GeneralizedFlow

Note: As always the ideas expressed here got clearer (and even better) through the cross polination of the different views in the group. You might want to read up the related maillist discussion.

The goal of this writing is

- to smash down and make public some thoughts floating around between SylvainWallez and MarcPortier following these discussions on cocoon-dev recently:
  - http://marc.theaimsgroup.com/?t=105588670600005&r=1&w=2 and
  - http://marc.theaimsgroup.com/?t=105587166400013&r=1&w=2
- to fuel a discussion on how Cocoon can be extended to control webapps
- to put down some bigger picture aswell as some naming and definitions
- propose a design for one and the other

Before getting there we will have

- catalogued 3 types of Controllers for WebApps
- said something about the use of HttpSessions
- given some opinions on Actions
- provided some challenging ways of using flowscripts

Basically we got off-list for some social talk and found ourselves exchanging loads of yepyep's on shared thinking about flowscripts. Before we knew it, this had led to a far better understanding of both flowscripts and how they relate to some other types of interaction-management.

InteractionState

Uhm, an other type of interaction management? Where is it? In your mind: It basically sits in realizing that server side state in webapps can be considered as temporary resources. (URI's with dynamic resource-id's in there) _(please read more about it in

- Pier's description - http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=105589388328188&w=2
- the xreporter documentation, since it is how the flow is managed over there. - http://xreporter.cocoondev.org
- more general similar concepts in many of the ReST related stuff - http://internet.conveyor.com/RESTwiki/moin.cgi/FrontPage

)_

In some detail:
Look at a specific use case of your webapplication: one that consists of a number of end-user-interaction-steps taking the end user via a number of screens through the complete use case.

Then *Starting* this use-case can be seen as: creating a server side 'context' for this interaction.
Let us call that context the "InteractionInstance".
And provide a unique identifier to it that can be encoded into URI's on the server.
(Like this there is some addressable temporary resource which is in fact holding the server side state of your application)

And finally *Continuing* this use case is then easily achieved by calling upon URI's that are holding this identifier.

(Pier's description perfectly clears out why this is different then using the session context for controlling your state)

As the biggest benefits for this kind of working you could

1. have one end user interact in two parallel disctinct InteractionInstances of the same use case without the data messing up
2. have a way to share (publish) these interaction URI's between different end users so more then one can interact with the same instance (think chatroom or helpdesk assisted form completion)

```
Browsers are requesting URI's.
URI's are something between names and addresses pointing to resources.
These resources are at each moment in time in some kind of 'state'.
Dynamic applications kind of implicitely indicate there should be something
as temporary resources.
```

(most of this should sound a bit ReSTy - see http://internet.conveyor.com/RESTwiki/moin.cgi/FrontPage)

WebContinuations rule

The important eye opener Sylvain brought at this time is that flowscript continuations inside Cocoon today effectively allow for this state management model:

In fact:

Starting a flow-interaction currently in Cocoon is like creating a temporary resource that captures (freezes) the state of your flowscript (== interaction description) in a WebContinuation.

indeed: the sitemap will match a URI that maps to the logical name of the usecase you want to engage, and you start that with <map:call function="..." />

Continuing the flow_interaction then actually becomes calling back into that frozen WebContinuation state to continue where it was left of

indeed: the remainder of the end user flow is directed to a *.kont URI that maps to the previously created WebContinuation with a <map:call continutation="{1}" />

So there is only a slight difference in implementation:
While the previous section spoke of ONE temporary resource grabbing the current state of a 'InteractionInstance' for the complete duration of the use case... the WebContinuation approach is allocating a different (unique) temporary resource for awaiting each end-user click during the complete flow.
Conceptually (and from a user perspective) there is hardly any distinction between both.

