Class: **Adventure**

Superclass: AdventureSimpleContainer
Category: Adventure-Kernel
Instance variables: startingPlace turnQueue turnInProgress isPaused registeredComponents unregisteredComponents debugMenusEnabled isOver nowhere
Class variables: DependencyCoordinator

I am the central object in an adventure. Notice that I am a kind of AdventureSimpleContainer. I contain every other object in the adventure. I also provice central control and synchronization mechanisms.

My class provides "utilities-" protocols that are used all over the adventure.

Instance Variables:
startingPlace <AdventurePlace> where the player starts at the beginning of the adventure.
turnInProgress <AdventureLockedFlag> Active elements in the game (players included) avoid stepping on each others toes by taking turns. When this is set, a turn is in progress.
turnQueue <AdventureSharedQueue> AdventureRobot's queue up here for the right to compete with the player for the turnInProgress flag.
isPaused <Boolean> I use this to remember if I'm paused or not. When a game is paused, non-players are inhibited from doing anything. isPaused is also useful for the UI to connect to as a dependent to update button labels.
registeredComponents <IdentityDictionary> Every AdventureThing ("component") in me has an id, which may be nil, or a unique Symbol. #registeredComponents provides access to components with a non-nil id. This mechanism is useful for references in code to components that may not exist at compile-time.
unregisteredComponents <OrderedCollection> Holds components with nil id's. This is maintained to satisfy programmer curiosity during debugging.
debugMenusEnabled <Boolean> This controls on a game-wide basis whether AdventureThing's display their menu items for debugging.
isOver <Boolean> UI's can become dependent on this to tell when the game has ended. They can also use it to ask me if I'm over.
*nowhere <AdventureSimpleContainer> that contains AdventureThing's that have been #destroy'ed. This is handy for debugging, and it means that destruction is not a special case, *and* the AdventureThing generates a #disappearanceEvent when it's destroyed. This makes detection of destruction in scripts the same as detection of disappearance.*

Class Variables:
DependencyCoordinator <AdventureDependencyCoordinator> All Adventure's use this dependency coordinator to coordinate their UI dependencies. Because this is in a class variable, BOSS ignores it.

Class Instance Variables:
product <AdventureProduct> that describes me.
saveFileVersionNumber <Integer> The ever-increasing version number of the adventure save file. I provide this for use by the UI.

Subclasses must implement methods for
- *instance protocol*
  - *initialize-release*
    - *build*
- *class protocol*
  - *product*

productAboutText
productCopyright
productCredits
productName
productVersion

connections

startingPlace
```
^startingPlace
```

startingPlace: aPlace
```
startingPlace := aPlace.
self changed: #startingPlace
```

accessing

debugMenusEnabled
```
^debugMenusEnabled isNil
    ifTrue: [debugMenusEnabled := false]
    ifFalse: [debugMenusEnabled]
```

debugMenusEnabled: aBoolean
```
debugMenusEnabled := aBoolean
```

isOver
```
^isOver isNil
    ifTrue: [isOver := false]
    ifFalse: [isOver]
```

isOver: aBoolean
```
isOver := aBoolean.
self changed: #isOver
```

isPaused
```
^isPaused isNil
    ifTrue: [isPaused := true]
    ifFalse: [isPaused]
```

isPaused: aBoolean
```
isPaused := aBoolean.
self changed: #isPaused
```

nowhere
```
^nowhere isNil
    ifTrue: [nowhere := AdventureSimpleContainer
                    id: #AdventureNowhere
                    name: 'Nowhere'
                    description: 'The signpost up ahead reads... ']
    ifFalse: [nowhere]
```

nowhere: aContainer
```
nowhere := aContainer
```

registeredComponents

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

registeredComponents: anIdentityDictionary

```
registeredComponents := anIdentityDictionary
```

turnInProgress

```
^turnInProgress
```

turnInProgress: aLockedFlag

```
turnInProgress := aLockedFlag
```

turnQueue

```
^turnQueue
```

turnQueue: aSharedQueue

```
turnQueue := aSharedQueue
```

unregisteredComponents

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

unregisteredComponents: anOrderedCollection

```
unregisteredComponents := anOrderedCollection
```

initialize-release

build

```
"Subclasses must define #build. It should set
adventure name and description. In must set startingPlace."

self subclassResponsibility
```

initialize

```
"Subclasses wishing to hook into initialization should do so by defining
#build. I set #adventure here last to propagate myself into the
#adventure’s of everything else, and to get everything else to register
itself with me."

super initialize.
self turnQueue: AdventureSharedQueue new.
self turnInProgress: (AdventureLockedFlag initially: false).
self build.
self nowhere container: self.
self
    performAndForward: #adventure:
    with: self
    performFirst: true
```

releaseAll

```
self performAndForward: #release performFirst: false
```

testing

isHeldBy: aContainer

```
	"I'm at the top of the container hierarchy."

	^false
```

registered components

componentAt: anIdSymbol

```
	^self registeredComponents at: anIdSymbol
```

register: aThing

```
	aThing id notNil
		ifTrue:
			[(self registeredComponents includesKey: aThing id)
				ifTrue: [self error: 'Id''s must be unique'].
			self registeredComponents add: aThing id -> aThing]
		ifFalse: [self unregisteredComponents add: aThing]
```

game control

endGame

```
	"End the game. Release myself and tell anyone interested that the game
	is over. This will be of interest to ApplicationModel's that are monitoring
	or controlling some aspect of my operation."

	self pauseAndTerminateAllProcesses.
	self endGameWithoutSynchronization
```

endGameWithoutSynchronization

```
	"End the game without synchronizing with players or actors. This was
	factored out of #endGame to allow a quick exit for programmers when
	the game is hung in #beginTurn during debugging. It has no other valid
	use. I #releaseAll last, since that breaks my dependents; I want #isOver: to
	signal any of my dependents that care about my state."

	self isOver: true.
	self releaseAll
```

executePaused: aBlock

```
	"Execute aBlock, ensuring that the game is paused while it executes. If
	I'm already paused, just execute aBlock. Answer aBlock's value."

	| alreadyPaused val |
	(alreadyPaused := self isPaused) ifFalse: [self pauseAndTerminateAllProcesses].
	val := aBlock value.
	alreadyPaused ifFalse: [self resumeAllProcesses].
	^val
```

pauseAndTerminateAllProcesses

```
	"Pause the game. If already paused, do nothing."

	self isPaused
		ifFalse:
```

```
			[Cursor wait showWhile: [self performAndForward: #pauseAndTerminateProcesses performFirst: true].
			self isPaused: true]
```

resumeAllProcesses

```
	"Resume the game."

	self performAndForward: #resumeProcesses performFirst: true.
	self isPaused: false
```

product

product

```
	^self class product
```

interface opening

postOpenWith: anAdventureControlPanel

```
	"This hook is called by anAdventureControlPanel after it has opened.
	This is a good place for subclasses of my class to add debugging
	hooks, such as sending enableSingleStepping to an actor."
	"super postOpenWith: anAdventureControlPanel."

	^self
```

saving

dependencyCoordinator

```
	^self class dependencyCoordinator
```

postSave

```
	self dependencyCoordinator restoreExternalDependenciesOn: self
```

postSaveAll

```
	self performAndForward: #postSave performFirst: false
```

preSave

```
	self dependencyCoordinator breakExternalDependenciesOn: self
```

preSaveAll

```
	self performAndForward: #preSave performFirst: true
```

save: aPlayer on: aBos

```
	self
		executePaused:
			[self preSaveAll.
			aBos nextPut: aPlayer.
			self postSaveAll]
```

suggestSaveFile

```
	^self suggestSaveFileUsingVersionNumber: self class saveFileVersionNumber
```

suggestSaveFileUsingVersionNumber: anInteger

```
	| serial |
	serial := (anInteger \\ 1000) printString.
```

```
    serial := ('00' copyFrom: 1 to: 3 - serial size)
                , serial.
    ^Filename defaultClass suggest: self name , ' ' , serial , Adventure suggestSaveFileSuffix
```

updateSaveFileSuggestion

```
    self class saveFileVersionNumber: self class saveFileVersionNumber + 1
```

turn synchronization

beginTurnAndTestGameOverWithSpinDelay: aDelay

```
    "Users: Any interaction with the adventure must begin with this. If I have
    to spin, wait aDelay between looks. Answer true if the game is over."

    [self turnGranted]
        whileFalse: [aDelay wait].
    ^self isOver
```

cancelTurnFor: aRequester

```
    "Cancel a queued turn request."

    self turnQueue remove: aRequester
```

endTurn

```
    "Any interaction with the adventure must end with this."

    self changed: #turnInProgress with: false.
    self turnInProgress clear
```

endTurnFor: aRequester

```
    "Robots: Any interaction with the adventure must end with this."

    (self testRightsToCompeteWithUserFor: aRequester)
        ifTrue:
            [self turnQueue next.
            self endTurn]
        ifFalse: [self error: 'Turn queue error']
```

queueTurnFor: aRequester

```
    "Queue a turn request."

    self turnQueue nextPut: aRequester
```

testRightsToCompeteWithUserFor: aRequester

```
    "Test if aRequester is at the head of the queue, and therefore can
    compete with the user for a turn."

    ^self turnQueue peek == aRequester
```

turnGranted

```
    "Test if the turn is granted."

    ^self turnInProgress testAndSet not
        and:
            [self changed: #turnInProgress with: true.
            true]
```

turnGrantedFor: aRequester

```
"Test if the turn is granted. If we’re at the head of the queue, we compete
with the user for the turnInProgress locked flag."

^(self testRightsToCompeteWithUserFor: aRequester)
    and: [self turnGranted]
```

restoring

postRestoreAll

```
self performAndForward: #postRestore performFirst: false
```

postRestorePostOpenAll

```
self performAndForward: #postRestorePostOpen performFirst: true
```

MetaClass: Adventure class

Instance variables: product saveFileVersionNumber

view creation

defaultPlayer

```
^AdventurePlayer
    id: #player
    name: ’Lee Q. Player’
    description: ’A nameless, faceless placeholder of a person.  Not unlike the implementor of this adventure, who was too lazy to provide a more interesting default player via Adventure class>>defaultPlayer.’
```

defaultUserInterface

```
^AdventureControlPanel
```

open

```
"Open an AdventureControlPanel on an instance of myself with a generic
player."

^self defaultUserInterface openOn: self new player: self defaultPlayer
```

product

flushAllProductCaches

```
"self flushAllProductCaches"

self product: nil.
self allSubclasses do: [:sub | sub product: nil]
```

product

```
^product isNil
    ifTrue: [product := AdventureProduct
                name: self productName
                version: self productVersion
                logo: self productLogo
```

```
                    copyright: self productCopyright
                    about: self productAboutText
                    instructions: self productInstructionsText
                    credits: self productCredits]
        ifFalse: [product]
```

product: aProductOrNil

```
    ""
    "self flushAllProductCaches"
    product := aProductOrNil
```

productAboutText

```
    self subclassResponsibility
```

productCopyright

```
    self subclassResponsibility
```

productCredits

```
    self subclassResponsibility
```

productInstructionsText

```
    ""
    "self flushAllProductCaches"

    ^Text fromString:
```

'Playing the Game
An Adventure is played by manipulating the world with the two lists displayed on the control panel. The list at the left displays the room enclosing your player, and the contents of that room. The list on the right displays your Player and what it is carrying. Each list has a menu associated with whatever item is currently selected in that list. You interact with the game through those menus.

Pausing the Game
The adventure is "realtime," meaning that time passes, and things take place, whether or not you do anything. Independent Actors interact with your player during the game. To pause their actions, select the "pause" item of the "Game" menu. Select "resume" under "Game" to resume play.

Saving the Game
A game can be saved to disk for later retrieval. It can be saved at any point, and as many times as you like. While a game is being saved, the game is automatically paused, so there''s no need to pause before saving. To save the game, select the "save as..." menu item under the "Game" menu.

Restoring Saved Games
To play a previously-saved game, select the "open..." item from the Adventure Launcher''s "File" menu, and select the saved game''s file.

