CS603 Organization of Programming Languages

Spring 2005

Smalltalk Control Structures

Joel Jones
Department of Computer Science
University of Alabama

Printing Paychecks

```
paycheckString
  | aStream |
  aStream := WriteStream on:
       (String new: 20).
  self printPaychecksOn: aStream.
  ^aStream contents
```

Transcript

Transcript is an instance of TextCollector.

TextCollector is not a subclass of Stream, but supports a lot of WriteStream protocol, such as `nextPut:`, `nextPutAll:`, `cr`, `space`, `tab`

show: and print:

`show:` is like `nextPutAll:`

Transcript doesn't display any characters until it is sent `show:`. Look at show: method to see how to display otherwise.

```
print: anObject
   anObject printOn: self
```

Uses of Inheritance

To be shared by all subclasses

- don't redefine

Default

- probably redefine

(continued)

Parameterized by subclasses (template method)

- rarely redefine
- change it by redefining the methods it calls

True abstract methods

- must redefine
- self subclassResponsibility

For an Abstract Class

...

Look for "subclassResponsibility" methods to see the core methods.

Look in subclasses to see examples.

Flow of control goes up and down the class hierarchy.

(So don't try to follow it!)

Don't send "new" to the class.

Collection hierarchy

Collection ()

- Bag ('contents')
- SequenceableCollection ()
  - ArrayedCollection ()
  - LinkedList ('firstLink' 'lastLink')
    - Semaphore ('excessSignals')
  - Interval (‘start’ ‘stop’ ‘end’)
  - OrderedCollection ('firstIndex' 'lastIndex')
    - SortedCollection ('sortBlock')
- Set ('tally')
  - Dictionary ()
  - IdentitySet ()

ArrayedCollection hierarchy

ArrayedCollection ()
- Array ()
- CharacterArray ()
- String ()
  - Symbol ()
- ByteArray ()
- FloatArray ()
- IntegerArray ()
- Text ('string' 'runs')
- RunArray ('runs' 'values' 'lastIndex' 'lastRun' 'lastOffset')

Collection as an Abstract Class

Collection has no instance variables.

Collection defines as `subclassResponsibility`

`do:`, `add:`, `remove:`, `at:`, `at:put:`

Collection as an Abstract Class

Template methods defined in terms of do:

```
select:,collect:,inject:into:,
detect:ifAbsent:,size
```

Abstract Classes

- Abstract class as template
  - most operations defined in terms of `do:`
  - improved program skeleton
- Abstract class as type
  - All collections understand same protocol (`do:`, `select:`, `collect`, etc.)

Find an Element

```
detect: aBlock ifNone: exceptionBlock
  "Evaluate aBlock with each of the
receiver's elements as the argument.
Answer the first element for which
aBlock evaluates to true."
  self do: [:each | (aBlock value: each)
                      ifTrue: [^each]].
  ^exceptionBlock value
```

Accumulate a Value

inject: thisValue into: binaryBlock

"Accumulate a running value associated with evaluating the argument, binaryBlock, with the current value and the receiver as block arguments. The initial value is the value of the argument, thisValue."

Accumulate a Value

```
inject: thisValue into: binaryBlock
| nextValue |
  nextValue := thisValue.
  self do: [:each |
     nextValue := binaryBlock value: nextValue
                               value: each].
  ^nextValue
```

Transform a Collection

```
collect: aBlock
    | newCollection |
    newCollection := self species new.
    self do: [:each |
              newCollection
                  add: (aBlock value: each)].
    ^newCollection
```

Abstract Classes

Classes defined to be used ONLY as superclasses.

Design of its subclasses.

Abstract methods are supposed to be defined by subclasses.

Define standard interface that all subclasses implement and any client can use.

