We.eXplained

Concepts and Internals

Michael Neumann

mneumann@ntecs.de

What the hell...

Decorations

Components

Snapshots

Callbacks

Continuations

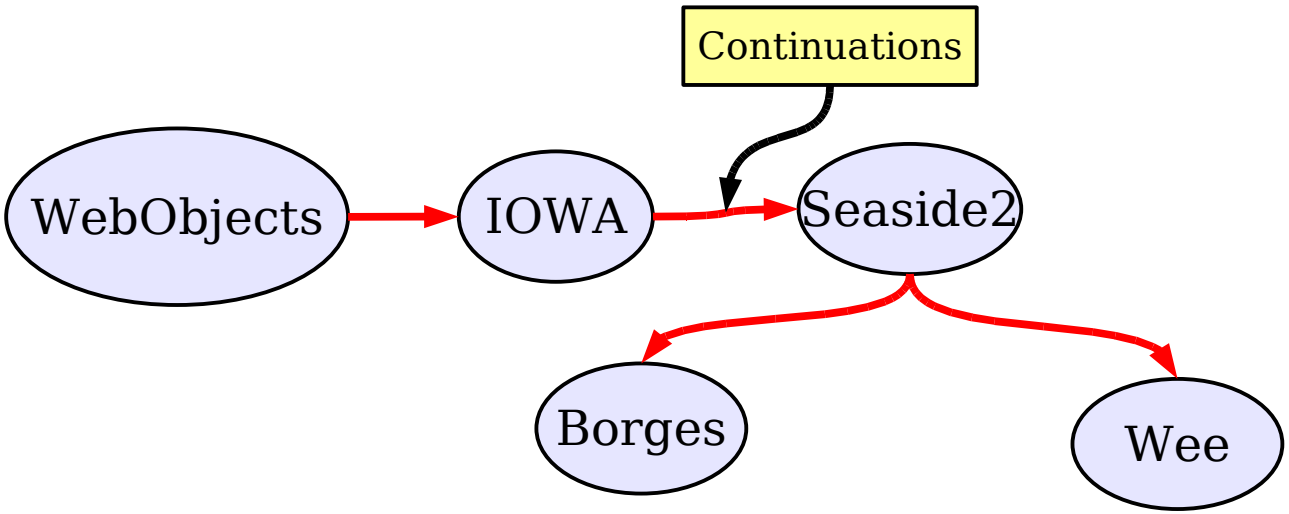
Backtracking

Stateful

What the hell...

Introduction

- What is Wee?
  - A **component oriented**, stateful web engine, largely inspired by Seaside2 and Avi Bryant.



  - See the demo.

Hello W(ee)orld

```
require 'wee'

class HelloWorld < Wee::Component
  def render; r.text „Hello World“ end
end

Wee.run(HelloWorld)
# => http://localhost:2000/app
```

