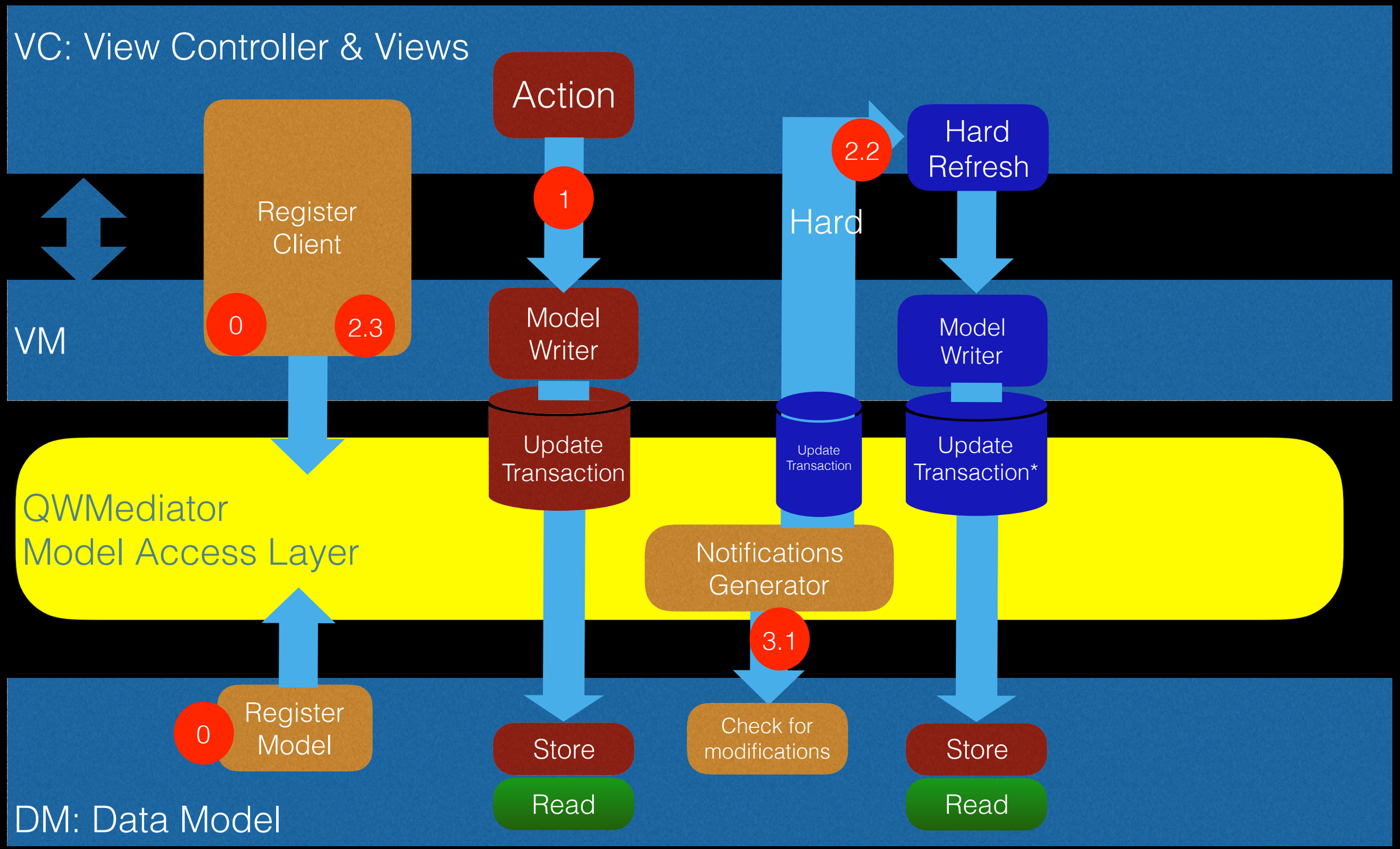
Step 1: Registration



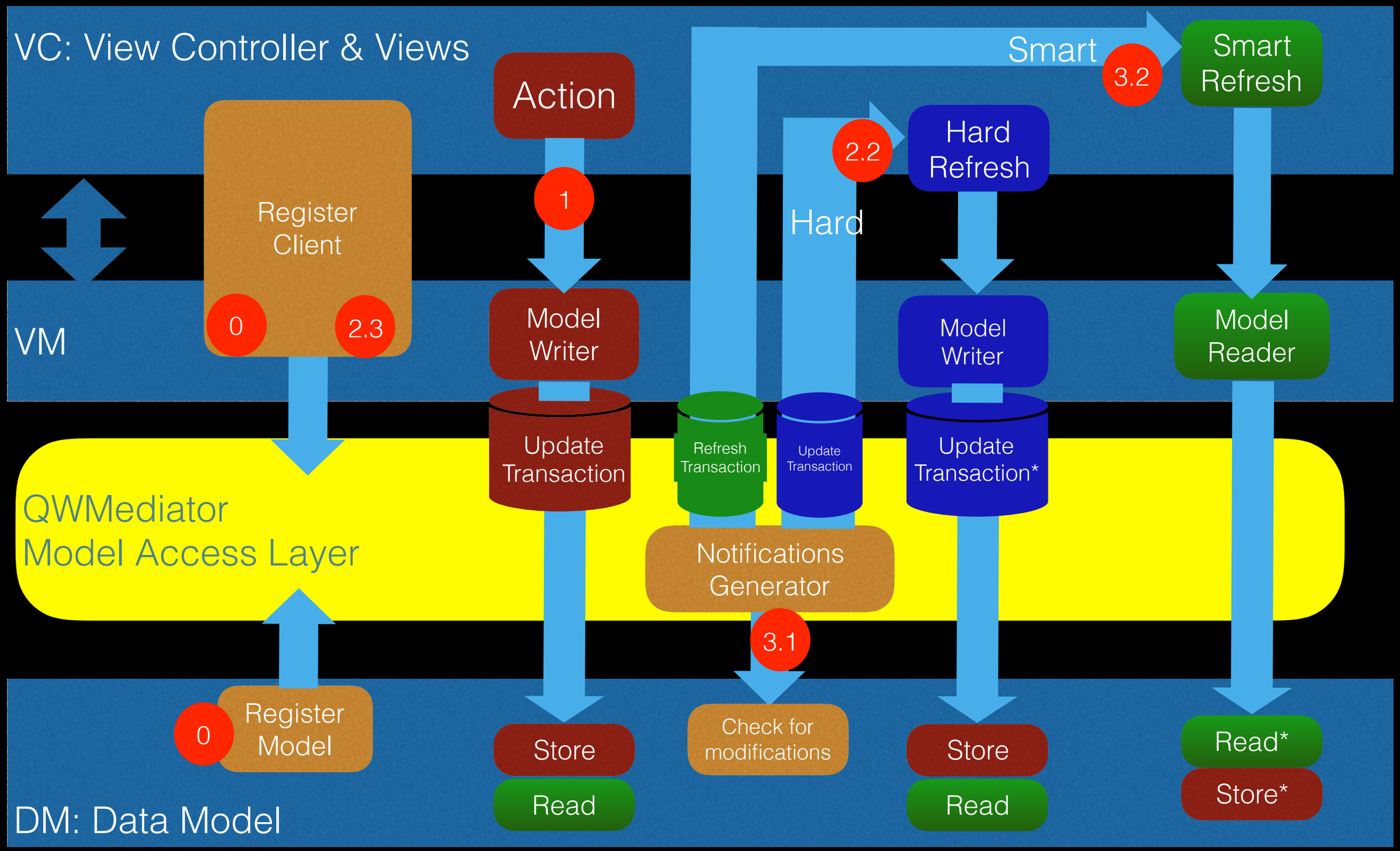
Step 2: Action + Model update



Step 3: Hard Scheduling



Step 4: Smart Scheduling



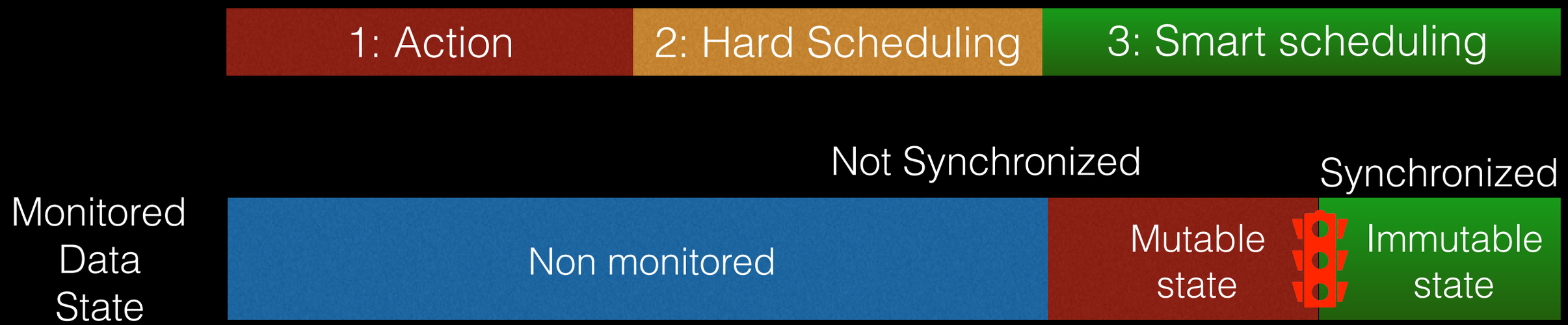