Polymorphism

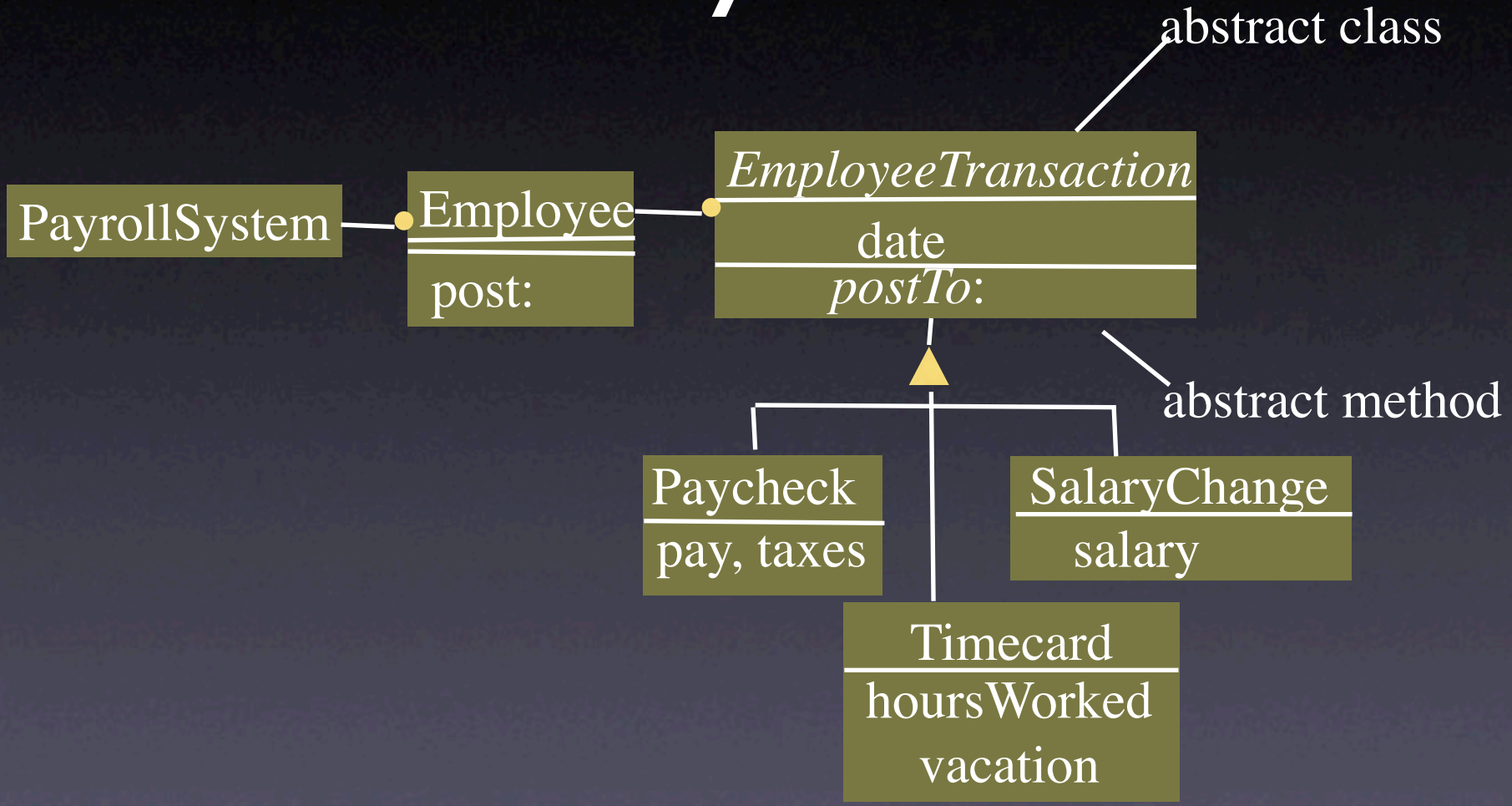
- Poly => many
- Morph => shape

- Variables take on many shapes, or many classes of objects.

Polymorphism

- Polymorphism in OOP is caused by late-binding of procedure calls (message lookup).
- Program can work with any object that has the right set of methods.

Object Model for Payroll



Examples

```
post: aTransaction

   aTransaction postTo: self.

   transactations add: aTransaction
```

```
numbers inject: 0 into: [:sum :i | sum + i]
```

The Case Against Case Statements

- Smalltalk has no case statement.
- OO programmers tend to avoid case statements, and to use message sending, instead.

Eliminating Cases

```
exp
   case: 1 do: [...];
   case: 15 do: [...].
```

Make classes for 1 and 15, with a method called `msg`.

```
exp msg or (dictionary at: exp) msg
```

Class UndefinedObject

- Class UndefinedObject has one instance -- `nil`.

- All variables are initialized to `nil`.
- Also used to indicate illegal value.

Don’t Test Classes

UndefinedObject

```
isNil
    ^true
notNil
    ^false
```

Object

```
isNil
    ^false
notNil
    ^true
```

Don’t test classes: use message sending and inheritance.

Choices

bad

```
x class = UndefinedObject ifTrue: [...]
```

OK

```
x = nil ifTrue: [...]
```

best

```
x isNil ifTrue: [...]
```

Another choice

```
x ifNil: [ … ]
```

Object

```
ifNil: aBlock
    ^self
```

UndefinedObject

```
ifNil: aBlock
    ^aBlock value
```

Magnitude

Magnitude ()
  Number ()
    ....
  Point ('x' 'y')
  Character (’value’)
  Time ('seconds' ‘nanos’)
  Timespan (’start’ ‘duration’)
    Date ()

Number Hierarchy

Number
  Float
  Fraction ('numerator' 'denominator')
  Integer ()
    LargePositiveInteger ()
      LargeNegativeInteger ()
    SmallInteger ()
  ...

