CS 535 Object-Oriented Programming & Design
Fall Semester, 2008
Doc 10 More Testing, Abstraction & Polymorphism
Sept 29 2008

References

Ralph Johnson Lecture notes, Lecture 3 Data Abstraction and Encapsulation, http://st-www.cs.uiuc.edu/users/cs497/lectures.html

Object-Oriented Design Heuristics, Riel, Chapter 2

Testing

What to Test

Everything that could possibly break

Test values

- Inside valid range
- Outside valid range
- On the boundary between valid/invalid

GUIs are very hard to test

- Keep GUI layer very thin
- Unit test program behind the GUI, not the GUI

Common Things Programs Handle Incorrectly

Adapted with permission from “A Short Catalog of Test Ideas” by Brian Marick

Any Object

nil pointer

Strings

Empty String

Collections

Empty Collection

Collection with one element

Collection with duplicate elements

Collections with maximum possible size

Numbers

Zero

The smallest number

Just below the smallest number

The largest number

Just above the largest number

Brian Marick paper can be found at http://www.exampler.com/testing-com/writings/short-catalog.pdf. See http://www.exampler.com/testing-com/writings.html for more of his papers.

Abstraction
Information Hiding
Polymorphism

Information Hiding

An object should hide design decisions from its users

Hide

- What is stored & what is computed
- Classes used

How does Point story its data?

How does OrderedCollection hold elements?

Heuristic 2.1

All data should be hidden within it class

Smalltalk instance variables in can be accessed in:

Instance methods of Class where they are defined

Instance methods of subclasses of the Class where they are defined

Language Support for Global Data

Smalltalk has shared variables

Use sparingly

Use for constants

What is a constant?

Hiding Instance Variables

Some argue that only two methods should access an instance variable

Class BankAccount
Instance variable: balance

```
balance
   ^balance

balance: aNumber
   balance := anumber

deposit: aNumber
   self balance: (self balance + aNumber)
```

Why

This protects the class from changes in instance variables

Makes easy to enforce constraints

```
balance: aNumber
   aNumber < 0 ifTrue: [ NegativeBalanceError raiseSignal].
   balance := aNumber
```

Hiding Instance Variables &Tools

Refactoring browser

- Lists all methods accessing an instance variable
- Change all accesses to be through access methods
- Removes all access through access methods

So don’t worry about hiding instance variables

If later you need to hide them it is easy to do

Abstraction

"Extracting the essential details about an item or group of items, while ignoring the unessential details."

Edward Berard

"The process of identifying common patterns that have systematic variations; an abstraction represents the common pattern and provides a means for specifying which variation to use."

Richard Gabriel

Pattern: Priority queue
Essential Details: length
items in queue
operations to add/remove/find item
Variation: link list vs. array implementation
stack, queue

How to Find Abstractions

Look at nouns in requirements specification or system description

A refrigerator has a motor, a temperature sensor, a light and a door. The motor turns on and off primarily as prescribed by the temperature sensor. However, the motor stops when the door is opened. The motor restarts when the door is closed if the temperature is too high. The light is turned on when the door opens and is turned off when the door is closed.

Look at these phrases. Some will be obvious classes, some will be obvious nonsense, and some will fall between obvious and nonsense. Skip the nonsense, keep the rest. The goal is a list of candidate objects. Some items in the list will be eliminated, others will be added later. Finding good objects is a skill, like finding a good functional decomposition

Ralph Johnson's Suggestions for Finding Abstractions

Do one thing
Eliminate duplication
Keep rate of change similar
Decrease coupling, increase cohesion
Minimize interfaces
Minimize size of abstractions
Minimize number of abstractions

Do One Thing

Methods should do one thing

Method's name should tell what it does

findString:startingAt:
asNumber
asUppercase
dropFinalVowels

Class should be what its name says

String
OrderedCollection
Array
ReadStream

Break complex classes/methods into simpler ones

Eliminate Duplication

(self asInteger - $a asInteger + anInteger) \\ 26 – (self asInteger - $a asInteger)

(self alphabetValue + anInteger) \\ 26 - self alphabetValue.