As Sylvain put it here (http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=105601203401382&w=2) :
(slightly rephrased)

```
The (complete) continuation tree *is* actually the "interaction instance".
```

Remark: It is to be noted that local variables inside a flowscript are frozen into the WebContinuation by reference, and not by value! This means that all distinct continuations inside one continuation tree (that is: inside parallelly started interaction sequences) are in fact pointing to the same values at all times. (explicitely for the interaction scheme where 2 distinct end users engage in one flow instance, this is an important note to make)

Limit transitions

Mind though that this equivalence only works in one direction. InteractionState approach is a special case of the WebContinuation approach (not vice versa)

Mathematically speaking we could write it down like this

```
lim WebContinuation = InteractionInstance
  number_of_sendPageAndWaits_in_the_flow_script -> 1
```

Indeed. Suppose your complete flowscript (including subroutines called) gets written down so it only holds one occurance of the sendPageAndWait() function (probably nested in a while loop). Then all the continuations in the complete tree will be pointing to the same local variables (by ref) and to the same flow-position to go back to. As such they become functionally equivalent. We might even consider in this case to invalidate each continuation as soon as the sendPageAndWait() returns, then the only noticeable difference between the two cases is that the WebContinuation approach will slide in different ID's in the URI upon each request.

The assumption of one sendPageAndWait per flow script might sound as a big limitation, however checking up on some of the current examples:

- numberguessing: GettingStartedWithFlow
- linotype (if you do not consider the login/security): see flow.js
- petstore (some, not all, of the functions there): see petstore.js

we see the same pattern recurring.

When looking at these limits or boundaries of usage there is one more transition we could make:

```
lim InteractionInstance  = Stateless FlowController
  number_of_interaction_clicks_per_instance -> 1
```

we end up in the case of classical servlet programming, where there is no need to remember any state. So we don't really need a temporary resource to be refered to and each request-response cycle is just independently dealt with. (adding ultimate scalability)

Some of us might not see this as genuine webapp-programming, but there really remains a great deal of nice things that can be covered by this easy mechanism. In fact starting again from WebContinuations, this is the special case where the first thing the flow-script will encounter is a mere sendPage(). (notWait)

We end up listing 3 separate models for controlling flow:

1. WebContinuations (each interaction-click creates another temporary (state-) resource on the server)
2. InteractionInstances (there is one temporary resource keeping track of the last (or current) state of the interaction on the server)
3. Stateless FlowControl (there is only some 'static' context and no state variables to take into account during the application flow)

All of these could hook into the sitemap in a similar way since from that point of view they are doing the same things for the sitemap:

1. Offering a set of meaningful (temporary) URI's pointing back to next state-transitions in the interaction flow.
2. Offering a resulting dataBean (bean-dict) with resulting data that can be merged into the publication line
3. Selecting the pipeline to use for presenting the answer to the end-user.

Remark: notice that even the stateless flowcontroller gets to maybe even dynamically provide a set of next links (ie. controlling the flow)... to aid immagination here are some cases where that will occur

- the session is assembling a growing xlink linkbase as the end user wades through your site
- a database table lists over time different 'news' or 'alert' items that need to be linked from the resulting page
- an LDAP server stores user-specific links (e.g those accessible or his/her preferences)
- a specific workflow engine backend is actually making all the decissions, your stateless flowcontroller is just the gateway to that service.

Given this 'special case of' understanding we think that:
If we smash down an API for the WebContinuations then it is very likely that we have passed stages of abstraction where the special cases could be handled directly.

Or put differently:

WebContinuations are obviously on the top of the foodchain here. However:

- the need to have a complete continuation tree managed in the more specialized case of InteractionState
- or the need to hook up your stateless flowcontrollers inside WebContinuations (and currently also javascript)

will at best be seen as an arbitrary (random) hurdle for a developer to take. While one could rate that as 'bad enough' there could even be some criticism on the effect of the additional server-side objects and layers passed on the overall performance and scalability.

Positively: catching these 3 ways of working under one umbrella will leave the publication pipelines in all cases very similar if not the same. As such there will be a gained knowledge re-use and forehoped a more hasardless transition of our web applications from one model to the other...

Events

One thing the InteractionInstance approach seem better at is handling random-end-user-navigation throught the flowscript.