Property Mutation Rule

A data registered by Smart Scheduling can only have 2 states:

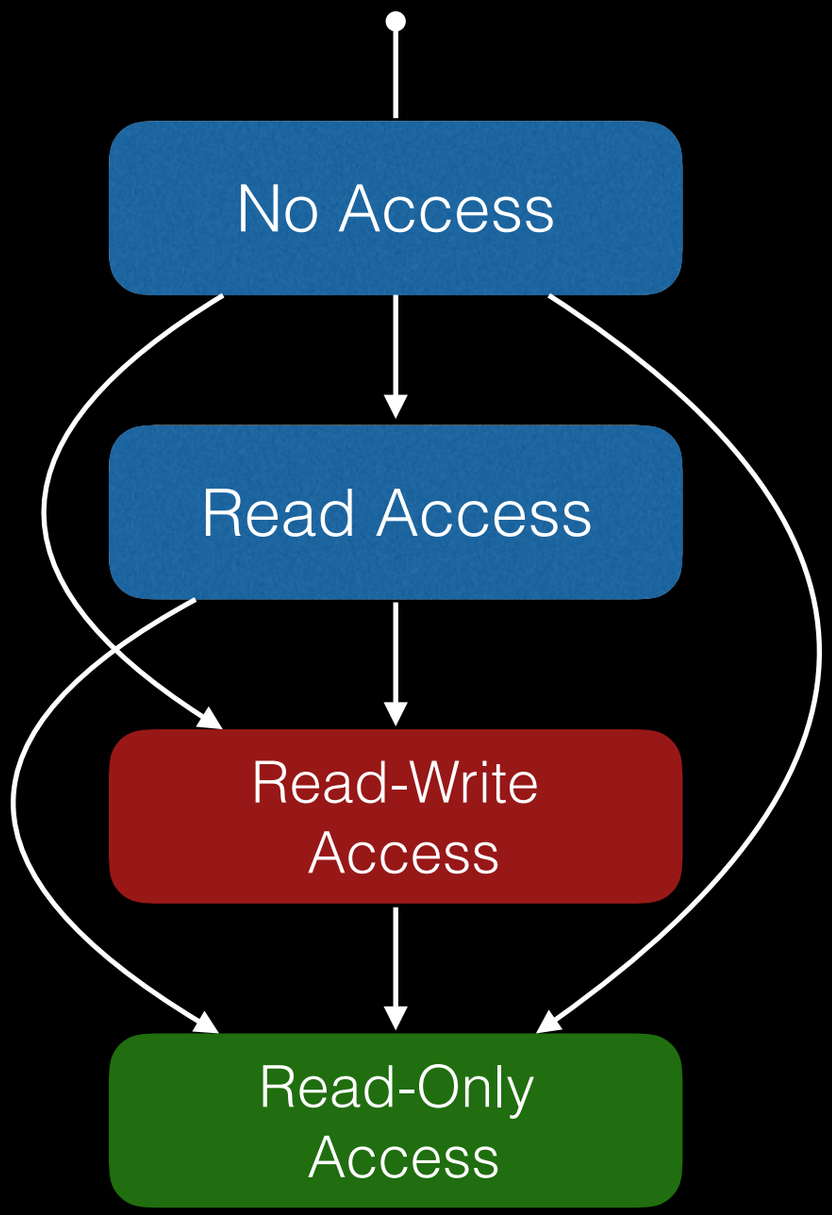
- **Mutable state**: Can belong to a single active Smart Registration Write Set. Can not belong to Smart registration Read Set.

