From Design to Implementation

The Presenters

- Rick Trotter, IBM Smalltalk Group
- Dave Maeda, IBM Smalltalk Group

Coding the video-store application

Today we will see how the video-store application Booch design and CRC cards covered in previous lectures can be translated into Smalltalk code.

A core principle of OO programming is that high-level design is a model of the application domain. This greatly contributes to requirements traceability and program maintainability.

In proceeding from high-level design to an initial implementation, we are going to continue in this vein.

We are going to begin by creating Smalltalk classes that provide implementations for the domain objects in our design.

We start by recalling the following Booch class diagram:
Building the domain model in VisualAge

Our implementation will consist of a set of domain model classes and a set of user interface classes.

We will create a $\text{VAST ENVY}$ application, \textbf{CSC517VAVideoStoreDomainModelApp}, to serve as a container for each of these.

From the VisualAge Organizer, click on the “Application-&gt;New...” menu. Enter the application name and then press “OK.”

![New Application dialog box]

Now we proceed with implementing our domain model classes. We'll start with a Customer class, \textbf{VaVCCustomer}. (Use the menu, “Parts-&gt;New...”.) Note that we create the class as a VisualAge "Nonvisual part".
After creating the class we will need to define an implementation of instance attributes and associations.

Our class diagram shows us relationships but not attributes. So in addition to our class diagram we refer to our Customer CRC card.

**Table 1 Customer CRC Card**

<table>
<thead>
<tr>
<th>Responsibilities</th>
<th>Collaborators</th>
</tr>
</thead>
<tbody>
<tr>
<td>know your name</td>
<td></td>
</tr>
<tr>
<td>know your customer number</td>
<td></td>
</tr>
<tr>
<td>know your address</td>
<td></td>
</tr>
<tr>
<td>know your current rentals</td>
<td></td>
</tr>
<tr>
<td>return a customer for an ID#</td>
<td></td>
</tr>
<tr>
<td>tell what rentals are overdue</td>
<td>TapeRental</td>
</tr>
</tbody>
</table>
In Smalltalk, instance attributes and associations are both represented using instance variables.

- There is no feature of the language which enables a designer to mark an instance variable as representing an attribute or an association.
- In some contexts the intent may be obvious or immaterial, while in other cases it is extremely important for the designer to indicate intent with method or class comments.

We define our new attributes and associations using the VAST Public Interface Editor.
We define each of our Customer model attributes and associations as a Customer object attribute in the Public Interface Editor (PIE).

From our CRC card we derive the attributes name, customerNumber, address, and currentRentals.

The PIE provides for specification of an attribute data type.

- We specify a data type of String for the attributes name and customerNumber, and
- a data type of Object for address, and OrderedCollection for currentRentals.

(Basically, for each attribute:

1. Type the attribute's name into the “Attribute name” field,

2. Click on the “Defaults” button,

3. Change the “Attribute data type” field as necessary, and

4. Click on the “Add" button.)

When this interface specification is complete, we generate and compile Smalltalk source code from it. ("File-->Generate Default Scripts...")
We then save our new interface definition, and we have completed the initial implementation of our first domain model class.

We'll next use the VAST Script Editor to review the effect of our work in PIE. (“View-->Script Editor”)
In the bottom pane is a Smalltalk class definition.

Look closely at the syntax. You will see that this is actually an imperative statement...

   It is an instruction to send the message
   #subclass:instanceVariableNames:classVariableNames:poolDictionaries:
   to the class AbtAppBlldrNonVisual.

   A new class definition is created by compiling and executing a source statement such as this.

   Accessor methods
You will also see that PIE has defined instance variables and accessor methods (sometimes called “getters and setters”) in the generated class definition.

Recall that all instance variables in Smalltalk are private to that class. To access the value of a variable from outside the class you must write a method to allow access.

Note that our mutator methods (or "setters") not only modify the state of an object, but signal that the object’s state has changed.

```smalltalk
address: anObject
  address := anObject.
  self signalEvent: #address with: anObject.
```

In the VAST visual programming environment, objects are "tied together" into a program by associating events (such an instance attribute value change) with actions (such as a view update or database table update).

The message sent above, #signalEvent:with:, is part of the internal implementation of this event-driven framework.

In addition to instance attribute accessors, we also find a number of private class methods generated by PIE. (Recall that a class method is a method that an instance’s class responds to.) There is also a generated private instance method, #abtBuildInternals. In general it is best to not to modify these private generated methods.
These private generated class methods are used by the VAST application builder tools and the VAST runtime framework.