Keep rate of change similar

An object should not contain both

- An instance variable that changes every second
- An instance variable that changes once a month

- Code that is different for each hardware platform
- Code that is different for each OS

Separate tax tables from employee data from time cards

Minimize interfaces

Use the smallest interface you can

Use Number instead of Float

Avoid embedding classes in names

add: instead of addNumber:

Minimize the size of abstractions

Lots of Little Pieces

Methods should be small

Median size is 3 lines
10 lines is starting to smell

Classes should be small

7 variables is starting to smell
40 methods is starting to smell

VW 7.6

	Average	Median	Max
Variables / class	2.1	1	72
Methods / class	16.6	8	359
LOC / method	3.0	2	156

Code used to generate Numbers

Variables Per Class

```
classes :=Smalltalk allClasses reject: [:each | each isMeta]
variablesInClass :=classes collect: [:each | each instVarNames size].
average :=((variablesInClass fold: [:sum :each | sum + each] )/
                variablesInClass size) asFloat.
median := variablesInClass asSortedCollection at: variablesInClass size // 2.
max := variablesInClass fold: [:partialMax :each | partialMax max: each]
```

Methods Per Class

```
classes :=Smalltalk allClasses reject: [:each | each isMeta]
methodsInClass :=classes collect: [:each | each selectors size].
average :=((methodsInClass fold: [:sum :each | sum + each] )/
                methodsInClass size) asFloat.
mean := methodsInClass asSortedCollection at: methodsInClass size // 2.
max := methodsInClass fold: [:partialMax :each | partialMax max: each]
```


Note how the above code could use the application of these ideas

LOC / Method

```
methodSizes := OrderedCollection new.
classes
	do: [:class |
		class selectors
			do: [:method |
				| periodCount |
				periodCount := (class compiledMethodAt: method) decompiledSource
					occurrencesOf: $..
				methodSizes add: periodCount + 1]].
average :=((methodSizes fold: [:sum :each | sum + each] )/
			methodSizes size) asFloat.
median := methodSizes asSortedCollection at: methodSizes size // 2.
max := methodSizes fold: [:partialMax :each | partialMax max: each]
```

Minimize number of abstractions

A class hierarchy 6-7 levels deep is hard to learn

Break large system into subsystems, so people only have to learn part of the system at a time

Polymorphism

Objects with the same interface can be substituted for each other at run-time

Variables take on many classes of object

Objects will behave according to their type

Code can work with any object that has the right set of methods

In C++ polymorphism requires
- Inheritance
- Pointers
- Virtual functions

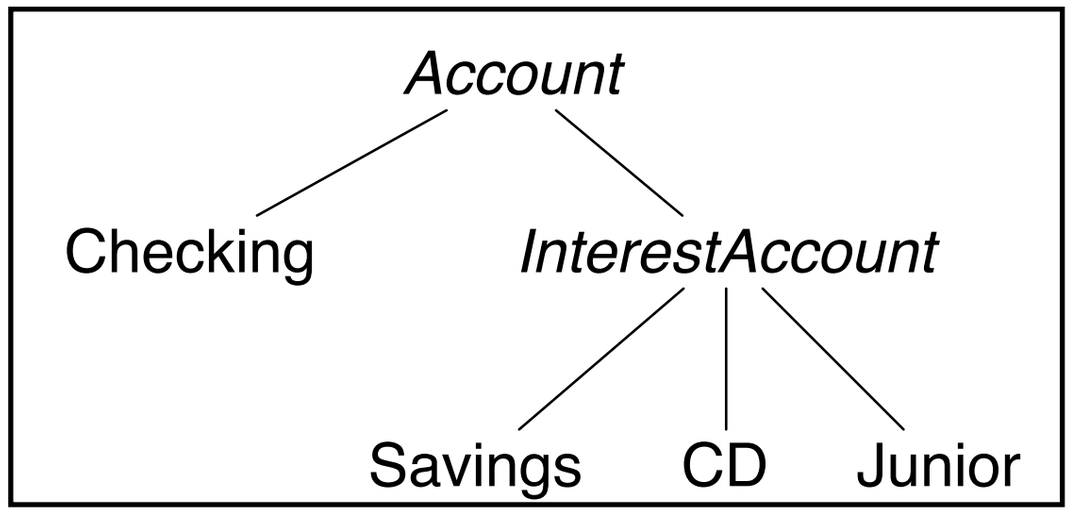
In Java polymorphism requires
- Inheritance or
- Interfaces

