



Control Flow

Lukas Renggli
www.lukas-renggli.ch

Outline

- ★ Define Flow
- ★ Convenience Methods
- ★ Call and Answer
- ★ Transactions
- ★ Example

Define Flow

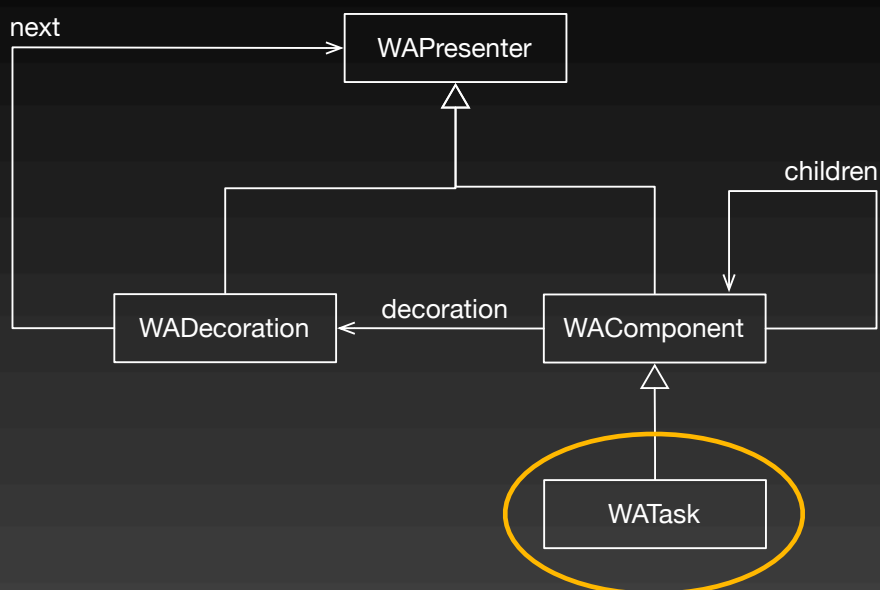
Control Flow

- ★ Create a subclass of `WATask`
 - ★ Implement the method `#go`
 - ★ Split the method `#go` into smaller parts to ensure readability
- ★ Create an callback handler (will be discussed later on)

Tasks

- ★ Tasks are a special kind of component having no visual representation by itself.
- ★ Tasks define a logical flow (or part of it) within the method `#go`.
- ★ Tasks call other components to delegate the generation of the output.

Tasks



Convenience Methods

Convenience Methods

`#inform:`

`#confirm:`

`#request:`

`#request:default:`

`#request:label:`

`#request:label:default:`

`#chooseFrom:`

`#chooseFrom:caption:`

#inform:

self inform: 'Hello World'

Hello World

Ok

#confirm:

bool := self confirm: 'Are you sure?'

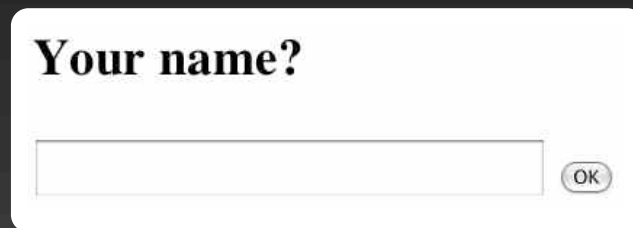
Are you sure?

Yes

No

#request:

```
string := self request: 'Your name?'
```



Your name?

#request:label:default:

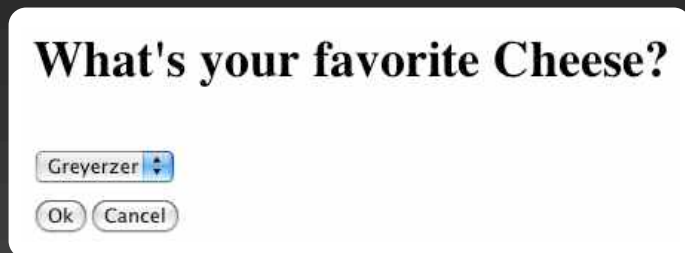
```
string := self  
    request: 'Your name?'  
    label: 'Accept'  
    default: 'Lukas Renggli'
```



Your name?

#chooseFrom:caption:

```
string := self  
  chooseFrom: #('Greyerzer' 'Tilsiter' 'Sbrinz')  
  caption: 'What's your favorite Cheese?'
```



Call and Answer

Call and Answer


- ★ #call: aComponent
 - ★ Transfer control to aComponent
 - ★ aComponent will be given control
- ★ #answer: anObject
 - ★ anObject will be returned from #call:
 - ★ Receiving component will be removed

x := A call: B



B answer: 



x := 



Transactions

Transactions

- ★ Sometimes it is required to prevent the user from going back within a flow.
- ★ Calling `#isolate`: treats the flow defined in the block as a transaction.
- ★ Users are able to move back and forth within the transaction, but once completed they cannot go back anymore.

#isolate:

```
self isolate: [  
    self doShopping.  
    self collectPaymentInfo ].  
self showConfirmation.
```

Example

Game Example

UserNumberGuesser>>go

```
| guess number |
guess := -1.
number := 100 atRandom.
self inform: 'I am thinking of a number between 1 and 100.'.
[ guess = number ] whileFalse: [
    guess := (self request: 'What is your guess?') asNumber.
    guess < number
        ifTrue: [ self inform: 'The number I am thinking of is bigger.' ].
    guess > number
        ifTrue: [ self inform: 'The number I am thinking of is smaller.' ].
    self inform: 'You got it!'
```

Transaction Example

UserNumberGuesser>>go

```
| guess number |
guess := -1. number := 100 atRandom.
self inform: 'I am thinking of a number between 1 and 100.'.
[ guess = number ] whileFalse: [
    self isolate: [
        guess := (self request: 'What is your guess?') asNumber.
        guess < number
            ifTrue: [ self inform: 'The number I am thinking of is bigger.' ].
        guess > number
            ifTrue: [ self inform: 'The number I am thinking of is smaller.' ] ] ].
    self inform: 'You got it!'
```

Register Starting Point

- ★ To register your task (or component) as a starting point of your application:
 - ★ Create a method `#canBeRoot` returning true on the class-side of your component.
 - ★ Register the component using the Seaside configuration interface.

```
UserNumberGuesser class>>canBeRoot  
^ true
```

Register Starting

- ★ To register your task (or component) as a starting point of your application:
 - ★ Create an `#initialize` method on the class-side of that component.
 - ★ Evaluate the initialize method.

```
UserNumberGuesser class>>initialize  
self registerAsApplication: 'ung'
```

Summary

- ★ Define application flow within the method `#go` of your subclass of `WATask`.
- ★ Use `#call:` or one of the convenience messages to delegate the rendering to other components.
- ★ Use `#isolate:` to control the use of the back button.