Now that we have created at least a minimal implementation of our customer class, we will likewise create initial versions of our other classes:

- VaVCAAddress
- VaVCCategory
Testing our domain model

Even with an implementation as simple as our Customer model is at present, it is worthwhile beginning to develop test cases and test suites.

We aren’t really going to do justice to the subject of test methodology in this lecture, but we really would be remiss if we didn’t mention testing at all.

As a rule, each component and subsystem should be independently testable. As the implementation of our domain model evolves we are likely to want to implement automated test drivers to verify use-case scenarios.

We would like to create a new ENVY application for our test-case classes. For present purposes, however, we will just be implementing a few simple test scripts as class or instance methods on our model classes.

We create a class method to return a sample customer model.

```smalltalk
exampleCustomer1 ^self new
  name: 'V.A. Small';
  customerNumber: 21120;
  address: VaVCAAddress exampleAddress;
  yourself.
```
We will inspect our model objects frequently during development, and we will find it useful to have sort of a summary view available to us.

To accomplish this, we implement an instance method like the following on our VaVCCustomer, providing us with a “debug” print string.

```smalltalk
debugPrintOn: writeStream
    writeStream
    nextPutAll: self class name;
    nextPutAll: '{
    cr; tab; nextPutAll: 'customerNumber: ';
    nextPutAll: self customerNumber
    printString;
    cr; tab; nextPutAll: 'name: ';
    nextPutAll: self name
    printString;
    cr; nextPut:$}
```

This gives us the following inspector view.
`VaVCCustomer`

```python
VaVCCustomer {
    customerNumber: '90210'
    name: 'Suzie Q. Marietta'
}
```

- `self`
- `eventDependents`
- `parentPart`
- `abtlisDestroyed`
- `userInfo`
- `components`
- `primaryPart`
- `attributeConnections`
- `instanceInterfaceSpec`
- `dragDropSpecClass`
- `customerNumber...
- `name...
- `address`
- `currentRentals`
Object interactions

Recall from the previous lecture our “Rent a Video” use case. We might outline the essential interactions, from a baseline scenario, between the clerk and our domain model as follows.

1. Find Customer for customer number
2. Begin a new transaction for this customer
3. Add rentals for selected videos to the transaction
4. Compute the sales tax and amount due
5. Commit the transaction

In a real system our domain objects would generally be persistent objects—objects with a representation external to our application that persists across invocations of our application.

This persistent representation would likely be maintained in some sort of database, e.g., a flat file, a relational database, or an object database.

To support our model-building demonstration we are going to make use of a mock in-memory object store, where class variables are used to hold collections of “persistent” objects.

*These objects are in fact persistent in the sense that the state of a class (including its class variable associations) is saved when the Smalltalk image is saved.*

We will add a class variable `AllCustomers` to our `VaVCCustomer` class definition.

```smalltalk
AbtAppBldrPart subclass: #VaVCCustomer
    instanceVariableNames: 'customerNumber name address currentRentals'
```
Then we add class methods for basic storage and retrieval operations. For present purposes, we need a lookup method.

```smalltalk
lookUpCustomerForNumber: aCustomerNumber

^self allCustomers detect: [ :each | each customerNumber = aCustomerNumber ] ifNone: []
```

We will need a similar lookup method on `VaVCVideoTape`.

```smalltalk
lookUpTapeForId: aTapeId

^self allTapes at: aTapeId ifAbsent: []
```

Initially, I am going to skip the sales tax and amount due calculations. We will come back to these, but for now we will finish up this iteration of our scenario implementation with a commit method on `VaVCCustomerTransaction`.

```smalltalk
commitTransaction: aTransaction

aTransaction finish.

self allTransactions add: aTransaction
```

Putting all of this together, we might have something like the following.

```smalltalk
testScenarioOne

| sampleCustomerId sampleTapeId customer tape rentalTransaction |

sampleCustomerId := '90210'.

sampleTapeId := 'Down by Law'.

(customer := VaVCCustomer lookupCustomerForNumber: sampleCustomerId)

isNil ifTrue: [^self error: ('No customer: "%1" bindWith: sampleCustomerId)].

(tape := VaVCVideoTape lookupTapeForId: sampleTapeId)

isNil ifTrue: [^self error: ('No tape: "%1" bindWith: sampleTapeId)].
```

---

Lecture 9 Object-Oriented Languages and Systems
ifTrue: [^self error: ('No tape: "%1" bindWith: sampleTapeId)].
rentalTransaction := VaVCCustomerTransaction new
customer: customer;
rentalDate: Date today;
addTapeRentalForTape: tape;
yourself.
VaVCCustomerTransaction commitTransaction: rentalTransaction.