- then **Immutable state**: Can only belong to registration Read set.

This constraint is essential to allow architecture changes without side effect: new components can be added or removed, data flow can be modified, but **the subscribed data will be in immutable state and sent only once per event loop.**



Property States during Smart Scheduling



Initial state at the beginning of smart scheduling

Used as Access Path of a scheduled Write Path
The read value may change
before the end of the event loop.

QWModel.root.nodeProp.prop_Write

Used as Mutable Path of a scheduled Write Path

Used as a scheduled Read Path

QWModel.root.nodeProp.prop_Read

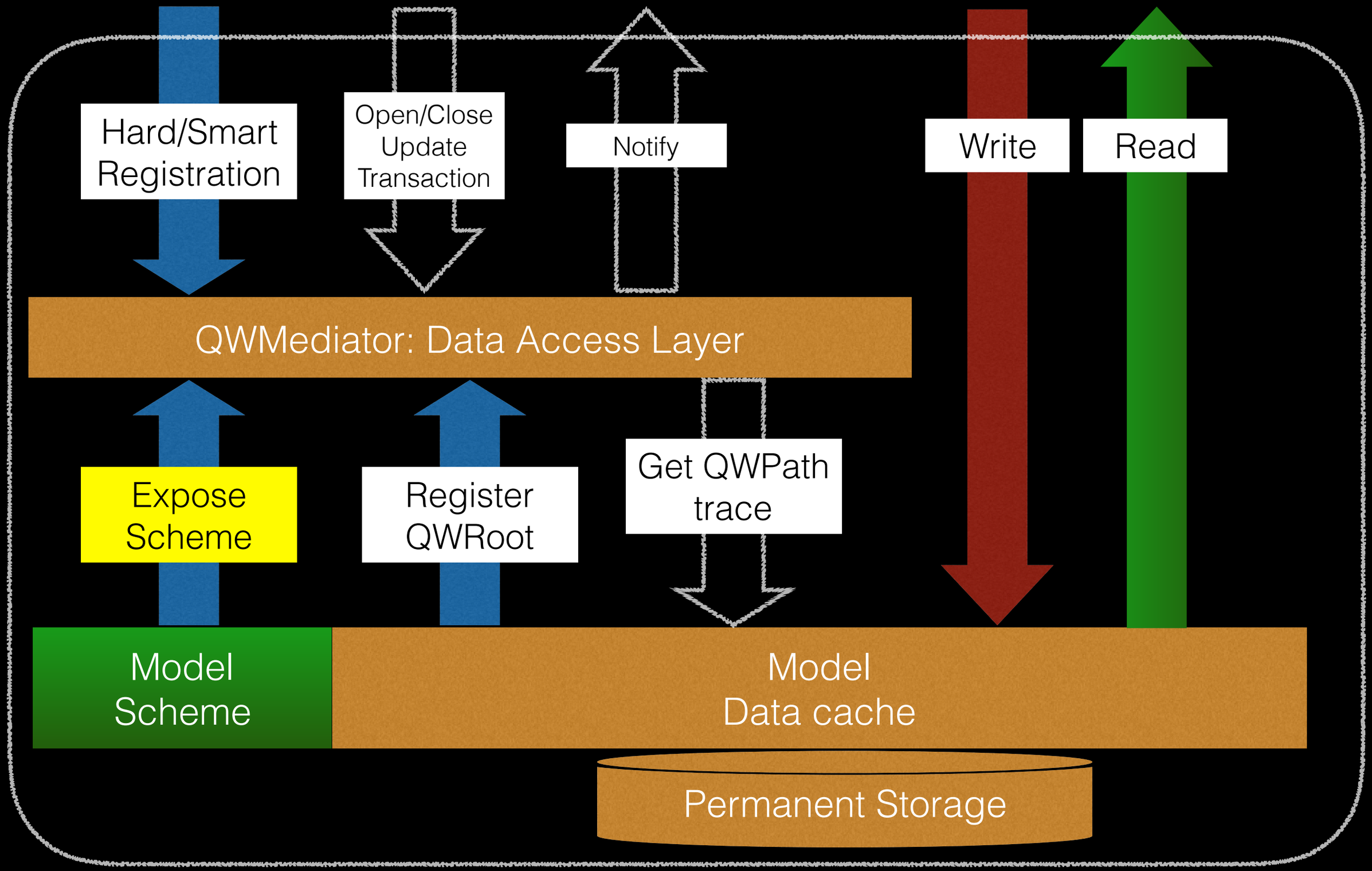
A Global, Holistic, Architecture

Every Data Model access shall be Quantwm-managed

Quantwm simplifies the development, because the respect of local rules guaranties a global behavior.
But the hidden cost is that ALL Data Model access shall be instrumented with an update transaction or a registration to a notification.
It is thus easier to use on a new project than converting an existing one.

