Class: **Machine**

Superclass: Model
Category: Adventure-State Machines
Instance variables: name states currentState

I'm a state machine. I am similar to the state machines described in "Object-Oriented Modeling and Design" by Rumbaugh et al (ISBN 0-13-629841-9). See my class protocol "examples".

To allow tracing of transitions, I do a 'self changed: #transition with: anArray' on each transition. anArray contains the following parameters, in order:
1 self
2 my state before the transition
3 my new state, or nil if no state change is involved. Note that this parameter may be equal to parameter 2.
4 the event that triggered the transition. This is a kindOf: MachineEvent.

By default, my initial currentState is the first state added.

To execute the entryBlocks of my current state (presumably the state I was initialized with), send me #boot. This removes the special case of the first state I'm in; sending me #boot executes the state entry block just like every other time I "enter" the state.

Instance Variables:
name <Symbol> Name of the state machine. Used for trace output.
states <IdentityDictionary> Dictionary of the machine's states. Indexed with state name symbols, so states must have unique names within a machine.
currentState <MachineState> My current state.

events

event: anEvent
"Deliver anEvent to the current state. Answer
the value of the transition's actionBlock, if any."

```
^self currentState event: anEvent from: self
```

adding

add: aState
"Add aState to the reciever. If this is the first state added, make it the
receiver's currentState. Answer aState."

```
self states isEmpty ifTrue: [self currentState: aState].
(self states includesKey: aState name)
    ifTrue: [self error: 'State names must be unique'].
self states add: aState name -> aState.
^aState
```

accessing

```
currentState
    ^currentState

currentState: aState
    currentState := aState

name
    ^name

name: aSymbol
    name := aSymbol

states
    ^states isNil
        ifTrue: [states := IdentityDictionary new]
        ifFalse: [states]

states: anIdentityDictionary
    states := anIdentityDictionary
```

states

```
atState: aSymbol
    "Answer the state named aSymbol"

    ^self states at: aSymbol
```

printing

```
printDetailsOn: aStream
    aStream nextPutAll: self name printString; space

printOn: aStream
    super printOn: aStream.
    aStream nextPut: $(.
    self printDetailsOn: aStream.
    aStream nextPut: $)
```

transitions

```
transitionTo: nextStateOrNil triggeredBy: anEvent
    "Make a transition to the nextStateOrNil. If we don't change states, then
    execute the current state's reentryBlock. anEvent is useful for tracing."

    self changed: #transition with: (Array
            with: self
            with: self currentState
            with: nextStateOrNil
            with: anEvent).
    nextStateOrNil notNil & (nextStateOrNil ~~ self currentState)
        ifTrue:
            [self currentState exitBlocks value: self.
            self currentState: nextStateOrNil.
            self currentState entryBlocks value: self]
        ifFalse: [self currentState reentryBlocks value: self]
```

booting

boot
"Assume that I'm in my initial state. Simulate normal entry into this state
be executing its entryBlocks."

self currentState entryBlocks value: self

utilities

terselyTraceTransitionFrom: fromState to: toState triggeredBy: anEvent on: aStream
"If the transition caused a state change, trace a transition from fromState
to state toState caused by anEvent on aStream."

```
toState notNil & (fromState ~~ toState) ifTrue: [self
        traceTransitionFrom: fromState
        to: toState
        triggeredBy: anEvent
        on: aStream]
```

traceTransitionFrom: fromState to: toState triggeredBy: anEvent on: aStream
"Trace a transition from fromState to state toState caused by anEvent on
aStream."

```
self name printOn: aStream.
aStream cr; tab; nextPutAll: 'currentState: '.
fromState traceOn: aStream.
aStream cr; tab; nextPutAll: 'event: '.
anEvent traceOn: aStream.
aStream cr; tab; nextPutAll: 'nextState: '.
toState isNil
    ifTrue: [aStream nextPutAll: '(same)']
    ifFalse: [toState traceOn: aStream].
aStream cr; endEntry
```

MetaClass: Machine class

instance creation

named: aSymbol
"Answer an instance of the receiver named aSymbol"

