Building Windows in VisualAge Smalltalk

The Presenters
- Ginny Ghezzo: Project Manager for IBM VisualAge Smalltalk and Level 3 support for the Application Builder
- John O’Keefe: Technical Lead for IBM VisualAge Smalltalk

A Little Review

Lecture 7: Analysis and Design
- Iterative refinement
- Understand the application requirements
- Decompose the application into object classes
- Determine the interactions and dependencies between objects

Lecture 8: Building the Model with VisualAge Smalltalk
- Transition from design to implementation
- Delegation and encapsulation
- Variables and where they fit in
- Representation of part attributes
Architectural Background

Model, View, Controller

[Liu, Ch. 11] In the beginning, Smalltalk supported a user-interface architecture called MVC. MVC was an acronym taken from the names of three key classes, Model, View and Controller.

Models are objects that represent the components of an application that perform information processing in the problem domain.

Models should represent “real world” entities:
- physical entities like a valve in a control system, or
- conceptual entities like a department in an office, or a contract between two businesses.

Views are objects that display some aspect of the model. They are the output mechanism for the models.

You could have a view that represents:
- the position of the valve or the temperature of a chemical vat (graphical)
- cells in a spreadsheet (tabular)
- the terms and conditions of a contract (textual)
- product installation instructions (video or audio)

Controllers are objects that control how user actions are interpreted. They are the input mechanism for the views and models.

For example, the interpretation of a double-click on a temperature gauge would be handled by the controller notifying the model in a way it agrees to respond to.

MVCs came in a triad, with communication between the components occurring as follows:
The primary communication path is from Model to View. Whenever the state of a model changed, the view would need to display itself differently.

For instance, consider a View that represents a car’s engine temperature.

If the engine temperature goes up (say, because a fan belt breaks) the gauge showing the temperature will need to redisplay its new value in response to that change.

It accomplished this feat through the use of a protocol called change/update. Change/update consists of two messages, one implemented in the abstract class Model, and another implemented in the abstract class View, and reimplemented in each subclass.

Let’s say we have a subclass of Model called Engine that represents our car engine. Engine will have instance variables for engine temperature, fuel level, RPM’s, etc.

To notify the View that it needs to change, the View must be listed in a special list of objects that care about changes in the Model.

This list is called the list of Dependents of the Model. Once a View has been added to the Dependents list of a Model, it will receive updates about what has changed. The process works like this.

- When a View is created, it adds itself to its Model’s list of Dependents by implementing code similar to the following:
anEngine addDependent: self

- The implementation of the message `addDependent:` in the superclass Model would look something like the following:

```smalltalk
addDependent: anObject

"Add @anObject as one of the receiver’s dependents."

| dependents |

dependents := Dependents at: self ifAbsentPut: [OrderedCollection new].

^dependents add: anObject
```

- The `Engine>>temperature:` message that sets the temperature of the engine contains the following code. The changed: message takes a single parameter, called an aspect. This identifies the particular piece of the Model that a View might be interested in.

```smalltalk
temperature: anInteger

"Change your temperature to @anInteger and notify your dependents that you changed"

temperature := anInteger.
self changed: #temperature.
```

Thus, the Model does not need to know what Views are interested in each of its aspects. It just needs to know that it has a list of objects that are dependent on it. Whenever it receives a changed: message, it sends an update: message (for the same aspect) to its dependents.

- The implementation of the message `changed:` in the superclass Model would look something like the following:
changed: anAspect

"Inform your dependents that your aspect has changed"

self dependents do: [:dependent | dependent update: anAspect].

- Now, each subclass of View would have to override the update: message and implement it differently to look for the particular aspect that they care about.

update: anAspect

"If anAspect is #temperature, redisplay. Otherwise ignore it."

anAspect == #temperature
  ifTrue: [self redisplay].

**MVC component relationships:**

- Views request data from the Model and displays them
- Controllers change the state of the View and/or Model in response to user actions.

The Controller can send messages to the View.

