New Notes on the ChangeApplier

The ChangeApplier API has lingered in its current form for several years, but during the course of FLUID-5024 "model relay" implementation work (and following the implementation of FLUID-4258, declarative change binding), it's become clear that it requires a significant overhaul to be ready for newer framework requirements - especially in terms of the original over-arching mission of FLUID-3674 ("Current idiom for applying changes to models with ChangeApplier is inadequate for large-scale cooperation on extended models") described in 2010.

On this page will be dumped implementation thoughts relating to this new semantic for the ChangeApplier. Strategically, the current plan is to keep the existing semantic in place for the Infusion 1.5 release (upcoming 1H 2014) but to opportunistically add in to the framework a new base class, "fluid.modelRelayComponent" implementing the new semantic/API alongside the existing one (fluid.modelComponent). "At leisure" we could move components over from the old to the new base class as their functionality can be vetted. For the Infusion 2.0 release, we would remove the old fluid.modelComponent and all of its associated ChangeApplier code, and rename the newly introduced fluid.modelRelayComponent to fluid.modelComponent.

Stability of base model reference

The most important changed functionality in the "new ChangeApplier" would be the removal of the idiom expecting the "model" object reference held in that.model in a cooperating component to remain stable. This is a very core assumption that has been baked into Infusion component code for as long as we have coded using models (2008 or so) and so represents a very significant change. The original thinking behind this design choice was mainly centred around efficiency and convenience of access. Also, in the old days, we were somewhat obsessed with "closing over" object references for the purpose of "safety", which is a kind of thinking we are now getting over. Since we expect that the majority of model references in new code are expressed declaratively, the instability of the "model" reference we expect will cause few problems to new implementations.

The reason to allow unstable "model" object references was baked right into the original conception of the FLUID-3674 issue report. In a set of cooperating components, whose "model" references are bound to scattered parts of "the same" model tree, it becomes extremely hard to implement the ChangeApplier semantics in a consistent way. The "old" ChangeApplier code already has a significant amount of special-cased code which treats the "root model object" reference in a very different way to nested objects - for example, a `DELETE` operation targetted at the root does not actually delete it, but instead operates a "fluid.clear" eliminating all its subobjects. Further, an `ADD` operation is converted to a `MERGE` operation, similarly in order to cause the root reference to remain stable. In fact, the `MERGE` operation itself was invented primarily because of this use case, and itself represents an odd anomaly in the old ChangeApplier API - in fact, users are unlikely ever to want the effect of the operation named `ADD` which could be implemented by a `DELETE` followed by a `MERGE` (With the introduction of "declarative change triggering" for FLUID-4258, this unnecessary proliferation of change types is a further annoyance). This inconsistency of this strange "relative" behaviour becomes magnified once we consider the case of a set of cooperating components. Since one component's "base model reference" would be another component's "nested model reference", we would observe inconsistent effects where "the same change" was applied using one component's ChangeApplier or another. This led to the concept of the "cautious ChangeApplier" mentioned in some reports /conversations, which is a requirement (though perhaps not an implementation) that clearly needs to be abandoned. Finally, this "stable root reference" requirement led to a number of anomalies even in simple implementing components. For example, the "InlineEdit" implementation is obliged to wrap its "real model" in a "fake key" named `value` - since `String` is an immutable type in JavaScript, it would be impossible with the old semantic to have a modelComponent whose entire model consisted of just a String, since the base model references was required to be stable. This unnatural "wrapping" is also seen in some of the panels of the UIOptions system.

All of these odd conditions taken together imply that the "new ChangeApplier" should be implemented in with the following properties - i) no expectation to "close over" the model reference in a component, but to re-read it on every access, ii) abolition of the `MERGE` change type, and the `ADD` change type to be implemented with the function of the old `MERGE`.

Scope of change tracking and null changes

Other important changes in ChangeApplier semantic now follow. Since early revisions, the configuration of the ChangeApplier supported a little-known option named "cullUnchanged", whose intention was to completely eliminate the reporting of changeRequests which in fact left the model's content unchanged. Due to the poor implementation of both the ChangeApplier's application model, as well as the "isNullChange" utility method, this option only offered poor support and was only good enough to detect "null changes" which affected just a single primitive value (no "trunk values"). Because of this weak semantic for detection of "equality of old and new models", this option was not widely advertised to the public because of the possibility for unexpected results.

