WoodyRefactoring

About

To get CFORMS into a stable block there is some upcoming refactoring required.

The goal is to assemble the actual coding ideas around various topics that need to be handled. Hopefully this yields some overview which allows us to plan and actually build.

Some of these topics have their own pages discussing the underlaying concepts/features in more detail:

- CubistForms on the roles and responsibilities surrounding 'cforms'
- WoodyWidgetLifecycle on the various states and transitions a widget experience during its lifetime
- WoodyScratchpad on the topics 'widget-reuse' and the 'choice' widget
- WidgetStates on the various states of a widget (e.g. input, output, hidden)

Widget Lifecycle management

Describing it: WoodyWidgetLifecycle

The goal of this refactoring is to assign semantically meaningful names to more atomic methods that independently handle parsing, validation, event-registration.

From there we should be able to more clearly combine these in the actual use cases of these fields

- generateSaxFragment
- read-from-request
- eventhandling
- programmatic manipulation (like binding)

while correctly handling internal processing issues:

- splitting values from aggregate-widget down to the part-fields
- choice-widget value leading to new widget instances
- eventhandling code setting values triggering events

Widget Re-use through @defines and @extends

The goal of this refactoring is to change the building process so we support widget reuse from local definitions and external repositories.

See WoodyScratchpad for the more conceptual discussion.

The introduction of the @defines and @extends attributes largely have an influence on the process of "Building The WidgetDefinitions". This process is about interpreting the form-definition files and making the corresponding WidgetDefinition instances out of it.

Without the @define and @extend attributes these WidgetDefinition instances only had one purpose: to produce Widget instances of them (through createInstance()) to be used in actual form-models used during the control of the end-user interaction with a cforms based application.

With the advent of these new attributes we are obliged to solve the following issues:

- @extend: requires us to solve
  - lookup of a registered, existing 'base-definition'
  - extending that 'base' with additional configurational info
- @define:
  - creation of a WidgetDefinition that has no 'id', and that because of that cannot be asked to produce actual widgets over the createInstance() (if you look at WidgetDefinitions as being Classes, you could see this state of base-definition as being 'abstract')
  - registration of that base-WidgetDefinition for later reference

Above can be grouped into two new aspects to cover:

1. registration/lookup of abstract-like base-definitions (no 'id', can't produce instances)
2. all of the actual creation (@define), the extension(@extends) and the 'concretization' (@id) of the WidgetDefinitions

The registration/lookup seems logical to be moved under a specific component that uses the tuple source-of-definition-file and basename-of-widgetdefinition as a key to find back the actual base-definitions. This needs some aditional thinking:

- IIRC there was the need to refer to base-definitions that only get loaded later in the file? (update: the reason has to be (again) the case of the recursive definition with the union/choice construct, some early thoughts on solving this were here: http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=107346318901387&w=2)

- in any case the widget-definitions have some parent-child relation that needs to be built up and kind of defies the all immutable idea. (tim: maybe not, see ImmutableWidgetDefinitions)
- And in the more general case one could be parsing a definition file which refers to a source that wasn't loaded yet --> nope: the @wdns attribute must be handled and trigger the load of the refered source before the builder continues with his own work
- we should also have some concurrency management on these things since we wouldn't want to have two concurrent calls for some form to be triggering the concurrent loading of the same definition-source twice?

AbstractDefinitionRegistry would thus minimally allow for:

WidgetDefinition lookup(String source, String baseName)

void register(String source, String baseName, WidgetDefinition newBaseDef)

The registration/lookup seems to be a task for the various WidgetBuilders. This would call for their buildWidgetDefinition method to take a second argument that gets passed down in the builder hierarchy and allows for the register/lookup. This should not be the actual AbstractDefinitionRegistry mentioned before but a Helper class that is tight to the processing of the current source. Like this it could automatically insert the current source in the register() action, but could also maintain the local prefix-to-source-mapping so it can assist to translate prefix to source during lookup.

so rather some DefinitionRegistryWrapper:

- <init> taking the String source of the currently processed file, and a ref to the actual AbstractDefinitionRegistry
- void pushPrefixMapping(String pfx, String source);
- String popPrefixMapping(String pfx);
- WidgetDefinition lookup(String baseNameWithPrefix);
- void register(String localBaseName, WidgetDefinition newBaseDef);