In Smalltalk polymorphism does not require inheritance

Simplistic Example

Bank offers various types of accounts:

- Checking
- Savings
- CD
- Junior savings accounts



Each type has different rules for processing a transaction

Processing a Transaction

Using Case Statement

```
ewCustomer := Bank createNewAccount: type.

Etc.

newCustomer class = Checking ifTrue:[ ...]
newCustomer class = Savings ifTrue:[ ...]
newCustomer class = CD ifTrue:[ ...]
newCustomer class = Junior ifTrue:[ ...]
```

Using Polymorphism

```
newCustomer := Bank createNewAccount: type.
newCustomer.processTransaction: amount
```

Which processTransaction is called?

Adding new types of accounts to program requires:

- Adding new subclasses
- Changing code that creates objects

Avoid checking the class of an object

Avoid Case Statements

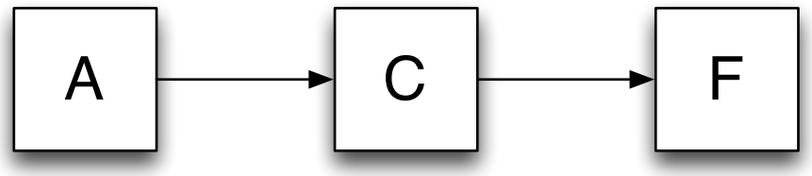
Smalltalk has no case statement

OO programers send a message to object instead

Each type of object handles the message according to its type

Case statements make it harder to add new cases

Linked List Example



Operations

- Add elements
- Test if list contains an element
- printOn:
- size

First Example

```
Smalltalk defineClass: #LinkedList
   superclass: #{Core.SequenceableCollection}
   instanceVariableNames: 'value next '
```

Class Methods

```
with: anObject
   ^super new setValue: anObject
```

Instance Methods

```
addLast: anObject
   next ifNotNil: [^next addLast: anObject].
   next := LinkedList with: anObject.

includes: anObject
   value = anObject ifTrue:[^true].
   next ifNotNil: [^next includes: anObject].
   ^false
```

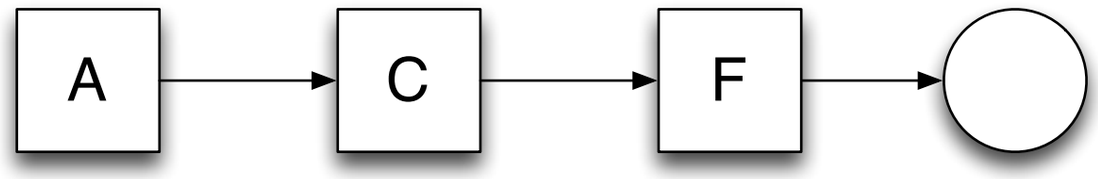
Instance Methods

```
printOn: aStream
   aStream
      print: value;
      nextPutAll: ' '.
   next ifNotNil: [next printOn: aStream]

setValue: anObject
   value := anObject.

size
   next ifNil: [^1].
   ^next size + 1
```

Using Polymorphism

Use two types of nodes

- LinkedList
- NilNode



NilNode

- Linked list terminator
- Ends messages sent through list
- List always ends with a nil node

NilNode

```
Smalltalk defineClass: #NilNode
   superclass: #{Core.Object}

                  Instance Methods
addLast: anObject
   self become: (LinkedList with: anObject)

includes: anObject
   ^false

printOn: aStream

size
   ^0
```

LinkedList with NilNode

```
Smalltalk defineClass: #LinkedList
   superclass: #{Core.SequenceableCollection}
   instanceVariableNames: 'value next '
```

