

VisualWorks applications

Views

Application Model

Domain Model

Domain Model

- Employee
- PayrollSystem

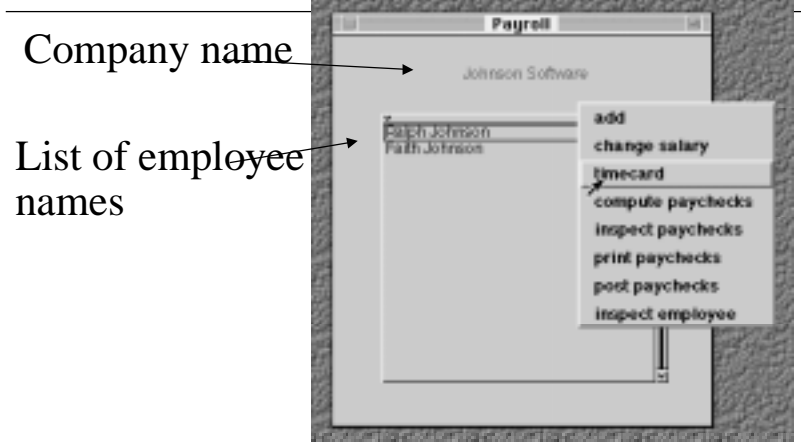
ApplicationModel

- PayrollSystemInterface

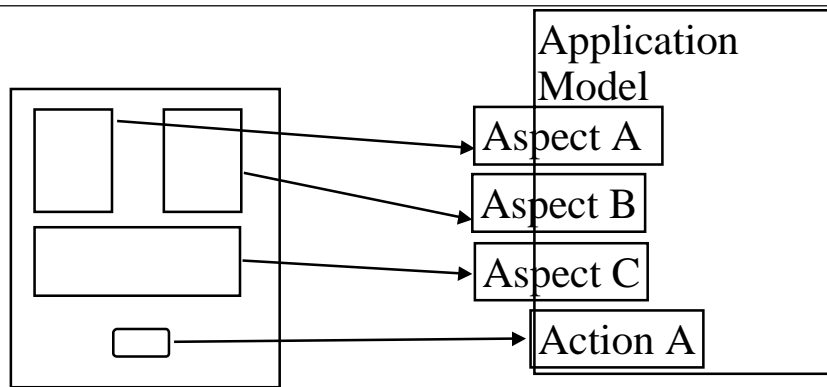
Views

- ComposedTextView

Payroll Interface



Aspects and Actions



Widget is a dependent of the aspect.

PayrollSystemInterface

Initialize a PayrollSystemInterface with the PayrollSystem it is the interface for.

payrollSystem: aPayrollSystem

payrollSystem := aPayrollSystem.

self companyNameHolder

value: payrollSystem name

ApplicationModel

Each VisualWorks application is a subclass of ApplicationModel.

Class will

- define window
- define aspects

You can override inherited methods, but most of the code is generated by GUI builder.

ApplicationModel

Make a separate ApplicationModel for each "tool".

Often customize initialize method.

Use #onChangeSend:to: to let model know when one of its aspects has changed.

Send #open to ApplicationModel class to create new applications.

Opening Application

ApplicationModel with no parameter can be created by sending #open to its class.

PayrollSystemInterface takes a PayrollSystem as a parameter.

So, use #openOn:, which takes an ApplicationModel as its parameter.

```
openOnPayroll: aPayrollSystem
  self openOn: (self new payrollSystem:
    aPayrollSystem)
```

Must also define #payrollSystem: as an instance method.

VisualWorks ApplicationModel

Each VisualWorks application is a subclass of ApplicationModel.

Methods tend to be either

- interface specification methods
- resource methods
- aspect methods
- action methods

```
windowSpec
  "UIPainter new openOnClass: self
  andSelector: #windowSpec"
  <resource: #canvas>
  ^#(#FullSpec
    #window:
      #(#WindowSpec #label: 'Payroll'
      #bounds: #(#Rectangle 1245 452 1544
      761 ) )
    ...
```

ApplicationModel

Provides hooks that subclasses can use to customize specs.

preBuildWith: aBuilder

build window

postBuildWith: aBuilder

open window

postOpenWith: aBuilder

ValueModel Variable

What should you call a variable that holds a ValueModel?

- Contents is important
- Use of a ValueModel is important.

Append "Holder" to the name. Add accessing methods to return the ValueModel and its value.

listHolder

^listHolder

list

^self listHolder value

list: aList

self listHolder value: aList

SelectionInList

Aspect of a list. Has holder for the list, holder for the selected object, and holder for index.

- list, list:, listHolder
- selection, selection:, selectionHolder
- selectionIndex, selectionIndex:, selectionIndexHolder

Other aspects

MultiSelectionInList - same as a SelectionInList, but can select a set of elements.

