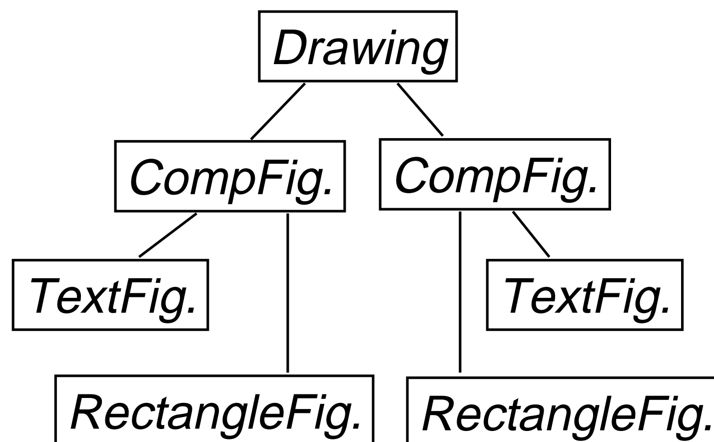
Centralized vs. Decentralized

Interpreter Pattern

Visitor Pattern

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Defining a picture



Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Defining a Language for Pictures

Figure - displayOn: is abstract method
- CompositeFigure
  - Drawing
- RectangleFigure
- TextFigure

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Interpreter Pattern

- Tree of objects is a kind of program
- Interpret program by sending message to root, which is recursively sent to entire tree.

Not about parsing!!!

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Object-Oriented Interpreters

To write a little object-oriented interpreter for a language L:

- 1) make a subclass of LParseNode for each rule in the grammar of L
- 2) for each subclass, define an interpreter method that takes the current context as an argument.

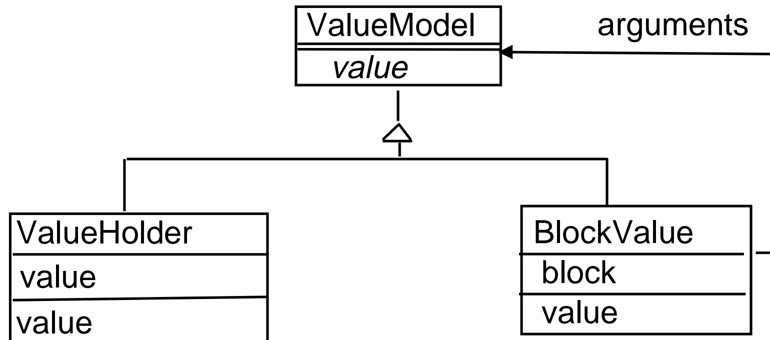
Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Object-Oriented Interpreters

- 3) define protocol for making a tree of LParseNodes.
- 4) define a program for L by building an abstract syntax tree.

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Example: ValueModels



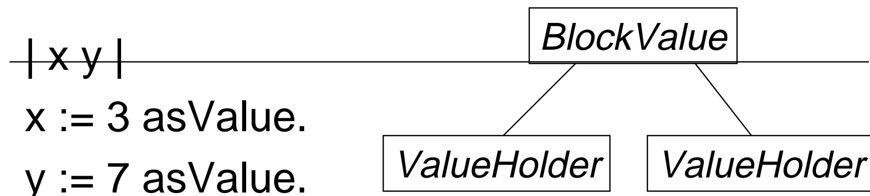
ValueModels

| x y |

x := 3 asValue.

y := 7 asValue.

x + y



ValueModel>> + aValue

^ BlockValue

block: [:a :b | a value * b value]

arguments: (Array with: self with: aValue asValue)