Now to the second aspect: the various instances of 'similar' WidgetDefinitions. The WidgetDefinitions are to be unique and distinct (and immutable IMHO) for any of their possible incarnations matching a certain occurence in some definition file. This means that there should be distinct instances for each @define, @define and @extend, @id, @id and @extend occurance of the widget-definition (all with their own Location position, and base/extended configuration settings). Making the actual members of these Widgets into final fields will help us in making sure that we're not altering exisiting WidgetDefinition instances, but really are making new ones.

As a consequence however we will need to provide so called copy-constructors for these WidgetDefinition classes that allow to instantiate new ones by copying over and extending the member fields into the new instance.

The WidgetDefinitionBuilder, having access to the registry (see above) could then be following this pattern.

Note: The instant idea of the Config inner class is to prevent the ugly long parameter-lists of the constructors as seen in todays Binding classes and to keep the configuration logic of interpreting the XML config central to the Builder class.

```
public class XYZWidgetDefinitionBuilder {

  Config interpretXMLConfig(Element elmConfig) {
    // read various parts of the config
    final TypeK newPartK = DomHelper.....(elmConfig);
    final TypeL newPartL = ...

    // Note: this should also have ready build children for those
    // widget-definitions that contain other widget-definitions
    // note however that these nested Definitions all will need to
    // have their @id being set!

    return new Object() {
      TypeK getPartK(TypeK basePartK) {
        //merges, overrides or keeps/clones basePartK over newPartK
        return newPartK;
      }
      TypeL getPartL(TypeL basePartL) {
        //merges, overrides or keeps/clones basePartL over newPartL
        return newPartL;
      }
    };
  }


  XYZWidgetDefinition buildWidgetDefinition(Element elm, DefinitionRegistryWrapper reg) {
    Config conf = interpretXMLConfig(elm);
    String location = DomHelper.getLocation(elm);

    String baseName = DomHelper.getAttribute("extends", null);
    XYZWidgetDefinition definition= null;
    if(baseName != null)  {
      definition = reg.lookup(baseName);
    }

    String defines = DomHelper.getAttribute("defines", null);
    if (defines != null) {
      definition = new XYZWidgetDefinition(null, location, definition , config);
      reg.register(defines, definition);
      config = null;
    }

    String id = DomHelper.getAttribute("id", null);
    if (id != null) {
      definition = new XYZWidgetDefinition(null, location, definition , config);
    }

    //possibly throw an exception if both @id and @defines where null?
    return definition ;
  }
```

Inside XYZWidgetDefinition's constructor something like the following could go on:

```
  private final String id;
  private final String location;
  private final TypeK partK;
  private final TypeL partL;
  ...

  XYZWidgetDefinition(String id, String location,
                      XYZWidgetDefinition base, XYZWidgetConfig config) {
    this.id = id;
    if (base != null} {
      this.lccation = location + "\n\tBased on defition at " + base.location;
      this.partK = config.getPartK(base.partK);
      this.partL = config.getPartL(base.partL);
      ...
    } else {
      this.location = location;
      this.partK = config.getPartK(base.partK);
      this.partL = config.getPartL(base.partL);
      ...
    }
  }

  // some more ideas:
  boolean isAbstract() {
    return (this.id == null);
  }

  XYZWidget createInstance() {
    if isAbstract() throw new ....
    // from here somethin like current instantiation could happen
  }
```

Extra features to the repeater widget

- enable the selection of rows: http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=108123604720257&w=2
- add 'identity' property to rows (needed for binding)

Binding refactoring

- cleaning out the interface
- making the binding-builders actual avalon components (like the definition-builders)

- repeater-binding
    - combine the various clones under one umbrella as described here:

http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=108059910416310&w=2

- combining some currently separated bindings
    - aggregate and context?
    - multivalue and various repeaters (some hints on that were expressed here: http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=108209653422163&w=2)
- do something about the binding/widget naming confusion as described here:

http://marc.theaimsgroup.com/?l=xml-cocoon-dev&m=108194795223412&w=2