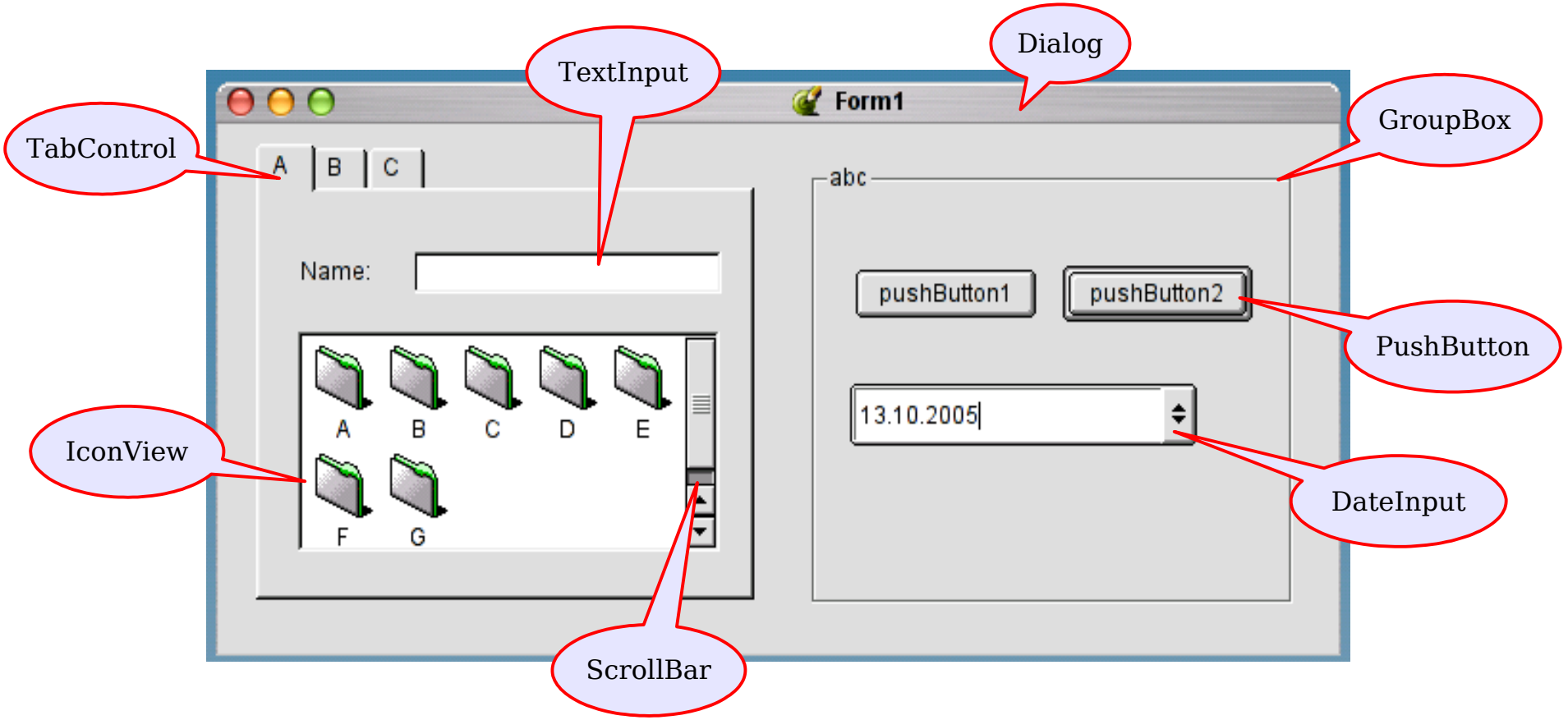
```
require 'wee'

class HelloWorld < Wee::Component
  def render
    r.h1.onclick_callback { @v = !@v }.with {
      r.text(@v ? "Bye bye world" : "Hello world")
    }
  end
end

Wee.run(HelloWorld)
```

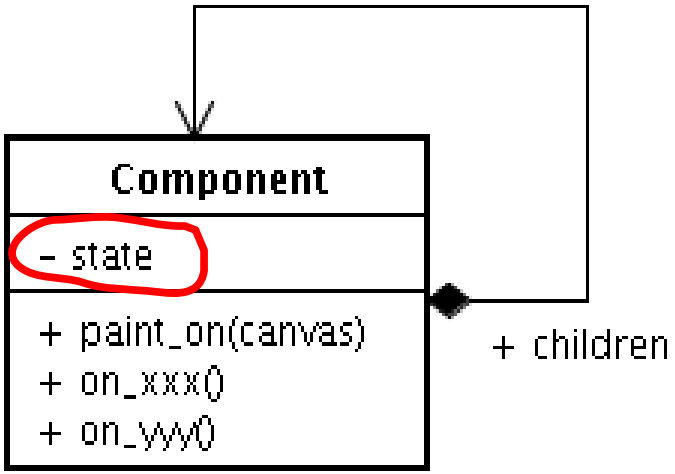
Analysing GUIs

- Modern GUIs are constructed out of individual **components** (or „widgets“).



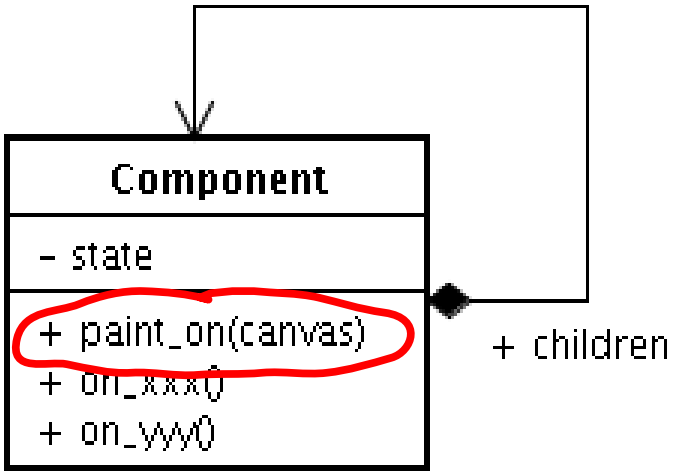
Analysing GUIs - State

- Each component stores it's own current visual UI **state**, e.g.
  - selected item
  - collapse info of a TreeView