Speaking
To make your player say something, select it in the right list. Then click mouse button 2 ("operate") on the right list to bring up your player''s menu. Select the "Say..." menu item. A dialog box will appear. Fill it in and press the "Say" button, or the Return key.

Manipulating Objects
Each object in the Adventure has a context-sensitive menu attached to it. It will display the actions that you as a Player can perform at the moment on the object. It''s not necessary for the player to hold an object in order to manipulate it. The only reason to "grab" something is to carry it with you, or to add its value to your score. If you have enabled "debug menus" (under the "Special" menu), each object''s menu will include game-debugging items below the separator line. These are not normally used during play. If

you know what you''re doing, this is a good way to cheat.

Unlocking Doors and Using Tools
Sometimes you want to use one object on another. For example, you may want to unlock a door with a key. In this case, you must first designate the object that is the subject or target of the operation. To support this, the Adventure has the concept of the Player target. At any given time, the Player is "pointing at" some object in his environment (or nothing.) This "target" determines what object tools apply themselves to. So, for example, to unlock a door:

1. Select the door as the Player target by clicking on it and selecting its "select as target" menu item.
2. Click on the key, and select its "unlock xxx" menu item, where "xxx" specifies the target.

Tips
- The game progresses in realtime, whether you do anything or not. Make liberal use of the "pause" feature.
- Before you try anything risky, save the game. If you make a mistake, you can restore the saved game. '

productLogo
```
""
"self flushAllProductCaches"

^CachedImage on: (Adventure readImageFromBOSSFile: 'chest.bos')
```

productName
```
self subclassResponsibility
```

productVersion
```
self subclassResponsibility
```

testing

isStartable
```
"Redefine in concrete subclasses."

^false
```

utilities-saving

suggestSaveFileSuffix
```
^Filename defaultClass suggest: '.adventure'
```

saving

binaryRepresentationVersion
```
^'Adventure 1'
```

dependencyCoordinator
```
"I keep this in a class variable, because BOSS ignores references in
class variables when writing an instance."
"self dependencyCoordinator: nil"

^DependencyCoordinator isNil
    ifTrue: [DependencyCoordinator := AdventureDependencyCoordinator new]
    ifFalse: [DependencyCoordinator]
```

dependencyCoordinator: aDependencyCoordinatorOrNil

```
DependencyCoordinator := aDependencyCoordinatorOrNil
```

utilities-filing out

fileOutCategories: aCollection on: aFileManager

```
| org |
org := Smalltalk organization.
aCollection do: [:category | org fileOutCategorySource: category asSymbol on: aFileManager]
```

fileOutInitializationOn: aFileManager

```
aFileManager store: 'AdventureLauncher open'
```

fileOutProduct

```
""
"self fileOutProduct"

| fileName default |
default := AdventurePath asFilename constructString: 'CastleMS.st'.
fileName := AdventureWideDialog requestNewFileName: 'File out on:' default: default.
fileName isEmpty ifTrue: [^nil].
self fileOutProductTo: fileName
```

fileOutProductTo: aFileName

```
| stream fileManager categories |
stream := aFileName asFilename writeStream.
fileManager := SourceCodeStream on: stream.
categories := #('Adventure-State Machines' 'Adventure-Kernel' 'Adventure-Acting' 'Adventure-Views'
'Adventure-Toys' 'Adventure-Microsloth' ).
self fileOutCategories: categories on: fileManager.
self fileOutInitializationOn: fileManager.
fileManager close
```

utilities-images

constructImageFileString: aString

```
"Construct and answer the image file string for file aString. If aString is not
absolute, apply AdventurePath to it."

| file |
file := aString asFilename.
file isAbsolute ifFalse: [file := AdventurePath asFilename construct: aString].
^file asString
```

convertPNMFile: pnmFile toBOSSFile: bossFile

```
| im |
im := Image readPNMFile: pnmFile.
(BinaryObjectStorage onNew: bossFile asFilename writeStream)
      nextPut: im; close
```

convertPNMFileToBOSSFile

```
""
"self convertPNMFileToBOSSFile"
```

```
	| input output |
	(input := DialogView request: 'Input (PNM) File:') = '' ifTrue: [^self].
	(output := DialogView request: 'Output (BOSS) File:') = '' ifTrue: [^self].
	self convertPNMFile: input toBOSSFile: output
```

flushAllImageCaches

```
	"Flush all image caches everywhere."
	"self flushAllImageCaches"

	AdventureThing flushAllImageCaches.
	AdventureContainerInspector flushImageCache.
	AdventureControlPanel flushImageCache
```

importPNMFile

```
	"Import a PNM file and save it as a BOSS file."
	"self importPNMFile"

	| file |
	(file := DialogView request: 'File?') = '' ifTrue: [^self].
	self convertPNMFile: (self constructImageFileString: file , '.ppm')
		toBOSSFile: (self constructImageFileString: file , '.bos')
```

readImageFromBOSSFile: aString

```
	"Answer the Image contained in the file aString. If aString does not
	specify an absolute path, use the contents of AdventurePath as the
	directory."

	| bos im fileString |
	fileString := self constructImageFileString: aString.
	Cursor read
		showWhile:
			[bos := BinaryObjectStorage onOld: fileString asFilename readStream.
			im := bos next.
			bos close].
	^im
```

storeClipboardToBOSSFile

```
	""
	"self storeClipboardToBOSSFile"

	| file |
	(file := DialogView request: 'Save to File:') = '' ifFalse: [self storeClipboardToBOSSFile: file]
```

storeClipboardToBOSSFile: aString

```
	(BinaryObjectStorage onNew: aString asFilename writeStream)
		nextPut: Pixmap fromClipboard asImage; close
```

accessing

saveFileVersionNumber

```
	""
	"self saveFileVersionNumber: nil"

	^saveFileVersionNumber isNil
		ifTrue: [saveFileVersionNumber := 0]
		ifFalse: [saveFileVersionNumber]
```

saveFileVersionNumber: anInteger

```
saveFileVersionNumber := anInteger
```

utilities-misc

inspectReferencedAdventures

```
"Collect garbage and inspect any remaining adventures."
"self inspectReferencedAdventures"
"Optional:    self dependencyCoordinator: nil"

ObjectMemory globalGarbageCollect.
self allSubclasses do: [:sub | sub allInstances isEmpty ifFalse: [sub allInstances inspect]]
```

Class: **AdventureAppearanceEvent**

Superclass: AdventureNonPropagatingEvent
Category: Adventure-Kernel

I represent an AdventureThing's appearance in a container.

type

type

```
^#appearanceEvent
```

MetaClass: AdventureAppearanceEvent class

Class: **AdventureBuilder**

Superclass: Object
Category: Adventure-Kernel
Instance variables: adventure components

I'm used by an Adventure to build and connect a series of AdventurePlace's with a series of AdventurePassage's. My function can be done manually, but it's a lot uglier.

I assume that my adventure responds to the messages that correspond to the parameters passed to '#connect: place1Id and: place2Id via: door1Id and: door2Id'. For example, my adventure should return an instance of AdventurePlace when I send it the message #place1Id.

All components being connected must have non-nil, unique id's.

Instance Variables:
adventure <Adventure> that I'm building.
components <IdentityDictionary> I use this to detect the existence of named components, and to detect duplicate (illegal) names.

private-accessing

adventure
```
^adventure
```

adventure: anAdventure
```
adventure := anAdventure
```

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

components: anIdentityDictionary
```
components := anIdentityDictionary
```

connecting

connect: place1Id and: place2Id via: door1Id and: door2Id
```
"Given the id symbols for two places and two doors, ensure that they all
exist and hook them up. I assume that the doors do not yet exist."

self ensureThatPlaceExists: place1Id.
self ensureThatPlaceExists: place2Id.
((self addComponent: (self adventure perform: door1Id))
    container: (self componentAt: place1Id))
    connectWith: ((self addComponent: (self adventure perform: door2Id))
            container: (self componentAt: place2Id))
```

private-connecting

addComponent: aComponent

```
"Add aComponent to me. Answer aComponent."

(self components includesKey: aComponent id)
    ifTrue: [self error: 'Id''s must be unique'].
self components add: aComponent id -> aComponent.
^aComponent
```

createAndAddComponent: idSymbol

```
"Create and add the component with id idSymbol. Assume that my
adventure implements a message that creates and returns the
component, with selector idSymbol. Answer the new component."

^self addComponent: (self adventure perform: idSymbol)
```

ensureThatPlaceExists: idSymbol

```
"Ensure that the place idSymbol exists. If it does not exist, add it. I
assume here that my adventure has a message to add the place with
selector idSymbol."

(self includesComponent: idSymbol)
    ifFalse: [(self createAndAddComponent: idSymbol)
            container: self adventure]
```

includesComponent: idSymbol

```
^self components includesKey: idSymbol
```

component accessing

componentAt: anIdSymbol

```
^self components at: anIdSymbol
```

MetaClass: AdventureBuilder class

instance creation

on: anAdventure

```
^self new adventure: anAdventure
```

Class: **AdventureClosingContainer**

Superclass: AdventureSimpleContainer
Category: Adventure-Kernel
Instance variables: isOpen

I’m a simple container that can be opened and closed.

Instance Variables:
isOpen <Boolean> Determines if I’m open or closed.

opening-closing

close
```
self setIsOpen: false
```

open
```
self setIsOpen: true
```

private

setIsOpen: aBoolean
```
"Private for simplicity-so all calls go through #open and #close. I report a
#contents change since the visibility of my contents (see
#contentsExposed) is treated like any other change in my contents. I
don’t bother to report an #isOpen change because nobody cares."

isOpen := aBoolean.
self changed: #description.
self changed: #contents
```

accessing

additionalDescription
```
"Answer a description of my open/closed state. This may be overridden
by subclasses that lock, for example."

^self isOpen
    ifTrue: [' It is open. ']
    ifFalse: [' It is closed. ']
```

description
```
^super description , self additionalDescription
```

isOpen
```
^isOpen isNil
    ifTrue: [isOpen := false]
    ifFalse: [isOpen]
```

testing

contentsExposed

```
^self isOpen
```

menu items

addContainerMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (self isOpen
        ifTrue: [AdventureMenuItem label: 'close' block: [self close]]
        ifFalse: [AdventureMenuItem label: 'open' block: [self open]])
```

MetaClass: AdventureClosingContainer class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: 'box.bos')
```

Class: **AdventureDeadboltLock**

Superclass: AdventureLock
Category: Adventure-Kernel

My instances emulate a deadbolt lock like that on a front door. They do not automatically latch. An open and locked door or container can’t be closed, for example.

menu items

addOwnerMenuItemsFor: aPlayer to: aCollection

```
self owner isLocked ifFalse: [self owner isOpen
        ifTrue: [aCollection add: (AdventureMenuItem label: ’close’ block: [self owner close])]
        ifFalse: [aCollection add: (AdventureMenuItem label: ’open’ block: [self owner open])]].
super addOwnerMenuItemsFor: aPlayer to: aCollection
```

addPlugMenuItemsFor: aPlayer to: aCollection onBehalfOf: aKey

```
"aKey has delegated menu building to me."

self owner isLocked
    ifTrue: [aCollection add: (AdventureMenuItem label: ’unlock ’ , self owner name block: [self
                    ifCompatibleWith: aKey
                    execute: [self owner unlock]
                    for: aPlayer])]
    ifFalse: [aCollection add: (AdventureMenuItem label: ’lock ’ , self owner name block: [self
                    ifCompatibleWith: aKey
                    execute: [self owner lock]
                    for: aPlayer])]
```

initialize-release

initialize

```
super initialize.
self description: ’key-operated lock’
```

MetaClass: AdventureDeadboltLock class

Class: **AdventureDependencyCoordinator**

Superclass: Object
Category: Adventure-Kernel
Instance variables: roots

I coordinate dependency breaking and restoring for a collection of 0 or more objects. I normally reside in a class variable, and that's how I am accessed by instances. Even though I point to objects (user interfaces, generally) that should not be BOSSed out with my clients ("root objects"), BOSS ignores me because I am held by the class.