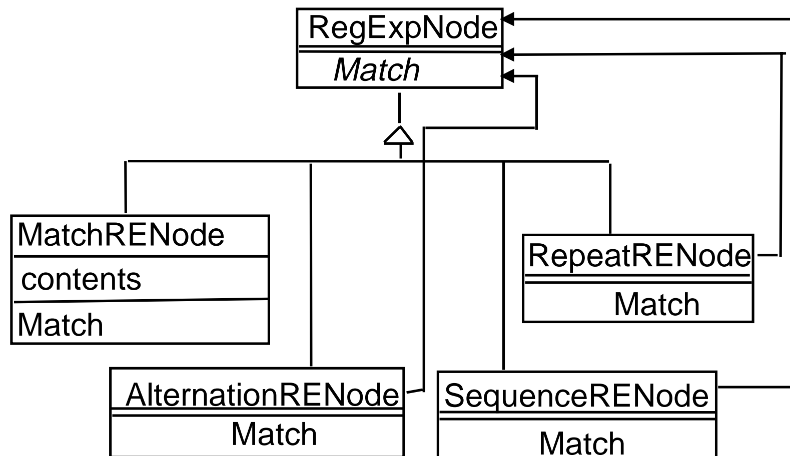
Example: Regular Expression Checker

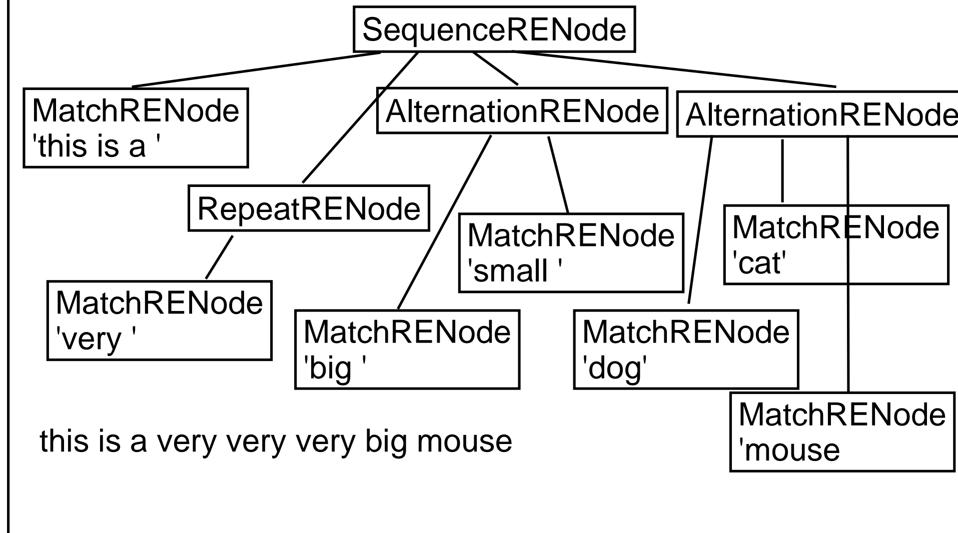
Grammar: exp ::= string | exp '+' exp |
exp '&' exp | exp
'repeat'

Step 1: define RegExpNode, MatchRENode, AlternationRENode, SequenceRENode, RepeatRENode

Step 2: define a **match:** method for each class

Regular Expression Classes





Matching

SequenceRENode>>match: inputState

```
^components
  inject: inputState
  into: [:nextState :exp |
              exp match: nextState]
```

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Matching

```
RepeatRENode>>match: inputState
| aState finalState |
aState :=  inputState.
finalState := inputState copy.
...
```


(continued)

```
[aState notEmpty]
   whileTrue:
      [aState := component match: aState.
      finalState addAll: aState].
^finalState
```

Matching

AlternationRENode>>match: inputState

```
| finalState |
finalState := REState new.
components do: [ :exp | finalState
addAll: (exp match: inputState)]
^finalState
```


Matching

MatchRENode>>match: inputState

```
| finalState  tStream |
finalState := REState new.
...
```

REState is a collection of streams.

(continued)

```
inputState
        do: [:stream |
   tStream := stream copy.
   (tStream nextAvailable:
        components size) = components
         ifTrue: [finalState add: tStream]].
^finalState
```


Protocol for Building Tree

Define ”+" and "repeat" and ”&" as methods on RegExpNodes and any literals that are to be treated as patterns, such as strings.

Then *((’dog ’ + ’cat ’) repeat & ’weather’) matches: ’dog dog cat weather’* is true.