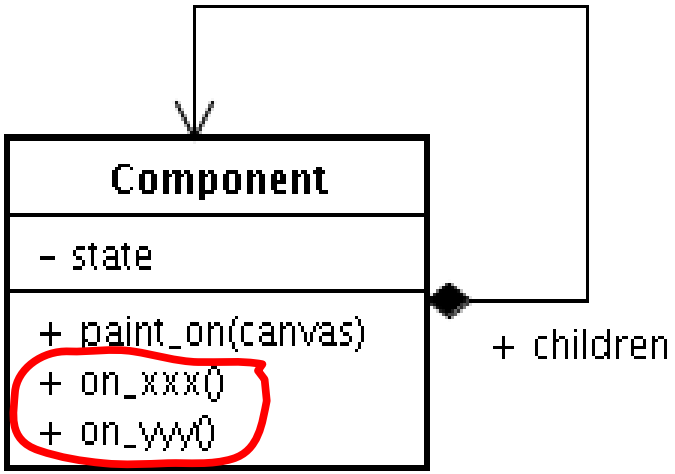
Analysing GUIs - Display

- A component can **paint** itself onto something (called a canvas here).



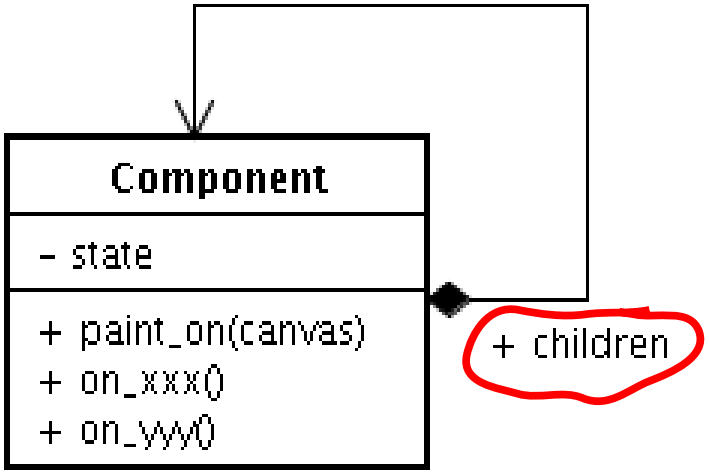
Analysing GUIs - Callbacks

- Interaction from outsite can solely happen through explicitly exposed **callbacks** which are bound to specific events. They **modify state**!



Analysing GUIs - Composition

- **Composition** is used to build more complex components.
- Parents...
  - ... can intercept events sent to their children. (?)
  - ... are responsible to paint their children. (?)



UIs in Wee

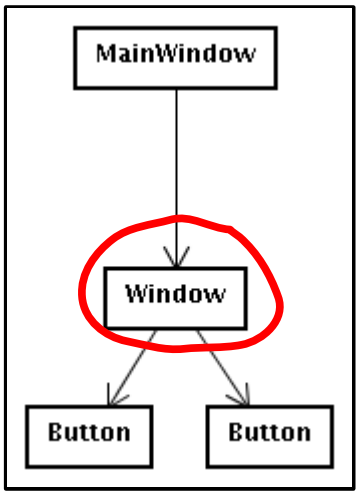
- Except **paint** is now called **render**, and the output is text/html, everything applies ;-)
- An application in Wee is constructed out of commponents. Thus:
  - You can reuse components!
  - BUT:

Reusable Components

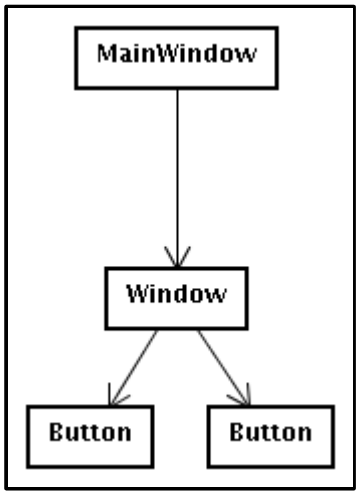
- **More a dream than reality!**
  - It's hard to build reusable components.
- Customization through:
  - Subclassing (overwrite methods)
  - CSS

A Compositional Problem

- Wrap Window inside a GroupBox!?
  - (Or draw a a border around the Window)

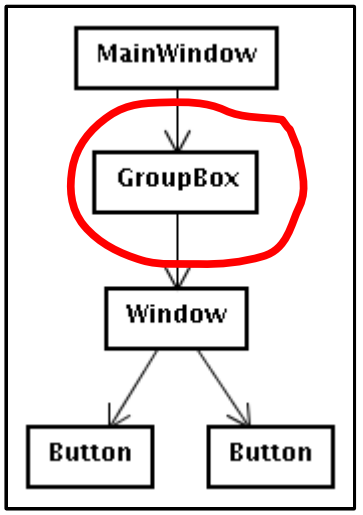


„Simple“ Solution



Initial Setting

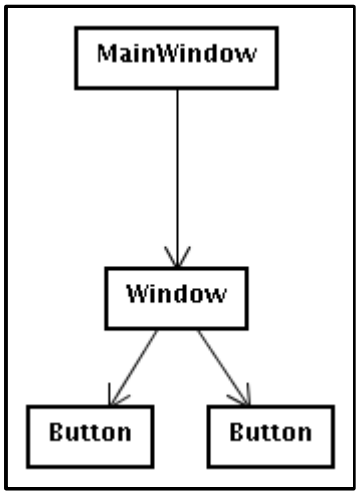
Modify Tree



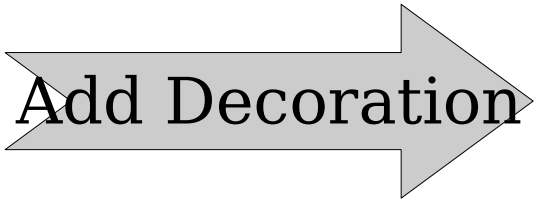
No Decorations

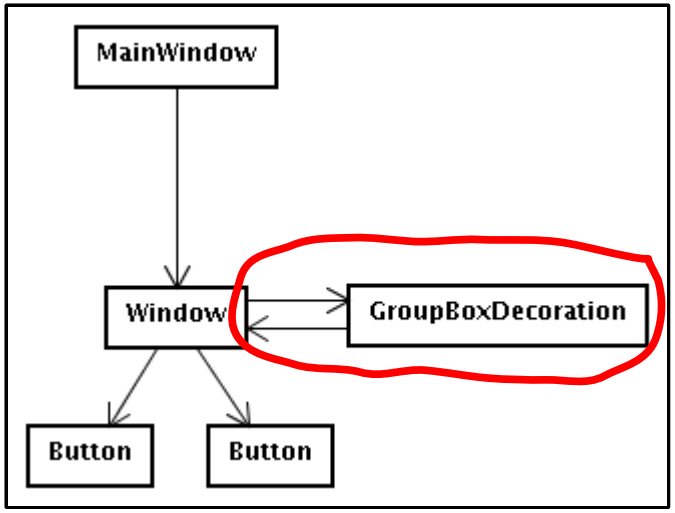
BUT: What if the parent is unknown?

The „decorative“ Solution



Initial Setting





With Decorations

Advantage: No need to modify the tree!

Decorations

- Changing the look and/or behaviour of components **without** modifying the compositional relation.
- Examples:
  - draw header/footer around them
  - intercept callback processing (e.g. LoginDecoration)

Uses of Decorations

- WrapperDecoration, PageDecoration, FormDecoration
- LoginDecoration
- Delegate, AnswerDecoration

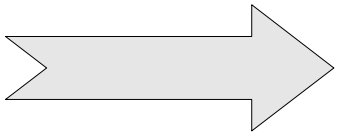
WrapperDecoration

```
class HeaderFooter < Wee::WrapperDecoration
  def render_wrapper
    r.text „header“
    yield # render the component
    r.text „footer“
  end
end
c = MyComponent.new.add_decoration(HeaderFooter.new)
```

Renders a header and footer around a component.