Short digression on circular change propagation

Also, one of the purposes for which "cullUnchanged" was targetted, was thought to be the use of the "source tracking system for changes" (which we still intend to maintain and improve - see comments on FLUID-4679) - this is the issue of "backwash prevention" or "indefinite circular propagation of changes". ChangeEvents and their listeners are often wired up in cyclic graphs, since their aim is to synchronise perception of model material held in common across a number of components, from whatever source the change is propagated. This situation leads to a fundamentally different geometry than is typical in non-model listeners, which usually have a natural sense of "forward progression" in the work sequence of handling a set of notifications. Both "unchanged value detection" as well as "source tracking" can act to mitigate circular propagation, but only the latter is a completely secure strategy - since there may in some cases be "slop" in the propagation of "the same value" across boundaries which lie outside the framework's view. Imagine the case of a media element with a volume control, which honours requests for a change in volume, say, in free floating point units, but in fact quantises them to the nearest integer in the range 1..10 or 1..100 before applying them to its own model. The resulting change notification triggered by the media element to its own listeners will not pass the "unchanged value test" which is seeking to damp the propagation of the "same change" back into the system. This could be seen as a case of "cryptic model relay" - that is, there is a "hidden transformation" operating as part of the model relay process. In general, two elements in a cooperating model system which have "mismatched expectations" as to the set of valid model values may tend to get into an "infinite fight" as they handle each other's change notifications, without a separate system for handling "source tracking" which explicitly prevents circular notifications by tracking the set of elements notified as part of the "current change". Note that in the case of the media element example just mentioned, there may still be problems with circularity since if the notification by the media element is asynchronous, there will be no "natural scheme" for matching the fresh set of changes as being part of the existing set. This "stack-frame based detection" thinking was behind the current implementation strategy of ChangeApplier source tracking which remains the final source of "ThreadLocal" state in the framework, now seen as an implementation risk.

Returning to the issue of unchanged value detection, we can observe there were two cooperating causes which led to the poor implementation quality - firstly, the opportunistic strategy of using jQuery.extend for applying composite changes (which at the time was seen as a "safe pair of hands"), and secondly, the simplistic implementation if isNullChange itself, which otherwise seemed to run the risk of importing a complex "deep equality" check algorithm into the framework. In fact, we had already just axed (as of 2010) our own assertDeepEq implementation from our testing framework in favour of jqUnit's, since our own implementation was discovered to be operating a naive algorithm with awful exponential time complexity costs. Since the "ginger world" implementation phase has ended up littering the framework with numerous recursive processors in which we have rather more confidence than before (through the establishment of the so-called "strategy API") we no longer regard jQuery.extend's hands particularly safer than our own, and it seems a good plan to clear up several of these problems at the same time.

We can gain a more performant, more powerful and more framework-appropriate implementation of "fluid.applyChange" by implementing the application process ourself. As well as taking the opportunity to operate any "strategy" supplied by the user to govern the particular model access, we can gain access to a cheap source of the effects of "isNullChange" by keeping simple records during the process. Finally, these records will be a more performant and more accurate source of the information needed to operate "modelChanged" listeners which expect to react to only genuine changes in model content. Ironically, this implementation also gives us access to the facility which we projected before under the name of the "cautious ChangeApplier", since we can arrange an implementation which ensures that it never replaces a "trunk reference" (plain Object or Array) with a different copy so long as it detects that the incoming change seeks to replace it with a trunk object of the same time. "Perfection is achieved only on the point of collapse" since our non-shared model reference idiom no longer places this requirement at a premium (but it is conceivable that a future version of the ChangeApplier may use this as an implementation optimisation, much in the same way that we imagine that a future version of the IoC system may revert to using prototypal chains similar to those seen in this-ist programming as an optimisation technique which is expected to be invisible to the user).

Rationalisation of change reporting and normalisation of transaction demarcation

The arguments to the original "modelChanged" (and other) ChangeApplier events were in general arcane and unhelpful. The implementation of FLUID-4258 has already required us to rationalise these to some extent (with changes to a few framework utilities), since in a declarative context, the unhelpfulness of the original event arguments was even more manifest. The old signature used to be (newModel, oldModel, \[changes]) where the first two arguments consisted of the entire values of the component model, irrespective of the path which the changeListener had been registered at, and the final argument was an array of change objects responsible for the change. In general, the user might have been expected to rummage around in these in order to see if there had indeed been a change in the part of the model which interested them (see reports such as FLUID-4739, for example, from 2012), and in general this "not properly baked" presentation of the change process was part and parcel of the implementation deficiencies stemming from the old $.extend and "isNullChange" strategy described in the previous section.