Detailled Model Architecture

Model 1: Expose Scheme

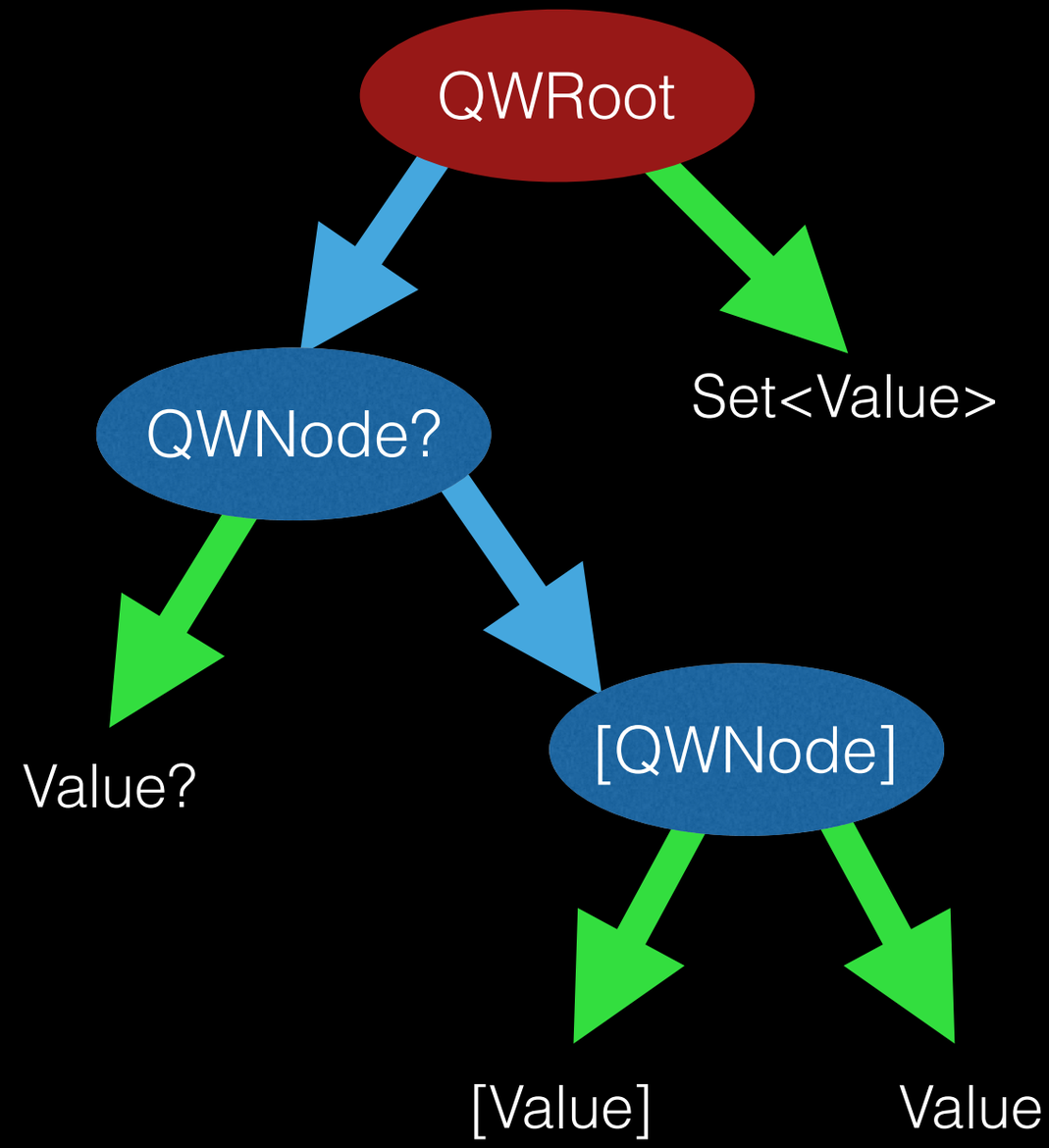


Model Scheme

The Model Scheme is a static representation of the model, describing the data model as a property tree.

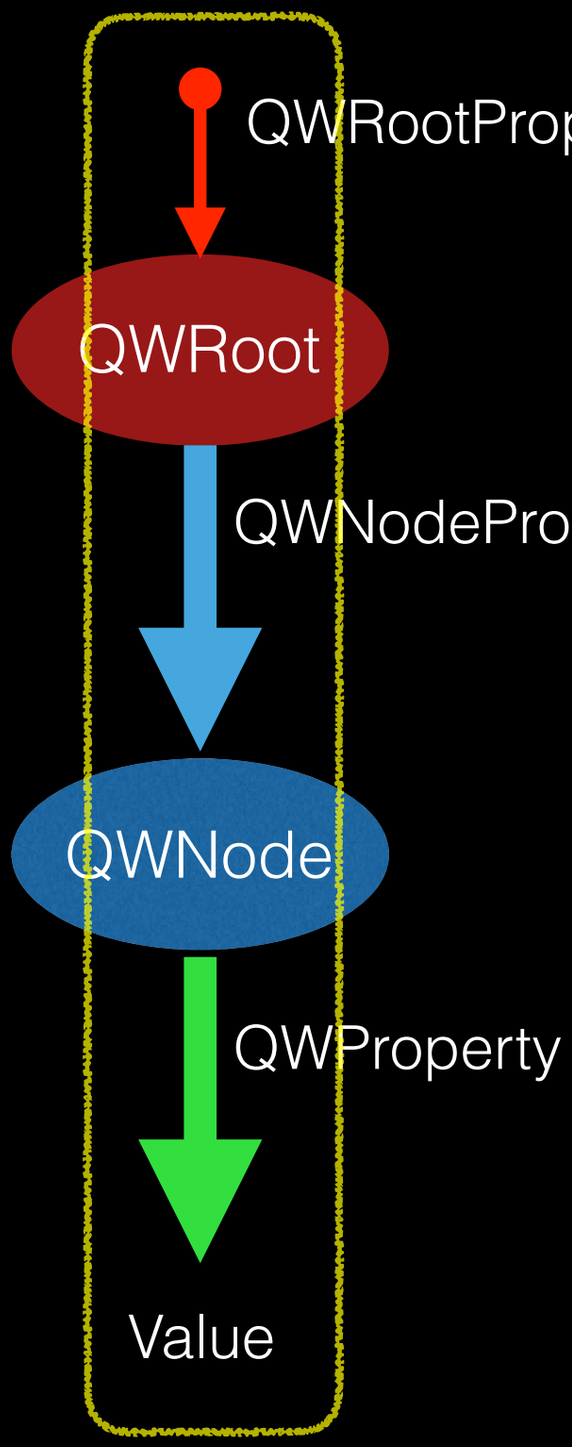
Each node and property is uniquely identified by a path starting from the unique root and composed of Node Property and Value Property.

This unique root is simpler for conceptual reason (application state save / restore / undo).