Dependents can register themselves with me. A root object can ask me to ask its registered dependents to break their dependencies (breakExternalDependenciesOn: self), and to restore their dependencies (restoreExternalDependenciesOn: self). When a registered dependent no longer has dependency on the root object, it should deregister itself. It's normal to repeatedly register and deregister as dependencies change.

Channel symbols are used to differentiate several dependencies from the same object.

Instance Variables:
roots <IdentityDictionary> of dependency information, keyed by root object.

Data Structure:
AdventureDependencyCoordinator
+ IdentityDictionary
** root object record*
+ key
+ root object
+ value
+ IdentityDictionary
** dependency record*
+ key
+ dependent object
+ value
+ IdentityDictionary
** channel record*
+ key
+ channel name symbol
+ value
AdventureExternalDependent
+ root object
+ dependent object
+ break-dependency selector
+ restore-dependency selector

Examples:
MyADC := AdventureDependencyCoordinator new.
MyADC register: #dependent onChannel: #ch1 asExternalDependentOf: #root.
MyADC register: #dependent2 onChannel: #ch1 asExternalDependentOf: #root2.
MyADC register: #dependent onChannel: #ch1 asExternalDependentOf: #root2.
MyADC register: #dependent onChannel: #ch2 asExternalDependentOf: #root2.

"Use Notifiers and the debugger to verify proper operation."

MyADC breakExternalDependenciesOn: #root.
MyADC breakExternalDependenciesOn: #root2.

MyADC deregister: #dependent onChannel: #ch1 asExternalDependentOf: #root2.
MyADC deregister: #dependent onChannel: #ch2 asExternalDependentOf: #root2.
MyADC deregister: #dependent2 onChannel: #ch1 asExternalDependentOf: #root2.
MyADC deregister: #dependent onChannel: #ch1 asExternalDependentOf: #root.

registration

deregister: aDependent onChannel: cSymbol asExternalDependentOf: anObject

```
    "Deregister aDependent on channel cSymbol. If this was the last dependent of anObject,
    remove our reference to anObject, so it can be garbage-collected if
    appropriate."

    self
        rootsAt: anObject
        dependentsAt: aDependent
        removeKey: cSymbol
```

register: aDependent onChannel: cSymbol asExternalDependentOf: anObject breakSelector: bSymbol restoreSelector: rSymbol

```
    "Register aDependent as an external dependent of anObject. Channel
    cSymbol differentiates different registrations from the same
    aDependent. aDependent promises to respond to selectors bSymbol
    and rSymbol to break and restore dependencies to anObject. These
    are 0-argument selectors."

    | xDependent |
    xDependent := AdventureExternalDependent
                on: anObject
                dependent: aDependent
                breakSelector: bSymbol
                restoreSelector: rSymbol.
    (self rootsAt: anObject dependentsAt: aDependent)
        at: cSymbol put: xDependent
```

rootsAt: anObject dependentsAt: aDependent

```
    "Answer the IdentityDictionary corresponding to the key
    anObject+aDependent. If the dictionary doesn’t exist, create it. This is
    used only for adding with at:put:."

    ^(self roots at: anObject ifAbsent: [self roots at: anObject put: IdentityDictionary new])
        at: aDependent ifAbsent: [(self roots at: anObject)
            at: aDependent put: IdentityDictionary new]
```

rootsAt: anObject dependentsAt: aDependent removeKey: cSymbol

```
    "Remove aDependent/aChannel for anObject. If this was the last
    dependent of anObject, remove our reference to anObject, so it can be
    garbage-collected if appropriate. If our roots are empty, nil them so we
    are back at our initial state."

    ((self roots at: anObject)
        at: aDependent)
        removeKey: cSymbol.
```

```
	((self roots at: anObject)
		at: aDependent) isEmpty ifTrue: [(self roots at: anObject)
			removeKey: aDependent].
	(self roots at: anObject) isEmpty
		ifTrue:
			[self roots removeKey: anObject.
			self roots isEmpty ifTrue: [self roots: nil]]
```

break and restore

breakExternalDependenciesOn: anObject

```
	(self roots at: anObject ifAbsent: [^self])
		do: [:dependents |
			dependents do: [:xDependent |
				xDependent break]]
```

restoreExternalDependenciesOn: anObject

```
	(self roots at: anObject ifAbsent: [^self])
		do: [:dependents |
			dependents do: [:xDependent |
				xDependent restore]]
```

accessing

roots

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

roots: anIdentityDictionary

```
	roots := anIdentityDictionary
```

MetaClass: AdventureDependencyCoordinator class

Class: **AdventureDisappearanceEvent**

Superclass: AdventureNonPropagatingEvent
Category: Adventure-Kernel

I represent an AdventureThing's disappearance from a container.

type

type
    ^#disappearanceEvent

MetaClass: AdventureDisappearanceEvent class

Class: **AdventureDoor**

Superclass: AdventurePassage
Category: Adventure-Kernel
Instance variables: isOpen

I'm a passage that can be opened and closed.

Instance Variables:
isOpen <Boolean> True if I'm open.

accessing

additionalDescription

```
^self isOpen
    ifTrue: [' It is open. ']
    ifFalse: [' It is closed. ']
```

description

```
^super description , self additionalDescription
```

isOpen

```
^isOpen isNil
    ifTrue: [isOpen := false]
    ifFalse: [isOpen]
```

opening-closing

close

```
self setIsOpen: false
```

open

```
self setIsOpen: true
```

menu items

addPassageMenuItemsFor: aPlayer to: aCollection

```
self isOpen
    ifTrue:
        [aCollection add: (AdventureMenuItem label: 'enter' block: [self inhale: aPlayer]).
        aCollection add: (AdventureMenuItem label: 'close' block: [self close])]
    ifFalse: [aCollection add: (AdventureMenuItem label: 'open' block: [self open])]
```

private

basicSetIsOpen: aBoolean

```
isOpen := aBoolean.
self changed: #description
```

setIsOpen: aBoolean

```
"Private for simplicity-so all calls go through #open and #close."
```

```
self basicSetIsOpen: aBoolean.
self mate basicSetIsOpen: aBoolean
```

MetaClass: AdventureDoor class

Class: **AdventureEmergencyExitBar**

Superclass: AdventureProperty
Category: Adventure-Kernel

I function as an emergency exit bar for a locking door.

menu items

addMenuItemsFor: aPlayer to: aCollection onBehalfOf: aThing actingAs: roleSymbol

```
roleSymbol = #owner ifTrue: [aCollection add: (AdventureMenuItem label: ’exit via’ block: [self owner inhale: aPlayer])]
```

initialize-release

initialize

```
super initialize.
self description: ’emergency exit bar’
```

MetaClass: AdventureEmergencyExitBar class

Class: **AdventureEvent**

Superclass: Object
Category: Adventure-Kernel
Instance variables: source

I’m an abstract class for events passed within the adventure.

Instance Variables:
source <Object> The original sender of the event.

Subclasses must implement methods for
accessing
type

accessing

source

```
^source
```

source: anObject

```
source := anObject
```

type

```
"Answer a Symbol corresponding to my type."

self subclassResponsibility
```

initialize-release

initialize

```
^self
```

propagation

afterPassingThrough: aContainer

```
"Answer a version of myself after passing through (either into or out of)
aContainer, or nil if nothing passes through."

^aContainer contentsExposed
    ifTrue: [self]
    ifFalse: [nil]
```

formatting

formattedSource

```
^’[’ , self source name asUppercase , ’]’
```

printing

printOn: aStream

```
    super printOn: aStream.
    aStream nextPutAll: ' source: ' , self source printString
```

MetaClass: AdventureEvent class

instance creation

source: anObject

```
    ^self new source: anObject
```

Class: **AdventureExternalDependent**

Superclass: Object
Category: Adventure-Kernel
Instance variables: root dependent breakSelector restoreSelector

I represent an object that is dependent on an AdventureThing.

Instance Variables:
root <AdventureThing> upon which #dependent is dependent.
dependent <Object> that is dependent upon #root.
breakSelector <Symbol> Selector of the message to send to #dependent to get it to restore its dependency on #root.
restoreSelector <Symbol> Selector of the message to send to #dependent to get it to break its dependency on #root.

accessing

breakSelector
^breakSelector

breakSelector: aSymbol
breakSelector := aSymbol

dependent
^dependent

dependent: anObject
dependent := anObject

restoreSelector
^restoreSelector

restoreSelector: aSymbol
restoreSelector := aSymbol

root
^root

root: anObject
root := anObject

initialize-release

on: aRootObject dependent: aDependent breakSelector: bSymbol restoreSelector: rSymbol
self root: aRootObject.
self dependent: aDependent.
self breakSelector: bSymbol.
self restoreSelector: rSymbol

break and restore

break

"Break dependencies."

```
self dependent perform: self breakSelector
```

restore

"Restore dependencies."

```
self dependent perform: self restoreSelector
```

MetaClass: AdventureExternalDependent class

instance creation

on: aRootObject dependent: aDependent breakSelector: bSymbol restoreSelector: rSymbol

```
^self new
	on: aRootObject
	dependent: aDependent
	breakSelector: bSymbol
	restoreSelector: rSymbol
```

Class: **AdventureHint**

Superclass: AdventureThing
Category: Adventure-Kernel
Instance variables: isShowing

I'm a hint. I can be left in difficult spots to give the player a little help.

Instance Variables:
isShowing <Boolean> True if I am visible to the user.

menu items

addMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (self isShowing
        ifTrue: [AdventureMenuItem label: 'hide' block: [self isShowing: false]]
        ifFalse: [AdventureMenuItem label: 'show' block: [self isShowing: true]]).
super addMenuItemsFor: aPlayer to: aCollection
```

accessing

description

```
^self isShowing
    ifTrue: [super description]
    ifFalse: ['I''m a hint. Use my "show" menu item to see me. There is no penalty for viewing me. ']
```

isShowing

```
^isShowing isNil
    ifTrue: [isShowing := false]
    ifFalse: [isShowing]
```

isShowing: aBoolean

```
isShowing := aBoolean.
self changed: #description
```

initialize-release

initialize

```
super initialize.
self name: 'Hint'.
self isMoveable: false
```

string: aString

```
self description: aString
```

MetaClass: AdventureHint class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: 'hint.bos')
```

instance creation

string: aString

```
^self new string: aString
```

Class: **AdventureInoperableDoor**

Superclass: AdventureDoor
Category: Adventure-Kernel

I’m a door that can’t be opened.

menu items

addPassageMenuItemsFor: aPlayer to: aCollection

```
"Ahhhhahahaha. Ha. There’s nothing you can do. You’re trapped on the
other side of me!!"

^self
```

MetaClass: AdventureInoperableDoor class

Class: **AdventureKey**

Superclass: AdventurePlug
Category: Adventure-Kernel

I'm a key. I'll open a lock if our #shape's match.

testing

type
^#key

MetaClass: AdventureKey class

resources

buildImage
^CachedImage on: (Adventure readImageFromBOSSFile: 'key.bos')

Class: **AdventureLatchingLock**

Superclass: AdventureLock
Category: Adventure-Kernel

I can only be unlocked-and-opened or closed-and-locked. The key unlocks-and opens me. I close-and-lock myself.

menu items

addOwnerMenuItemsFor: aPlayer to: aCollection

```
    "My owner has delegated menu building to me."

    self owner isOpen ifTrue: [aCollection add: (AdventureMenuItem label: 'close and lock'
                block:
                    [self owner close.
                    self owner lock])].
    super addOwnerMenuItemsFor: aPlayer to: aCollection
```

addPlugMenuItemsFor: aPlayer to: aCollection onBehalfOf: aKey

```
    "aKey has delegated menu building to me."

    self owner isLocked ifTrue: [aCollection add: (AdventureMenuItem label: 'unlock and open ' , self
owner name block: [self
                    ifCompatibleWith: aKey
                    execute:
                        [self owner unlock.
                        self owner open]
                    for: aPlayer])]
```

initialize-release

initialize

```
    super initialize.
    self description: 'latching lock'
```

MetaClass: AdventureLatchingLock class

Class: **AdventureLock**

Superclass: AdventureSocket
Category: Adventure-Kernel

I am the abstract superclass for all locks.

menu items

addOwnerDebugMenuItemsFor: aPlayer to: aCollection

```
""
"super addOwnerDebugMenuItemsFor: aPlayer to: aCollection"

self owner isOpen ifFalse: [aCollection add: (AdventureMenuItem
            label: 'unlock and open'
            block:
                [self owner unlock.
                self owner open]
            type: #debug)].
super addOwnerDebugMenuItemsFor: aPlayer to: aCollection
```

plugType

```
"Answer the type that a compatible plug will present when asking for
menu items."

^#key
```

MetaClass: AdventureLock class

Class: **AdventureLockedFlag**

Superclass: Object
Category: Adventure-Kernel
Instance variables: semaphore flag

I am a Boolean flag wrapped in a Semaphore for controlling access to me. I was created so that the testAndSetTo: feature was available. I am initially #false by default.

