From Design to Implementation

The Presenters

- Rick Trotter, IBM Smalltalk Group
- Rob Pungello, IBM Smalltalk Group

Coding the video-store application

Today we will see how the Booch design for the video-store application can be translated into Smalltalk code.

There are two caveats about today’s code:

- It will not include any user-interface elements (windows, buttons, dialogs, etc.). That will be covered in the next lecture.
- It will not be a true “production-quality” implementation. This is a small example application, but there are still many compromises that must be made to fit the coding into a single lecture.

Some review and comment on methodology

Recall that OO practitioners often employ a process methodology that can be thought of as a sequence of iterative refinements.

This is often referred to as a “spiral model’ of software development.

- This methodology has really evolved concurrently with OO design. It is not inherently an OO methodology, but
- in the spirit of OO being a synthesis of evolving best practices it is commonly used by OO practitioners.

Getting all the requirements correct and complete up-front for non-trivial systems and applications is, for practical purposes, impossible. Developing new applications and systems is very much a process of discovery.
• Users of business applications typically don't have a well-defined formal understanding of their business processes and business rules, even though they are experts at what they do.

• Inventors of new technologies usually don't foresee the requirements that an initial deployment to a user community will generate.

The software-engineering community continues to struggle with the problems of developing with incomplete and evolving requirements in a competitive marketplace and with conflicting objectives, e.g.,

• time versus money,
• performance versus interoperability).

There really is no language and no process which is recognized as enabling software development to be very predictable and mechanical.

There continues to be innovation on the process front. A relatively recent example of this is the Extreme Programming approach. Nevertheless, it remains true that building and delivering high-quality software on time and in budget is hard.

With the objectives and constraints of this course, we are not going to do justice to the issue of software engineering methodology. We do hope to provide some snapshots, which will hopefully enlighten more than they mislead.

Today’s exercise

Previous lectures illustrated how we can use CRC cards to identify key domain objects and their responsibilities, and how this understanding of the domain can be carried forward into a high-level application design.

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 VAST ENVY application to serve as a container for each of these.

We begin by creating an ENVY application for our VA domain model.
Now we proceed with implementing our domain model classes. We'll start with a Customer class. Note that we create this as a VisualAge "Nonvisual part".
After creating the class we want 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.

When this interface specification is complete, we generate and compile Smalltalk source code from it.
Then we 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.

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

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 included 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.

**Pool dictionaries**

The particular form of a class definition method that we see in the Script Editor also provides for class variable definition and pool dictionary declarations.

A class variable is a variable that is shared by all instances of a class and its subclasses.

A pool dictionary is a namespace-extension mechanism. A class that declares the use of a pool dictionary may reference variables in that pool by name as if they were defined locally.

Pool dictionaries are most often used as namespaces for symbolic constants. The pool dictionary CfsConstants, for example, defines the pool variables `ORDONLY`, `ORDWR`, and `OWRONLY` (among others).
In addition to instance attribute accessors, we also find a number of class methods generated by PIE. (Recall that a class method is a method that an instance’s class responds to.)

These generated class methods are used by the VAST application builder tools and the VAST runtime framework.

Note that these methods are categorized as “private” methods. This is a statement of design intent. Nothing in the Smalltalk language restricts visibility of methods, even “private” methods.

Generally, Smalltalk is well-suited for clear-box designs. Smalltalk, its development tools, and its common coding practices strive to make design intent evident without restricting usage which may have been beyond the expectation of the original developer.
Most of these private class methods serve to provide a runtime “interface spec” for the class.

The internal representation of this interface spec is a pool dictionary named `IS_VaVCCustomer`, defined in the class method `_PRAGMA_IS_`.

Each of the pool variables in this pool dictionary has as its value an interface spec for one of the attributes we defined in PIE, or an interface spec for an instance of the class.

Note that the class method `IS_name` has a variable reference of the form `foo::bar`. This is a reference to the variable named `bar` in the pool dictionary `foo`.

In this case, this pool dictionary is actually defined by the class, in the method `_PRAGMA_IS_`, and so a reference of the form `bar` alone would be syntactically correct. The PIE code generator uses this fully-qualified form of a pool variable reference in order to minimize the possibility of namespace collisions.