QWPath

QWPath define a path from the root node.




```
The unique rootId of the Model tree = a QWNodeProperty(_,Root).
```

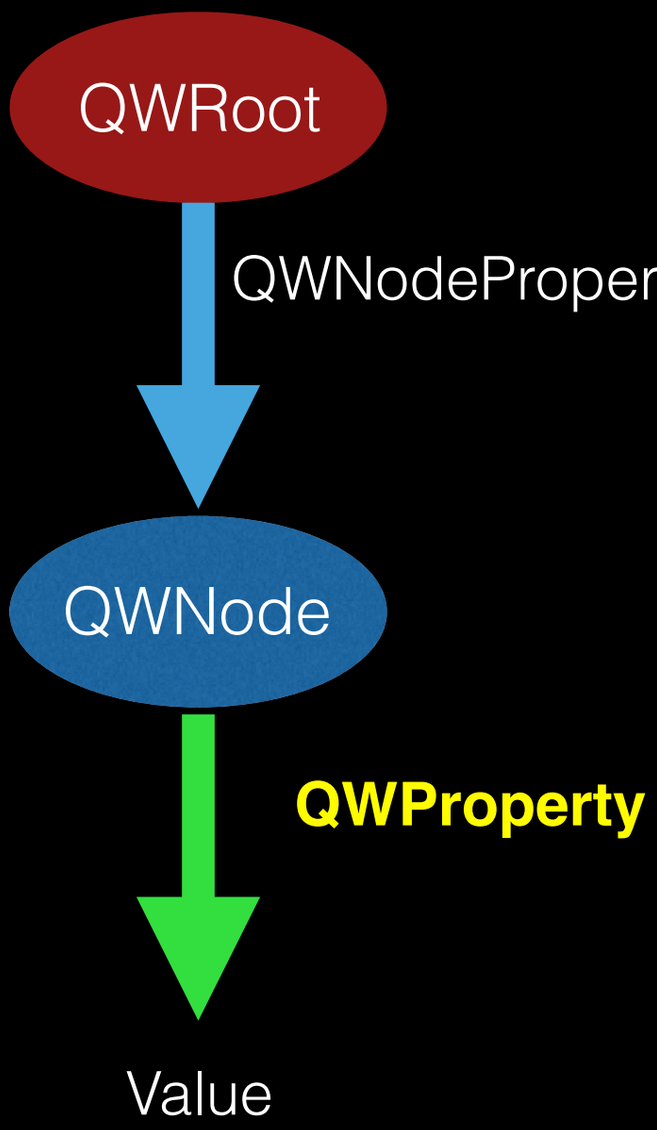
```
The root class, compliant with sourcery-generated QWRoot:QWNode
protocol
```

```
A Swift KeyPath wrapper, with an auto-generated (QWNode)->[QWNode]
closure.
```

```
A Class or Struct, compliant with the sourcery generated QWNode
protocol.
```

```
A Swift KeypPath wrapper.
```

QWProperty



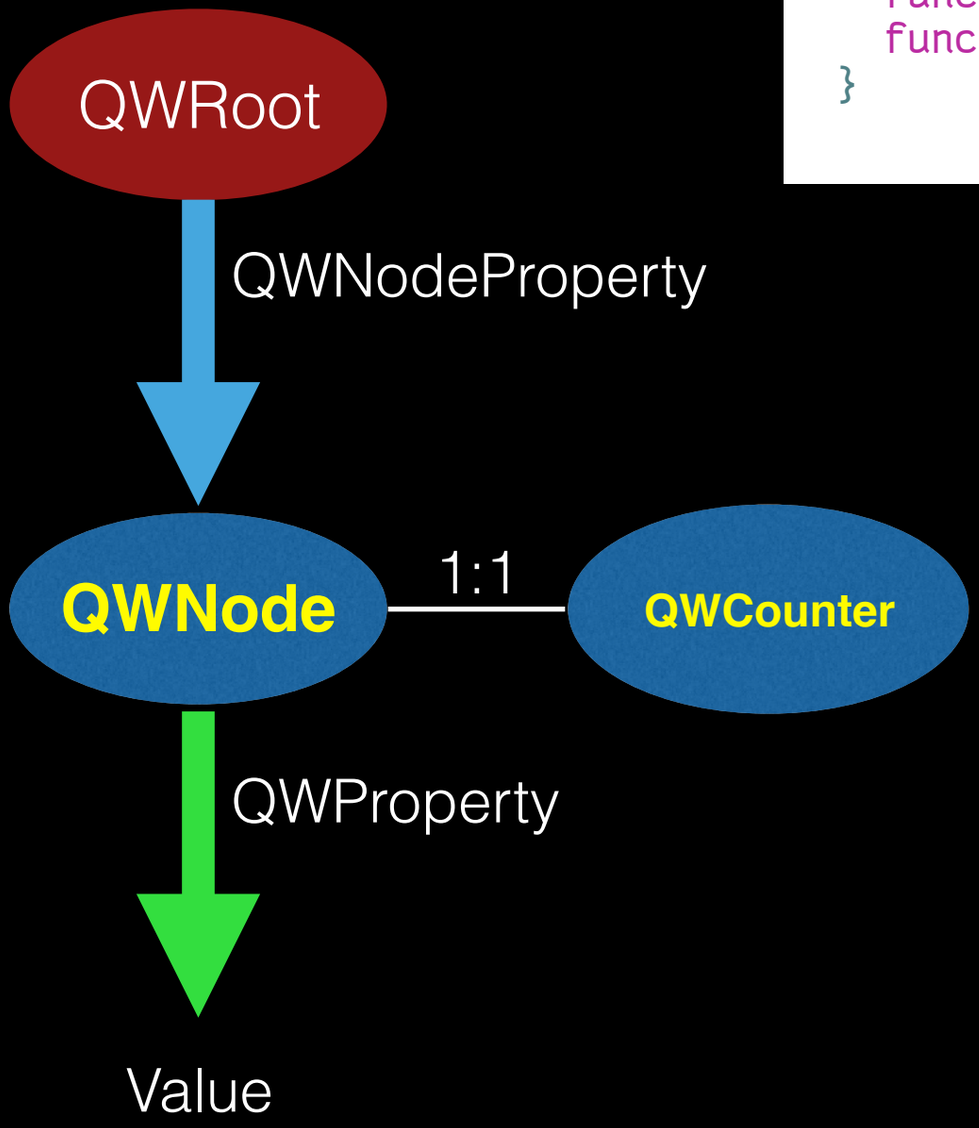

```
// Definition
class QWProperty: Hashable, Encodable {

  init<Node:QWNode, Value>(
    propertyKeypath: KeyPath<Node,Value>,
    description: String)
    ...
}
```

A Property is defined statically as a wrapper of Keypath<QWNode,Value>
The propertyKeypath is the key uniquely identifying the property relatively to its node.

```
// Usage
// Quantwm Property: myProperty
static let myPropertyK = QWProperty(
        propertyKeypath: \Myclass.myProperty,
        description: “myProperty")
```

QWNode

```
// Definition
public protocol QWNode
{
  func getQWCounter() -> QWCounter
  func getQWPropertyArray() -> [QWProperty]
}
```