Other problems with the old ChangeApplier event system related to the multiplicity of updates. In general we worked hard to ensure that a "single listener" received just a "single notification" at the end of a complete transaction in which it was interested. Unfortunately the botched conception of this interfered with straightforward cases such as a "single listener" being impossible to register at more than a "single path" leading to defect reports such as FLUID-5151 which needs to be fixed with high priority. In fact, a "single listener" should expect to receive multiple updates from a single transaction even when it has been registered against what the old system would consider a "single path", for example, "synths.*". The old "invalidation detection" system (partially discussed in the previous section together with "isNullChange") was extremely insensitive and operated a very simple scheme of matching the "base path" values of each incoming changeRequest against the list of modelChanged listeners, insensitively to the contents of the changes. This system was inherited directly from the "ListBeanInvalidationModel" implemented in RSF in 2006 which was already known to be inadequate at that time. Our new "fluid.applyChange" implementation can also help with this task by outputting a data structure which can more quickly be queried for invalidation results by modelChanged listeners, as well as giving more accurate results due to its ability to see inside the detailed effects of changes on the model. However, the use of this structure is not necessarily the best idea to meet one use case of the ChangeApplier ("thin models") described in the next section.

The case of "thin models" and costs of reporting changes

One use case which was dimly conceived of in the "old ChangeApplier" but only partially handled, related to the handling of "large models". It's possible to imagine use cases where we are frequently handling many changes which are "very small" in comparison to the size of the overall model, which is, we quantify as O(N). We would like to be able to process sequence of changes which are all individually O(1) without the incurring of work proportional to O(N) for each change. This implies, at the very least, that we do not try to maintain full transactional semantics and do not copy the entire model at the start of every transaction. This was the thinking behind the "thin = true" option accepted by the old ChangeApplier, which has also never been seen outside test cases since its inception in 2010.

The "improved applier" described in the previous section acts against this use case by requiring a wildcard path (such as "synths.*") to be matched against every path present in the actual new model in order to detect all cases of updates, in order to service the multiple notifications which we have now decided are necessary (rather than the single notification that the "old ChangeApplier" would provide). **NOTE** - this is actually untrue, the "improved applier" does indeed guarantee to hold only records for genuinely updated individual paths in the case of "overly broad changes targetted at root" - this section is left here as an example of incorrect thinking.

However, the requirement for such "thin models" appears to be an extreme minority case - useful, for example, in large, frequently updating views on large tables - which we will have other framework issues to contend with under new rendering models, before the ChangeApplier becomes the primary bottleneck.

Elimination of old "guard" semantic and events, in favour of universal use of the modelRelay system

In the course of finally trying to use the historical "guards/postGuards" system implemented in 2010 for the use case it had originally been designed, managing the model state of the Fluid Pager component, it was discovered that it was not even adequate for this precise use case. Given that the Pager was not an actively maintained component, this only came to light during the 2013 phase of updating all Infusion components in our image to the post "ginger world" scheme following FLUID-4330, eliminating all manual initialisation code). In general a scheme based on manual inspection of ChangeRequest event streams cannot be expected to be workable, since the source and content of these is hard to regulate/interpret and there are many different event streams leading to the same final results. The particular difficulty of this case stems from the possibility that changes to one piece of model state may invalidate constraints which are used to validate another piece. These "cross-invalidation" rules are extremely hard for the user to configure correctly, and in one case required the user to explicitly defeat the existing (rather poor) "unchanged value semantics" in order to explicitly re-trigger the evaluation of a related constrained even when the related change had not been triggered by the first constraint author.

A system based on the "integral tendency" where it is the model **invariants** which are specified (in fact, in the form of model transformation rules operated by the model relay system), rather than "validation rules" which expect to operate directly on changes (deltas) to the model state and either accept or reject them, is hugely more likely to lead to consistent results, as well as being vastly more easy to configure, since the "cross-invalidation rules" can be automatically detected by the framework based on the "read profiles" of the mappings defining the invariants. That is, it is necessary to specify a model relay relationship, exactly which model values the relay expects to read in order to operate the relationship, from which the necessary invalidation semantics can be inferred - the same is not true of arbitary code written in "guards".

The conception behind "guards" was historically derived from previous IoC systems such as the Spring Framework's "Validators". Given these were based around a "flat" conception of IoC, they could never be expected to have insight into fine-grained read and write patterns operated by framework elements. Only with the implementation of the fine-grained IoC resolution system, together with the similarly fine-grained model relay system for expressing relationships between cooperating model-bearing components, could we expect to gain the necessary insight to schedule possibly conflicting model constraints. However, such power was taken for granted in early end-user development systems such as spreadsheets, which naturally expected to schedule updates (possibly in multiple phases triggered from a single change) across all of the user's values in a consistent way.