Note that I've included simple checks for the scenarios “customer not found” and “tape not found”, which signal an exception.

If we were not prepared to catch such exceptions and one occurred in our development image, a debugger such as the one below would appear.
Let's move on and consider the issue of calculating our transaction total, which is the sum of rental fees, late fees, and sales tax.

```apl
totalDue
 | subTotal |
 subTotal := self rentals
        inject: (ScaledDecimal fromInteger: 0)
        into:[:sum :each | sum + each feeOrFine].
 ^subTotal + self salesTax
```

This looks good; we just need implementations of `#feeOrFine` and `#salesTax`.

```apl
feeOrFine
 ^self isOverdue
  ifTrue:[self overdueFine]
  ifFalse:[self rentalFee]
```

Consider a business rule that indicates that rental fees vary by movie category—drama, comedy, new release, and so forth. Thus, we can implement `#rentalFee` on `Category` and delegate from rental to tape to category.

```apl
VaVCTapeRental>>#rentalFee
 ^self tape rentalFee

VaVCMovie>>#rentalFee
 ^self category rentalFee

VaVCCategory>>#rentalFee
 (self name = 'new releases')
  ifTrue:[^self newReleasesFee]
  ifFalse:[^self allOthersFee]
```

Now consider the calculation of sales tax.
Since we have a rental object representing a line item in a customer transaction, a rental object may represent either a new rental or a late rental with a consequent overdue fine.

In our example application, overdue fines are not taxable, but new rentals are taxable at 5%.

We choose to implement this business rule in the rental object, since the rental knows if it is representing an overdue fee or a new rental. So it follows that this is a good place to encapsulate the dependency of sales tax on #isOverdue.

Thus the calculation of sales tax will require us to write several methods in different classes. This distribution of behavior gives us two benefits:

- It enforces encapsulation. Methods should not “snoop on” other objects to get their data and operate on it; behavior should go with the data it operates on.
- It makes the individual methods in each class shorter, which makes them easier to understand, modify and maintain.

Let’s follow the trail of messages and examine them one at a time, starting with VaVCCustomerTransaction>>salesTax.

```smaller

salesTax
"Return the sales tax for all rentals"
^self rentals
  inject: (ScaledDecimal fromInteger: 0)
  into:[:sum :each | sum + each salesTax].
```

So, all a transaction knows how to do is sum the tax required for each of its line items.

Now let’s consider VaVCTapeRental>>salesTax.
salesTax
"Overdue fines are not taxable. New Rentals are taxable at 5%"
^self isOverdue
  ifTrue: [##(ScaledDecimal fromInteger: 0)]
  ifFalse: [self rentalFee * self taxRate].

Notice two things about the previous method.

• I did not figure out the rental fee directly in this method; I passed it off to another method.

  This is because a rental does not have all of the information to know what its rental fee is. This it must obtain from the category that the movie belongs to.

  We factor out a dependency on a business rule that is beyond the scope of responsibility of the method being implemented.

• I did not directly encode the tax rate as a constant in this method; it was coded as another method.

  In effect, tax rate is an abstraction that it used by the sales tax calculator. It may be constant or derived from some other business rule. It may be constant today but derived from a new business rule six months from now.

  Once again, we factor out a dependency on a business rule that is beyond the scope of responsibility of the method being implemented.

At the end of this path we have our #taxRate implementation.

taxRate
  TaxRate isNil
    ifTrue: [TaxRate := ScaledDecimal fromString: '0.05'].
  ^TaxRate

Lecture 9 Object-Oriented Languages and Systems
Just to make my program a bit easier to configure and maintain, I've implemented a class variable to hold the tax rate in use. (If my video store customer is successful, they might expand into an adjoining region with a different sales tax rate)

Now that we've seen the entire path of messages and how they are implemented, we can try out the full version of our test method and see if it works.

### testVideoRentall

"try the following scenario. 
1. Find the customer with Customer Number 10001.
2. Create a new customer transaction
3. Have him rent video number 10001 (clueless).
4. Show the sales tax and amount due"

"self testVideoRentall"

| customer transaction |
customer := VCCustomer customerForNumber: 10001.
transaction := self new
customer: customer;
getOverdueTapes;
addTapeForId: 10001;
yourself.
transaction finishTransaction.
Transcript show: ('Sales Tax ', transaction salesTax printString);cr.
Transcript show: ('Total Due ', transaction totalDue printString);cr.

### Summary

We've discussed:
- Transition from design to implementation
- Delegation and encapsulation
- Representation of attributes and associations