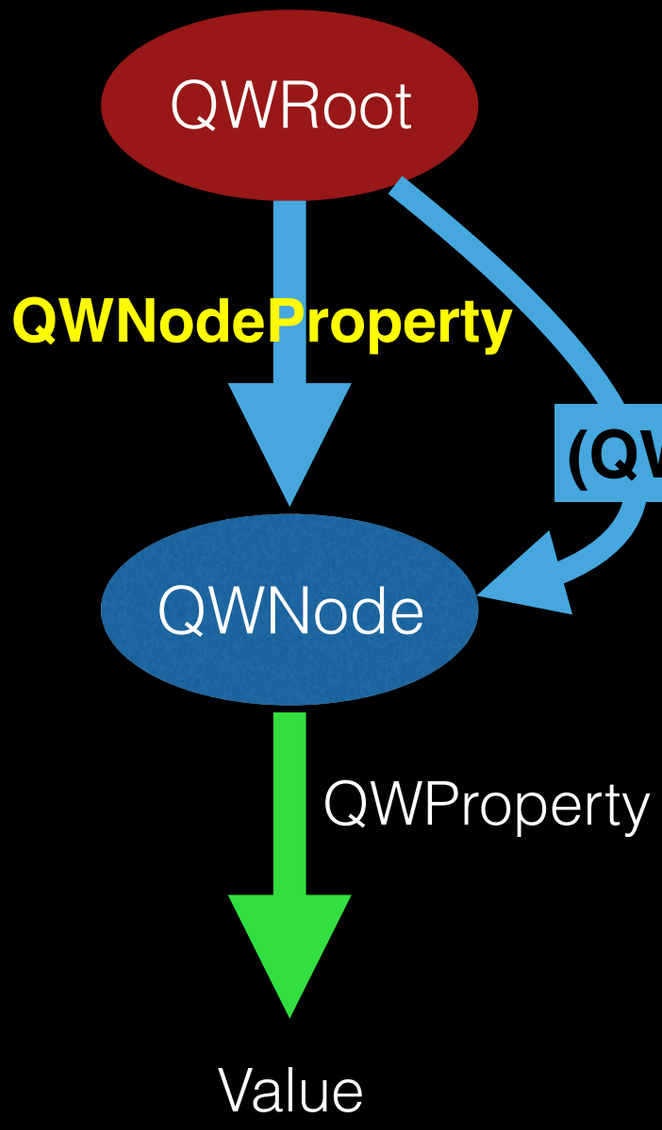


QWNode protocol defines:

- A QWCounter object.
- An property array, used to iterate through all the children of a node.

QWCounter is wrapping Property Counters which are incremented at each monitored property write or read.

QWNodeProperty

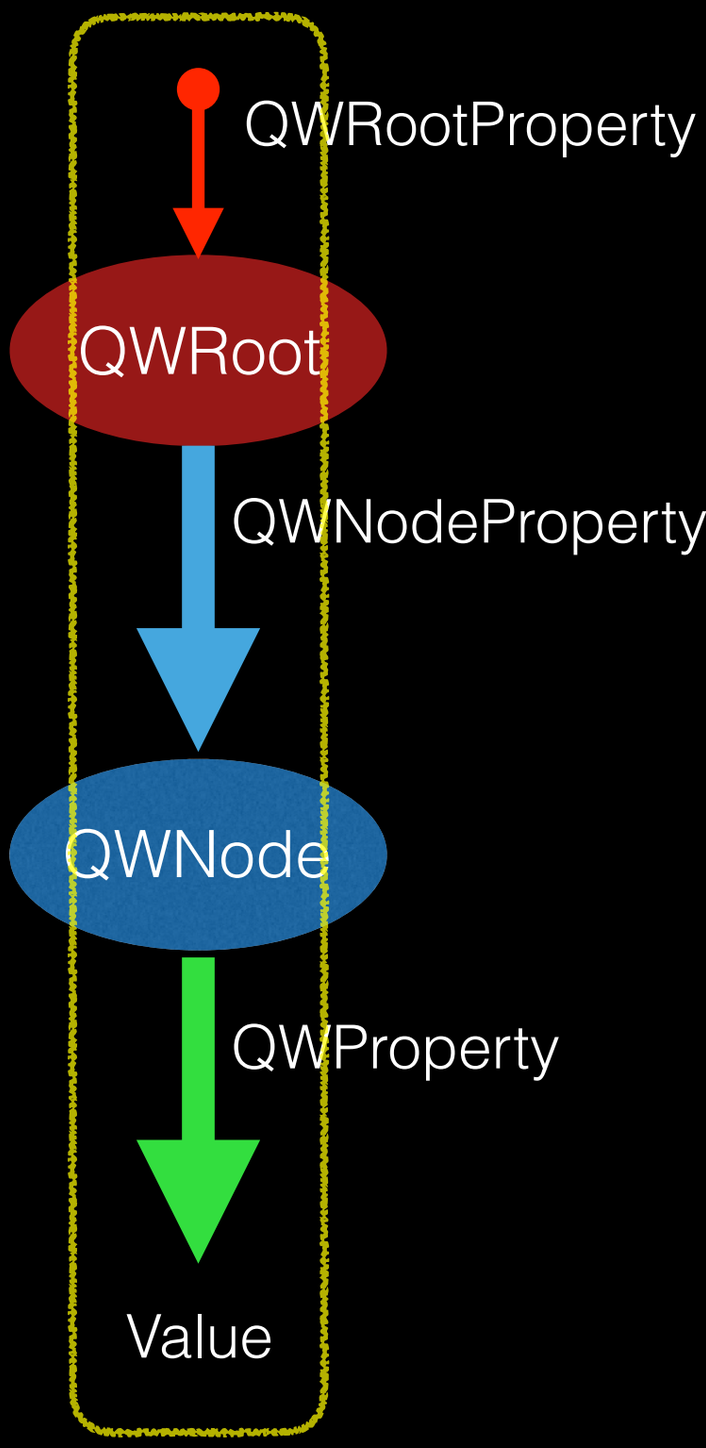



```
// Usage
static let myNodePropertyK = QWNodeProperty(
    keypath: \DataModel.myNodeProperty,
    description: "myNodeProperty")
```

QWNodeProperty is a QWProperty pointing toward a QWNode or a collection of QWNode, associated to a customizable closure:

(QWNode) -> [QWNode]