Interface method

RegExpNode>>matches: anArg

```
| inputState |
inputState := (streamOrCollection isKindOf:
Collection)
    ifTrue: [REState with: (ReadStream on:
streamOrCollection)]
    ifFalse: [REState  with:
streamOrCollection].
(self match: inputState) do: [:stream | stream
```


Interface methods

Define +, &, *repeat*, and *asRENode* in RegExpNode and String

+ anArg

```
^AlternationRENode new
  components: (Array with: self with:
  anArg asRENode)
```

Other examples of Interpreter pattern:

- • producing Postscript for a document
- • figuring out the value of an insurance policy
- • spreadsheet
- • compiling a program

C program would use case statement

Replacing Cases with Subclasses

Advantages

- • instead of modifying case statements, add a new subclass
- • easier to parameterize
- • can use inheritance to make new options

Replacing Cases with Subclasses

Disadvantages

- • program is spread out,
  - + harder to understand
  - + harder to replace algorithm
- • state of object can change, but class can not

How To Centralize Algorithms

Define isAlternative, isRepeat, isSequence, isMatch, childrenDo:

fix: anRENode

```
anRENode isMatch
    ifTrue: [anRENode contents:
(anRENode contents capitalize)]
    ifFalse: [anRENode childrenDo: [:child
| self fix: child]]
```

When to Centralize Algorithm

Use centralized algorithm when you need to

- • change entire algorithm at once
- • look at entire algorithm at once
- • change algorithm, but not add new classes of components

Visitor pattern

Visitor lets you centralize algorithm, lets you create a family of algorithms by inheritance, and makes it easy to create new algorithms.

Major problem is that adding a new kind of parse node requires adding a new method to each visitor.

Visitor pattern

- • two kinds of classes: nodes and node visitor
- • nodes have *traverse* method with visitor as an argument
- • traverse method sends *handle* message

SequenceRENode>>traverse: aVisitor

- aVisitor handleSequence: self

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Visitor pattern

- each node class sends a different handle message to visitor

handleSequence:, handleAlternation:, handleRepetition:, handleMatch:

- visitor defines a handle method for each class of parse tree node
- uses double dispatching

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

REVisitor and MatchVisitor

An REVisitor implements handleSequence:, handleAlternation:, handleRepetition:, handleMatch:

MatchVisitor is an REVisitor with one instance variable, *state*, and methods to access it.

MatchVisitor>>handleSequence: sequence

```
^state := sequence components
  inject: state
  into: [:nextState :exp |
      visitor := MatchVisitor with:
  nextState.
      exp match: visitor.
      visitor state]
```

MatchVisitor>>handleAlternation: aNode

```
| initialState finalState |
initialState := state.
finalState := REState new.
components do: [ :eachComponent |
    state := initialState.
    eachComponent match: self.
    finalState addAll: state]
state := finalState
```


MatchVisitor>>handleRepeat: aNode

```
| finalState |
finalState := state copy.
[state notEmpty]
  whileTrue:
      [component traverse: self.
      finalState addAll: state].
^finalState
```

FixVisitor

FixVisitor>>handleMatch: aMatch

aMatch contents: (aMatch contents capitalize)

FixVisitor>>handleSequence: aSequence

aSequence
  childrenDo: [:child | child traverse: self]]

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Visitor Pattern

Strengths

- Can add a new algorithm to operate on a class hierarchy without changing existing classes.

Weakness

- Adding a new node class requires changing visitors.

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

ParseTreeEnumerator

Smalltalk compiler parse tree hierachy rarely changes.

Many programs need to traverse tree:

- • consistency checkers
- • program transformation
- • metrics

Keep them out of the parse tree.

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

VisualWorks GUI Specs

A tree of Specs describes the components of a window.

You build a window for an application by asking the Spec for that window to do it.

UIPolicy is a Visitor that converts a tree of Specs into a tree of VisualComponents.

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

dispatchTo: policy with: builder

ActionButtonSpec

policy actionButton: self into: builder

CompositeSpec

policy composite: self into: builder

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson

Summary

Two related patterns:

Interpreter - distribute code over class hierarchy

Visitor - centralize code in single class

Object-oriented Programming and Design - Copyright 1998 by Ralph E. Johnson