^(self new) name: aSymbol

examples

lightSwitch
"Answer an example state machine."
"
MyMachine := Machine lightSwitch.

```
MyMachine event: (MachineEvent named: #on).
MyMachine event: (MachineEvent named: #off).
MyMachine event: (MachineEvent named: #toggle).
MyMachine := nil.
"

| switch |
switch := (Machine named: #lightSwitch)
	add: (MachineState named: #on);
	add: (MachineState named: #off);
	yourself.

(switch atState: #on)
	addEntryBlock: [:machine |
		Transcript cr; show: 'Entering ' , machine currentState printString];
	addReentryBlock: [:machine |
		Transcript cr; show: 'Re-entering ' , machine currentState printString];
	add: (
		(MachineTransitions forEvent: #on)
			add: (
				MachineTransition new
					actionBlock: [:machine :event |
						Transcript cr; show: 'Light is already on: ' ,
							machine printString , ' ' , event printString]
				)
	);
	add: (
		(MachineTransitions forEvent: #off)
			add: (
				MachineTransition new
					actionBlock: [:machine :event |
						Transcript cr; show: 'Light being turned off: ' ,
							machine printString , ' ' , event printString];
					nextState: (switch atState: #off)
			)
	);
	add: (
		(MachineTransitions forEvent: #toggle)
			add: (
				MachineTransition new
					guardBlock: [:machine :event |
						DialogView confirm: 'Really toggle?'];
					actionBlock: [:machine :event |
						Transcript cr; show: 'Light being toggled off: ' ,
							machine printString , ' ' , event printString];
					nextState: (switch atState: #off)
			)
	).

(switch atState: #off)
	addExitBlock: [:machine |
		Transcript cr; show: 'Exiting ' , machine currentState printString];
	add: (
		(MachineTransitions forEvent: #on)
			add: (
				MachineTransition new
```

```
                    actionBlock: [:machine :event |
                        Transcript cr; show: 'Light being turned on: ' ,
                            machine printString , ' ' , event printString];
                    nextState: (switch atState: #on)
                )
        );
        add: (
            (MachineTransitions forEvent: #off)
                add: (
                    MachineTransition new
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light is already off: ' ,
                                machine printString , ' ' , event printString]
                )
        );
        add: (
            (MachineTransitions forEvent: #toggle)
                add: (
                    MachineTransition new
                        guardBlock: [:machine :event |
                            DialogView confirm: 'Really toggle?'];
                        actionBlock: [:machine :event |
                            Transcript cr; show: 'Light being toggled on: ' ,
                                machine printString , ' ' , event printString];
                        nextState: (switch atState: #on)
                )
        ).

    ^switch
```

Class: **MachineBlockCollection**

Superclass: OrderedCollection
Category: Adventure-State Machines

I exist for the purpose of collecting and then executing blocks of code. I execute the blocks in the order that they were added.

evaluating

value: aParameter
```
self do: [:aBlock | aBlock value: aParameter]
```

MetaClass: MachineBlockCollection class

Class: **MachineEvent**

Superclass: Object
Category: Adventure-State Machines
Instance variables: name

I represent events to be dropped into a state machine. I have a name symbol that identifies me to the machine. The message #description is provided for more complex subclasses to override in order to provide more informative printOn: and state machine trace displays.

Instance Variables:
name <Symbol> My name. This is a symbol because I am inserted in an IdentityDictionary.

accessing

name
```
^name
```

name: aSymbol
```
name := aSymbol
```

printing

printOn: aStream
```
super printOn: aStream.
aStream nextPutAll: '(' , self name printString , ')'
```

traceOn: aStream
```
"Append a representation of myself to transition trace stream aStream."

aStream nextPutAll: self name printString
```

MetaClass: MachineEvent class

instance creation

named: aSymbol
```
"Answer an instance of myself named aSymbol."

^self new name: aSymbol
```

Class: **MachineState**

Superclass: Object
Category: Adventure-State Machines
Instance variables: name transitionSets entryBlocks reentryBlocks exitBlocks

I'm a machine's state.

When my entry, exit, and reentry block collections are executed, the machine's currentState is set to me. These collections are passed [:aMachine |]. Collections of blocks are used instead of single instance variables to allow subclasses of my class to add default entry, exit, and reentry behavior while preserving the ability to add instance-specific behavior.

Instance Variables:
name <Symbol> used to index me in the machine's collection of states. Also used in trace output. This must be a symbol to allow machines to use IdentityDictionary's.
transitionSets <IdentityDictionary> of transition sets indexed by event name.
entryBlocks <MachineBlockCollection> executed when we enter this state.
reentryBlocks <MachineBlockCollection> executed when a transition is taken that leaves the machine's state unchanged (still set to me.)
exitBlocks <MachineBlockCollection> executed when we exit this state.

events

event: anEvent from: aStateMachine
"Process anEvent. Answer the value of the transition's actionBlock"

