A Tiny Workflow in Spring

Andrew Cooke[*]

August 2006

Abstract

I define a simple workflow within Spring, give some examples, and discuss its use. While is works well for prototyping and structuring simple flow–based processes, it doesn't scale well if the control becomes complex.

Contents

1 Motivation

A workflow is a software tool that separates 'control', 'state' and 'process'. A workflow engine is often a 'composition language' used to express the high level functionality of a system.

There are many complex, powerful workflow engines available, but their associated cost makes them attractive only at the higher levels of a system. Since I have just defined workflows in terms of 'high level functionality' this might seem reasonable.

However, 'high level' is a relative judgement (and the definition of 'business logic' is more flexible than many people pretend). I felt a lightweight workflow would be a useful tool that could help clarify the structure of many systems[1].

2 Design

Figure 1 is a class diagram for the Tiny Workflow library.

The design exploits the 'nesting facade' pattern[1]: a workflow contains a `Flow`; many implementation of that interface can contain further embedded `Flow` instances.

The most abstract view of a workflow presents two separate ideas: decisions and processes. These suggest `Test` and `Action` interfaces. To reconcile this with the pattern, I make `Flow` inherit from both and give appropriate semantics: the `Action` in a `Flow` is called if the `Test` returns true[2].

A useful practical class is `Pair` which binds a `Test` with an `Action`. I can then develop separate decisions and process, combining them with `Pair` as needed. Alternatively, instead of aggregation, I can use inheritance: `Null` is a `Flow` that can be subclassed as required (by default the test is `Always` true and the `Action` does nothing).

[*]andrew@acooke.org

[1]Ad-hoc 'workflow–like' classes emerge naturally in Spring–based systems. All I do here is provide a little more structure.

[2]Assuming that the workflow has 'reached' the flow in question.

