**Distributed/Multi-Threaded Error Monitoring**

The idea for a generic inter-process error-handling framework came from working on HBASE-6055 (snapshots). Distributed snapshots require tight time constraints in taking a snapshot to minimize offline time in face of errors. However, we often need to coordinate errors between processes and the current Abortable framework is not sufficiently flexible to handle the multitude of situations that can occur when coordinating between all region servers, the master and zookeeper.

There are a couple of overall design goals that drove the structure of this framework.

1. Real time error notification
   1. This was a primary requirement to ensure that we don’t spend extra time doing work when we need to return regions to an available state during snapshots.
   2. Currently this is implemented as single threaded notification between all error listeners, but can easily be parallelized and asynchronous (see ErrorOrchestrator.java) without hindering the real-time properties.
2. Error propagation between all tasks
   1. If a subtask throws an error, then the other subtasks should quickly be informed of that error (when they are ready to deal with it).
3. Enable task specific error handling with minimal overhead
   1. You don’t want to have to rely on a generic ‘receiveError’ method when you really care about the information that is propagated. It is really too easy to pass in the wrong order of arguments because you forgot the right order. This is really about using the static typing that java provides as much as possible.
4. Providing a single method to check for an error with the option to bail with an exception if an error is found
   1. This was actually pretty easy to implement, but still was a necessary feature since if you pass in error, you better be able to find it.
5. Optional operation timing
   1. An operation can be the overall, distributed task or subcomponents of each task. Either way, you frequently require time-based SLAs on sub-task which, if any of them exceed that time bound, should cause the entire process to fail.
6. Easy error injection
   1. This just makes it easy to test different failure scenarios (its proven imminently useful for testing snapshotting).

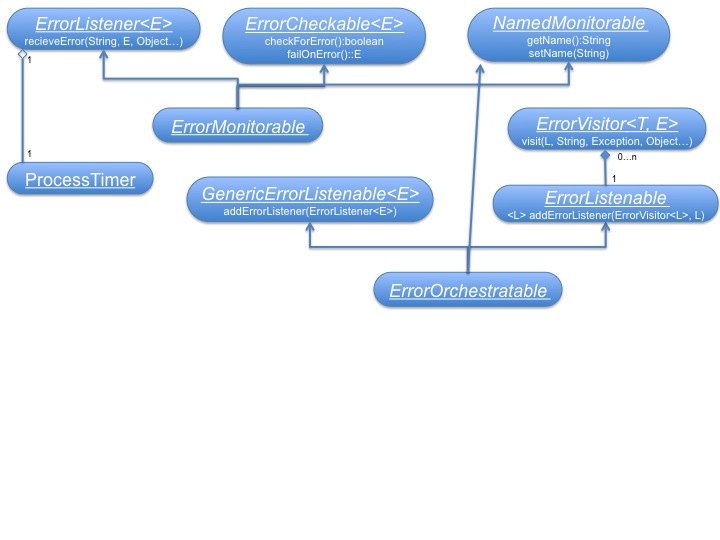
*NOTE: The following diagrams use the following key:*

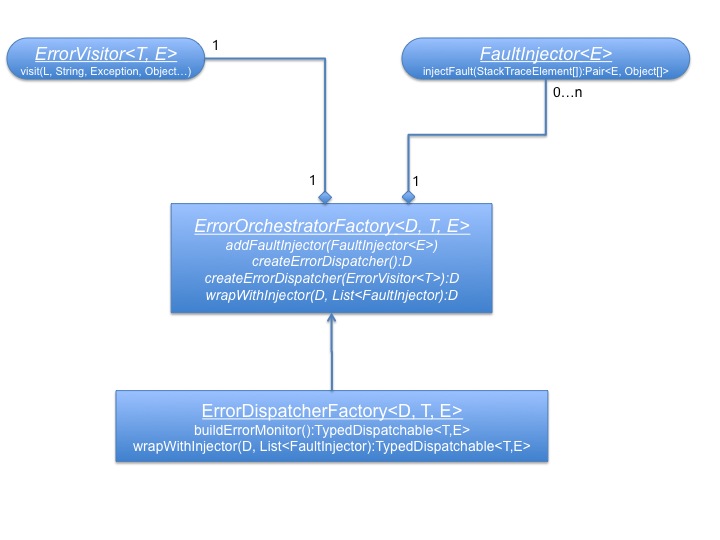
*Rounded corners, italics = interface*

*Square corners, italics = abstract class*

*Square corners = concrete class*

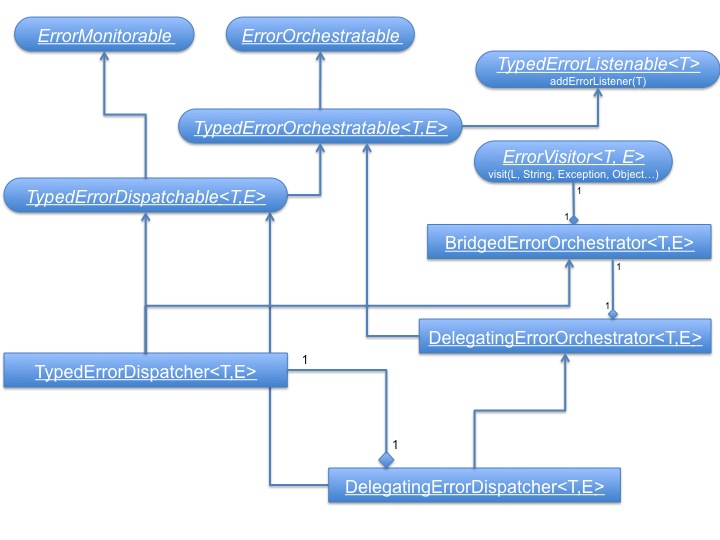
Below is diagram of the generic error handling interfaces. These are enough to implement most of the above requirements.

****

The one thing missing from the above diagram is how we inject errors. This is a much simpler diagram and also encompasses the helper factories that can be used to implement some of the above interfaces.

The above two class hierarchies are enough to enable all the goals that we originally set for a generic error-handling framework.

However, to make it easier to deal with arbitrary types of error listeners (rather than ErrorListeners and generic error propagation), we also need to add a couple of utility layers. These layers let us more easily bridge between a typed error listener (which gets specifically typed error notifications) and the generic listener/dispatch framework. Further, we also add delegating implementations of the major interfaces to make it easy for developers to add their own extensions, implementing their known interfaces. Below is an extension of the first diagram to help illuminate this ‘usability’ layer.

****

The biggest known issue is the large number of marker interfaces. However, to make it easier to developers to reuse the framework – particularly for making their own factories – using marker interfaces was found to be the cleanest solution. See the error handling for in HBASE-6055 for an example implementation.

Hopefully this document will help illuminate the corresponding code and why we even need to have a generic, inter-process error-handling framework in the first place.