Smalltalk

First Examples

```
hello
    Transcript show: 'Hi World'

hello5
    1 to: 5 do: [:i | (Transcript show: 'Hi World') cr]

hello: times
    1 to: times do: [:i | (Transcript show: 'Hi World') cr]
```

Variables and Parameters

```
hello: times say: text
    "Prints the text (several times) in Transcript window"
    (times > 100)
        ifTrue: [ Transcript show 'You will get bored!']
        ifFalse: [1 to: times do:
                        [:i | (Transcript show: text) cr]]

hellox2: times say: text
    | timestwo |
    timestwo := 2 * times.
    self hello: timestwo say: text
```

Simple Syntactic Elements

name: uppercase first letter only for class variables
and global variables (including classes), otherwise lowercase

comment: in double quotes ("comment")

string: in single quotes ('x', 'x''', '')

character: following $ ($x, $3, $<, $$)

symbol: following # (#x, #'x', #do:, #ifTrue:ifFalse:)

constant array: #(1 2 + 3), #(1 2 (3 #(4)) 5)

number: 12, 3.14e-10, 2r101, 8r177, 16rFF, 2r1.1e2

Messages

assignment: var := expr or var ← expr (← = _)

pseudo variables: nil, true, false, self, super, thisContext

unary message: 1.5 tan rounded (= (1.5 tan) rounded)

binary message: 3 + 4 * 5 (= 35, strictly left to right)

3 + 4 factorial (= 27, stronger binding for unary message)

(4/3) * 3 = 4 (= true, accurate representation of fractions)

(3/4) == (3/4) (= false, different objects)

keyword message: 1 to: 3 do: b (sends #to:do: to 1)

(1 to: 3+4) do: b (#to: with 7 to 1, #do: to result)

Composing Expressions

expression sequence: box ← 20@30 corner: 60@90. "dot necessary!"
box containsPoint: 40@50. "dot optional at the end"

expression cascade: receiver unary; +23; at: 23 put: value

block: [1. 2. 3], [: p1 p2 | p1+p2], [: p || v | v←p*2. v+p]

evaluation of block: aBlock value (block without arguments)
b value: a. b value: a1 value: a2 (up to 4 arguments)
b valueWithArguments: anArray (arguments in array)

answer expression: ↑ 2+3 (↑ = ^, terminates method)

Important Predefined Methods

conditional execution: messages to aBoolean:
#ifTrue:, #ifFalse:, #ifTrue:ifFalse, #ifFalse:ifTrue:

messages to arbitrary objects, parameters are blocks:
#ifNil:, #ifNotNil:, #ifNil:ifNotNil:, #ifNotNil:ifNil:

iterations: receivers and parameters are blocks:
#whileTrue, #whileTrue:, #whileFalse, #whileFalse:

counting iterations: messages to anInteger:
#timesRepeat:, #to:do:, #to:by:do:

message to aCollection: #do:

arguments following do: are Blocks with 1 parameter
other arguments are integers

Brace Arrays and Classes

brace array: array with dynamically computed values:

{1. 2. 3}, {$a. #brace. array}, {1 + 2}

selection: aSymbol caseof: {[#a] -> [1]. ['b' asSymbol] -> [2]}

aSymbol caseof: {[#a] -> [1]. [#b] -> [2]} otherwise: [3]

class definition:

```
SuperClass subclass: #NameOfClass
    instanceVariableNames: 'instVarName1 instVarName2'
    classVariableNames: 'ClassVarName1 ClassVarName2'
    poolDictionaries: ''
    category: 'Major-Minor'
```

Method Definition

```
lineCount
    "Answer the number of lines represented by the receiver
     where every cr adds one line."

    | cr count |
    cr := Character cr.
    count := 1 min: self size.
    self do:
        [:c | c == cr ifTrue: [count := count + 1]].
    ^ count
```