^(self transitionSets at: anEvent name)
event: anEvent from: aStateMachine

accessing

entryBlocks
^entryBlocks

entryBlocks: aMachineBlockCollection
entryBlocks := aMachineBlockCollection

exitBlocks
^exitBlocks

exitBlocks: aMachineBlockCollection
exitBlocks := aMachineBlockCollection

name
^name

name: aSymbol
name := aSymbol

reentryBlocks
```
^reentryBlocks
```

reentryBlocks: aMachineBlockCollection
```
reentryBlocks := aMachineBlockCollection
```

transitionSets
```
^transitionSets isNil
    ifTrue: [transitionSets := IdentityDictionary new]
    ifFalse: [transitionSets]
```

transitionSets: anIdentityDictionary
```
transitionSets := anIdentityDictionary
```

printing

printOn: aStream
```
super printOn: aStream.
aStream nextPutAll: '(' , self name printString , ')'
```

traceOn: aStream
```
"Append a representation of myself to transition trace stream aStream."

aStream nextPutAll: self name printString
```

adding

add: aTransitionSet
```
"Add aTransitionSet to myself. Answer aTransitionSet."

(self transitionSets includesKey: aTransitionSet eventName)
    ifTrue: [self error: 'Transition Set names must be unique'].
self transitionSets add: aTransitionSet eventName -> aTransitionSet.
^aTransitionSet
```

addEntryBlock: aBlock
```
self entryBlocks add: aBlock
```

addExitBlock: aBlock
```
self exitBlocks add: aBlock
```

addReentryBlock: aBlock
```
self reentryBlocks add: aBlock
```

initialize-release

initialize
```
self entryBlocks: MachineBlockCollection new.
self exitBlocks: MachineBlockCollection new.
self reentryBlocks: MachineBlockCollection new
```

MetaClass: MachineState class

instance creation

named: aSymbol
	"Answer an instance of myself named aSymbol."

	^self new name: aSymbol

new
	^super new initialize

Class: **MachineTransition**

Superclass: Object
Category: Adventure-State Machines
Instance variables: guardBlock actionBlock nextState

I represent a possible transition to another state.

Instance Variables:
guardBlock <BlockClosure | nil> If non-nil, this block is passed [:aMachine :aMachineEvent |]. It answers a Boolean indicating whether the transition should be taken. If guardBlockis nil, the transition is always taken.
actionBlock <BlockClosure | nil> If non-nil, this block is executed if the transition is taken. It is passed [:aMachine :aMachineEvent |].
nextState <MachineState | nil> to change to if I'm taken. If nil, the state is not changed.

transitions

for: aStateMachine triggeredBy: anEvent
"Take the transition represented by the reciever for aStateMachine,
triggered by anEvent. Answer the value of the actionBlock, or nil if there
isn't one. Execute the actionBlock before making the transition."

```
| val |
val := self actionBlock notNil
        ifTrue: [self actionBlock value: aStateMachine value: anEvent]
        ifFalse: [nil].
aStateMachine transitionTo: self nextState triggeredBy: anEvent.
^val
```

accessing

actionBlock
```
^actionBlock
```

actionBlock: aBlockOrNil
```
actionBlock := aBlockOrNil
```

guardBlock
```
^guardBlock
```

guardBlock: aBlockOrNil
```
guardBlock := aBlockOrNil
```

nextState
```
^nextState
```

nextState: aStateOrNil
```
nextState := aStateOrNil
```

printing

printOn: aStream

"Append to the argument, aStream, the elements of the receiver's class and name"

```
super printOn: aStream.
aStream nextPutAll: ' to: ' , self nextState printString
```

MetaClass: MachineTransition class

Class: **MachineTransitions**

Superclass: Object
Category: Adventure-State Machines
Instance variables: transitions eventName

I'm an ordered collection of possible transitions for a given event in a given state. Because of guardBlock's, it makes sense to have several possible transitions for a single event. Transitions are tested (guardBlock is evaluated) in order of addition.

Instance Variables:
transitions <OrderedCollection> of MachineTransition's.
eventName <Symbol> Name of the event I apply to.

adding

add: aTransition
```
"Add aTransition to the end of the transitions so that the order of guard
condition evaluation is the same as the order of adding. This lets the
programmer optimize performance by putting likely-to-succeed
transitions first. Answer self (this makes state machine building more
convenient)."

self transitions addLast: aTransition
```

events

event: anEvent from: aStateMachine
```
"Process anEvent with the event's parameter as the argument to
the guardConditionBlock of the Transition. Answer the value of
the transition's actionBlock"

self transitions do: [:transition | transition guardBlock isNil
        ifTrue: [^transition for: aStateMachine triggeredBy: anEvent]
        ifFalse: [(transition guardBlock value: aStateMachine value: anEvent)
                ifTrue: [^transition for: aStateMachine triggeredBy: anEvent]]]
```

accessing

eventName
```
^eventName
```

eventName: aSymbol
```
eventName := aSymbol
```

transitions
```
^transitions isNil
    ifTrue: [transitions := OrderedCollection new]
    ifFalse: [transitions]
```

transitions: anOrderedCollection

transitions := anOrderedCollection

printing

printOn: aStream
"Append to the argument, aStream, the elements of the
receiver's class and event"

super printOn: aStream.
aStream nextPutAll: '(' , self eventName printString , ')'

MetaClass: MachineTransitions class

instance creation

forEvent: aSymbol
"Return an instance of myself with aSymbol as the event name"

^self new eventName: aSymbol