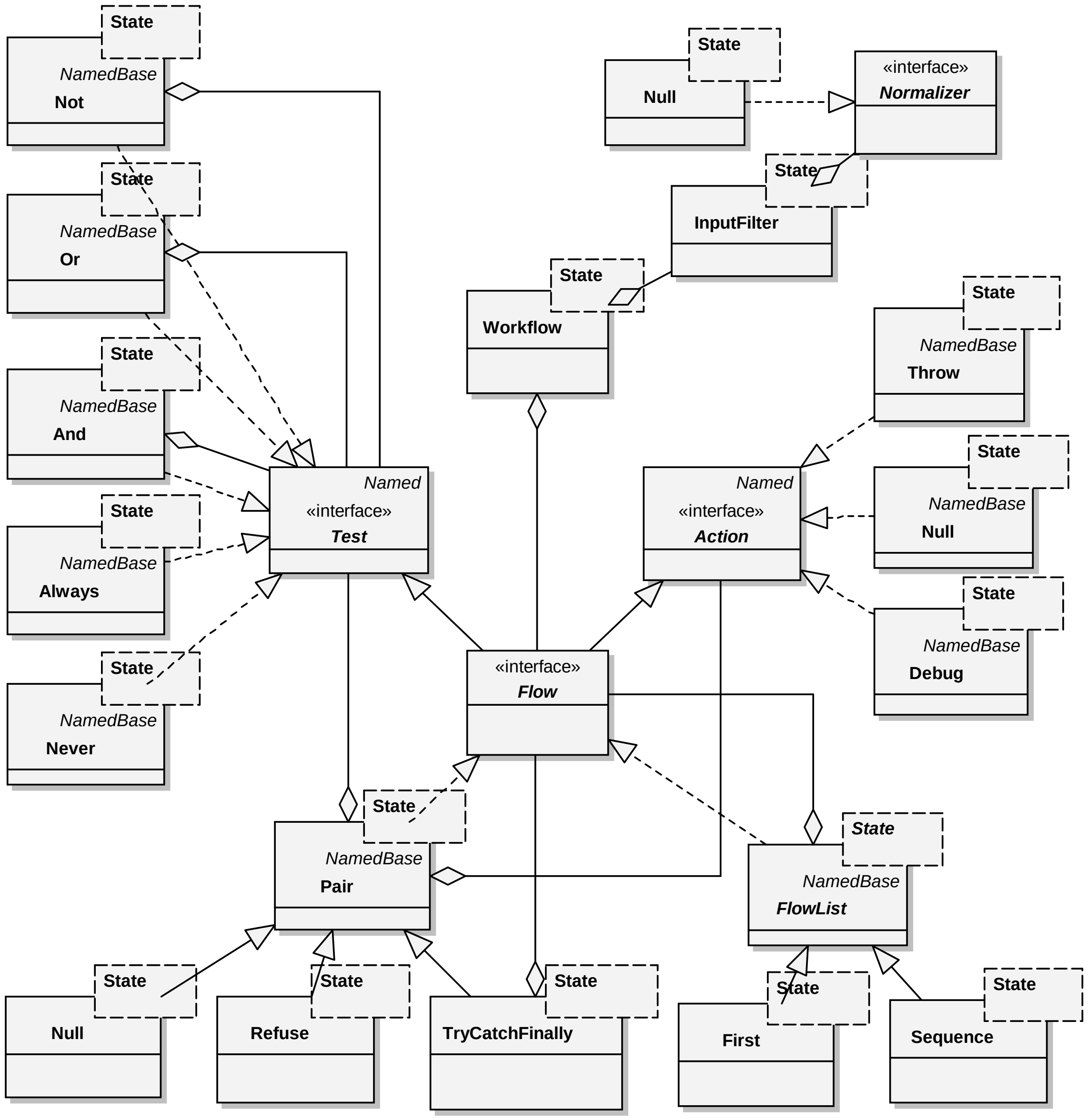

Figure 1: The basic structure of the Tiny Workflow.

Implicit in the discussion so far is the idea of `State`. This single interface is used both as the basis for decisions, and as the data to be processed. This is not an interface, but a generic parameter to almost all classes. In this way, the workflow is type–safe, but not intrusive. `Test` interface's main method to take a `State` and returns a `boolean`; `Action` takes and returns `State`.

I can also define `Test` instances that are injected with other tests (nesting facades again). The library contains the obvious `Or` and `And`, as well as `Always` and `Never`.

The library also contains a set of composite `Flow` instances. `Sequence` takes a list of `Flow` instances and calls them in turn. `First` also takes a list, but stops running through the contents as soon as a `Flow` whose `Test` returns true is found (and the corresponding `Action` invoked).

Finally, the workflow includes some support for converting incoming data to the internal `State` representation: `Normalizer` and `InputFilter`.

3 Examples

3.1 Pseudo–Code

Figures 2 and 3 show the same ideas expresed in a Java–like syntax and as Tiny workflow components (for clarity some components have been omitted, typically where default values are used).

The advantages of structuring code like this include:

- The ability to restructure the logic without recompiling the software.
- Explicit state allows serialisation (see dicussion in [1]).
- Clear separation of decisions and process (often confused in OO code).
- Use of a standard, simple interface allows composition of generic components.

```
if (testA(state)) {
  try {
    if (! testB(state)) {
      state = processB(state)
    } else if (testC(state)) {
      state = processC1(state)
      state = processC2(state)
      state = processC3(state)
    } else if (testD(state)) {
      state = processD(state)
    } else {
      throw
    }
  } catch {
    state = processE(state)
  }
}
```

Figure 2: A fragment of pseudo–code; see figure 3 for the equivalent workflow structure.

3.2 DB Loader

Figure 4 shows part of a fairly complex flow (further loaders have been omitted for clarity). The `First` element will select the first sub–flow whose test is successful. By ordering the loaders from most to least specific the most complex that is consistent with the data will be chosen. If none are chose an exception is thrown; this, or any other exception thrown by a failing loader, is then caught and a generic loader called.

Figure 5 is part of the test for `mosaicSkyFlatImageDataProductLoader`. Three different tests are combined with `Or`.

(This system used the XML–based representation of state discussed in section 4.1, hence the `tiny.xml` namespace.)

4 Experience

In general, my experience with the Tiny Workflow has been positive. It was simple to develop, is easy to understand, and has proved to be very flexible. However, a number of issues restrict its use. It is not a replacement for a 'real' workflow, but is a useful

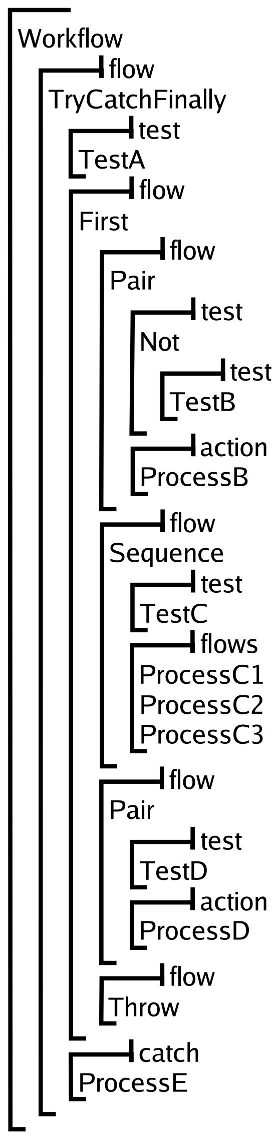

Figure 3: A workflow structure that corresponds to the pseudo–code in figure 2.

additional tool.