Class Methods

```
with: anObject
   ^super new setValue: anObject
```

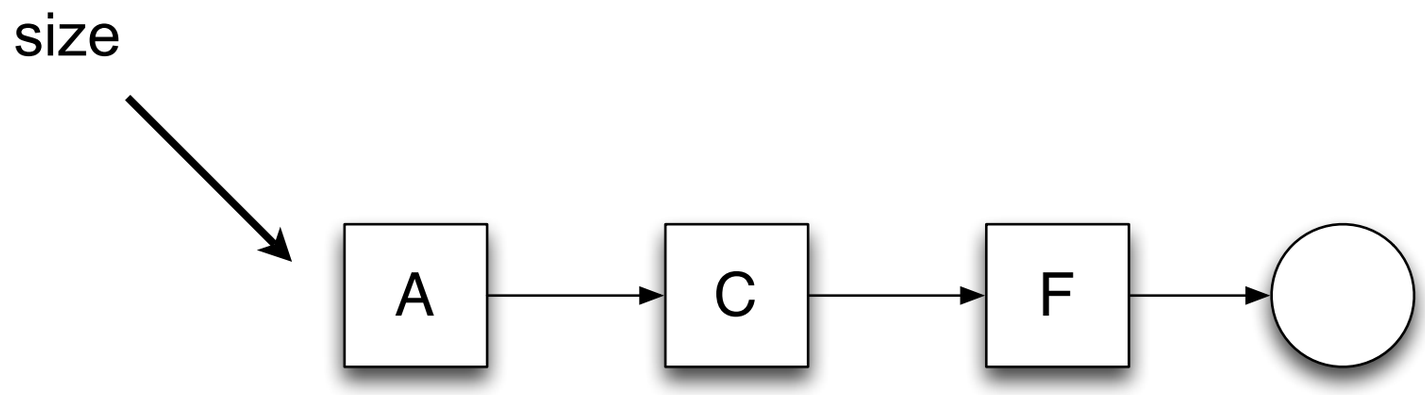
Instance Methods

```
addLast: anObject
   next addLast: anObject

includes: anObject
   value = anObject ifTrue:[^true].
   ^next includes: anObject
```

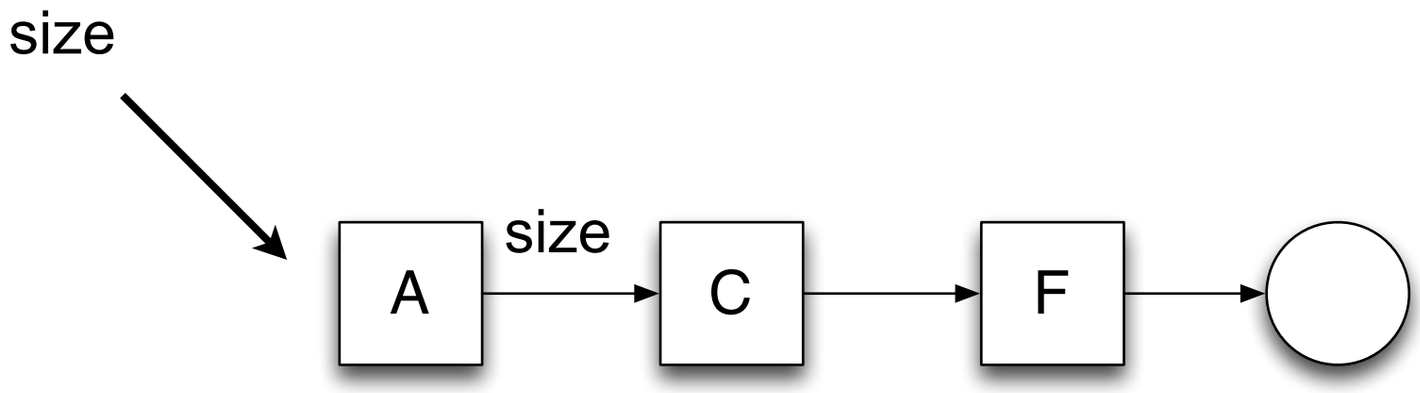
Instance Methods

```
printOn: aStream
   aStream
      print: value;
      nextPutAll: ' '.
   next printOn: aStream

setValue: anObject
   value := anObject.
   next := NilNode new.

size
   ^next size + 1
```