Magnitude comment

- Magnitude has methods for dealing with linearly ordered collections. Subclasses represent dates, times, and numbers

Magnitude comment

Subclasses must implement the following messages:

comparing

<

=

hash

Numbers

- Numbers are part of the class hierarchy, not built into compiler.
- Numbers understand +, -, *, /, <, <=, etc.
- 3 / 6 => 1 / 2
- 2 sqrt => 1.41421

- 3 + 4 * 2 =>

ArithmeticObject

- Points, Numbers, Complex (not in image)
- squared
- "Answer the receiver multiplied by itself."
- ^self * self

Arithmetic

```
= aNumber
  aNumber isNumber ifFalse: [^false].
  aNumber isInteger ifTrue:
    [aNumber negative == self negative
      ifTrue: [^(self digitCompare: aNumber) = 0]
      ifFalse: [^false]].
  ^aNumber adaptToInteger: self andSend: #=
```

Number

- Number is also a magnitude.
- Adds truncation operations
- FixedPoint, Fraction, Integer, Double, Float

Number

```
// aNumber
   "Integer quotient defined by division with
truncation toward negative
   infinity.  9//4 = 2,  -9//4 = -3.  -0.9//0.4 =
-3.
   \\ answers the remainder from this division."
   ^(self / aNumber) floor
```

Number

```
\\ aNumber
    "modulo.  Remainder defined in terms of /
/.  Answer a Number with the same sign as
aNumber.  e.g.    9\\4 = 1,    -9\\4 = 3, 9\\
-4 =  -3,      0.9\\0.4 = 0.1"
    ^self - (self // aNumber * aNumber)
```

Uses of Polymorphism

- Method lookup finds the method that is right for the receiver.
  - method always knows the class of the receiver
  - different classes lead to different methods

Uses of Polymorphism

- Methods often depend radically on class of receiver.
  - isNil
  - ifTrue:ifFalse:
  - double dispatching

Double-dispatching: the Problem

	Si	Fl	Fr	Fi	M
Smallinteger	*	Fl	Fr	Fi	Sc
Float	Fl	*	Fl	Fl	Sc
Fraction	Fr	Fl	*	Fi	Sc
FixedPoint	Fi	Fl	Fi	*	Sc
Matrix	Sc	Sc	Sc	Sc	*

Arithmetic

Number * Matrix

Multiply each element by number

Matrix * Number

Multiply each element by number

Matrix * Matrix

Standard matrix multiplication

Arithmetic

Integer * Integer

Primitive int operations

Integer * Float

Convert int to float

Float * Integer

Convert int to float

Float * Float

Primitive float operation

Double dispatching: the Solution

- Primary method (+, *, etc) sends a second message to argument, encoding the class of the receiver.
- Second message knows class of both its argument and its receiver.

Primary operations

- (Conceptually) Send a second message, encoding the class of the original receiver in the name of the message.

```
+ anArg

  ↑ anArg sumFromInteger: self
```

- Details are trickier

Double Dispatching Methods

- Knows the class of receiver and argument.
- sumFromInteger: anInteger
- 	^anInteger asFloat + self

The first message dispatch

- 37 + 8.9
-
- SmallInteger + aNumber
- <primitive: 1>
- ^ super + aNumber
-
- 37 + 8.9

Uses + in Integer

The second message dispatch

- `8.9 adaptToInteger: 37 andSend: #+`
- `Float adaptToInteger: rcrv andSend: selector`

  `^ rcvr asFloat perform: selector with: self`

`8.9 + 37.0`

in SmallInteger

Finishing Up

- 8.9 + 37.0

- ```
  Float  + anArg

  <primitive: 41>

  ^aNumber adaptToFloat: self andSend: #+
  ```
  ```

Multiplication

- Fraction has two instance variables, numerator and denominator.
- `* aNumber`

```
"Result is a new Fraction unless the
argument is a Float, in which case the
result is a Float."

^aNumber productFromFraction: self
```

Shared Responsibilities

- Sometimes need to select a method based on class of several objects:
  - Displaying object -- depends on both the kind of object and the windowing system
  - Arithmetic -- depends on the types of both arguments

Double dispatching

- Three kinds of messages
  - primary operations
  - double dispatching methods
  - forwarding operations
- Implement inheriting from superclass of argument
- Implement commutativity

Cost of Double Dispatching

- Adding a new class requires adding a message to each of the other classes.
- Worst case is N*N methods for N classes.
- However, total lines of code is not much larger.