PageDecoration

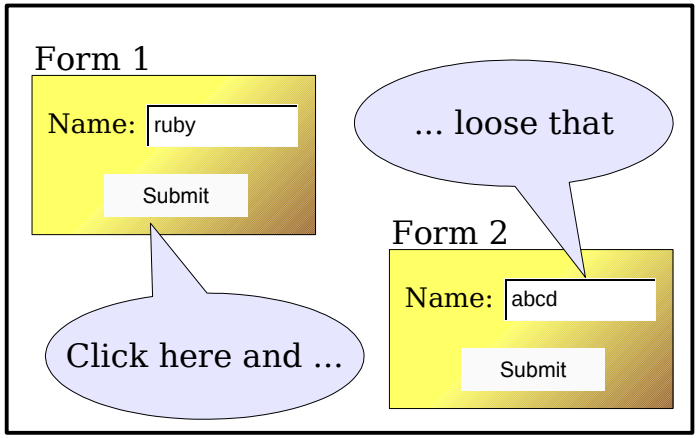
```
class Wee:PageDecoration < Wee::WrapperDecoration
  def initialize(title='')
    @title = title
    super()
  end
  def global?() true end
private
  def render_wrapper
    r.page.title(@title).with { yield }
  end
end
```



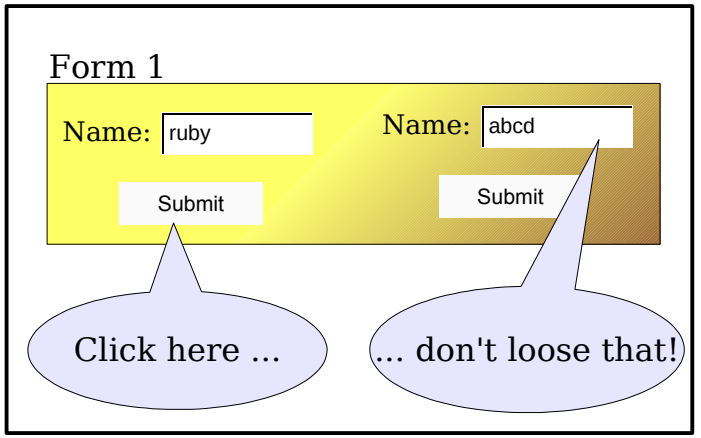
```
<html>
  <head>
    <title>...</title>
  </head>
  <body>
    ...
  </body>
</html>
```

FormDecoration

- Simply renders a <form> tag around a component.
- Why? By wrapping a set of components inside of **one** form tag, the entered values will not get lost upon submit!



Multiple Form Tags

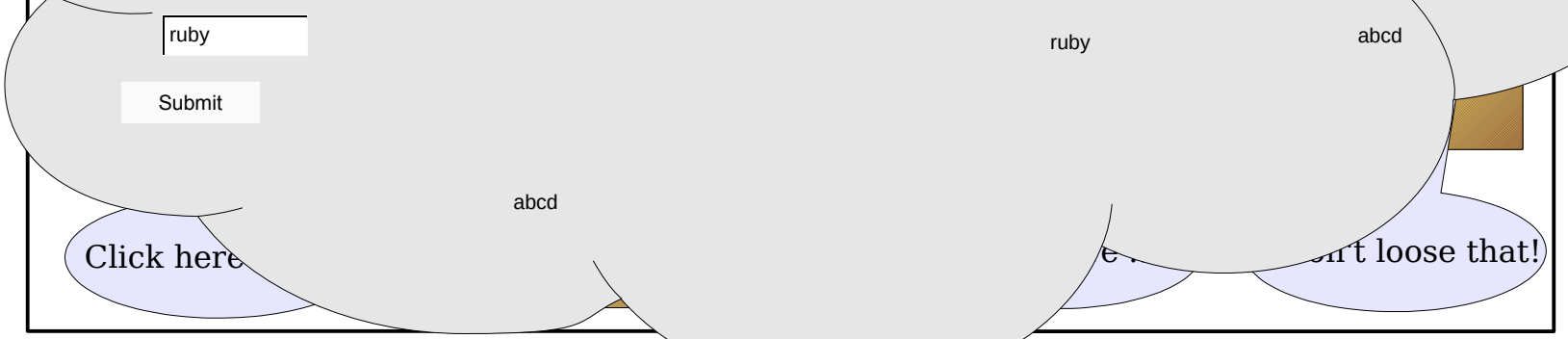


One Form Tag

FormDecoration

- Sim renders a <form> tag around a com nt
- Wh s ins

Today: Use AJAX!?