SelectionInTable - same as a SelectionInList, but for a 2-D table

Displaying Lists and Tables

List UI features

- select element and perform operation with it
- select more than one element

Table UI features

Displaying Lists

Widget is `ListView` or a `MultiSelectionListView`.

Aspect is `SelectionInList` or `MultiSelectionInList`

Aspect has `list`, `selection`, and `selectionIndex` (an integer).

`List` is a sequenceable collection, usually a `List`.

Facts about SelectionInLists

Canvas Tool produces aspects that contain empty lists.

Lists usually require more work with than input fields.

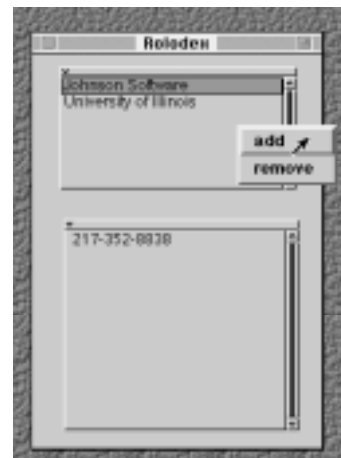
- Write methods to add and remove
- Write methods to get list from another object, if necessary.

Example: Rolodex

Simple rolodex: name and contents

List contains
RolodexEntries

RolodexEntry prints its
name, which causes list to
display correctly.



Top view is the list, bottom view is the contents of selected entry.

List menu: add, remove, rename

Text menu: accept, cancel

Problem

Text editor is not connected to list.
Selecting a new element from list has no effect on text editor.

Solution:

- Make ApplicationModel (Rolodex) depend on list.
- When list changes, Rolodex will change text editor.

Problem:

What if ApplicationModel has to depend on several lists?

How will #update: method know which has changed?

Solution:

- Use onChangeSend:to:

Using onChangeSend:to:

Register dependence with subject:

"send #listChanged to me when selection of entryList changes"

```
self entryListHolder selectionHolder  
onChangeSend: #listChanged to: self
```

Using onChangeSend:to:

Define what to do on change

listChanged

```
self entryListHolder selection isNil ifTrue:  
  [textHolder value: nil. ^self].
```

```
textHolder value: self entryListHolder  
  selection text
```

Problem:

- If you type in text and accept it, the RolodexEntry is never told.

Solution:

- Make ApplicationModel (Rolodex) depend on text field, so that it will know when you accept text.

Register dependency:

- self textHolder onChangeSend:
#textChanged to: self

Define what to do when text changes:

textChanged

- self entryListHolder selection text:
(textHolder value)

Decisions

Where to register dependency?

- In #initialize method ← This one
- In aspect method

If we register dependencies in #initialize method, should we initialize holders there, too?

Register Dependencies in Aspect?

textHolder

^textHolder isNil

ifTrue:

[textHolder := String new asValue.

self textHolder onChangeSend:

#textChanged to: self]

ifFalse: [textHolder]

Problem: Have to change computer generated code. Lots of different methods might be changed.

Solution: Avoid changing aspect methods.

Register dependences in #initialize

initialize

super initialize.

self entryListHolder selectionHolder

onChangeSend: #listChanged to: self.

self textHolder onChangeSend:

#textChanged to: self

Tables

TableInterface describes grid lines, widths, formats, labels, and contains SelectionInTable

SelectionInTable is the holder for TableAdaptor or TwoDList

TableAdaptor or TwoDList is a two-dimensional collection.

Table of UFO Sightings

initialize

| list |

super initialize.

list := TwoDList on: #('Vulcans' 188 173 192
'Romulans' 26 26 452) copy

columns: 4

rows: 2.

sightingsTable := SelectionInTable with: list.

"Create a table interface and load it with
the sightings."

tableInterface := TableInterface new
selectionInTable: sightingsTable.

tableInterface columnWidths: #(100 40).

Labels

tableInterface

- columnLabelsArray: #('Visiting Race' '1992' '1993' '1994');
- rowLabelsArray: #(1 2);
- rowLabelsWidth: 20.

...

(continued)

tableInterface

columnFormats: #(#left #right #right #right);
columnLabelsFormats: #(#left #right #right #right);
rowLabelsFormat: #right.

Table for ValueWithHistory

Suppose “history” is a variable containing a ValueWithHistory. Then you can make a TwoDList with two columns, one of which has the dates of changes and the other has the new value, by

ValueWithHistory

table := TwoDList columns: 2 rows:
(history datesOfChanges size).

1 to: history datesOfChanges size
do: [:eachIndex | | eachDate |
...]

Must write ValueWithHistory
datesOfChanges to return "dates"

(continued)

eachDate := history datesOfChanges at:
eachIndex.

table at: (1@eachIndex) put: eachDate.

table at: (2@eachIndex) put: (history at:
eachDate)

Summary

Framework:

- reuse of large-scale design
- describes how to break problem into objects
- consists of classes (usually abstract) and the way they interact
- not just static interface, but invariants and abstract algorithms