Example - size

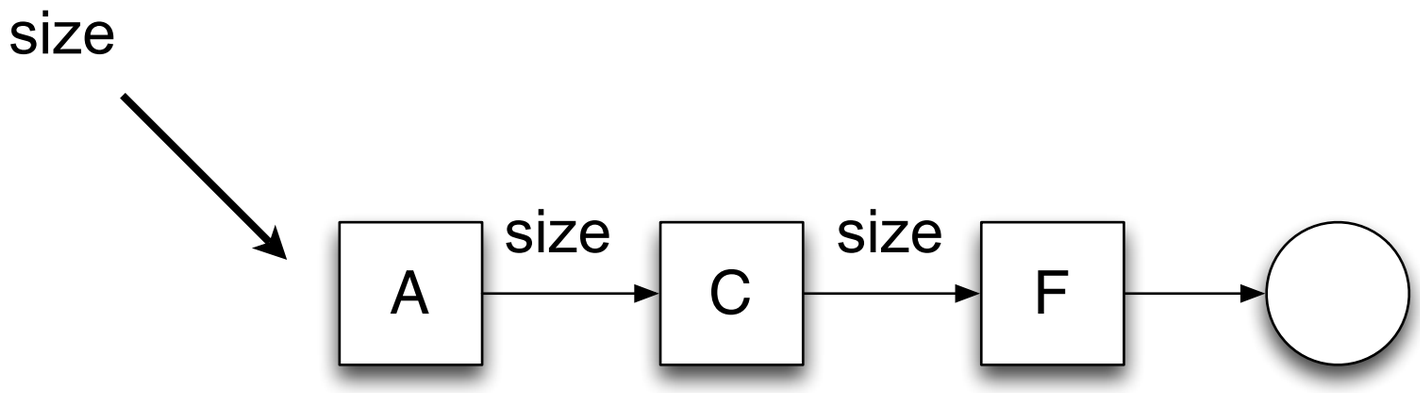


Example - size



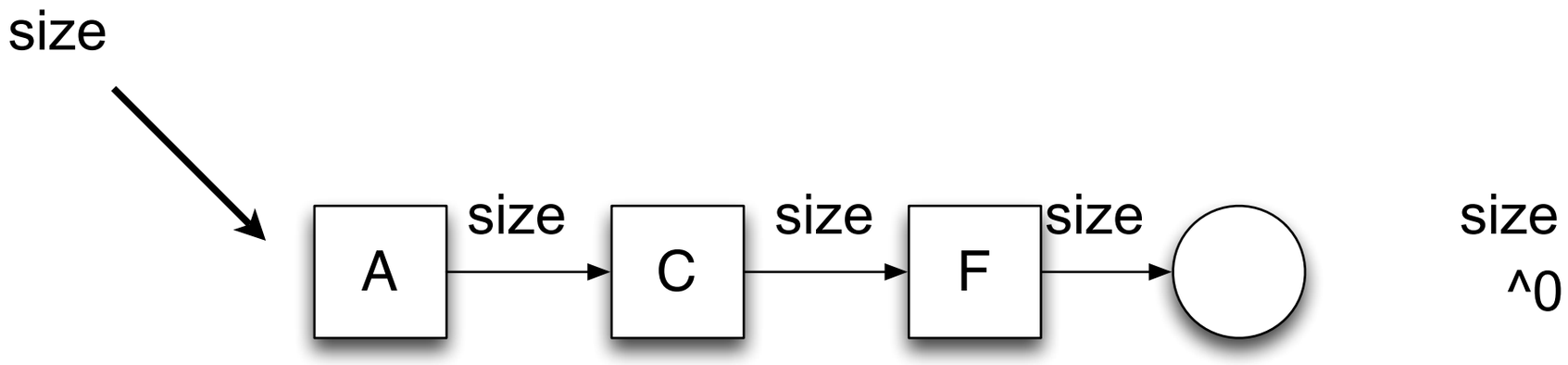

```
size
    ^next size + 1
```

Example - size



