WoodyScratchpad

Various notes about some ongoing thoughts.

Please add comments, clarify wording, contribute ideas, etc.

These ideas are here for discussion, and have not been implemented yet.

Update: The implementation ideas around these (and other) concepts have moved to a separate page WoodyRefactoring

Macros and Types – id/define/expand/inherit

A macro is a definition (or blueprint) consisting of a list of widget definitions. When a macro is "expanded" it creates instances of those widget definitions in the surrounding container. No "macro" widget or container is created, because a macro is *only*
a definition.

So this:

```
<fd:macro define="mymacro">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:macro>

<fd:struct id="mystruct">
  <fd:macro expand="mymacro"/>
</fd:struct>
```

is equivalent to this:

```
<fd:struct id="mystruct">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:struct>
```

...except that now you have a reusable macro named "mymacro" that you could expand in several places in your form instead of typing the same list of widget definition over and over in each place where you need them.

Sometimes you may want to define a general macro, and then create variants based on it. Think of this as creating a blueprint that contains most of your details, and then making photocopies of this blueprint to fill in different finishing touches to customize each copy. In terms of forms, this would consist of adding, deleting, and/or modifying the widget definitions in the copies of the main macro.

So this:

```
<fd:macro define="mymacro">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:macro>

<fd:macro define="yourmacro" inherit="mymacro">
  <fd:add>
    <fd:field id="anotherfield">
      ...
    </fd:field>
  </fd:add>
</fd:macro>

<fd:struct id="mystruct">
  <fd:macro expand="mymacro"/>
</fd:struct>

<fd:struct id="yourstruct">
  <fd:macro expand="yourmacro"/>
</fd:struct>
```

would be equivalent to:

```
<fd:struct id="mystruct">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:struct>

<fd:struct id="yourstruct">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
  <fd:field id="anotherfield">
    ...
  </fd:field>
</fd:struct>
```

This is all well and nice, but if your customizations are only used in one place it would be nice to be able to specify your changes right where you use them.

So this:

```
<fd:macro define="mymacro">
  <fd:field id="myfield">
    ...
  </fd:field>
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:macro>

<fd:struct id="yourstruct">
  <fd:macro expand="yourmacro">
    <fd:del>
      <fd:field id="myfield"/>
    </fd:del>
  </fd:macro>
</fd:struct>
```

would be equivalent to:

```
<fd:struct id="yourstruct">
  <fd:field id="yourfield">
    ...
  </fd:field>
</fd:struct>
```

While thinking about how nice it would be to be able to define, inherit from, expand, and modify macros as explained above, you may realize you would like to be able to do the same to individual widget definitions...

Create a widget:

(Note: This is unchanged from current Woody/CForms.)

```
  <wd:field id="field-a">
    <!-- validation, etc. -->
  </wd:field>
```

Create a type:

```
  <wd:field define="type-a">
    <!-- validation, etc. -->
  </wd:field>
```

Create a widget inheriting from a type:

```
  <wd:field id="field-b" inherit="type-a">
    <!-- validation, etc. -->
  </wd:field>
```

Create a type inheriting from an existing type:

```
  <wd:field define="type-b" inherit="type-a">
    <!-- validation, etc. -->
  </wd:field>
```

Create a widget *and* create a new type based on this widget definition:

```
<wd:field id="field-c" define="type-c">
  <!-- validation, etc. -->
</wd:field>
```

Create a widget inheriting from an existing type and create a new type based on this widget definition:

```
<wd:field id="field-d" define="type-d" inherit="type-c">
  <!-- validation, etc. -->
</wd:field>
```

Create a widget inheriting from a type without knowing what kind of widget it is(e.g. Field/Repeater/etc.):

(Note the use of the generic "wd:widget" element.)

```
<wd:widget id="widget-e" inherit="type-c">
  <!-- validation, etc. (limited to aspects that do not depend on the kind of widget represented) -->
</wd:widget>
```

Repository – wd:import/wd:widget

After types are implemented (as described above), we can add type repositories to Woody/Cforms. A repository could simply be a Source identified by URI and given a prefix for later reference. This will allow us to store widget type definitions anywhere that can be referenced as a source:

```
cocoon://some/path/and/filename
context://some/other/path/and/filename
cvs://...
webdav://...
```

Each repository would contain a set of type definitions.