Mind you that the current continuations-based implementation of the flow would at first glance give you the impression that there is only ONE next link to be followed. And thus leave the impatient flow investigator with the understanding that it cannot control a more random or multi-path end-user interaction with the application. After all, making use of the continuations paradigm will require all interaction links to carry the continuation-id and will thus, through the <map:call continuation="{1}" /> all point to the exact same place in the script.

Let us take up the challenge. Assuming one of these typical UI's that show this multipath behaviour:

1. the tabbed-wizard:

1.
    - suppose for this case a very classic wizard-like UI hooking up several stages of screens to be filled in.
    - typically the navigation allows for stepping with next> <previous links through the various screens.
    - this 'tabbed-wizard' would allow direct links to each of the logically named sub-stages

2. the datarecord browser

- a typical record-detail browsing page would have the following links
    - next-record,
    - [input-box-for-id] + goto-record-button,
    - pevious-record,
    - save-changes,
    - create-new,
    - delete-current

(In fact, again linotype offers a nice example in its edit(..) function) We can easily end up writing a flowscript that handles these multiple events and paths through which the end-user might take your application, in the more general case it will look somewhat like:

```
while (!finished){
  sendPageAndWait("view.html");
  // following gets a <map:parameter> from the <map:call continuation>
  String command = cocoon.parameters["command"];
  if (command == "save") {
    doSave();
  } else if (command == "other") {
    doOther();
  }
  finished = // some logical function of having saved or something else...
}
```

where the choosen 'action' got somehow encoded into the URI (and extracted by the sitemap) As such we have effectively written an event-handling loop for dealing with the end-user interaction.

```
Inside a web-application URI's are not as much requesting resources
as they are communicating end-user-selected events?
```

Having declared the notion of events we might just as well slide in the notion of eventhandlers and maybe introduce a notation like:

```
handlers = {"save": doSave, "other": doOther};
sendPageAndWait(view.html);
handleCommand(handlers);
```

The publishing pipelines now will need some mechanism to encode these 'save' and 'other' events into call-back URI's (with temporary URI)

And honesty requires to indicate we haven't seen the light on what the clean way for doing that would really be.

SylvainWallez suggested that this could just be put down in the bean-dict object to get it working without revolutionising current investment and untill we gain some experience with what this will actually be used for...

Indeed: It is quite unclear if this 'set-of-next-flow-links' is to be separated into a possible third argument (of a possible overloaded version) of the sendPageAndWait method:

```
handlers = {save: doSave, other: doOther};
bean_dict = {data : x, message : say };
sendPageAndWait(view.html, bean_dict, buildUriMapForEvents(handlers));
handleCommand(handlers);
```

could easily be handled by

```
handlers = {save: doSave, other: doOther};
bean_dict = {data : x, message : say, flow-links: buildUriMapForEvents(handlers)};
sendPageAndWait(view.html, bean_dict);
handleCommand(handlers);
```

Finally, if we could assume this, we might even allow the jxtransformer to react on more meaningful flow-indicating parameters then only the #{$continuation/id} construct (as in : <form action="#{$continuation/id}.kont" method="POST">)

this specific one could get a name detached of the continuation implementation by calling it flow.next, and likewise the named events could get a flow.save, flow.other, etc. etc.

We have to think some more on this encoding of 'events' cause people would want to encode the next links in very variant (even deviant) ways: form/@action as above, but equally some input[type='submit']/@value and surely a/@href, possibly even clientside-javascript{location=..}

