Università
della
Svizzera
italiana

**Facoltà
di scienze
informatiche**

Programming Fundamentals 2 - Smalltalk **2004 - 2005**

Prof. Michele Lanza Paolo Bonzini, Mircea Lungu and Romain Robbes

Hands on 3 : more on Seaside **16/03/2005 @ 10:30**

This hands-on session will tell you some more advanced concepts in Seaside, namely call/answer, forms, tasks. It consists of two relatively short exercises because you first have to synchronize with your partner regarding Assignment 2. In other words, if you have not completed Assignment 2 successfully, do not start this hands on before your partner has explained you how to complete it.

If nobody in your group has completed the assignment, reach at least item 3 in exercise 3 before proceeding with this hands on. In other words, your table should be rendered as several separate `WAPlayer` components, all of which are children of `WAPlayerPool`.

Exercise 1 - How forms work

The assignment required you to make the statistics of a player modifiable, using `request:` to have the user enter the new value. With this exercise, you will implement your own web form.

First of all, look at the existing Seaside forms to understand how they work. Look at `WAInputDialog` in particular.

So you decided to skip this? No, open your browser and look at how this component is implemented—it is a very small one, but it is also instructive.

The key method is of course `renderContentOn:`, which I'll copy down here for clarity.

```
html form: [
    html defaultAction: [self answer: value].
    html textInputWithValue: value callback: [:v | value := v].
    html space.
    html submitButtonWithText: self label.
]
```

A form is defined using a block, just like tables, table rows, and so on. The peculiar thing about forms, is that you can have *callbacks* defined within them. Here there are two, one for the default action and one for the textual input. When you press the form's submit button, Seaside will evaluate callbacks in this order:

- cancel callbacks; these are created when you use the HTML renderer's `cancelButtonWithAction:` method.
- value callbacks, such as the one-argument block `[:v | value := v]`.
- submit callbacks; these are created when you use the HTML renderer's `submitButtonWithAction:text:` method. Here, the submit button has no attached action, because a different method is used.
- the default callback, created by `defaultAction:` as in this case.

Forms are often invoked using `call:`, so that at any time a callback can use `answer:` to stop the evaluation of callbacks. Indeed, cancel callbacks are evaluated first so that subsequent callbacks may be stopped.

An example of using `call:` is given by `request:` itself. You can find it in `WAComponent`, under the *convenience* category. However, since it is a bit more complex than this, look at `inform:` and `confirm:` in the same place. You can see that you basically create your own component (with `new`) and pass it to the method:

```
inform: aString
    self call: (WAFormDialog new addMessage: aString)
```

Exercise 2 - Creating superclasses

Your task, now, is to *define a component for an editable player*. We want to create a superclass of `WAPlayer`, which will also be the superclass of `WAPlayerComponent`.

1. Create a `WAPlayerComponent` class. Do this by going to *WAPlayer* and changing the name of the class in the code pane to `WAPlayerComponent`. Then remove the instance variable names, and *Accept* the definition.
2. Make the class a *superclass* of `WAPlayer`. Go back to `WAPlayer` and change the superclass from `WAComponent` to `ProFund.WAPlayerComponent`.
3. You want to move the instance variable that holds the `FMPlayer` instance up to `WAEditablePlayer`. In the class definition for `WAPlayer`, highlight the name of the variable and pick the *Push up* refactoring (Edit → Refactoring → Push Up).
4. Similarly, push up the accessor methods for your instance variable. The refactoring is now under Methods → Refactoring → Push Up.
5. If you have an instance variable that points from the `WAPlayer` back to the `WAPlayerPool`, push it up as well.
6. Create the `WAEditablePlayer` subclass of `WAPlayerComponent`. While the two `renderContentOn:` methods for `WAPlayer` and `WAEditablePlayer` will be different, they will be overall similar.
7. Start by changing `WAPlayer`, removing your previous attempt at making statistics modifiable. Then copy the methods that take care of rendering into your new class.
8. Go to `WAEditablePlayer` and remove anything related to folded view. This should get rid of all references to undeclared variables (that is, no more red underlines in the methods).

Exercise 3 - Create the editable player component

1. Go to `WAPlayerPool` and (temporarily) make it create instances of `WAEditablePlayer` instead of `WAPlayer`. Fix any bugs you have introduced, then go on.
2. Using the information you found out in exercise 1, complete the functionality of `WAEditablePlayer`. Use callbacks to store data in the `FMPlayer`, like you used to do with the return value of `#request:`.

   If necessary, you can use the default action to update the list of players shown by the `WAPlayerPool` (see the penultimate item of the assignment). Otherwise, just use an empty block for the default action.

Now, your football manager will be composed exclusively of `WAEditablePlayer` components, but you should have at least restored the functionality of the player.

Exercise 4 - Use call/answer

This part is actually the easiest.

1. First of all, switch `WAPlayerPool` back to creating instances of `WAPlayer`.
2. Add an anchor to the side of the fold/unfold button in `WAPlayer`, named *Modify*. Leave the action empty for now. If you do not have a fold/unfold button, put it to the right of the name.
3. For the anchor's action, use `call:` to switch to a `WAEditablePlayer`. Use the accessor methods so that the instance variables of the `WAEditablePlayer` are equal to those of the calling `WAPlayer`.
4. Go to the `WAEditablePlayer`'s default action, and append a call to `self answer` to get back to the `WAPlayer`.

Note that in our case, we already stored the value in the `FMPlayer` during the callbacks, so we need not use the one-argument `#answer:` method that you saw in exercise 1.