A form would bring the set of types into scope via an import statement:

```
<wd:import prefix="my-types" src="cocoon://MyOwnTypes.xml"/>
```

Type repositories may be originally be implemented for different projects. When they are later used together there needs to be a way to resolve the naming conflicts that arise. This is the purpose of the "prefix" attribute.

```
<wd:widget id="my-widget" inherit="my-types.some-type-definition"/>
```

Alternative suggestion (*added 20040329 by mpo*):

(tim: +1 to this alternative; nice syntax, and conveys the right ideas more clearly.)

SylvainWallez: -0.5. I consider a wd:import as nothing more than an externally-defined container widget and therefore the "." notation can be used as everywhere else. Accordingly "prefix" on wd:import should actually be "id". Let's also strees the concept: what if a repository includes another repository. Can we have several colons in a widget reference? Also (unrelated to repositories, actually) we need a way to crawl up the ancestor hierarchy if a widget in a container (e.g. repeater) is of a type defined outside the container.

TimLarson: I will answer this in several steps. First, the "wd:import" is meant as direct parallel to java's "import" statement. It should not create or insert any widget definitions on its own, but just bring widget definition types into scope. The "inherit" syntax defined in the section above is used to actually create widget definitions based on the types defined in the imported source. Second, imports are private to the form or repository that declares them. If a repository imports another repository and wants to make its types available, it will have to do "<wd:widget inherit="nested-ns:type" defines"type"/> for each imported type it wants to expose, and these would then become part of its own namespace. Internally this will just add some references, not not waste memeory on unchanged copies of the definitions. We could also introduce a mass syntax for this. Third, could you explain further about this crawling need?

Given the fact that this starts looking like namespaces a lot, why not also adopt the syntax of namespaces?

```
<wd:any-scope wdns:my-types="cocoon://MyOwnTypes.xml"
              wdns:other-types="cocoon://OtherTypes.xml"/>
```

and reference some definition in there with: (preferig the colon over the dot)

```
<wd:widget id="my-widget" inherit="my-types:some-type-definition"/>
```

IMHO having the prefix-to-source binding declared as attributes makes the scoping a lot more transparant. (otherwise the lookup for resolving would not only be about looking in the ancestor axis, but also into the preceding-sibling which IMHO 'blurs' the relation between containment and scope?)

This would also stress nicely that prefixes only have local scope, that the sources are the real important things, and that the form-manger could cache these repositories based on the source. (tim: +1 the pre-built definition caching is one of the reasons for importing instead of just using (x|c)include)

Additionally we could have the form-manager entry in the xconf kind of pre-load repositories like this by adding the sources (without prefixes since those have only local meaning) to the configuration?

Questions:

- Where should imports be allowed?
  *Top of the form definition.
  *Top of any container widget definition.
  *Anywhere a widget definition is allowed. (+1 mpo, +1 tim, +1 sylvain)
- Where should imported types be registered (affects namespace)? (mpo: I don't get this question) (tim: let's ignore it, I don't think it makes sense anymore.)
  *Children of the form definition.
  *Children of the container widget definition that contains the import statement.
- What should be importable?
  *Widget definition prototypes. (+1 mpo)
  *Widget definitions that will have instances created. (tim: Does this have any usecases?)
  *Other?
- Are forward references allowed?
  *No. This avoids cyclic dependencies in the definition (and also eases the implementation).
  **TimLarson: Yes, otherwise we break recursive GUI editor forms. (fyi, they are the reason I am working on this.)*

Conditional – choice/case

The following structure supports dynamic (runtime) widget selection and lazy (a.k.a. on-demand) widget creation based on a static form definition. This is intended to replace the "union" widget.

Two (mutually exclusive) versions are available:

- Single string-valued expression selects a case (almost like a switch statement in the 'C' language).
- Boolean-valued expression on each case (like a chain of if...else if...else if...else).