Multiple Form Tags

One Form Tag

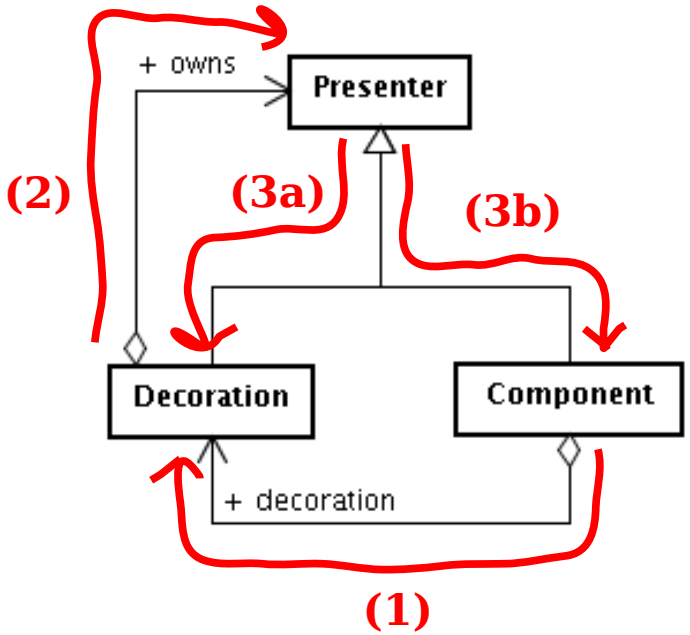
LoginDecoration

- Redirect all requests to a Login page unless logged_in?
- Else: Pass all requests through
- See: login_page.rb

Delegate decoration

- Like LoginDecoration, but unconditionally.
- Introduced later (together with AnswerDecoration) when we look at the **call/answer** mechanism.

Implementation of Decorations



(1) The component will pass control to it's first decoration in the chain.

(2) A decoration owns all „following“ decorations. It might proceed or stop here.

(3a) The next decoration is in control. Proceed with (2).

(3b) We reached „self“ (the initial component). We're ready!

Note that Decoration and Component share a similar interface.
That's why 'decoration' can point back to self.

call/answer mechanism

- Think of a „**procedure call stack**“ (call/return).
- Demo

```
class Counter < Wee::Component
  def initialize; @cnt = 0; super end
  def render
    r.anchor.callback(:dec).with('--')
    r.text „ #{ @cnt } “
    r.anchor.callback(:inc).with('++')
  end
  def dec
    if @cnt == 0
      call MessageBox.new('Go Negative?'),
           proc {|cond| @cnt -= 1 if cond }
    else
      @cnt -= 1
    end
  end
  def inc; @cnt += 1 end
end
```

```
class MessageBox < Wee::Component
  def initialize(title)
    @title = title; super()
  end
  def render
    r.form do
      r.h1 @title
      r.submit_button.value('YES').
        callback(:answer, true)
      r.submit_button.value('NO').
        callback(:answer, false)
    end
  end
end
```

call/answer (cont')

- Components can replace themself temporaryly with another component (**call**), until the called component gives control back to the calling component (**answer**).

Implementation of call

- Simply add a Delegate decoration to the calling component:

```
def call(component, on_answer=nil)
  delegate = Delegate.new(delegate_to=component)
  self.add_decoration(delegate)
  ...
```

- But that's not enough. We need an AnswerDecoration as well.

```
  ...
  answer = AnswerDecoration.new
  answer.on_answer = on_answer
  component.add_decoration(answer)
  ...
```

Implementation of call (2)

- Why an AnswerDecoration?
  - To be able to use a component calling **answer** like any other component.
  - „return without call“!

Implementation of call (3)

- Still not enough!

```
  ...
  throw :wee_abort_callback_processing
end
```

- Only slightly different in the „continuation“ case, but **usage** is much easier:

```
class Counter < Wee::Component
  ...
  def dec
    @cnt -= 1 if @cnt > 0 or call MessageBox.new('Go Negative?')
  end
  ...
end
```

Implementation of answer

- Simply removes the Delegate decoration from the calling component and invokes the on_answer callback.