Instance Variables:
flag <Boolean> flag that holds my state.
semaphore <Semaphore> that controls access to #flag.

MyLock := AdventureLockedFlag new.
MyLock := AdventureLockedFlag initially: true.
MyLock test.
MyLock set.
MyLock clear.
MyLock testAndSet
MyLock testAndClear

private-accessing

flag
```
^flag
```

flag: aBoolean
```
flag := aBoolean
```

semaphore
```
^semaphore
```

semaphore: aSemaphore
```
semaphore := aSemaphore
```

initialize-release

initialize
```
self semaphore: Semaphore forMutualExclusion.
self clear
```

initially: aBoolean
```
self flag: aBoolean
```

testing

clear
```
self flag: false
```

set
```
self flag: true
```

test

```
| val |
self semaphore wait.
val := self flag.
self semaphore signal.
^val
```

testAndClear

```
^self testAndSetTo: false
```

testAndSet

```
^self testAndSetTo: true
```

testAndSetTo: aBoolean

```
| val |
self semaphore wait.
val := self flag.
self flag: aBoolean.
self semaphore signal.
^val
```

MetaClass: AdventureLockedFlag class

instance creation

initially: aBoolean

```
^self new initially: aBoolean
```

new

```
^super new initialize
```

Class: **AdventureLockingContainer**

Superclass: AdventureClosingContainer
Category: Adventure-Kernel
Instance variables: isLocked

I'm a closing container that can be locked.

Instance Variables:
isLocked <Boolean> True if I'm locked.

accessing

additionalDescription

```
^(self isOpen
    ifTrue: [' It is open']
    ifFalse: [' It is closed'])
    , (self isLocked
            ifTrue: [' and locked. ']
            ifFalse: [' and unlocked. '])
```

isLocked

```
^isLocked isNil
    ifTrue: [isLocked := true]
    ifFalse: [isLocked]
```

menu items

addContainerMenuItemsFor: aPlayer to: aCollection

```
"I delegate all menu-building to my lock. It may query me to build the
proper menu items."

^self
```

locking-unlocking

lock

```
self setIsLocked: true
```

unlock

```
self setIsLocked: false
```

private

setIsLocked: aBoolean

```
"This is private for simplicity-all locking and unlocking must pass through
#lock and #unlock."

isLocked := aBoolean.
self changed: #description
```

MetaClass: AdventureLockingContainer class

Class: **AdventureLockingDoor**

Superclass: AdventureDoor
Category: Adventure-Kernel
Instance variables: isLocked

I’m an door that can be locked.

Instance Variables:
isLocked <Boolean> True if I’m locked.

accessing

additionalDescription

```
^(self isOpen
    ifTrue: [' It is open']
    ifFalse: [' It is closed'])
    , (self isLocked
            ifTrue: [' and locked. ']
            ifFalse: [' and unlocked. '])
```

isLocked

```
^isLocked isNil
    ifTrue: [isLocked := true]
    ifFalse: [isLocked]
```

menu items

addPassageMenuItemsFor: aPlayer to: aCollection

```
self isOpen ifTrue: [aCollection add: (AdventureMenuItem label: 'enter' block: [self inhale: aPlayer])]
```

locking-unlocking

lock

```
self setIsLocked: true
```

unlock

```
self setIsLocked: false
```

private

basicSetIsLocked: aBoolean

```
isLocked := aBoolean.
self changed: #description
```

setIsLocked: aBoolean

```
"This is private for simplicity-all locking and unlocking must pass through
#lock and #unlock."

self basicSetIsLocked: aBoolean.
self mate basicSetIsLocked: aBoolean
```

MetaClass: AdventureLockingDoor class

Class: **AdventureMenuItem**

Superclass: Object
Category: Adventure-Kernel
Instance variables: block label type

I represent a menu item. I have a type that users of me can jam with something when they build me. Later, when they build the actual menu, they can use the type to sort menu items into classes. I default the type to #normal.

Instance Variables:
block <BlockClosure> Block that is executed when the menu item is selected.
label <String> Menu label string.
type <Symbol> Either #normal for normal menu items, or #debug for menu items that are only useful for programmer debugging. Default is #normal.

accessing

block
```
^block
```

block: aBlock
```
block := aBlock
```

label
```
^label
```

label: aString
```
label := aString
```

type
```
^type
```

type: aSymbol
```
type := aSymbol
```

printing

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

initialize-release

label: aString block: aBlock
```
self
    label: aString
    block: aBlock
    type: #normal
```

label: aString block: aBlock type: aSymbol

```
    self label: aString; block: aBlock; type: aSymbol
```

MetaClass: AdventureMenuItem class

instance creation

label: aString block: aBlock

```
    ^self new label: aString block: aBlock
```

label: aString block: aBlock type: aSymbol

```
    ^self new
        label: aString
        block: aBlock
        type: aSymbol
```

Class: **AdventureNonPropagatingEvent**

Superclass: AdventureEvent
Category: Adventure-Kernel

I'm an event that never propagates out of my container.

propagation

afterPassingThrough: aContainer

```
"A never propagate to any container."

^nil
```

MetaClass: AdventureNonPropagatingEvent class

Class: **AdventurePassage**

Superclass: AdventureThing
Category: Adventure-Kernel
Instance variables: mate

A pair of me is used to connect two places. Two objects are used so I don't have to break the rule that an AdventureThing has only one container. Also, in the case of a door, the door can have a different appearance and lock property on each side.

Instance Variables:
mate <AdventurePassage> that I connect to.

connections

connectWith: aPassage

```
self mate: aPassage.
aPassage mate: self
```

accessing

mate

```
^mate
```

mate: aPassage

```
mate := aPassage
```

name

```
"I test my mate here for debugging ease only (so printOn: doesn't
blow up)."

^super name , ' leading ' , (self mate isNil
        ifTrue: ['nowhere']
        ifFalse: [self position pathTo: self mate position])
```

menu items

addMenuItemsFor: aPlayer to: aCollection

```
self addPassageMenuItemsFor: aPlayer to: aCollection.
super addMenuItemsFor: aPlayer to: aCollection
```

addPassageMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (AdventureMenuItem label: 'enter' block: [self inhale: aPlayer])
```

actions

inhale: aPlayer

```
"Make aPlayer enter me."

aPlayer container: self mate container
```

initialize-release

initialize

```
super initialize.
self isMoveable: false
```

MetaClass: AdventurePassage class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: ’door.bos’)
```

Class: **AdventurePlace**

Superclass: AdventureSimpleContainer
Category: Adventure-Kernel

I'm a container that represents a physical place.

Note: I do not need to be set 'isMoveable: false' because movement logic prevents a container from being into its contents. Since a place is the highest-level container accessible to a player, this issue never arises.

connections

setPositionFromContainer

```
"Do nothing because it is inconvenient to have my position
hammered when I am inserted in my container (the adventure) at
adventure build time."

^self
```

events

propagateUpwards: anEvent

```
"Events never propagate out of me."

^self
```

menu items

addDebugMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (AdventureMenuItem
        label: 'transport player'
        block: [aPlayer container: self]
        type: #debug).
super addDebugMenuItemsFor: aPlayer to: aCollection
```

MetaClass: AdventurePlace class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: 'room.bos')
```

instance creation

id: idSymbol name: nameString description: descriptionString position: aPosition

```
^(self new) id: idSymbol; name: nameString; description: descriptionString; position: aPosition
```

Class: **AdventurePlayer**

Superclass: AdventureSimpleContainer
Category: Adventure-Kernel
Instance variables: score target gameIsOver

I represent the physical player in the game.

Instance Variables:
score <Number> My score in whatever units you like.
target <AdventureThing **|** *nil> What my virtual finger is pointing at. I use this to designate the object upon which a tool (gun, key, etc.) should act.*
gameIsOver <Boolean> I set this true when I die, win, or lose. This allows the UI in #closeRequest to distinguish a game-initiated exit from a user-initiated (window menu "close") exit. Changing this doesn't generate an update message because player outcomes (win/lose/die) generate their own update. This value is tested after the fact.

initialize-release

initialize

```
"Make me open so my contents are visible by default in inspectors.
Inspectors that should not display my contents must suppress my
expansion explicitly. isMoveable is set false so I can't be grabbed or
moved-to-target."

super initialize.
self isMoveable: false.
self ear: TextCollector new.
self soundComprehensionTime: 4000
```

accessing

adventure: anAdventure

```
"The test asks if we're part of a new game, or we've been restored from a
saved game. If we're being restored, we want to stay exactly where we
are."

self adventure isNil
    ifTrue:
        [super adventure: anAdventure.
        self container: self adventure startingPlace]
```

gameIsOver

```
^gameIsOver isNil
    ifTrue: [gameIsOver := false]
    ifFalse: [gameIsOver]
```

gameIsOver: aBoolean

```
gameIsOver := aBoolean
```

score

```
^score isNil
    ifTrue: [score := 0]
    ifFalse: [score]
```

score: aValue

```
score := aValue.
self changed: #score
```

connections

container: newContainer

```
super container: newContainer.
(self target notNil and: [(self target isHeldBy: newContainer) not])
    ifTrue: [self invalidateTarget]
```

invalidateTarget

```
self target: nil
```

invalidateTargetIfInaccessible

```
(self target isHeldBy: self container)
    ifFalse: [self invalidateTarget]
```

target

```
^target
```

target: aThingOrNil

```
"Set my target to aThingOrNil. Senders of #target must be prepared to
deal with a nil answer. Set #target before sending #noLongerTargeted
so that any triggered menu rebuild finds that I'm no longer targeting
oldTarget. (Previous sentence is obsolete now that I build menus
dynamically.)"

| oldTarget |
oldTarget := self target.
target := aThingOrNil.
oldTarget notNil
    ifTrue:
        [oldTarget noLongerTargeted.
        oldTarget retractInterestIn: #isExtant for: self.
        oldTarget retractInterestIn: #container for: self].
self target notNil
    ifTrue:
        ["I care if the Thing is destroyed, or it changes containers, Any
        other changes in the thing do not alter its accessibility to me, so I
        don't care about them."
        self target
            expressInterestIn: #isExtant
            for: self
            sendBack: #invalidateTarget.
        self target
            expressInterestIn: #container
            for: self
            sendBack: #invalidateTargetIfInaccessible.
        self target nowTargeted].
self changed: #target
```

actions

eat: aFood

```
"Destroy aFood before me because, if it’s contained in me, then it will be
destroyed when I kill myself, and again (highly illegal) when I
explicitly destroy it here."

aFood transferValueTo: self.
aFood destroy.
aFood isDeadly
    ifTrue: [self die]
    ifFalse: [self say: ’Yum.  I just ate a ’ , aFood name , ’.’]
```

requestAndSay

```
| sd sentence |
sd := AdventureSayDialog new.
sd open
    ifTrue:
        [sentence := sd sentenceField value.
        sentence isEmpty ifFalse: [self say: sentence]]
```

menu items

addDebugMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (AdventureMenuItem
        label: ’win’
        block: [self win]
        type: #debug); add: (AdventureMenuItem
        label: ’lose’
        block: [self lose]
        type: #debug); add: (AdventureMenuItem
        label: ’die’
        block: [self die]
        type: #debug).
super addDebugMenuItemsFor: aPlayer to: aCollection
```

addMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (AdventureMenuItem label: ’say...’ block: [self requestAndSay]).
super addMenuItemsFor: aPlayer to: aCollection
```

user I/O

warn: aString

```
"Warn the physical player with error message aString."

DialogView warn: aString
```

outcomes

die

```
self outcome: #hasDied
```

lose

```
self outcome: #hasLost
```

outcome: aSymbol

```
self gameIsOver: true.
self destroy.
self changed: aSymbol
```

win

```
self outcome: #hasWon
```

restoring

postRestorePostOpen

```
| heard |
super postRestorePostOpen.
heard := self ear value.
self ear clear.
(self ear) nextPutAll: heard; endEntry
```