Normalisation of transaction demarcation

Having talked around the previous few points, we can return to the issue of transaction demarcation, which was also strongly vexed under the old framework. It was never really clear who was meant to be responsible for starting and stopping transactions, which in any case could only be done by a cumbersome procedural sequence started by applier.initiate() (in the absence of any declarative framework preceding FLUID-4258). With the intention of using modelRelay for all of the cases which were previously handled by "guards", it seems we can get enough insight into the structure of the graph of change listeners in order to make the transaction issue clear. In particular, we can consider that all propagation of changes wired by the graph of modelRelay listeners takes place **INSIDE** a particular transaction - and that notification of listeners bound to these change events wired to listeners outside the modelRelay system occurs sequentially **OUTSIDE** the transaction, after it has concluded. This simple distinction seems to capture all of the thinking that originally motivated the complex rules scheduling the action of guards/postGuards wrt a transaction, which in any case had little to do with the conventional notion of transaction demarcation as such. Therefore only the modelRelay system gets to participate in transactions, which are currently considered to be completely synchronous and completely irrevocable. It's possible that some future use case may bring back the conception of "cancelling" transactions, but in the development of persistence thinking since 2008 it has become clear that a much more helpful modern conception revolves around "failure to **synchronise**" state, rather than "failure to **commit**" state. There is no good reason why one participant in a network of related holders of state-bodies should be compelled by another to revert their intended model for the state (as was required under SQL-like state semantics), especially if they have kept an orderly sequence of records of their local state history indexed by GUIDs.

More detailed implementation notes

The awkward coupling of the "model events system" and the "standard events system" can be improved significantly with this work. We have historically had an implementation warning on the old "fireToListeners" method in Fluid.js, mentioning that it is possibly poorly conceived. The old ChangeApplier implementation followed a bizarre and confusing workflow where a model event which was required to be fired was first "pre-fired" - that is, it was dispatched to Fluid.js firing, but only fired a "wrapper" which produced a "fireSpec" structure consisting of records only of those listeners which had not otherwise been defeated by the base event system. This, for example, handled the case of namespacing of events, which might lead to "loss of listeners", as well as handling the case of "listener priority", both of which were features of the base event system considered valuable also to model events. After this "fireSpec" had been accumulated, it could be further picked over within DataBinding.js to determine which listeners had paths which matched those of outstanding changeRequests, thus leading to their (under the old model, just one single) actual firing to the user listener.

Given the subtleties involving "the same listener" being attached to multiple paths, those "core event features" are appearing increasingly useless. In general, we don't expect listener priority is a reliable way to juggle with notification orders across multiple constituencies - as well as, it no longer being clear how to resolve the issue of namespacing of listeners against "the same listener" attached to multiple paths - if two different listeners are attached with the same namespace to two different paths, should one of them be eliminated? If the same listener is attached to two different paths, with the same namespace, should one if its instances eliminate the other? What are our criteria for resolving against "the same path" where one path may have a wildcard element which ends up comprising elements of the other? All of this suggests that we should abandon subleties wrt. change listeners and abandon the user of the "fireSpec". This would simplify the implementation enormously, as well as allowing the elimination of the anomalous "fireToListeners" method in Fluid.js, which allowed firing to a set of user-nominated listeners rather than necessarily the ones actually registered for a particular event.

We can also take the opportunity to bring the implementation of core events in line with our general thinking relating to "publicity" - by making the core records (such as the "byId") table public, we can assist in implementation sharing with DataBinding.js, as well as increasing the transparency of event firers that are being inspected from the debugger before entering their closure stack frame. This implementation dates from the darkest days when we still believed in the so-called "data-hiding" as a valid implementation technique.

The elimination of the confusing "guards" and "postGuards" event further streamlines not only user binding code but also the core implementation. In particular the cumbersome and confusing "fireAgglomerated" method, which in retrospect could be seen as expensively ensuring exactly the wrong notification semantics (at most one notification per listener per transaction) can be eliminated.

Similarities were seen between the new "cautious applier" and the existing "merge strategy" held in Fluid.js, but these seemed to be insufficiently strong to warrant implementation sharing in this code which is in any case the most performance-critical part of the framework. In particular, the changeApplier doesn't operate a mergePolicy, and there seems to be no real reason it should, although it clearly should be improved to fully support a user's "strategy", for example implementing a schema-aware piece of model state - even though most such implementations should just be concerned with populating "default values" in freshly-allocated model state which should already have been seen in the trunk path on the way to applying a change. However, given that we may be "in the air" **BOTH** wrt the target model **AND** the invalidation model on our way to detecting that a change actually invalidates part of the model seems to suggest a fundamental difference between these algorithms. In addition we noticed an odd check against fluid.VALUE in the merge implementation which we need to rediscover the purpose for.