State

The State Design Pattern

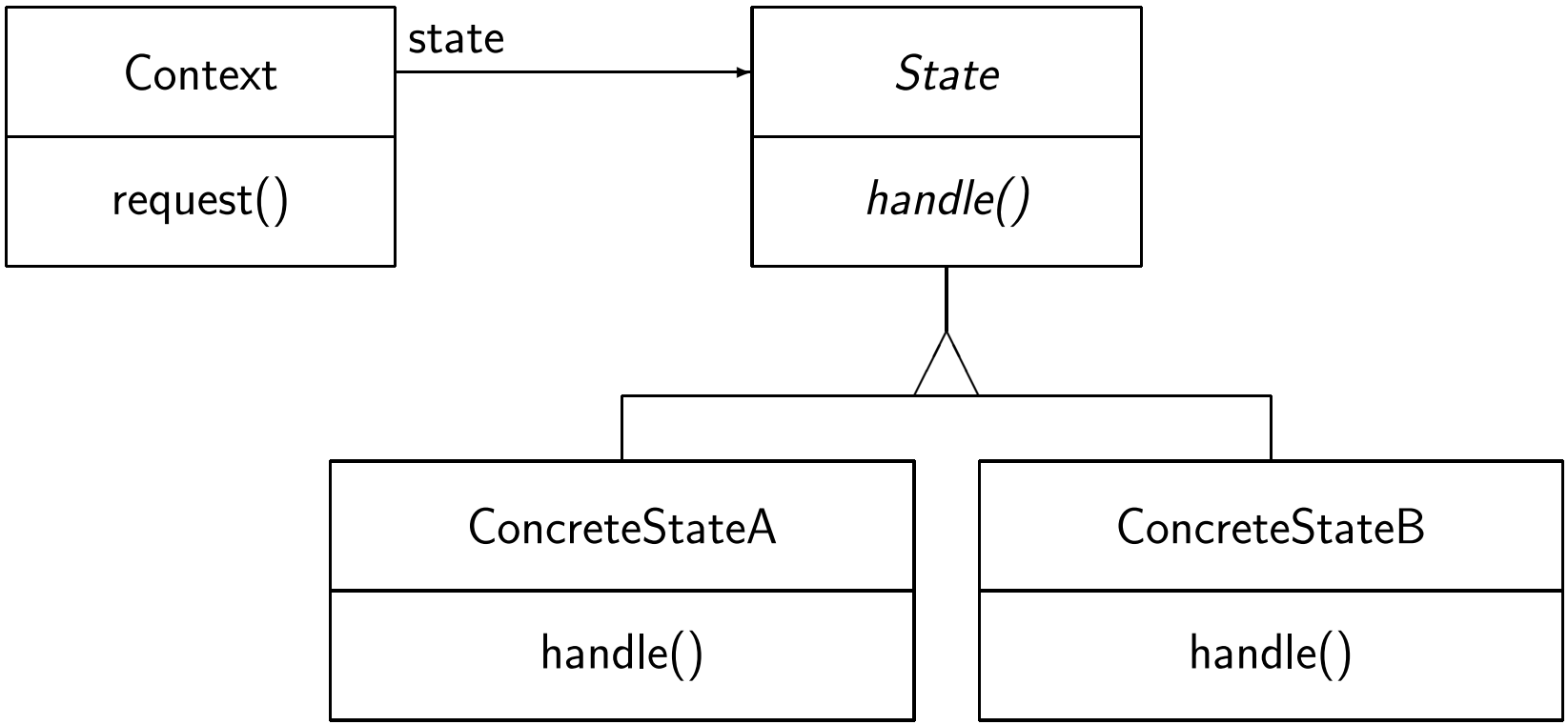
allows an object to change its behaviour if the internal state changes;
seems to change the class

use cases:

behaviour depends on object state and
changes at runtime according to the state

to avoid conditional statements depending on the object state
(often expressed through constants),
each branch of the conditional statement in its own class

Structure


Context
request()
state
State
handle()
ConcreteStateA
handle()
ConcreteStateB
handle()

Consequences

localises state-specific behaviour and separates behaviour in different states

→ easily extensible with states and state transitions,

→ code distributed over many classes

state transitions become explicit

→ all states are self-consistent because of atomic state transitions

state objects can be shared by several contexts

Implementation

who causes/defines state transitions?

Context: subsequent state does not depend on current state,
State: strong dependence between subclasses

often we need only a single object per state (singleton)

state transition table as alternative

focus on state transitions, not on behaviour,
state transition table can be generated (by parser or lexer)

dynamic inheritance as alternative (e.g. in Self)