MetaClass: AdventurePlayer class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: 'player.bos')
```

Class: **AdventurePlug**

Superclass: AdventureThing
Category: Adventure-Kernel
Instance variables: shape

I'm the abstract class for things that plug into other things, like keys and tokens.

Instance Variables:
shape <Symbol> used to match against sockets when inserting. This is defaulted to my #id.

accessing

shape

```
^shape isNil
    ifTrue: [shape := self id]
    ifFalse: [shape]
```

shape: aSymbol

```
shape := aSymbol
```

testing

isCompatibleWith: aSocket

```
^self shape = aSocket shape
```

MetaClass: AdventurePlug class

Class: **AdventurePosition**

Superclass: Object
Category: Adventure-Kernel
Instance variables: x y z

I'm a position in three-d coordinates. You can ask me what direction it is (in English) to another AdventurePosition.

Instance Variables:
x <Number> X coordinate.
y <Number> Y coordinate.
z <Number> Z coordinate.

printing

printOn: aStream
```
"Append a representation of myself to aStream."

super printOn: aStream.
aStream nextPutAll: '(' , x printString , ' ' , y printString , ' ' , z printString , ')'
```

accessing

x
```
^x isNil
    ifTrue: [x := 0]
    ifFalse: [x]
```

x: aValue
```
x := aValue
```

x: newX y: newY z: newZ
```
x := newX.
y := newY.
z := newZ
```

y
```
^y isNil
    ifTrue: [y := 0]
    ifFalse: [y]
```

y: aValue
```
y := aValue
```

z
```
^z isNil
    ifTrue: [z := 0]
    ifFalse: [z]
```

z: aValue

```
    z := aValue
```

comparing

= aPosition

```
    ^(self x = aPosition x and: [self y = aPosition y])
        and: [self z = aPosition z]
```

arithmetic

+ aPosition

```
    ^AdventurePosition
        x: self x + aPosition x
        y: self y + aPosition y
        z: self z + aPosition z
```

- aPosition

```
    ^AdventurePosition
        x: self x - aPosition x
        y: self y - aPosition y
        z: self z - aPosition z
```

pathTo: aPosition

```
    "Finish the statement that reads 'the path from me to aPosition leads...'"

    | delta yName xName zName |
    delta := aPosition - self.
    delta = AdventurePosition zero ifTrue: [^'here'].
    yName := #('south' '' 'north' ) at: ((delta y max: -1)
                    min: 1)
                    + 2.
    xName := #('west' '' 'east' ) at: ((delta x max: -1)
                    min: 1)
                    + 2.
    zName := #('down' '' 'up' ) at: ((delta z max: -1)
                    min: 1)
                    + 2.
    (yName size + xName size > 0 and: [zName size > 0])
        ifTrue: [zName := ' and ' , zName].
    ^yName , xName , zName
```

MetaClass: AdventurePosition class

instance creation

x: x y: y z: z

```
    ^self new
        x: x
        y: y
        z: z
```

zero

```
^self
    x: 0
    y: 0
    z: 0
```

Class: **AdventureProduct**

Superclass: Object
Category: Adventure-Kernel
Instance variables: name version logo copyright about instructions credits

My instances are relatively passive holders of data about software products.

Instance Variables:
name <String> Product name.
version <String> Version used for display to humans.
logo <CachedImage> 64x64 image.
copyright <String> List of copyright holders.
about <Text> Information "about" the product.
instructions <Text> Instructions on how to use the product.
credits <Collection> Collection of AdventureCredit's.

accessing

about
^about

about: aText
about := aText

copyright
^copyright

copyright: aString
copyright := aString

credits
^credits

credits: aCollection
credits := aCollection

instructions
^instructions

instructions: aText
instructions := aText

logo
^logo

logo: anImage
logo := anImage

name
^name

name: aString
name := aString

version
^version

version: aString
version := aString

MetaClass: AdventureProduct class

instance creation

name: nString version: vString logo: lImage copyright: cString about: aText instructions: iText credits: crString
^(self new) name: nString; version: vString; logo: lImage; copyright: cString; about: aText; instructions: iText; credits: crString

Class: **AdventureProperties**

Superclass: IdentityDictionary
Category: Adventure-Kernel

I'm an IdentityDictionary containing properties. At this point, I exists only so that I can provide an intelligent description of the properties.

description

appendSingularPrefixTo: aString

```
"Answer aString prefixed with 'a ' or 'an '."

^((aString at: 1) isVowel
    ifTrue: ['an ']
    ifFalse: ['a '])
    , aString
```

description

```
"Answer a description of myself."

| result descriptions last |
result := String new.
self isEmpty
    ifFalse:
        [descriptions := OrderedCollection new.
        self do: [:x | x description isEmpty ifFalse: [descriptions add: x description]].
        descriptions isEmpty
            ifFalse:
                [result := 'It has '.
                last := descriptions removeLast.
                descriptions isEmpty
                    ifFalse:
                        [descriptions do: [:y | result := result , (self appendSingularPrefixTo: y) , ', '].
                        result := result , 'and '].
                result := result , (self appendSingularPrefixTo: last) , '. ']].
^result
```

MetaClass: AdventureProperties class

Class: **AdventureProperty**

Superclass: Object
Category: Adventure-Kernel
Instance variables: owner description

I am an invisible property that is associated with an AdventureThing. I add additional behavior to the AdventureThing.

Instance Variables:
owner <AdventureThing> that I'm a part of.
description <String> that describes me. If #description is not empty (''), it becomes part of my thing's description.

Subclasses must implement methods for
menu items
addMenuItemsFor:to:onBehalfOf:actingAs:

initialize-release

initialize
```
^self
```

pause and resume

pauseAndTerminateProcesses
```
"This message is forwarded by my owner."

^self
```

resumeProcesses
```
"This message is forwarded by my owner."

^self
```

accessing

adventure: anAdventure
```
"This message is forwarded by my owner."

^self
```

description
```
^description isNil
    ifTrue: [description := String new]
    ifFalse: [description]
```

description: aString
```
description := aString
```

owner

```
    ^owner
```

owner: aThing

```
    owner := aThing
```

setPosition: newPosition

```
    "This message is forwarded by my owner."

    ^self
```

printing

printOn: aStream

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

menu items

addMenuItemsFor: aPlayer to: aCollection onBehalfOf: aThing actingAs: roleSymbol

```
    "roleSymbol determines the capacity in which aThing is acting. For
    example, a key with properties would ask its properties for menu items
    as #owner, but ask the player target lock property for menu items as a
    #key."

    self subclassResponsibility
```

saving

postSave

```
    "This message is forwarded by my owner."

    ^self
```

preSave

```
    "This message is forwarded by my owner."

    ^self
```

restoring

postRestore

```
    "This message is forwarded by my owner."

    ^self
```

postRestorePostOpen

```
    "This message is forwarded by my owner."

    ^self
```

forwarding

performAndForward: aSymbol performFirst: aBoolean

```
    "I don't have anywhere to forward this."
```

```
    self perform: aSymbol
```

performAndForward: aSymbol performFirst: aBoolean to: aCollection

```
    "Iterate over a copy of the collection in case the perform operation wants
    to alter the collection. If aBoolean is true, do the perform first. Otherwise,
    forward first."

    aBoolean
        ifTrue:
            [self perform: aSymbol.
            aCollection copy do: [:elem | elem performAndForward: aSymbol performFirst: aBoolean]]
        ifFalse:
            [aCollection copy do: [:elem | elem performAndForward: aSymbol performFirst: aBoolean].
            self perform: aSymbol]
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean

```
    "I don't have anywhere to forward this."

    self perform: aSymbol with: aParameter
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean to: aCollection

```
    "Iterate over a copy of the collection in case the perform operation wants
    to alter the collection. If aBoolean is true, do the perform first. Otherwise,
    forward first."

    aBoolean
        ifTrue:
            [self perform: aSymbol with: aParameter.
            aCollection copy do: [:elem | elem
                    performAndForward: aSymbol
                    with: aParameter
                    performFirst: aBoolean]]
        ifFalse:
            [aCollection copy do: [:elem | elem
                    performAndForward: aSymbol
                    with: aParameter
                    performFirst: aBoolean].
            self perform: aSymbol with: aParameter]
```

connections

basicDestroy

```
    "This message is forwarded by my owner."

    ^self
```

actions

basicTransferValueTo: aPlayer

```
    "This message is forwarded by my owner."

    ^self
```

MetaClass: AdventureProperty class

instance creation

description: aString

```
^self new description: aString
```

new

```
^super new initialize
```

Class: **AdventureRandomValueHolder**

Superclass: Object
Category: Adventure-Kernel
Instance variables: available chosen
Class variables: RandomGenerator

Send me #value, and I'll answer one of my contents at random. I can hold anything. I won't ever return a value twice before all other values have been returned. If I'm empty and you ask for my value, I generate an error.

Send me #reset to put me in my initial state. In this state, all values are #available, so I will produce a sequence that appears maximally random.

If the objects consuming me send #value to get my random value, then a simple object such as a String will also suffice (Object implements #value as "^self".)

Instance Variables:
available <OrderedCollection> of values that I haven't chosen yet. When a value is chosen from here, it is moved to chosenValues. The effect is to give a better variety of values than simple random selection.
chosen <OrderedCollection> of values that have been chosen.

Class Variables:
RandomGenerator <Random> number generator shared by all instances.

private-accessing

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

available: aValue
```
available := aValue
```

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

chosen: aValue
```
chosen := aValue
```

randomGenerator
```
^RandomGenerator isNil
    ifTrue: [RandomGenerator := Random new]
    ifFalse: [RandomGenerator]
```

values

reset

```
"A new sequence of values is requested. Make myself my most random.
Without this hook, a sequence of values may appear to repeat more
than necessary because such a sequence may start when my
#available values collection has been partially depleted."

[self chosen isEmpty]
    whileFalse: [self available add: self chosen removeFirst]
```

value

```
self available isEmpty
    ifTrue:
        [| temp |
        temp := self available.
        self available: self chosen.
        self chosen: temp].
^self chosen add: (self available removeAtIndex: (self randomGenerator next * self available size)
truncated + 1)
```

initialize-release

add: aValue

```
"Add aValue to my #available values. Answer aValue."

^self available add: aValue
```

addAll: aCollection

```
"Add aCollection's elements to my #available values. Answer
aCollection."