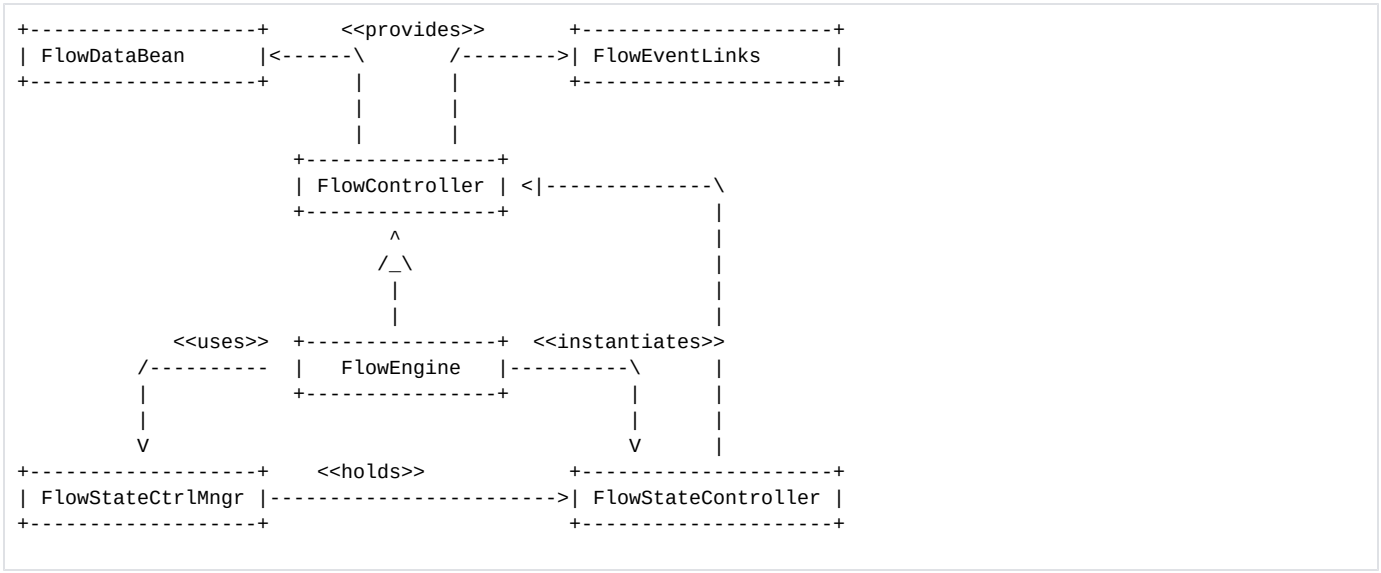
Names, Definitions and Design Proposal

Now, off for a question: How do you control your webapp?

As a developer? You don't! The web is purely re-active. The end-user (browser) is in charge at all times by the mere pull nature of the web. It's his/her click (incoming HTTPRequest) that is deciding what's up next. The smart developer however immediately understands that dynamically generated HTML, showing an abundance of seducing widgets is in fact exposing only a very limited list of sensible 'next' actions the end user can engage upon. Indeed it is only declaring those URI's to be followed by the end user.

This "Providing the set of possible links" is precisely what the Cocoon FLOW (in more general terms) is about.

Here is how Flow deciding components could be mapped onto some interfaces and their interrelation.



- **FlowController** is a thing that

1. selects a pipe,
2. can set in the environment/objectmodel a FlowDataBean (bean-dict) ready to be mixed into the pipe,
3. can declare FlowEventLinks for having call backs into the FlowController user customisation should include easy writing their own FlowControllerImpl (ActionNG) for stateless arbitrary control logic

- **FlowEngine** extends FlowController
  *depicts implementations of the FlowController that will create temporary resources that are tied to unique URI's and are managed by FlowStateControllerManager
  *has two foreseen inplementations WebContinuationsFlowEngine and InteractionInstanceFlowEngine

- **FlowStateControllerManager** is a cache (manager) of FlowStateController instances
- **FlowStateController** extends FlowController
  - depicts implementations of the FlowController that are stateful, tied to a unique idenifier and
  - has two foreseen implementations WebContinuation and InteractionInstance
- **FlowEventLinks** is an object that enlists the guided next links an end-user should use in this application. Each of these event-links will map onto a separate unique uri.
- **FlowDataBean** is a simple JXPath accessible object that holds 'resulting' data from the executed arbitrary logic (function of the current bean-dict)

Hooking into current discussions:

- the FOM (as being discussed on the list recently) in fact resembles the arguments and instance-variables that are accessible inside the WebContinuationFlowEngine and should be made accessible to the js-flowsctipts
- Flow as such has been tied to WebContinuations inside the Cocoon community. If this small document made any sense at all, we should try to avoid this name-coupling in future?

faq

Q: Why don't we just use Actions for the stateless flow control?