URLs in Wee

- You'll never have to touch them manually.

  http:// hostname / mount-point / info-part / separator / session-id / page-id ? callbacks
  http://localhost/demo/arbitrary/info/___/b2fb...7cb/1

- Page-Id will increase after every „action“ (see Backtracking).

A Wee Cycle

- Imagine following code:

```
def render
  r.form do
    r.text_input.value(@name).callback {|@name|}
    r.submit_button.value(„OK“).callback { p @name }
  end
end
```

- This will generate HTML like this:

```
<form method=“GET“ action=“/..../___/sid/120“>
  <input type=“text“ value=““ name=“1“ />
  <input type=“submit“ value=“OK“ name=“2“ />
</form>
```

A Wee Cycle (cont')

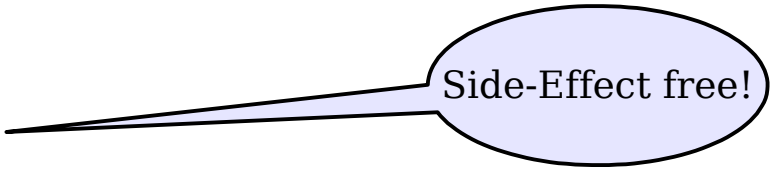
- Upon form submission the following happens:
  - The Page-id is extracted from the submitted URL.
    - http://...../___/sid/120?1=blah&2=OK
  - Lookup the page with that id from the PageStore. A page is just the snapshot of a certain state of the components tree + registered callbacks.
  - Restore the components tree to that state.
  - Invoke the supplied callbacks.
    - All input callbacks (e.g. text-fields) before the final action callback (submit-button, anchor).
  - Generate new Page-Id. Take a snapshot of the whole components tree and store it under that id.
  - Redirect to the new Page-id excluding the callbacks:
    - http://...../___/sid/121

Modifies State!

```
@name = „blah"
p @name
```

A Wee Cycle (cont')

- The client follows the redirect (http://...../___/sid/121):
  - Lookup page 121 and apply it's state to the components tree.
  - Render the components tree.



    - Callbacks are registered and stored in the page object of id 121.
    - All generated URLs include the current page id (121), or in other words, the context in which the callbacks should be invoked.

```
def render
  r.form do
    r.text_input.value(@name).callback {|@name|}
    r.submit_button.value(„OK“).callback { p @name }
  end
end
```

Backtracking

- „Undo/Redo“-Facility.
- You can use the back-button again!
- See the demo.

Backtracking - Implementation

- Simply take a snapshot of all the objects that are of interest (e.g. the current value of a counter).
- Later, restore them!
- Page-id's to make unique URLs and to reference the snapshots.

Continuations

- Try the same without continuations... you need a state machine!

```
  ...
  def action

    loop do
      if (call MessageBox.new('2 + 3 = 5?')) and
         (call MessageBox.new('3 + 4 = 7?')) then
        call TextBox.new('Genius!')
        break
      else
        call TextBox.new('Please try again!')
      end
    end

  end
  ...
end
```

- **But:** In some (rare?) situations, continuations leak memory! Be careful.
- **And**: You need one thread per session!

Is Wee for me?

- Maybe if...
  - your application is **very** dynamic.
  - you can easily recognize and extract parts of the page as components (it has a „regular“ structure).
  - you need to operate on very complex in-memory objects.
  - you don't like Javascript ;-)
  - your application is more complex than just CRUD.
  - bookmarkable URLs are not a requirement.
  - you think backtracking is a nice thing.
  - you really want to use continuations.
  - you hire me to continue Wee's development ;-)

Reasons why not

- No community. Development stalled.
- URLs are not bookmarkable (this is just impossible due to it's dynamic nature).
- It's stateful.
- Maybe too dynamic for you ;-)
- It's easy to introduce inconsistencies when using Og (ORM) due to object caching. Solution: Disable caching or manually reload the domain objects.

Questions?

Thank you!

Resources

- Wee: http://rubyforge.org/projects/wee
- Seaside2: http://www.seaside.st/
- Borges: http://borges.rubyforge.org/
- IOWA: http://enigo.com/projects/iowa/