If you get a double-click in a cell, the cell should be highlighted. The Model probably doesn’t need to get involved if the cell just needs to
highlight — it’s just something having to do with the visual representation.

Both the View and the Controller can send messages to the Model
• the Controller can tell the Model, “I’ve received a button click here.”
• The View could tell the Model, “I need some data from you. I need to know where to set the temperature gauge, so what is the temperature?”

However, the only way for the Model to send messages to the View or Controller is indirectly by sending changed: messages to itself.
• Why?

If the Model had knowledge of the state (or even the existence) of the View or the Controller, it would be very difficult to add a new type of View or Controller.
• Does a Model need to know that a View has three temperature gauges on it? Probably not.

All the model has to know is that when it receives the change: message, it needs to send the update: message to all of its dependents.

Some of the views may care about the argument temperature and some may not.

In general, the update: method will check to see if it is interested in the aspect it is being sent since all Views will be notified whenever any aspect changes. While this may seem inefficient, it is in fact very useful, because it decouples the Model from having to have any knowledge at all about its Views, thus reinforcing information hiding.

Now, this whole process worked fine for the first few versions of Smalltalk, with a few changes to make things easier to write. So why isn’t it generally used any longer?
• Because it had major flaws
Strengths of MVC
- The Model is decoupled from any knowledge of Views or Controllers that are associated with the Model.
- The developer can focus on the job at hand:
  - the Model can be efficient and elegant
  - the Views can be inventive and creative
  - there can be many Views on the same Model.

Weaknesses of MVC
- The View/Controller distinction is difficult to maintain
  - Controller objects have visual aspects
  - Drag and drop appearance has complex relationship to both model and view.
- Every View receives every update: message regardless of its interest in the aspect
  - Full refresh on each change is inefficient
- Validating and checking constraints force Model logic up to the View or Controller.

Model, View

So what’s changed in VisualAge Smalltalk?

- The functions of the Controller are merged into the View. The View handles the input and the output.
- The list of Dependents is enhanced to become a list of EventDependents
  - Generalized by moving protocol to Object
- Interest is registered in particular event – addDependent: replaced by abtWhen:perform:

```smalltalk
anEngine
    abtWhen: #temperature
    perform: (DirectedMessage
        receiver: self
        selector: #updateTemperature).
```
• When updating the Model, changed: replaced by signalEvent:

```
  temperature: anInteger

  "Change your temperature to @anInteger and notify your dependents that you changed"

  temperature := anInteger.
  self signalEvent: #temperature.
```

• Send of generic update: message replaced by send of a DirectedMessage

```
  signalEvent: anEvent

  "Get the list of messages that have been registered as dependent on the primitive event anEvent and cause each one to be sent without replacing any arguments. Return the receiver."

  | msgs |

  (msgs := self abrPartiesInterestedIn: anEvent) == nil ifFalse: [ msgs abtIsDependentsCollection ifTrue: [msgs signalEvent] ifFalse: [msgs do: [:msg | msg == nil ifFalse: [msg abrSend]]]]
```

• No general solution to problem of validation logic in the View
Underlying Architecture of GUI

Motif

Motif is the industry standard graphical user interface defined by IEEE 1295 specification.

- Motif provides portability across platforms.
- The core components of the Motif technology include an extensible user interface toolkit; a stable application-programming interface, a user interface language, and a window manager.

A Widget is a user interface component in Motif.
Examples: Label, Combo Box, Shell, …

A graphical user interface is built by creating a tree of widgets. Every widget, except the topmost, has a parent widget.

- Each parent is responsible for sizing and positioning its children.
- A parent-child relationship is not the same as the class-subclass relationship.

The life of a Widget

1. Create: The widget is created, named and given a parent. It is not displayed on the screen.
2. Manage: The widget’s size and position are specified. If not managed neither it nor its children will participate in geometry management.
3. Map: The widget is mapped to specify that it should be displayed when realized. If not mapped it will stay invisible.
4. Realize: The widget and all children are fully instantiated and made visible.
5. Destroy: The widget is removed from the display and memory is released. In addition all of the widgets children are destroyed.