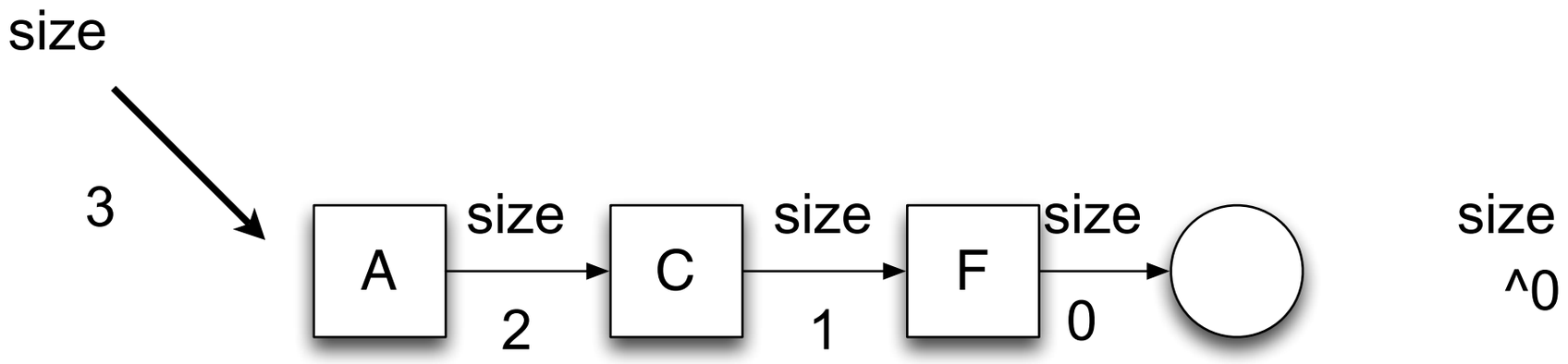

```
size
    ^next size + 1
```

Example - size



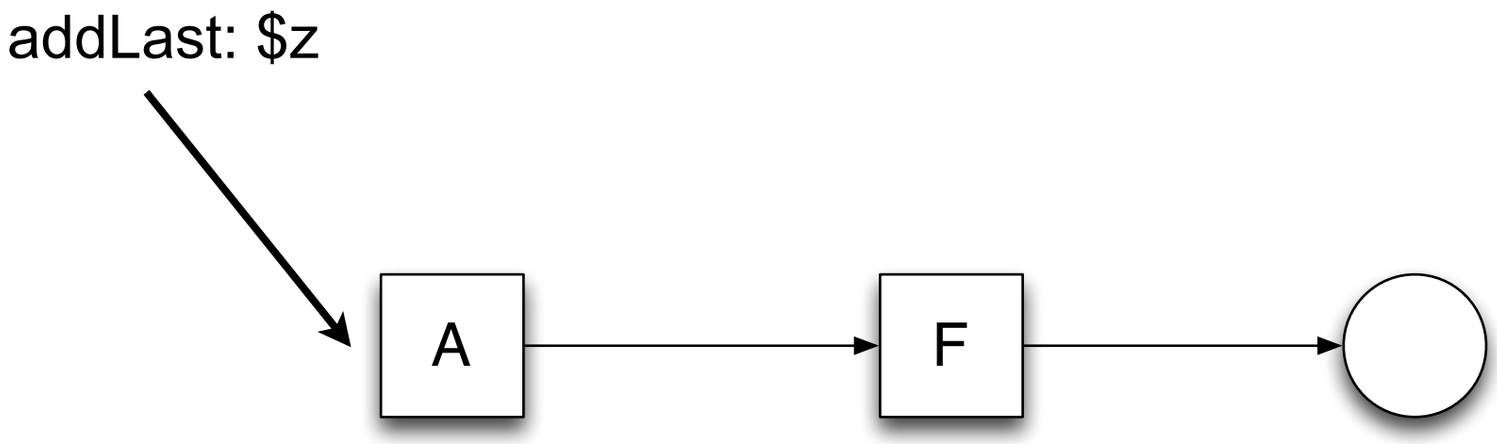

```
size
    ^next size + 1
```

Example - size