^self available addAll: aCollection
```

enumerating

do: aBlock

```
"Evaluate aBlock with each of my possible values as the argument."

self reset.
self size timesRepeat: [aBlock value: self value]
```

accessing

size

```
^self available size + self chosen size
```

MetaClass: AdventureRandomValueHolder class

instance creation

withAll: aCollection

```
^(self new) addAll: aCollection; yourself
```

Class: **AdventureRobot**

Superclass: AdventureThing
Category: Adventure-Kernel
Instance variables: process pauseRequestIsPending pauseAcknowledgement pauseEternity singleSteppingEnabled hasPermissionToProceed hasPermissionToStep hasAnnouncedReadinessToStep turnRequest spinDelay

I am an abstract class. The instances of my subclasses represent autonomous AdventureThing's that execute asynchronously with the rest of the adventure.

Instance Variables:
process <Process> that animates me.
pauseRequestIsPending <Boolean> True when I have asked my process to pause.
pauseAcknowledgement <Semaphore> that my process signals to indicate that it has ack'd my request for it to pause.
pauseEternity <Semaphore> that hangs my process permanently during the pause process.
singleSteppingEnabled <Boolean> True if single-stepping is enabled.
hasPermissionToProceed <Boolean> True if my process has been told to proceed and end stepping.
hasPermissionToStep <Boolean> True if my process has been told to take a single step.
hasAnnouncedReadinessToStep <Boolean> True if my process has already signalled (changed:) its readiness to take another step.
turnRequest <AdventureTurnRequest> used to request a turn.
spinDelay <Delay> When my process has to spin waiting for a turn, it uses this Delay.

Subclasses must implement methods for
behavior
behaviorBlock

pause and resume

makeRequestToSynchronizeWithProcess
```
self pauseAcknowledgement: Semaphore new.
self pauseEternity: Semaphore new.
self pauseRequestIsPending: true
```

pauseAndTerminateProcesses
```
"Nil my process so it doesn't come along when I'm BOSSed out."

super pauseAndTerminateProcesses.
self makeRequestToSynchronizeWithProcess.
self waitForAcknowledgementFromProcess.
self resetProcessSynchronization.
self process terminate.
self process: nil
```

resetProcessSynchronization
```
"Nil pauseEternity to avoid references to the process."
```

```
    self pauseEternity: nil.
    self pauseRequestIsPending: false
```

resumeProcesses

```
    super resumeProcesses.
    self process resume
```

synchronizeWithPause

```
    "This MUST BE executed by my process in its behavior loop at a safe
    place to be #terminate'd. Failure to do so will result in a hang when the
    game is saved or paused."

    self pauseRequestIsPending
        ifTrue:
            [self turnRequest cancel.
            self pauseAcknowledgement signal.
            self pauseEternity wait]
```

waitForAcknowledgementFromProcess

```
    self pauseAcknowledgement wait
```

initialize-release

initialize

```
    super initialize.
    self singleSteppingEnabled: false.
    self pauseRequestIsPending: false
```

release

```
    super release.
    self getProcess notNil ifTrue: [self process terminate]
```

accessing

getProcess

```
    ^process
```

hasAnnouncedReadinessToStep

```
    ^hasAnnouncedReadinessToStep
```

hasAnnouncedReadinessToStep: aBoolean

```
    hasAnnouncedReadinessToStep := aBoolean
```

hasPermissionToProceed

```
    ^hasPermissionToProceed
```

hasPermissionToProceed: aBoolean

```
    hasPermissionToProceed := aBoolean
```

hasPermissionToStep

```
    ^hasPermissionToStep
```

hasPermissionToStep: aBoolean

```
    hasPermissionToStep := aBoolean
```

pauseAcknowledgement
```
^pauseAcknowledgement
```

pauseAcknowledgement: aSemaphore
```
pauseAcknowledgement := aSemaphore
```

pauseEternity
```
^pauseEternity
```

pauseEternity: aSemaphore
```
pauseEternity := aSemaphore
```

pauseRequestIsPending
```
^pauseRequestIsPending
```

pauseRequestIsPending: aBoolean
```
pauseRequestIsPending := aBoolean
```

process
```
^process isNil
    ifTrue: [process := self behaviorBlock newProcess]
    ifFalse: [process]
```

process: aValue
```
process := aValue
```

singleSteppingEnabled
```
^singleSteppingEnabled
```

singleSteppingEnabled: aBoolean
```
singleSteppingEnabled := aBoolean
```

spinDelay
```
^spinDelay isNil
    ifTrue: [spinDelay := Delay forMilliseconds: 250]
    ifFalse: [spinDelay]
```

spinDelay: aDelay
```
spinDelay := aDelay
```

turnRequest
```
^turnRequest isNil
    ifTrue: [turnRequest := AdventureTurnRequest on: self adventure]
    ifFalse: [turnRequest]
```

turnRequest: aTurnRequest
```
turnRequest := aTurnRequest
```

behavior

behaviorBlock
```
"Answer a block containing my instance behavior."

self subclassResponsibility
```

single stepping

enableSingleStepping

```
AdventureSingleStepper openOn: self.
self hasPermissionToProceed: false.
self hasPermissionToStep: false.
self hasAnnouncedReadinessToStep: false.
self singleSteppingEnabled: true
```

proceed

```
self hasPermissionToProceed: true
```

requestToStep

```
"Answer whether or not I can take a step."

self singleSteppingEnabled ifFalse: [^true].
self hasAnnouncedReadinessToStep
    ifFalse:
        [self changed: #singleStep.
        self hasAnnouncedReadinessToStep: true].
self hasPermissionToStep
    ifTrue:
        [self hasAnnouncedReadinessToStep: false.
        self hasPermissionToStep: false.
        ^true].
self hasPermissionToProceed
    ifTrue:
        [self singleSteppingEnabled: false.
        ^true].
^false
```

step

```
self hasPermissionToStep: true
```

restoring

postRestorePostOpen

```
"After a user interface has been opened on my after restoration from a
BOSS file, this is sent to me."

super postRestorePostOpen.
self singleSteppingEnabled
    ifTrue:
        [AdventureSingleStepper openOn: self.
        self hasAnnouncedReadinessToStep: false]
```

turn synchronization

beginTurn

```
[self synchronizeWithPause.
self requestToStep ifFalse: [self spinDelay wait]
    ifTrue: [self turnRequest pollForTurn
            ifTrue: [^self]
            ifFalse: [self spinDelay wait]]] repeat
```

endTurn

```
self turnRequest endTurn
```

executeTurn: aBlock

```
"Execute aBlock with turn synchronization. Answer aBlock’s value."

| val |
self beginTurn.
val := aBlock value.
self endTurn.
^val
```

MetaClass: AdventureRobot class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: ’widget.bos’)
```

Class: **AdventureSharedQueue**

Superclass: SharedQueue
Category: Adventure-Kernel

I'm just like a SharedQueue, except that an element can be removed from me.

removing

remove: anObject

```
"Remove and answer anObject."

[accessProtect critical: [contents remove: anObject]] valueUninterruptably.
^anObject
```

MetaClass: AdventureSharedQueue class

Class: **AdventureSimpleContainer**

Superclass: AdventureThing
Category: Adventure-Kernel
Instance variables: contents

I'm the simplest kind of a container. My contents are always visible.

Instance Variables:
contents <OrderedCollection> of AdventureThing's. My contents.

adding

add: aThing

```
"Add aThing to myself and establish my dependencies on it. Do not use
this to place an object in me. Instead, send #container: to the object.
Add aThing at my head for UI convenience. Answer aThing."

self contents addFirst: aThing.
self expressInterestIn: aThing.
self reportContentsChange.
^aThing"By convention."
```

expressInterestIn: aThing

```
aThing
    expressInterestIn: #event
    for: self
    sendBack: #processEventFromContents:.
aThing
    expressInterestIn: #name
    for: self
    sendBack: #reportContentsChange.
aThing isContainer ifTrue: [aThing
        expressInterestIn: #contents
        for: self
        sendBack: #reportContentsChange]
```

removing

remove: aThing

```
"Remove aThing from myself and retract my dependencies on it. Do not
use this to remove an object from me. Instead, send #container: to the
object to move it somewhere else. Answer aThing."

self contents remove: aThing.
self retractInterestIn: aThing.
self reportContentsChange.
^aThing"By convention."
```

retractInterestIn: aThing

```
aThing retractInterestIn: #event for: self.
```

```
	aThing retractInterestIn: #name for: self.
	aThing isContainer ifTrue: [aThing retractInterestIn: #contents for: self]
```

accessing

contents

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

contents: anOrderedCollection

```
	contents := anOrderedCollection
```

value

```
	"Answer my value plus the value of my contents."

	^super value + (self contents inject: 0 into: [:total :thing | total + thing value])
```

weight

```
	"Answer my weight plus the weight of my contents."

	super weight + (self contents inject: 0 into: [:total :thing | total + thing weight])
```

connections

reportContentsChange

```
	"Something in one of my contents has changed, so tell my dependents."

	self changed: #contents
```

events

processEventFromContents: anEvent

```
	"Having received anEvent from my contents via the #event aspect, I
	signal the event with its actual type. This allows my contents to establish
	DependencyTransformer's for the particular events they're interested in."

	self changed: anEvent type with: anEvent.
	self propagateUpwards: anEvent.
	self changed: #debugEvent with: anEvent
```

propagateUpwards: anEvent

```
	"This behavior is factored to allow subclasses to alter their propagation.
	Having received anEvent from my contents. (Previous sentence likely
	BS) I pass it through myself. If there's anything left, I re-signal it to my
	container via the #event aspect."

	| newEvent |
	newEvent := anEvent afterPassingThrough: self.
	newEvent notNil ifTrue: [self changed: #event with: newEvent]
```

testing

canAcceptMoveFrom: aThing

```
	"Answer whether a player can sensibly move aThing into me. Suppress
```

```
a move if I already contain aThing. Allow a move into me if aThing
isHeldBy: me if aThing is not one of my immediate contents."

^(self ~~ aThing container)
& (self ~~ aThing)
& ((self isHeldBy: aThing) not)
& (aThing isMoveable)
& (self contentsExposed)
```

contentsExposed

```
"An ordinary container's contents are always exposed-like a bowl."

^true
```

isContainer

```
^true
```

menu items

addContainerMenuItemsFor: aPlayer to: aCollection

```
"Subclasses may redefine for more interesting container-specific
behavior. This method is meant to be redefined without a call to super.
That way, a subclass can completely replace a superclass's menu
behavior."

^self
```

addDebugMenuItemsFor: aPlayer to: aCollection

```
aCollection add: (AdventureMenuItem
        label: 'spawn'
        block: [AdventureStandaloneContainerInspector openOn: self player: aPlayer]
        type: #debug).
super addDebugMenuItemsFor: aPlayer to: aCollection
```

addMenuItemsFor: aPlayer to: aCollection

```
"#addContainerMenuItemsFor:to: is factored to allow replacement of a
superclass's menu behavior, instead of the appending allowed by
#super."

self addContainerMenuItemsFor: aPlayer to: aCollection.
super addMenuItemsFor: aPlayer to: aCollection
```

initialize-release

initialize

```
super initialize.
self volume: 100
```

forwarding

performAndForward: aSymbol performFirst: aBoolean

```
self
    performAndForward: aSymbol
    performFirst: aBoolean
    to: self contents
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean

```
self
    performAndForward: aSymbol
    with: aParameter
    performFirst: aBoolean
    to: self contents
```

MetaClass: AdventureSimpleContainer class

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: 'box.bos')
```

Class: **AdventureSocket**

Superclass: AdventureProperty
Category: Adventure-Kernel
Instance variables: shape

I am the abstract superclass for things that accept AdventurePlug's, like locks and vending machines.

Instance Variables:
shape <Symbol> used to match against plugs when inserting.

Subclasses must implement methods for
menu items
addPlugMenuItemsFor:to:onBehalfOf:
testing
plugType

accessing

shape
```
^shape
```

shape: aSymbol
```
shape := aSymbol
```

testing

plugType
```
"Answer the type that a compatible plug will present when asking for
menu items."

self subclassResponsibility
```

menu items

addMenuItemsFor: aPlayer to: aCollection onBehalfOf: aThing actingAs: typeSymbol
```
typeSymbol = self plugType
    ifTrue: [self
            addPlugMenuItemsFor: aPlayer
            to: aCollection
            onBehalfOf: aThing]
    ifFalse: [typeSymbol = #owner ifTrue: [self addOwnerMenuItemsFor: aPlayer to: aCollection]]
```

addOwnerDebugMenuItemsFor: aPlayer to: aCollection
```
""
"super addOwnerDebugMenuItemsFor: aPlayer to: aCollection"

^self
```

addOwnerMenuItemsFor: aPlayer to: aCollection
```
""
```

```
	"super addOwnerMenuItemsFor: aPlayer to: aCollection"

	self owner debugMenusEnabled ifTrue: [
		self addOwnerDebugMenuItemsFor: aPlayer to: aCollection]
```

addPlugMenuItemsFor: aPlayer to: aCollection onBehalfOf: aPlug

```
	self subclassResponsibility
```

ifCompatibleWith: aPlug execute: aBlock for: aPlayer

```
	(aPlug isCompatibleWith: self)
		ifTrue: [aBlock value]
		ifFalse: [self issueIncompatiblePlugWarningTo: aPlayer with: aPlug]
```

issueIncompatiblePlugWarningTo: aPlayer with: aPlug

```
	aPlayer warn: aPlug name , ' doesn''t fit ' , self owner name
```

MetaClass: AdventureSocket class

instance creation

shape: aSymbol

```
	^self new shape: aSymbol
```

Class: **AdventureSoundEvent**

Superclass: AdventureEvent
Category: Adventure-Kernel
Instance variables: string volume

I represent a sound. This lame simulation is thanks to the lack of cross-platform sound support.