Common Widgets

Common Widgets is VisualAge Smalltalk’s implementation of the Motif widget data structures
- Examples: CwLabel, CwComboBox, CwShell, …
- See the Smalltalk class CwWidget
- For documentation see the Programmer’s Reference in the Information Center.

All Common Widgets are subclasses of one of the following:
- CwShell: A topmost widget. A shell can only have one child. Example: CwTopLevelShell
- CwComposite: A widget that has zero or more children. Example: CwMainWindow
- CwPrimitive: The simplest building blocks. A primitive has no children. Example: CwLabel

Widgets are configured and controlled by resources and functions
- **Resources** define the behavior and appearance. Example: width, backgroundColor, …
- **Functions** are messages telling the widget to do something. Example: setString,
Program Responses
An *event* is a mechanism that notifies an application when a user performs an action. These are low-level interactions.
- Examples: Mouse pointer motion, button presses

A *callback* is a mechanism that notifies an application when a higher-level action is performed on a widget.
- Examples: Widget was destroyed, Pushbutton pressed
How to Build a Window in the Composition Editor
VisualAge Organizer

Left pane shows all applications.
Right pane shows classes contained or extended in that application.

To create a new application and a new part use the Quick Start.,
Tools-> Quick Start
Or to create a new part (class), with an application selected, Parts-> New -> Parts...

This will automatically create a new part called NCSUVideoStoreCheckIn that is a subclass of AbtAppBldrView and will open the Composition Editor.
Composition Editor

- **Free-Form Surface**: Large canvas that parts can be place on.

- **Parts Palette**: Some important available parts that can be used on the free-form surface
  - **Categories**: The left column is organized groups of parts.
  - **Parts**: The right column lists the parts that can be placed on the free-form surface.

- **Tool Bar**: Various tools used to test, layout and manipulate the parts on the free-form surface.

- **Properties Sheet**: Dialog to update the attributes of the part.
How to hook up the View to the Model

At this point, having created our View class and generated its methods, we can start our interface and see if it looks the way we intended. This is done by pressing the “Test” button on the Composition Editor toolbar. The window will open, the text field will accept characters, and the button will depress, but nothing will happen beyond that.

This is because we haven’t yet connected our View class to the domain model classes we built in the previous lectures. That is the next phase of building GUIs.

Let’s take a look back at one of the design diagrams we built in the previous lecture.
This object-message diagram shows the flow of messages in the process of returning a tape before it is due.

In the previous lectures we avoided getting into the class we were calling “Rental UI” because we were not ready to talk about window coding. Now we are ready to finish that part of the design and implementation.

We can see that this portion of the Rental UI is actually carried out by the class we have been constructing, the VaVCVideoCheckinView. We could replace the “Rental UI” in the above diagram with this class, and see the messages and objects that our VaVCVideoCheckinView needs to know about. The message that will tie the View and the Model together is the message called when we press the button on the UI. That is the user’s signal to begin the process of returning a tape.

In the diagram above, the first step is obtaining an instance of VaVCVideoTape from the class VaVCVideoTape for a particular ID number. In our example, how would we find the video tape corresponding to the number the user typed in?

```smalltalk
findTape: aTapeId
    "Answer either aVaVCVideoTape identified by @aTapeId or anAbtError if the tape doesn't exist."
```

| tape |
tape := VaVCVideoTape lookupTapeForId: aTapeId.
tape isNil ifTrue: [ 
tape := AbtError errorText: ('Tape "%1" is not currently rented' bindWith: aTapeId)].
^tape

Show working application
- Add action-to-script connection
- Add error popup
- Add Variable part representing an instance of domain model