A very good question indeed. In fact looking at it from the WebContinuations angle there is quite little distinction to be made between

```
         <map:call function="fn_name" /> and <map:act type="start-flow" src="fn_name" />
or even  <map:call continuation="{1}" /> and <map:act type="cont-flow" src="{1}" />
```

Actions however have been quite largely seen as 'bad', a stigma they probably got while the community extended the original Action interface so it could be a haven for 'arbitrary control logic'. Having said that, we believe that current webapps still have room for a more classic 'stateless flow control'. If we now try to hook in arbitrary logic for the FlowControl, we should envision it having a simple mechanism for doing stateless stuff.

Rather then wanting us to write up the flow with the map:act equivalent we would hope for a flow-umbrella API that allows for stateless implementations as well. Small changes in the wording of the sitemap-elements and attributes could achieve a lot:

```
call/@function becomes:   <map:call controller=".." />
call/@continuation:       <map:call state=".." /> -- not to be used for stateless controllers
```

Also check the remark in section 3 for a couple of examples where even these stateless implementations could be setting FlowEventLinks and a FlowDataBean (without them needing to be Actions, so we can have Actions become what they were intended for?)

Q: Why do you hardly mention sessions in this story?

Binding server side state to the infamous sessions (often cookie based) is an orthogonal way of working which is obviously accessible to any of the 3 mechanisms we have catalogued.

As became clear in the story of Pier the session is in fact a kind of store (Map) tied to the user-agent-instance (the one remembering the cookie) as such it declares a clear context to store anything a developer would want. (That of course includes Developers of FlowEngine Implementations)

How could development look like

The danger of any example is that narrows the reasoning too much down to a specific case. This specific case one could argue will best fit one implementation or the other. The hope is that we could avoid this argumentation all together and just see how 'different' things can fit in.

Suppose the classic guestbook. For some reason we split up the 'make-your-entry' process in 3 steps. The following 3 simple jx-pages could be grabbing the pages the end user will see.

```
entry-part1.jx
<form action="#{$flow/next}">
  Name:       <input name="name"  value="#{name}"  />
  Email:      <input name="email" value="#{email}" />
  WebAddress: <input name="www"   value="#{www}"   />
  <input type="submit" name="ok"/>
</form>

entry-part2.jx
<form action="#{$flow/next}" />
  Entry:
   <textarea name="entry">#{entry}</textarea>
</form>

entry-part3.jx
<h1>Preview your entry</h1>
  Your details: <a href="#{$flow/change-name}"> Change those </a>
  <a href="mailto:#{email}>#{name}</a> from <a href="#{www}">#{www}</a>

  Entry: <a href="#{$flow/change-entry}"> Change that </a>
  <p>#{entry}</p>
```

above pages get published through some:

```
<map:match pattern="guestbook/entry/part/*" />
  <map:generate type="jx" src="entry-part{1}.jx" />
  <map:serialize />
</map:match>
```

obviously there will be some thx.html and guestbook.html pages to complete the show.

Solution 1 - WebContinuations.

```
<map:flow>
<!-- where flowengine-type="js-continuations" -->
     <map:script src="guestbook-flow.js"/>
</map:flow>

<map:pipeline>
  <map:match pattern="**/*.kont">
    <map:call state="{2}"/>
  </map:match>

  <map:match pattern="guestbook-entry">
    <map:call controller="make_entry()" />
  </map:match>
</map:pipeline>


// simple version if we're not using the 'named links' stepping back in the flow
function make_entry() {
  var name  = "(your name here)";
  var email = "your.name@here";
  var www   = "http://there";
  var entry = "(your guestbook entry)";

  var i = 1;
  while (i<=3){
    var dataBean = {name: name, email: email, www: www, entry: entry};
    sendPageAndWait('guestbook/entry/part/' + i, dataBean, flowLinks );
    i++;
  }
  // actual code writing the entry to file obviously...
  sendPage('guestbook/thx.html');
}
```