Note also that these pool variables are initialized on demand using a technique known as “lazy initialization,” where the object associated with a variable is created and assigned to the variable only on first invocation of the accessor method.

We obtain an initial implementation of our remaining domain model classes in the same way we have for our Customer model.

I emphasize “initial” implementation because we have been focusing on structural representation and have not yet addressed the operational behavior of our domain model.

**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.

```small
exampleCustomer1
  ^self new
  name: 'V.A. Small';
  customerNumber: 21120;
  address: VaVCAddress 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.

```small
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.
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 add a class variable `AllCustomers` to our `VaVCCustomer` class definition.

```smalltalk
AbtAppBldrPart subclass: #VaVCCustomer
    instanceVariableNames: 'customerNumber name address currentRentals'
    classVariableNames: 'AllCustomers'
    poolDictionaries: ''
```

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) ].
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.
As we develop our automated test tools, where we have a driver that runs through a large set of test cases, we want to make sure that an exception in one test case does not cause our driver to fail completely.

So let’s examine how we might set up the code that invokes our test case so that an exception such as this is trapped and handled in a manner more appropriate to our needs.

Trapping an exception in Smalltalk involves wrapping the code to be evaluated in a Block and sending this Block instance the message #when:do:, as in the following example.

```smalltalk
[VaVCCustomerTransaction testScenarioOne]
  when: ExError
  do: [:sig |]
```
In this example, we indicate an interest in trapping the exception ExError, and any of its “children.” If any such exception occurs, our exception handler will simply write a log message to record the occurrence of the exception, and then cause a normal return (#exitWith:) from the stack frame wrapped by our exception handler.

Suppose we want to have a convenient technique for employing assertions in our test cases. We can create a new kind of exception for this purpose and have an #assert method throw this exception if an assertion test fails.

Our test code will include an expression such as the following.

```
[(customer := VaVCCustomer
   lookUpCustomerForNumber: sampleCustomerId)
  notNil] assert.
```

We create our new exception using the expression “ExError newChild”. This will yield an instance of ExceptionalEvent.

A common technique for managing custom exceptions is to make use of pool variables.

The system exception ExError, for example, is an instance of ExceptionalEvent bound to the pool variable SystemExceptions::ExError. (Note the method Core class>>#_PRAGMA_SystemExceptions)

I want to use this example, however, to demonstrate global variables in Smalltalk.

A global variable in Smalltalk is a variable defined in the global namespace, or the Smalltalk system dictionary (which is itself referenced by the global variable #Smalltalk).

```
System globalNamespace
```
Then we can define our new exception as follows.

```smalltalk
VaVCTestAssertionFailure := ExError newChild
markReadOnly: false;
description:
' (VaVCTestAssertionFailure) An assertion failed.';
markReadOnly: true;
yourself.
```

Next we will create a class extension to extend the behavior of Block with our custom assert method.

As we’ve seen, classes in IBM Smalltalk are defined within the scope of a controlling ENVY application or subapplication. Class definitions can, however, be extended with behavior not implemented within their controlling application. We are going to extend Block in our domain model application with an instance method #vcAssert. (It is generally good practice to use a prefix on method extensions, minimizing the possibility of namespace collisions and marking the origin of the extension).

So now the code in our test method looks something more like the following.

```smalltalk
| sampleCustomerId sampleTapeId customer tape rentalTransaction |
sampleCustomerId := '90210'.
sampleTapeId := 'Down by Law'.
[(customer := VaVCCustomer lookupCustomerForNumber: sampleCustomerId)
notNil] vcAssert.
[(tape := VaVCVideoTape lookupTapeForId: sampleTapeId) notNil] vcAssert.
rentalTransaction := VaVCCustomerTransaction new
customer: customer;
rentalDate: Date today;
addTapeRentals: tape;
yourself.
VaVCCustomerTransaction commitTransaction: rentalTransaction.
```

And we might choose to have our driver trap assert failures as follows.

```smalltalk
[VaVCCustomerTransaction testScenarioOne]
when: VaVCTestAssertionFailure
do: [:sig|
```
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.

```
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.

```
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.

```
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 \( \# \text{isOverdue} \).

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`.

```smalltalk
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`.

```smalltalk
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

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.
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
- Variables: global, pool, class, instance, method
- Blocks
- Exceptions
- Processes