Relating View Internals to GUI Architecture

abtBuildInternals
"** Do not modify or delete ** See: AbtAppBldrPart
class>>#about_abtBuildInternals"

| gui window pushButton1 pushButton2 rentalNumber label1 aVideoTape messagePrompter1 conn0 conn1 conn2 conn4 conn5 conn6 conn16 |
gui := self class abtSeparatedConstants.
window := AbtShellView abtCreatePart: 'Window'
parent: nil owner: self .
rentalNumber := AbtTextView abtCreatePart: 'RentalNumber' parent: window.
label1 := AbtLabelView abtCreatePart: 'Label1' parent: window.
 partClass: VaVCVideoTape.
messagePrompter1 := AbtMessagePrompter abtCreatePart: 'Message Prompter1' parent: self
 parentVisual owner: self.
self
 primaryPart: window.
window
 framingSpec: (AbtViewAttachmentConstraint new
 leftEdge: (AbtEdgeConstant new offset: 31);
 rightEdge: (AbtEdgeConstant new offset: 426);
 topEdge: (AbtEdgeConstant new offset: 17);
 bottomEdge: (AbtEdgeConstant new offset: 209));
 title: (gui at: 1) " 'Video Check-in' " .
pushButton1
 object: (gui at: 2) " 'Check in tape' " ;
 framingSpec: (AbtViewAttachmentConstraint new
 leftEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHFORM; offset: 77);
 rightEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHNONE);
 topEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHFORM; offset: 119);
 bottomEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHNONE)).
pushButton2
 object: (gui at: 3) " 'Cancel' " ;
 framingSpec: (AbtViewAttachmentConstraint new
 leftEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHFORM; offset: 251);
 rightEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHSELFOPPOSITE; offset: 97);
 topEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHFORM; offset: 119);
 bottomEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHNONE)).
rentalNumber
 framingSpec: (AbtViewAttachmentConstraint new
 leftEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHFORM; offset: 75);
 rightEdge: (AbtRunEdgeAttachmentConstraint new
 attachment: XmATTACHSELFOPPOSITE; offset: 263);
topEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHFORM; offset: 71);
bottomEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHNONE)).
label1
  object: (gui at: 4)          " 'Please enter
video tape number to be checked in:' " ;
framingSpec: (AbtViewAttachmentConstraint new
  leftEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHFORM; offset: 50);
  rightEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHNONE);
  topEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHFORM; offset: 35);
  bottomEdge: (AbtRunEdgeAttachmentConstraint new
  attachment: XmATTACHNONE)).
messagePrompter1
  iconType: 1;
  buttonText: 1.
window
  abtWhenPrimitive: #gettingFocus
  perform:
    (DirectedMessage new
      receiver: rentalNumber;
      selector: #setFocus;
      arguments: #()).
conn1 := AbtEventToCodeHookConnection new
  connectSource: pushButton1
  featureName: #clicked
  feature: AbtPushButtonView IS_clicked;
  receiver: self selector: #findTape:
  selectorArgumentCount: 1.
conn1
  forParameter: #parameter1
  perform:
    (DirectedMessage new
      receiver: rentalNumber;
      selector: #abtGetValueOfAttribute:;
      arguments: (Array with: AbtTextView
      IS_object)).
pushButton2
  abtWhenPrimitive: #clicked
  perform:
    (DirectedMessage new
      receiver: window;
      selector: #closeWidgetCommand;
      arguments: #()).
conn1
  abtWhenPrimitive: #errorResult
  perform:
    (DirectedMessage new
      receiver: messagePrompter1;
      selector: #promptFor:;
      arguments: (Array new: 1)).
pushButton1
   abtWhenPrimitive: #clicked
   perform:
      (conn6 := AbtEventToActionConnection new
           source: pushButton1;
           eventName: #clicked;
           actionProvider: aVideoTape
           variableFeatureName: #clearRental
           featureSelector: #IS_clearRental).
  
conn1
   abtWhenPrimitive: #normalResult
   perform:
      (conn16 := AbtEventToActionConnection new
           source: conn1;
           eventName: #normalResult;
           actionProvider: aVideoTape
           variableFeatureName: #self
           featureSelector: #IS_self).
   self initializeAttributeConnections.
   self finalInitialize.
Summary

We’ve discussed:
- MVC
- MV
- Motif and CommonWidgets
- Building a View
- Connecting the View to the Model
- Quick peek at the internals