Solution 2 - InteractionState

```
<map:flow>
<!-- where flowengine-type="InteractionState" -->
    <map:class="com.some.cocoon.interactions.GuestBookEntry" name="GuestBookEntry"/>
</map:flow>

<map:pipeline>
  <map:match pattern="**/*.kont">
    <map:call state="{2}"/>
  </map:match>

  <map:match pattern="guestbook-entry">
    <map:call controller="GuestBookEntry" />
  </map:match>
</map:pipeline>


// the following must be doable with a lot less code
// (by making this class his own eventHandler probably),
// Having it as explicit as here however pushes me to think some classes like this
// could be build from XML config files, maybe slide in some scripting in there
// and end up with something that could be called FSP: FlowScriptingPages?
package com.some.cocoon.interactions;

import..;

public class GuestBookEntry
implements FlowStateController, LogEnabled, Serviceable, ..  {

    SimpleFlowEventLinks flowEvents = new SimpleFlowEventLinks(super.getStateRef());
    FlowDataBean dataBean = new RequestFillableDataBean();
    String showStage = "guestbook/entry/part/1";
```

```
    public GuestBookEntry  {
        flowEvents.add("change-name", new FlowEventHandler(){
            public boolean handleEvent() {
                GuestBookEntry.this.showStage = "guestbook/entry/part/1";
                GuestBookEntry.this.flowEvents.setNextEvent("change-entry");
                return FlowStateController.CONTINUE;
            }
        });

        flowEvents.add("change-entry", new FlowEventHandler(){
            public boolean handleEvent(){
                GuestBookEntry.this.showStage = "guestbook/entry/part/2";
                GuestBookEntry.this.flowEvents.setNextEvent("check");
                return FlowStateController.CONTINUE;
            }
        });

        flowEvents.add("check", new FlowEventHandler(){
            public boolean handleEvent() {
                GuestBookEntry.this.showStage = "guestbook/entry/part/3";
                GuestBookEntry.this.flowEvents.setNextEvent("final");
                return FlowStateController.CONTINUE;
            }
        });

        flowEvents.add("final", new FlowEventHandler(){
            public boolean handleEvent() {
                GuestBookEntry.this.showStage = "guestbook/thx.html";
                // actual code to save the entry somewhere
                return FlowStateController.END;
            }
        });

       flowEvents.setNextEvent("change-entry");
    }


    public boolean controlFlow(FlowRequest request, FlowResponse response) {
        dataBean.updateDataFromRequest(request);
        boolean isFinished = flowLinks.handleEvent(request.getEventName())
        response.setPage(this.showStage);
        response.setDataBean(this.dataBean);
        response.setFlowLinks(this.flowEvents.getLinks());
        return isFinished;
    }
}
```

Solution 3 - Stateless

```
<map:flow>
<!-- should there be no flowengine here?  -->
  <map:class="GuestBookChangeName"  name="guestbook-change-name"/>
  <map:class="GuestBookChangeEntry" name="guestbook-change-entry"/>
  <map:class="GuestBookCheck"       name="guestbook-check"/>
  <map:class="GuestBookProcess"     name="guestbook-process"/>
</map:flow>

<map:pipeline>
  <map:match pattern="guestbook/change-name" >
    <map:call controller="guestbook-change-name" />
  </map:match>
  <map:match pattern="guestbook/change-entry" >
    <map:call controller="guestbook-change-entry" />
  </map:match>
  <map:match pattern="guestbook/check" >
    <map:call controller="guestbook-check" />
  </map:match>
  <map:match pattern="guestbook/process" >
    <map:call controller="guestbook-process" />
  </map:match>
</map:pipeline>
```

with the quite cumbersome four different classes... public class GuestBookChangeName extends FlowController {...}

which would easily be compared with classic servlet (not so smart) coding we have left behind us some time ago. Biggest difference to those servlets still would be that these controllers only decide and prepare, but have separated out the actual compilation and styling of the return HTML (a huge benefit if you attack this from a servlet background IMHO).