Instance Variables:
string <String> representing my sound in English. Ex: 'phhhhht'.
volume <AdventureVolume> My conceptual volume (conceptual since I can't make a real sound, even though all platforms support sound. Grrrrr!)

copying

postCopy
```
super postCopy.
self string: self string copy
```

accessing

string
```
^string
```

string: aValue
```
string := aValue
```

volume
```
^volume isNil
    ifTrue: [volume := AdventureVolume normal]
    ifFalse: [volume]
```

volume: aSymbol
```
volume := aSymbol
```

printing

printOn: aStream
```
super printOn: aStream.
aStream nextPutAll: ' string: ' , self string printString
```

formatting

formattedString
```
^self volume formatString: self string
```

attenuation

attenuated
```
"Answer a copy of myself with an attenuated volume."
```

```
	^self copy volume: self volume attenuated
```

type

type

```
	^#soundEvent
```

propagation

afterPassingThrough: aContainer

```
	"Answer a version of myself after passing through (either into or out of)
	aContainer, or nil if nothing passes through."

	| newSound |
	newSound := aContainer contentsExposed
			ifTrue: [self]
			ifFalse: [self attenuated].
	^newSound volume intensity = #silent
		ifTrue: [nil]
		ifFalse: [newSound]
```

MetaClass: AdventureSoundEvent class

instance creation

source: anObject string: aString

```
	"Answer an instance of myself with #normal volume."

	^(self new) source: anObject; string: aString
```

source: anObject string: aString volume: aSymbol

```
	^(self new) source: anObject; string: aString; volume: aSymbol
```

Class: **AdventureSpeechEvent**

Superclass: AdventureSoundEvent
Category: Adventure-Kernel
Instance variables: style

My instances represent speech of some sort. Speech is characterized with a style that modifies its #formattedString.

Instance Variables:
style <Symbol> representing the nature of my speech. May be #normal or #narration.

accessing

style
```
^style isNil
    ifTrue: [style := #normal]
    ifFalse: [style]
```

style: aSymbol
```
style := aSymbol
```

formatting

formattedString
```
self volume intensity = #silent ifTrue: [^String new].
self style = #normal ifTrue: [^super formattedString].
self style = #narration ifTrue: [^'[' , super formattedString , ']'].
self error: 'Invalid style'
```

type

type
```
^#speechEvent
```

MetaClass: AdventureSpeechEvent class

instance creation

source: anObject string: aString style: aSymbol
```
^(self new) source: anObject string: aString; style: aSymbol
```

source: anObject string: aString volume: aVolume style: aSymbol
```
^(self new)
    source: anObject;
    string: aString;
    volume: aVolume;
```

style: aSymbol

Class: **AdventureSwingingDoor**

Superclass: AdventureDoor
Category: Adventure-Kernel

I'm a swinging door. Like my name suggests:
- *I am always closed.*
- *I never change my open/close state.*
- *I always present a menu item that lets a player enter me.*

menu items

addPassageMenuItemsFor: aPlayer to: aCollection
    aCollection add: (AdventureMenuItem label: 'enter' block: [self inhale: aPlayer])

MetaClass: AdventureSwingingDoor class

Class: **AdventureThing**

Superclass: Model
Category: Adventure-Kernel
Instance variables: id name description weight volume value container position image adventure isExtant ear properties isMoveable debugStream haltsOnChange soundTimestamp soundComprehensionTime

I am the simplest kind of visible object in the adventure.

Instance Variables:
id <Symbol | nil> If non-nil, this uniquely identifies me withing the adventure. This feature is used for runtime lookups that would be impossible or inappropriate at Adventure-build-time.
name <String> My name. The UI displays this.
description <String> My description. Ends with a space, by convention (mine.)
weight <Number> How heavy I am in arbitrary units.
volume <Number> How big I am in in arbitrary units.
value <Number> How much I'm worth in arbitrary units.
container <AdventureSimpleContainer> containing me.
position <AdventurePosition> Wherever I am, I'm there.
image <CachedImage | nil> My image is a 64x64 icon. Normally, I get my image from my class. If #image is non-nil, it overrides the class image.
adventure <Adventure> that I'm part of.
isExtant <Boolean> True if I haven't been #destroy'ed.
ear <Stream | nil> If non-nil, I format sound events and send them here.
properties <AdventureProperties> holding my properties, if any.
isMoveable <Boolean> If true, a player can move me around.
debugStream <Stream | nil> If non-nil, various debugging information is sent here.
haltsOnChange <Boolean> If true, each time I send #changed:, I halt.
soundTimestamp <Integer> value of the system clock when I last heard anything.
soundComprehensionTime <Integer> number of milliseconds it takes me to comprehend a sound. This and soundTimestamp are used to throttle output to my ear.

Class Instance Variables:
image <CachedImage | nil> If non-nil, the 64x64 image that applies to my instances.

connections

basicDestroy
```
self container: self adventure nowhere.
self isExtant: false
```

container
```
^container
```

container: newContainer
```
"Set my container and position."

self container notNil
    ifTrue:
```

```
		["container is nil when setting my container for the first time."
		self broadcastEvent: (AdventureDisappearanceEvent source: self).
		self retractInterestInContainer.
		self container remove: self].
	container := newContainer.
	self expressInterestInContainer.
	newContainer add: self.
	self setPositionFromContainer.
	self broadcastEvent: (AdventureAppearanceEvent source: self).
	self changed: #container
```

destroy

```
	self performAndForward: #basicDestroy performFirst: false
```

expressInterestInContainer

```
	self container
		expressInterestIn: #debugEvent
		for: self
		sendBack: #logDebugEvent:.
	self canHear
		ifTrue:
			[self container
				expressInterestIn: #soundEvent
				for: self
				sendBack: #hearEvent:.
			self container
				expressInterestIn: #speechEvent
				for: self
				sendBack: #hearEvent:]
```

retractInterestInContainer

```
	self canHear
		ifTrue:
			[self container retractInterestIn: #soundEvent for: self.
			self container retractInterestIn: #speechEvent for: self.
			self container retractInterestIn: #debugEvent for: self]
```

setPositionFromContainer

```
	"Assume the position of my container. Suppress redundant position
	changes."

	self container position ~= self position ifTrue: [self position: self container position]
```

accessing

adventure

```
	^adventure
```

adventure: anAdventure

```
	"Set my adventure and register myself with anAdventure."

	adventure := anAdventure.
	anAdventure register: self
```

debugStream

```
	^debugStream
```

```
debugStream: aStream
    debugStream := aStream

description
    ^self basicDescription , self properties description

description: aValue
    description := aValue.
    self changed: #description

ear
    ^ear

ear: aStream
    ear := aStream

haltsOnChange
    ^haltsOnChange isNil
        ifTrue: [haltsOnChange := false]
        ifFalse: [haltsOnChange]

haltsOnChange: aBoolean
    haltsOnChange := aBoolean

id
    ^id

id: aValue
    id := aValue

image
    "If I have an instance-specific image, answer it. Otherwise, delegate to
    class."

    ^self getImage isNil
        ifTrue: [self class image]
        ifFalse: [self getImage]

image: anImage
    image := anImage

isExtant
    ^isExtant isNil
        ifTrue: [isExtant := true]
        ifFalse: [isExtant]

isExtant: aBoolean
    isExtant := aBoolean.
    self changed: #isExtant

isMoveable
    ^isMoveable isNil
        ifTrue: [isMoveable := true]
        ifFalse: [isMoveable]
```

isMoveable: aBoolean
```
isMoveable := aBoolean
```

name
```
^name isNil
    ifTrue: [name := String new]
    ifFalse: [name]
```

name: aValue
```
name := aValue.
self changed: #name
```

position
```
^position isNil
    ifTrue: [position := AdventurePosition x:0 y: 0 z: 0]
    ifFalse: [position]
```

position: newPosition
```
self
    performAndForward: #setPosition:
    with: newPosition
    performFirst: true
```

properties
```
^properties isNil
    ifTrue: [properties := AdventureProperties new]
    ifFalse: [properties]
```

properties: aDictionary
```
properties := aDictionary
```

setImageFromBOSSFile: filenameString
```
self image: (CachedImage on: (Adventure readImageFromBOSSFile: filenameString))
```

setPosition: newPosition
```
position := newPosition.
self changed: #position
```

soundComprehensionTime
```
^soundComprehensionTime isNil
    ifTrue: [soundComprehensionTime := 0]
    ifFalse: [soundComprehensionTime]
```

soundComprehensionTime: anInteger
```
soundComprehensionTime := anInteger
```

soundTimestamp
```
^soundTimestamp isNil
    ifTrue: [soundTimestamp := Time millisecondClockValue - self soundComprehensionTime]
    ifFalse: [soundTimestamp]
```

soundTimestamp: anInteger
```
soundTimestamp := anInteger
```

value
```
^value isNil
```

```
        ifTrue: [value := 0]
        ifFalse: [value]
```

value: aNumber

```
    value := aNumber
```

volume

```
    "The default volume was chosen to allow for objects that are particularly
    big or small."

    ^volume isNil
        ifTrue: [volume := 10]
        ifFalse: [volume]
```

volume: aValue

```
    volume := aValue
```

weight

```
    "The default weight was chosen to allow for objects that are particularly
    heavy or light."

    ^weight isNil
        ifTrue: [weight := 10]
        ifFalse: [weight]
```

weight: aValue

```
    weight := aValue
```

testing

canAcceptMoveFrom: aThing

```
    "Answer whether a player can sensibly move aThing into me."

    ^false
```

debugMenusEnabled

```
    ^self adventure debugMenusEnabled
```

isContainer

```
    ^false
```

isHeldBy: aContainer

```
    "Answer whether I'm contained by aContainer, or its container,
    recursively."

    ^self container == aContainer
        ifTrue: [true]
        ifFalse: [self container isHeldBy: aContainer]
```

type

```
    "Answer the type of thing I am. This is used to query the player target
    properties for menu items."

    ^#thing
```

menu items

addDebugMenuItemsFor: aPlayer to: aCollection

```
    ""
    "super addDebugMenuItemsFor: aPlayer to: aCollection"

    aCollection add: (AdventureMenuItem
            label: 'inspect'
            block: [self inspect]
            type: #debug).
    aCollection add: (self debugStream isNil
            ifTrue: [AdventureMenuItem
                    label: 'enable debug trace'
                    block: [self debugStream: Transcript]
                    type: #debug]
            ifFalse: [AdventureMenuItem
                    label: 'disable debug trace'
                    block: [self debugStream: nil]
                    type: #debug]).
    aCollection add: (self haltsOnChange ifFalse: [AdventureMenuItem
                label: 'enable change halt'
                block: [self haltsOnChange: true]
                type: #debug]
            ifTrue: [AdventureMenuItem
                    label: 'disable change halt'
                    block: [self haltsOnChange: false]
                    type: #debug])
```

addGrabDropMenuItemsFor: aPlayer to: aCollection

```
    (self isHeldBy: aPlayer)
        ifTrue: [aCollection add: (AdventureMenuItem label: 'drop' block: [self container: aPlayer
container])]
        ifFalse: [self ~~ aPlayer & (aPlayer isHeldBy: self) not & self isMoveable ifTrue: [
                aCollection add: (AdventureMenuItem label: 'grab'
                    block:
                        ["Put myself in aPlayer. Give aPlayer any value I
                        have
                        (left)."
                        self container: aPlayer.
                        self transferValueTo: aPlayer])]]
```