QWPath



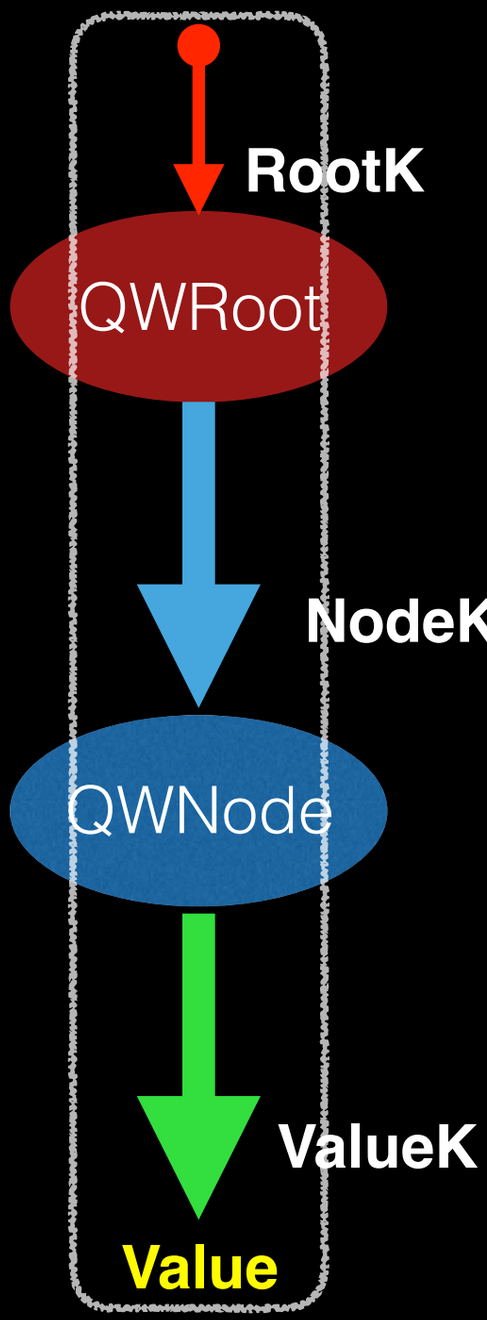

```
// Definition
struct QWPath: Hashable, Equatable, Encodable
{
    let root: QWRootProperty
    let chain: [QWProperty]
    let andAllChilds: Bool = false
    let type: QWPathType
    let access: QWAccess
}
```

QWPath define a path from the root node to a property, or a node, or a node and the set of all its children.

```
// Usage with auto-generated Model Scheme
static let directoryDescriptorChangedPath =
    QWModel.root.dataFileManager.directoryDescriptorChanged_Read
```

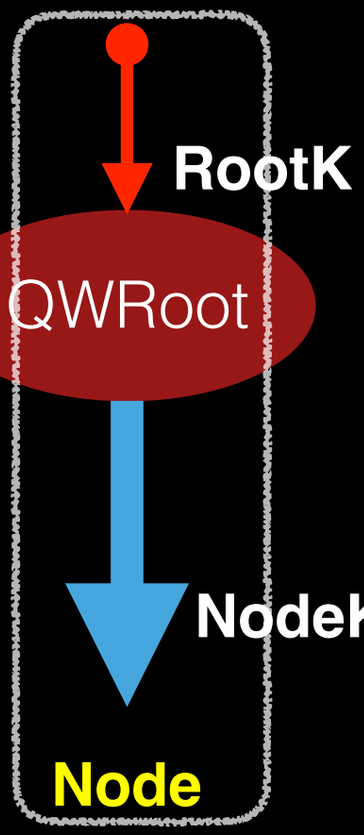
```
// Usage with manual generation
static let directoryDescriptorChangedPath =
    QWPath(root: DataModel.dataModelK,
        chain:[DataModel.dataFileManagerK,
        DataFileManager.directoryDescriptorChangedK],
        andAllChilds: false).readWrite(read: true)
```