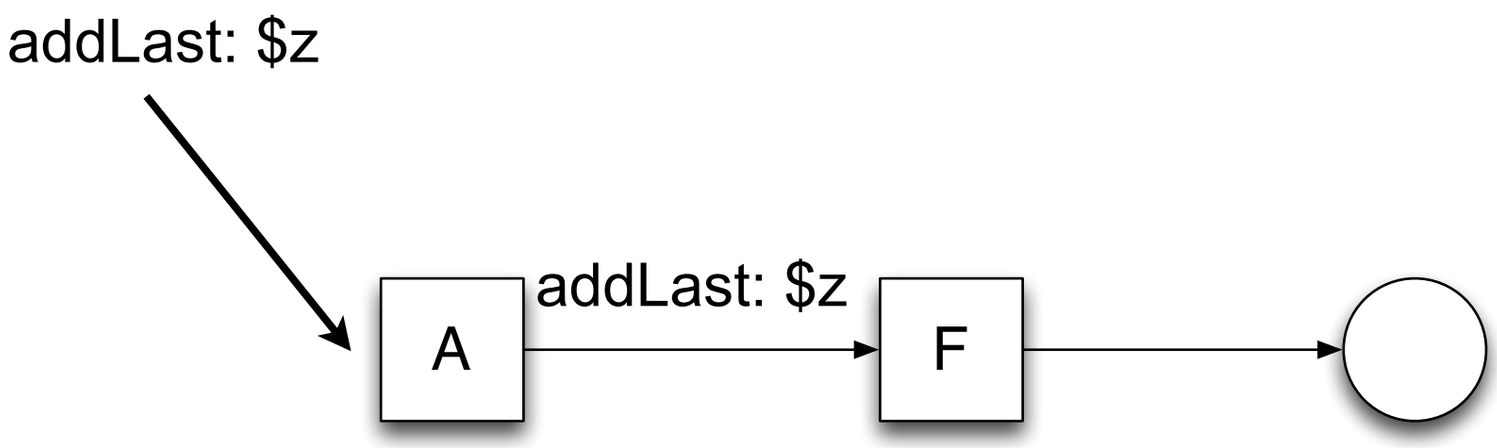

```
size
    ^next size + 1
```

Example - addLast:



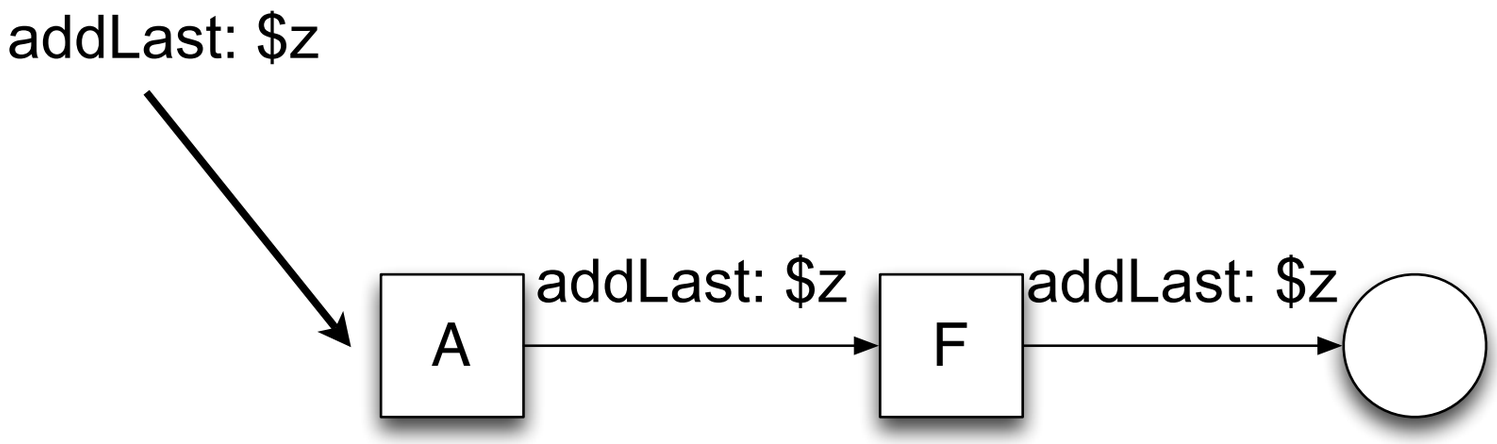

```
addLast: anObject
    next addLast: anObject
```