4.1 XML Tests

An early iteration forced the state to conform to a specific interface, which provided an XML 'view' of the contents. This had the advantage that tests could be written using XPath, and an XPath–based `Test` could be used in all workflows. Often, no additional, state–specific tests were needed.

However, this approach had some drawbacks. Implementation details were exposed in the workflow configuration (this may have been due to poor design of my XML). Maintaining consistency between the state and the XML was expensive: either the state was XML, in which case actions were inefficient, or the state was Java objects, in which case test were inefficient. Also, XPath is often not very compact or readable; I found myself writing state–specific tests to provide a more friendly interface to the user.

I eventually refactored the code so that the state was generic; the XML–aware interfaces were then (trivially) re–implemented as more specific versions of the generic base.

4.2 Verbosity

XML is not very compact. Spring configuration is not very compact, even for XML. Writing complex control flow logic in Spring XML soon becomes rather messy. To some extent this can be managed by standard programming practices (in particular, separating the workflow into a hierarchy of processes and placing each level in a different file), but even that soon becomes inadequate.

Tiny was used successfully in a variety of services, but when it was most useful — when the flexibility was really necessary — I switched to a different configuration mechanism once I understood the issues involved[2]. This was not really a failure, since the library had proved to be very useful, but did show that the logic scales poorly.

4.3 Lack of Features

This workflow provides a way to structure related processes, but it does not have any other workflow feature. In particular, it does not provide a mechanism for persisting state. Sub–flows cannot be stopped/started in response to reliable asynchronous messaging, or to provide continuity across system failures.

4.4 Persistent State

If this workflow were able to persist state it would simplify the interface with asynchronous messaging[1].

The next release of Java is rumoured to contain support for continuations, but the Tiny is so simple that this would be possible to add by hand; `State` and the call stack would need to be serialisable[3]. Extending `State` to track the call stack would be one approach.

In addition, some mechanism would be needed to interface this functionality with the messaging user, but again, at first glance, this does not seem to be a hard problem.

4.5 Pluggability

Where Tiny really shines is in assembling simple, composite services. Combined with the approach described in [1], the `Flow` interface makes building modular, composite services, whose sub–services can be configured independently, trivial. Sometimes it is not even necessary/appropriate to expose the workflow externally; the internal structure works well and a wrapper Java class can simplify configuration.

References

[1] A. Cooke, A. Egaña 2006; Spring, Mule, Maven: Lightweight SOA with Java
http://www.acooke.org/andrew/papers

[2] A. Cooke, A. Egaña, S. Lowry 2006; FITS Files and Regular Grammars: A DMaSS Design Case Study:
http://www.acooke.org/andrew/papers

[3]The simple structure Tiny imposes on the code is what makes this so simple; there is no need to store any further information from the Java stack and heap.

```
<bean name="loaderFlow"
  class="edu.noao.nsa.util.workflow.tiny.xml.flow.TryCatchFinally">
  <property name="action">
    <bean class="edu.noao.nsa.util.workflow.tiny.xml.flow.First">
      <property name="name" value="loadFlow.action"/>
      <property name="flows"><list>
        <ref bean="mosaicDomeFlatImageDataProductLoader"/>
        <ref bean="mosaicSkyFlatImageDataProductLoader"/>
        <ref bean="mosaicBiasImageDataProductLoader"/>
        <ref bean="mosaicReducedObjectImageDataProductLoader"/>
        <ref bean="mosaicRawObjectImageDataProductLoader"/>
        ...
        <bean class="edu.noao.nsa.util.workflow.tiny.xml.flow.Pair">
          <property name="action">
            <bean class="edu.noao.nsa.util.workflow.tiny.xml.action.Throw">
              <property name="message" value="No match with specific loaders"/>
            </bean></property>
        </bean>
      </list></property>
    </bean></property>
  <property name="catch">
    <ref bean="genericDataProductLoader"/>
  </property>
</bean>
```

Figure 4: A flow for loading data; the sub–flows are ordered from most to least specific and only the first that succeeds is used.

```
<bean name="_obsTypeSkyFlat" class="edu.noao.nsa.util.workflow.tiny.xml.test.Or">
  <property name="name" value="_obsTypeSkyFlat"/>
  <property name="tests"><list>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SFLAT"/>
    </bean>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SKYFLAT"/>
    </bean>
    <bean class="edu.noao.nsa.util.properties.tiny.PropertyTest">
      <property name="property" value="OBSTYPE"/>
      <property name="value" value="SKY FLAT"/>
    </bean>
  </list></property>
</bean>
```

Figure 5: A composite test for FITS header values in a remediation pipeline.