3 kinds of QWPath



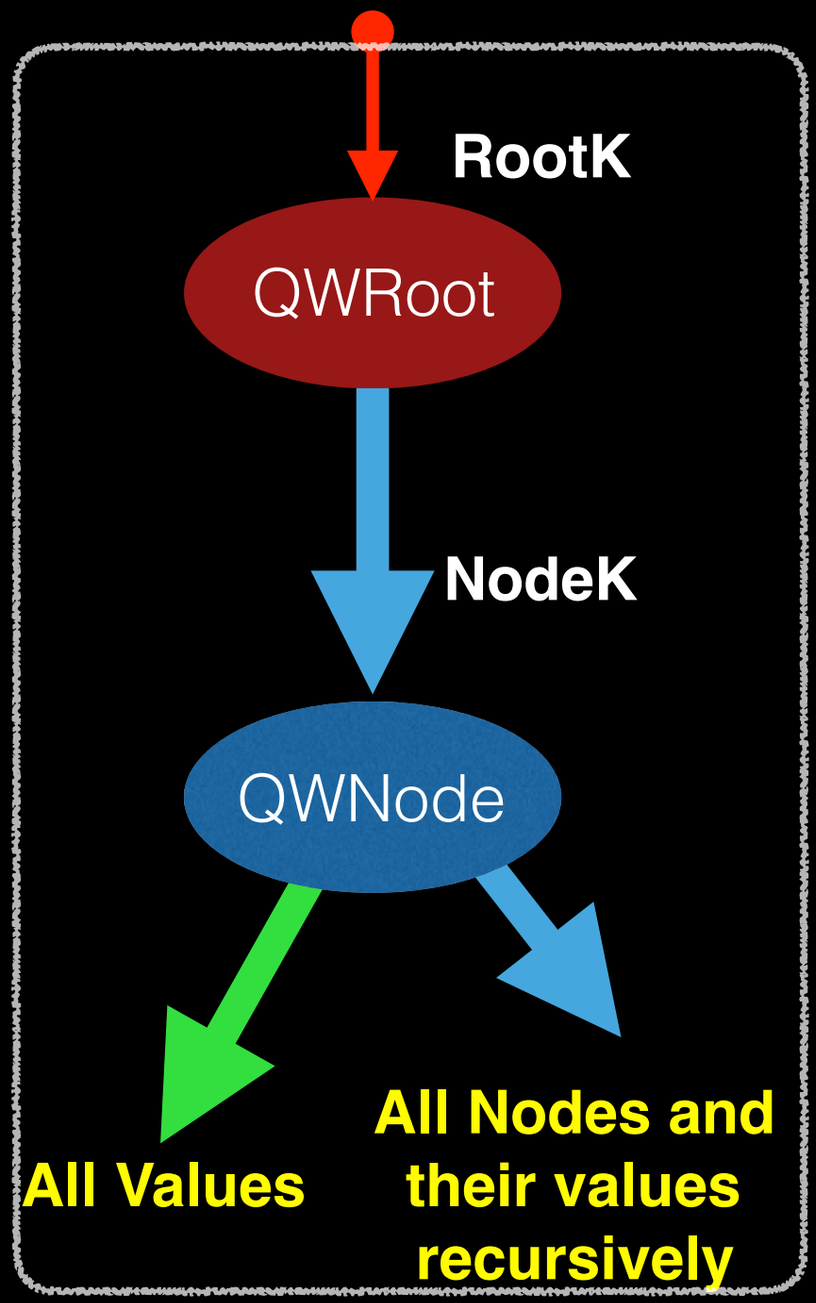
QWModel.root.node.value_Read

Value QWPath



QWModel.root.node_Read

Node QWPath



QWModel.root.node_allRead

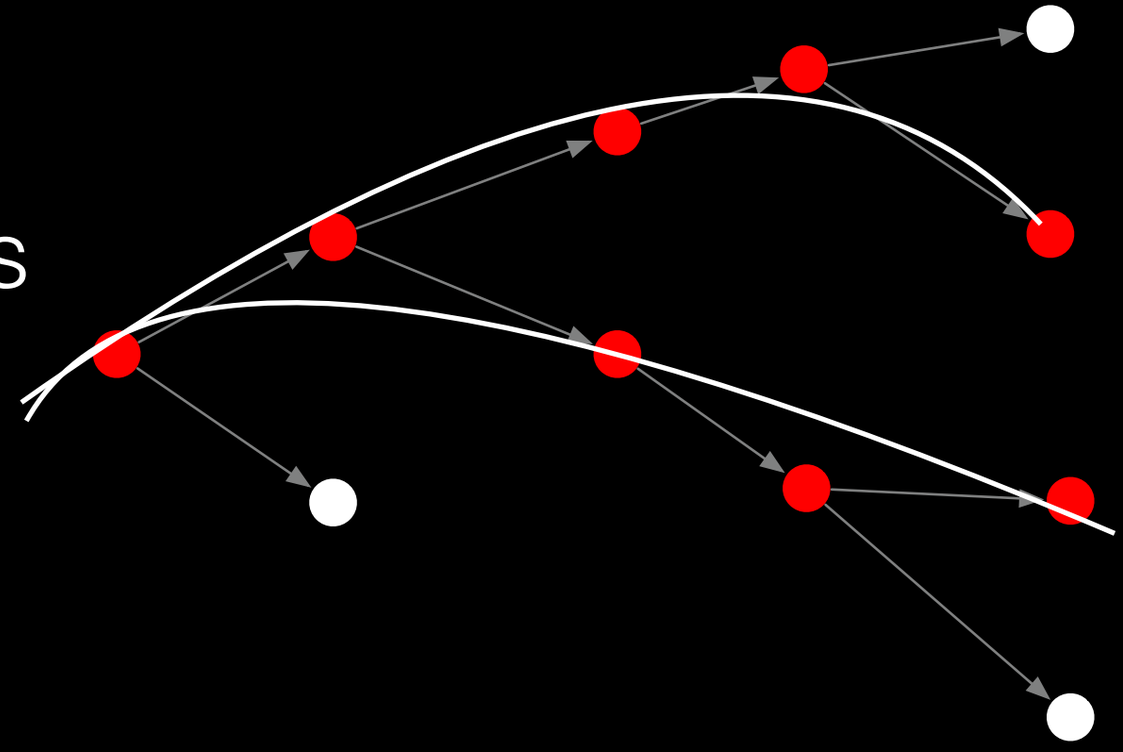
Subtree QWPath

QWMap

QWMap is a set of QWPath
QWMap can be composed with '+' operator.

```
// Definition
struct QWMap
{
  let qwPathSet : Set<QWPath>
}
```

A registration ReadSet is a static QWMap, composed of a set of static QWPath of kind value, node or subtree



Sourcery Property generation

Write this:

```
// Definition

class MyClass: QWNode_S {

    // sourcery: property
    fileprivate var _myNode : Int = 0

    // sourcery:inline:MyClass.QuantwmDeclarationInline
    // sourcery:end
}
```

Sourcery Magic

And Sourcery does the rest. Your Model is Quantwm compliant

```
class MyClass: QWNode_S {

    // sourcery: property
    fileprivate var _myNode : Int = 0

    // sourcery:inline:MyClass.QuantwmDeclarationInline

   // QWNode protocol
    func getQWCounter() -> QWCounter {
      return qwCounter
    }
    let qwCounter = QWCounter(name:"MyClass")
    func getPropertyArray() -> [QWProperty] {
        return MyClassQWModel.getPropertyArray()
    }


    // Quantwm Property: myNode
    static let myNodeK = QWPropProperty(
        propertyKeypath: \MyClass.myNode,
        description: "_myNode")
    var myNode : Int {
      get {
        self.qwCounter.read(MyClass.myNodeK)
        return _myNode
      }
      set {
        self.qwCounter.write(MyClass.myNodeK)
        _myNode = newValue
      }
    }
    // sourcery:end
}
```

Quantwm Benefits

Quantwm Benefits

- Robust. Very robust because completely synchronous.
- Detect framework misuse rapidly via assert()
- Synchronous debugging
- Stateful architecture for free. Easy undo.
- Software architecture is freed from common pitfall. Teamwork is easy. Each function has a clear context.
- Data Model usage can be exactly tracked via QWMap registration (during Smart scheduling only).
- Fast. No extra memory. Data is read from the model.
- No communication between Controller = No problem. Easy refactoring.

Quantwm Benefits

- Complete control of registration and update during the View Hierarchy change with Hard scheduling, including both synchronous and asynchronous View Controller creation.
- Easy View Model stubbing, including registration to notifications.
- Clear contract on reception of notification on a set of observables.
- No extra work for registration on non existing or transient data model properties.
- Architecture is Day One free of technical debt. Written code is free from new features side effects by design.
- Your architecture is your Data Model hierarchy definition + Registration set.
- Many other ....