Example - addLast:




```
addLast: anObject
   next addLast: anObject
```

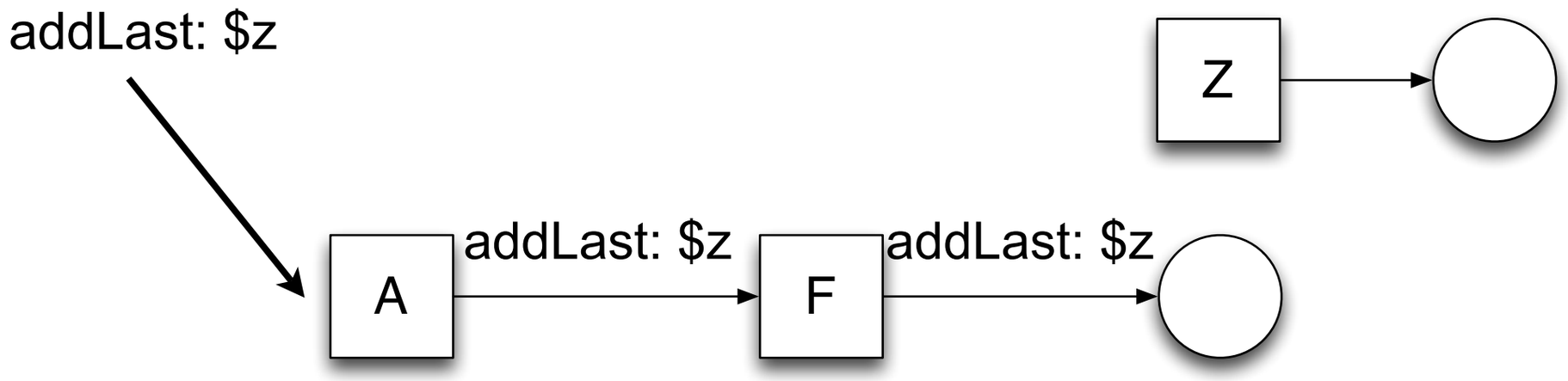
Example - addLast:




```
addLast: anObject
    next addLast: anObject
```

```
addLast: anObject
    self become: (LinkedList with: anObject)
```

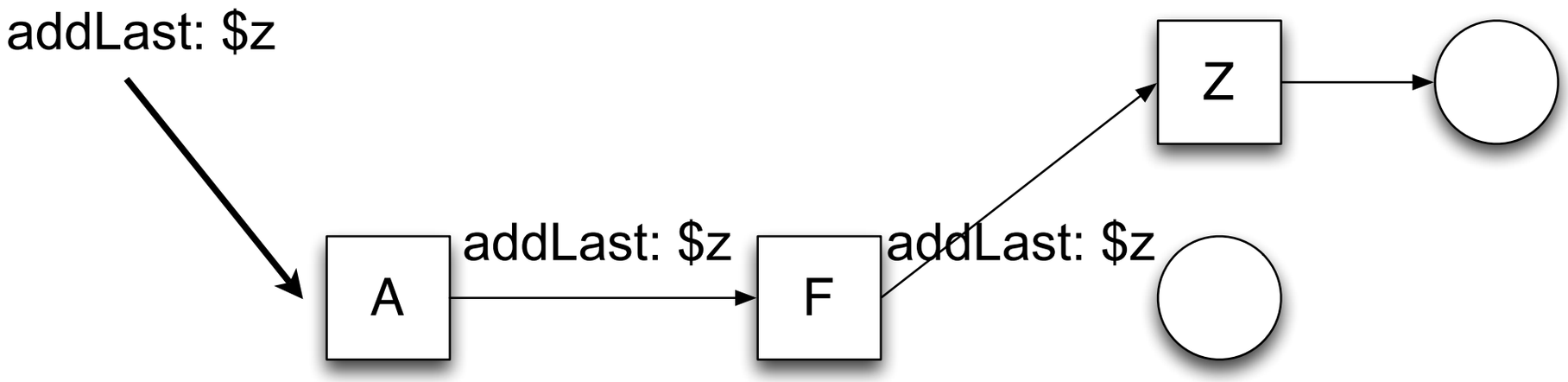
Example - addLast:




```
addLast: anObject
    next addLast: anObject
```

```
addLast: anObject
    self become: (LinkedList with: anObject)
```

Example - addLast:




```
addLast: anObject
    next addLast: anObject
```

```
addLast: anObject
    self become: (LinkedList with: anObject)
```


become: is an unusual operation. After performing "a become: b" all references to a now refer to b and all references to b now refer to a. That is a becomes b and visa-versa. A pointer from the nil node to F would have alleviated the need for using become:. The goal of the entire NilNode example is to provide an example of how to replace case (if) statements with polymorphism. The goal is to send messages to objects and have them do the right thing. The example is rather simplistic but does illustrate the basic idea.

What is wrong here?

```
Transcript
        nextPut: $a;
        print: #( $b $c $d) ;
        nextPutAll: 'cat';
        print: 5
```