(*mpo: I like the latter most, reads like <xsl:choose><xsl:when test="..">..<xsl:otherwise>..*) (*tim: I was thinking of offering both forms, not just one-or-the-other, because I have usecases for both.*)

Single string-valued expression selects a case:

Semantics:
The expression is evaluated to produce a string matching one of the case id's. The widgets referenced or contained by this selected case are created if they do not yet exist.

Questions:

- Instead of using an empty "id" attribute should we use <wd:default/> as the default case? (+1 tim, +1 mpo, +1 jh)
- How should we indicate when we do not want to allow a default selection?
  *Use 'required="true"'.
  *Do not supply a default case. (+1 mpo)
- What should we do if the expression evaluates to the default anyway? (*mpo: seems more logic in the when@test/otherwise filosophy?*) (*tim: I do not understand; could you clarify your comment?*)
- What namespace (read: generated request parameter names) should the widget id's be in?
  *widget-id – Same as widgets outside the choice. (+1 tim, +1 mpo, +1 jh)
  *choice-id.widget-id – wrapped by the choice's namespace. (-0 tim, +0 jh)
  *case-id.widget-id – wrapped by th case's namespace. (-0 tim, -1 mpo, see below, -1 jh)
  *choice-id.case-id.widget-id – wrapped by the choice's and the case's namespaces. (-1 tim, -1 mpo since the cases are exclusive, -1 jh)

Comments:
Please add some comments.

```
Form model syntax:
<wd:choice expr="some-string-valued-expression-producing-a-case-id">
  <!-- Zero or more widget definitions -->
  <!-- One or more cases -->
  <wd:case id="some-case-id">
    <!-- list of inline widget definitions and/or
         references to the widgets defined above -->
  </wd:case>
  <!-- Optional default case specified by empty "id" attribute: -->
  <wd:default>
    <!-- list of inline widget definitions and/or
         references to the widgets defined above -->
  </wd:default>
</wd:choice>

Example:
<wd:choice expr="some-expression">
  <wd:field id="field-A".../>
  <wd:repeater id="repeater-B" .../>
  <wd:case id="case-0">
    <wd:ref id="field-A"/>
  </wd:case>
  <wd:case id="case-1">
    <wd:ref id="field-A"/>
    <wd:ref id="repeater-A"/>
  </wd:case>
  <wd:case id="case-2">
    <wd:booleanfield id="bool-C".../>
  </wd:case>
  <wd:case id="">
    <wd:ref id="field-A"/>
    <wd:field id="field-B".../>
  </wd:case>
</wd:choice>

Form template syntax:
<ft:choose path="path-to-some-data-widget">
  <!-- Other non-template elements -->
  <ft:when value="some-string-value">
    <!-- Templates and other elements -->
  </ft:when>
  <!-- Other non-template elements -->
  <ft:when value="some-other-string-value">
    <!-- Templates and other elements -->
  </ft:when>
  <!-- Other non-template elements -->
</ft:choose>
(Minor thought: Should we allow other template elements where
it is currently marked "Other non-template elements", to allow
for easy conditional sequencing of conditional and non-conditional
templates?)
```

Expression on each case:

Semantics: The case expressions are evaluated in sequence. The first case with an expression that evaluates to true is selected, expression evaluation stops, the widgets referenced or contained by the selected case are created if they do not yet exist.

Questions:
Pretty much the same questions as above.

Comments:
Please add some comments.