And while we mentioned 'not so smart' there is no real need to look down on this approach:
The use for stateless stuff like this is very likely to be useful when your back-end has all the power and we practically just need to provide some simple gateway, in which case we reduce all this to

```
<map:flow>
  <map:flowcontroller class="GuestBookGateWay" name="guestbook-gw"/>
</map:flow>

<map:pipeline>
  <map:match pattern="guestbook/*" >
    <map:call controller="guestbook-gw" >
      <map:parameter name="event" value="{1}" />
    </map:call>
  </map:match>
</map:pipeline>
```

Where the controller could just read the parameter, and use it to decide which back-end service to call. Backend-return just holds all of flowLinks and flowDataBean that might needs to be wrapped, mapped or translated...

leftovers

apart from all the code to actually do this (and that do-experience modifying more then just some details), the biggest thing to hope for on short terms is some challenging remarks on this

some of what we realise already:

- If we really want these FlowController instances be accessible by more then one person at a time we will need

1. some kind of locking on them (since they *are* statefull)

2. some access control management as well. (since the session will not do it for us)

- To be useable in practice the InteractionState approach might require dynamic loading (possibly even compilation?) of the implementation classes

- left unclear how we actually best encode the event in the URI, having some dislike to the ?event= approach, but can't put it down yet. I remember some remark about how matching **/*.kont was really easy, and I should check upon the greadiness of the * matcher to see if some **/*.kont/** would work where **3** would catch the event-subresource-path

- related: the nature of the FlowEventLinks might not be really a map of URIs: Which events the controller can respond to is a part of his interface, and the publishing pipelines (matching that contract) should just decide how they encode that themselves... only thing that remains then is that 'logic' might want to indicate which state-transitions are 'allowed' or 'enabled' at any given moment in time (ie in a particular state of the interaction)... so it ends up more of a UI thingy then. So just passing back a list of 'allowed links now' would do.
- another idea came from SylvainWallez to load off the uri-encoding issue to the LinkRewriter: <a href="flow-event:event-name"> maybe...

- without wanting to re-open the discussion on short notice (and thus jeopardize some implicit release planning) this might lead to reconsidering that one sitemap would like to have more then one flowControllerEngine active inside one sitemap. No real opinions yet, but it seems to be opening a logic 'expectation' after enhancing the reuse of publishing pipelines across the different programming models.

- and some old mindspin that pops up again: the mount/processer similarity in all of this:

what to think about something that could dynamically mount the same subsitemap.xmap file as separate sitemap-instances on dynamically generated sub-resource-id's?

something like uri at /whatever/mount-name/** creates a dynamic ID and uses it to

- add some sort of dynamic mount point at /whatever/mount-name/MOUNT-ID.DYN

- redirects the browser to it with /whatever/mount-name/MOUNT-ID.DYN/{1}

after that all requests to /whatever/mount-name/MOUNT-ID.DYN/** are handled by a separate instance of that sub-sitemap?

only thing left would be to allow this subsitemap to declare room for some member-variables in his own context. (ie allowing the sitemap to have the same kind of attribute-holding possibilities as session or request contexts)

parent sitemap could look like:

```
<map:match pattern="mount-name/*.DYN/**">
  <map:mount instance="{1}" uri-prefix="mount-name/{1}.DYN"/>
</map:match>

<map:match pattern="mount-name/**">
  <map:act type="dynamount" >
    <map:redirect src="mount-name/{mount-id}.DYN/{1}/>
  </map:act>
</map:match>
```

the subsitemap in guestbook/sitemap.xmap would just have some simple pipelines exchanging objects with the sitemap's own context... leaving sub-uri management as we know it today.

- all the things we haven't thought of yet.