mpo comments: I would prefer the case/@expr and default to become when/@test and otherwise

tim comments: when/@test would be fine with me; should we also copy the word 'choose' from xslt, instead of using 'choice'? 'choice' seems more declarative, but I would be ok with either word.

```
<wd:choice id="some-id">
  <!-- Zero or more widget definitions -->
  <!-- One or more cases -->
  <wd:case id="some-case-id" expr="some-boolean-valued-expression>
    <!-- list of inline widget definitions and/or
         references to the widgets defined above -->
  </wd:case>
  <!-- Optionally, a default case can be given like this: -->
  <wd:case id="another-case-id" expr="some-boolean-valued-expression-that-equals-true">
    <!-- list of inline widget definitions and/or
         references to the widgets defined above -->
  </wd:case>
</wd:choice>

<wd:choice id="some-id">
  <wd:field id="field-A".../>
  <wd:repeater id="repeater-B" .../>
  <wd:case id="case-0" expr="case-1-expression">
    <wd:ref id="field-A"/>
  </wd:case>
  <wd:case id="case-1" expr="case-1-expression">
    <wd:ref id="field-A"/>
    <wd:ref id="repeater-A"/>
  </wd:case>
  <wd:case id="case-2" expr="case-2-expression">
    <wd:booleanfield id="bool-C".../>
  </wd:case>
  <wd:case id="case-3" expr="true">
    <wd:ref id="field-A"/>
    <wd:field id="field-B".../>
  </wd:case>
</wd:choice>
```

Masks

Evolution of an idea: union -> choice/case -> masks

What if instead of choosing between disjoint sets (union), or choosing between cherry-picked widgets (choice/case), we could specify action masks for trees of widgets (masks!)

A mask is similar to a "case" from the "choice/case" proposal above except that it does not choose between states, but instead it performs actions on a tree of widgets, changing various attributes as it goes. This would allow us to individually control attributes like existence, visibility, validation, enabled /disabled, processing of request parameters, and even changing sets of labels on the fly.

I picture something similar to Jetty's configuration files, where calls to java methods with parameters and calls to simple setters and getters are all encoded into an XML format. We can create a simplified XML format for these action masks, and then either embed them in the form model files or supply them externally via pipelines (using XSLT, etc. for freeform transformations) for mask builders to parse into mask objects. If we provide a good API we could even build our own masks dynamically from Java or Javascript.

A mask would sit in the middle between being declarative and being procedural. Instead of choosing between cases, we could apply one or more masks in sequence to apply a series of sweeping changes to trees of wigets. You could think of it as a type of shorthand for changes, possibly backed up with some internal mechanism to make mass changes ligher weight than individually performing each attribute change.

WDYT?

Here are some options/stages we could use on the way to masks:

Hacky, but we get full control now with no changes to CForms:
For each piece of data that needs to be controlled, make a set of widgets wrapped in a union, and use flowscript, java or widget event code to copy data between these widgets when switching between "union" cases, to make them roughly simulate one controllable widget. If we have code that reacts to valueChanged events we would of course have to filter out the events that are caused by copying values between these widgets.

Clean, procedural, requires light changes to CForms codebase:
Modify CForms to make more attributes dynamically controllable. Then we can make calls from flowscript, java, or widget event code to change the attributes at runtime, with no need for "unions" or for using sets of widgets to simulate more flexible widgets.

Clean, declarative, requires medium changes to CForms:
The "masks" idea presented earlier. This builds on the changes listed above (making more attributes dynamically controllable), by providing a way to specify the changes via masks encoded as XML fragments. This would involve creating the XML mask syntax, and making classes that would build and execute mask objects based on XML fragments that are written in this syntax. This would allow us to dynamically apply masks that we processed via XSLT, as well as masks that we defined statically in our form model files.

Here is a sample set of "masks", using some plausible syntax. We would define our widgets normally in our form model, and then also either include these mask in our form model or supply them via a pipeline where we do our XSLT or other processing. Note that none of this is coded yet, this is just a proposal:

```
<mask id="make-stuff-editable">
  <!-- ...and if it does not exist yet, create it. -->
  <widget id="someWidget" exist="true" mode="edit"/>
  <widget id="someOtherWidget" mode="edit"/>
</mask>
<mask id="and-now-change-it-again">
  <widget id="someWidget" exist="true" mode="read"/>
  <!-- Don't need "exist" because this widget is static. -->
  <widget id="someOtherWidget" mode="hide"/>
</mask>
<mask id="remove-that-widget">
  <!-- If widgets exists, delete it, otherwise do nothing -->
  <widget id="somewidget" exist="false"/>
</mask>
```