addMenuItemsFor: aPlayer to: aCollection

```
    "This is the hook for subclasses to add their own menu entries."
    "super addMenuItemsFor: aPlayer to: aCollection"

    self addPlayerTargetPropertyMenuItemsFor: aPlayer to: aCollection.
    self
        addPropertyMenuItemsFor: aPlayer
        to: aCollection
        onBehalfOf: self
        actingAs: #owner.
    self addGrabDropMenuItemsFor: aPlayer to: aCollection.
    self addMoveToMenuItemsFor: aPlayer to: aCollection.
    self addTargetingMenuItemsFor: aPlayer to: aCollection.
    self adventure debugMenusEnabled ifTrue: [self addDebugMenuItemsFor: aPlayer to: aCollection]
```

addMoveToMenuItemsFor: aPlayer to: aCollection

```
    | aTarget |
    aTarget := aPlayer target.
    (aTarget notNil and: [aTarget canAcceptMoveFrom: self])
        ifTrue: [aCollection add: (AdventureMenuItem label: 'move to ' , aTarget name block: [self
container: aPlayer target])]
```

addPlayerTargetPropertyMenuItemsFor: aPlayer to: aCollection

```
    | target |
    target := aPlayer target.
    target notNil ifTrue: [target
            addPropertyMenuItemsFor: aPlayer
            to: aCollection
            onBehalfOf: self
            actingAs: self type]
```

addPropertyMenuItemsFor: aPlayer to: aCollection onBehalfOf: aThing actingAs: roleSymbol

```
    "Ask my properties to add their menu items for aThing acting the role of
    roleSymbol."

    self properties do: [:x | x
            addMenuItemsFor: aPlayer
            to: aCollection
            onBehalfOf: aThing
            actingAs: roleSymbol]
```

addTargetingMenuItemsFor: aPlayer to: aCollection

```
    aCollection add: (self == aPlayer target
            ifTrue: [AdventureMenuItem label: 'deselect as target' block: [aPlayer invalidateTarget]]
            ifFalse: [AdventureMenuItem label: 'select as target' block: [aPlayer target: self]])
```

menuItemsFor: aPlayer

```
    "Player aPlayer is requesting menu items from me; answer them."

    | m |
    m := OrderedCollection new.
    self addMenuItemsFor: aPlayer to: m.
    ^m
```

actions

basicTransferValueTo: aPlayer

```
    "If I have any value (left) give it to aPlayer."

    self value > 0
        ifTrue:
            [aPlayer score: aPlayer score + self value.
            self value: 0]
```

noLongerTargeted

```
    "Was a hook for menu updates."
    "self changed: #menu"
```

nowTargeted

```
    "Was a hook for menu updates."
    "self changed: #menu"
```

transferValueTo: aPlayer

```
"If I have any value (left) give it to aPlayer."

self
	performAndForward: #basicTransferValueTo:
	with: aPlayer
	performFirst: false
```

printing

displayString

```
^self name
```

printOn: aStream

```
"Append a representation of myself to aStream."

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

initialize-release

initialize

```
^self
```

properties

propertyAt: aSymbol

```
^self properties at: aSymbol
```

propertyAt: aSymbol put: aProperty

```
"Set the my property named aSymbol to aProperty. Answer aProperty."

aProperty owner: self.
^self properties at: aSymbol put: aProperty
```

debugging

changed: anAspectSymbol with: aParameter

```
"Halt if I'm flagged that way. If I have a debugStream, log the change
there."

self haltsOnChange ifTrue: [self halt].
self debugStream notNil ifTrue: [(self debugStream) nextPutAll: self printString; cr; tab; nextPutAll:
'changed: ' , anAspectSymbol printString; cr; tab; nextPutAll: 'with: ' , aParameter printString; cr;
endEntry].
super changed: anAspectSymbol with: aParameter
```

logDebugEvent: anEvent

```
"Someone has generated a debugEvent, so log it if I have a
debugStream."

self debugStream notNil ifTrue: [(self debugStream) nextPutAll: self printString; cr; tab; nextPutAll:
'heard debug event: '; cr; tab; tab; nextPutAll: anEvent printString; cr; endEntry]
```

events

broadcastEvent: anEvent

```
self changed: #event with: anEvent
```

broadcastSpeech: aString volume: aVolume style: aSymbol

```
self broadcastEvent: (AdventureSpeechEvent
        source: self
        string: aString
        volume: aVolume
        style: aSymbol)
```

canHear

```
^self ear notNil
```

delayForSoundComprehension

```
"Ensure that enough time has passed since I last heard something for
me to have comprehended it."

| ms |
ms := self soundTimestamp + self soundComprehensionTime - Time millisecondClockValue.
ms > 0 ifTrue: [(Delay forMilliseconds: ms) wait]
```

hearEvent: anEvent

```
"Send a #speechEvent or #soundEvent event to my ear."

| notMe |
notMe := anEvent source ~~ self.
notMe ifTrue: [self delayForSoundComprehension].
(self ear) nextPutAll: anEvent formattedSource; space; nextPutAll: anEvent formattedString; cr;
endEntry.
notMe ifTrue: [self soundTimestamp: Time millisecondClockValue]
```

narrate: aString

```
self
    broadcastSpeech: aString
    volume: AdventureVolume normal
    style: #narration
```

say: aString

```
self
    broadcastSpeech: aString
    volume: AdventureVolume normal
    style: #normal
```

say: aString or: bString

```
self sayOneOf: (Array with: aString with: bString)
```

say: aString or: bString or: cString

```
self sayOneOf: (Array
        with: aString
        with: bString
        with: cString)
```

say: aString or: bString or: cString or: dString

```
self sayOneOf: (Array
        with: aString
```

```
            with: bString
            with: cString
            with: dString)
```

sayOneOf: aCollection
```
    self say: (AdventureRandomValueHolder withAll: aCollection) value
```

shout: aString
```
    self
        broadcastSpeech: aString
        volume: AdventureVolume high
        style: #normal
```

whisper: aString
```
    self
        broadcastSpeech: aString
        volume: AdventureVolume low
        style: #normal
```

private

basicDescription
```
    ^description isNil
        ifTrue: [description := String new]
        ifFalse: [description]
```

getImage
```
    ^image
```

pause and resume

pauseAndTerminateProcesses
```
    ^self
```

resumeProcesses
```
    ^self
```

saving

postSave
```
    "I have been saved to a BOSS file. Prepare to resume the game. If I have
    an instance-specific image, then I converted it from a CachedImage to
    an Image in #preSave. Now convert it back for display performance
    reasons."
    "super postSave."

    self getImage notNil ifTrue: [self image: (CachedImage on: self getImage)]
```

preSave
```
    "Prepare to be saved to a BOSS file. If I have an instance-specific
    image, it’s a CachedImage. Since CachedImage’s can’t be
    BOSSed-out, convert it to a normal Image."
    "super preSave."

    self getImage notNil ifTrue: [self image: self getImage image]
```

restoring

postRestore

```
"Do whatever it takes to make me game-worthy after I've been restored
from a BOSS file. Since my image, if any, was converted from a
CachedImage to an Image so it could be BOSSed-out, convert it back
here. I nil my soundTimestamp so that it will be lazy-initialized to
something sane for the current image's millisecondClockValue."
"super postRestore."

self getImage notNil ifTrue: [self image: (CachedImage on: self getImage)].
self soundTimestamp: nil
```

postRestorePostOpen

```
"After a user interface has been opened on my after restoration from a
BOSS file, this is sent to me."
"super postRestorePostOpen."

^self
```

forwarding

performAndForward: aSymbol performFirst: aBoolean

```
self
    performAndForward: aSymbol
    performFirst: aBoolean
    to: self properties
```

performAndForward: aSymbol performFirst: aBoolean to: aCollection

```
"Iterate over a copy of the collection in case the perform operation wants
to alter the collection. If aBoolean is true, do the perform first. Otherwise,
forward first."

aBoolean
    ifTrue:
        [self perform: aSymbol.
        aCollection copy do: [:elem | elem performAndForward: aSymbol performFirst: aBoolean]]
    ifFalse:
        [aCollection copy do: [:elem | elem performAndForward: aSymbol performFirst: aBoolean].
        self perform: aSymbol]
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean

```
self
    performAndForward: aSymbol
    with: aParameter
    performFirst: aBoolean
    to: self properties
```

performAndForward: aSymbol with: aParameter performFirst: aBoolean to: aCollection

```
"Iterate over a copy of the collection in case the perform operation wants
to alter the collection. If aBoolean is true, do the perform first. Otherwise,
forward first."

aBoolean
```

```
ifTrue:
    [self perform: aSymbol with: aParameter.
    aCollection copy do: [:elem | elem
            performAndForward: aSymbol
            with: aParameter
            performFirst: aBoolean]]
ifFalse:
    [aCollection copy do: [:elem | elem
            performAndForward: aSymbol
            with: aParameter
            performFirst: aBoolean].
    self perform: aSymbol with: aParameter]
```

MetaClass: AdventureThing class

Instance variables: image

instance creation

id: idSymbol name: nameString description: descriptionString

```
^(self new) id: idSymbol; name: nameString; description: descriptionString
```

id: idSymbol name: nameString description: descriptionString value: aNumber

```
^(self new) id: idSymbol; name: nameString; description: descriptionString; value: aNumber
```

new

```
^super new initialize
```

initialization

flushAllImageCaches

```
"Flush my image cache, and those of my subclasses."
"self flushAllImageCaches"
"self allSubclassesDo: [:s | Transcript cr; show: s name; space; show: s
canBuildClassSpecificImage printString ]"

self image: nil.
self allSubclassesDo: [:s | s image: nil]
```

images

getImage

```
"If I can build my own image, do so. Otherwise, ask my superclass for its
image. Don't override in subclasses."

^self canBuildClassSpecificImage
    ifTrue: [self buildImage]
    ifFalse: [self superclass image]
```

image

```
"Don't override in subclasses."
```

```
^image isNil
      ifTrue: [image := self getImage]
      ifFalse: [image]
```

image: anImage

```
image := anImage
```

testing

canBuildClassSpecificImage

```
^self class includesSelector: #buildImage
```

resources

buildImage

```
^CachedImage on: (Adventure readImageFromBOSSFile: ’thing.bos’)
```

Class: **AdventureToken**

Superclass: AdventurePlug
Category: Adventure-Kernel

I'm a plug with a #plugType of #token. I'm used as a kind of consumable key, like currency.

testing

type
^#token

MetaClass: AdventureToken class

Class: **AdventureTokenOperatedLock**

Superclass: AdventureLock
Category: Adventure-Kernel

I'm a lock operated by inserting a token with a compatible #shape. I expect that my owner is a locking container.

initialize-release

initialize

```
    super initialize.
    self description: 'token-operated lock'
```

menu items

addOwnerMenuItemsFor: aPlayer to: aCollection

```
    "My owner has delegated menu building to me."

    self owner isOpen ifTrue: [aCollection add: (AdventureMenuItem label: 'close and lock'
                block:
                    [self owner close.
                    self owner lock])].
    super addOwnerMenuItemsFor: aPlayer to: aCollection
```

addPlugMenuItemsFor: aPlayer to: aCollection onBehalfOf: aKey

```
    "aKey has delegated menu building to me."

    self owner isLocked ifTrue: [aCollection add: (AdventureMenuItem label: 'deposit and open ' , self
owner name block: [self
                    ifCompatibleWith: aKey
                    execute:
                        [self owner open.
                        self owner unlock.
                        aKey destroy]
                    for: aPlayer])]
```

plugType

```
    "Answer the type that a compatible plug will present when asking for
    menu items."

    ^#token
```

MetaClass: AdventureTokenOperatedLock class

Class: **AdventureTokenOperatedMechanism**

Superclass: AdventureSocket
Category: Adventure-Kernel
Instance variables: actuateBlock processTokenBlock

I’m a general-purpose mechanism that’s operated by inserting a token with a compatible #shape. I offer process-token and actuate hooks to customize my behavior.

Instance Variables:
actuateBlock <BlockClosure> that is called after the token is processed.
processTokenBlock <BlockClosure> that is called when a compatible token is inserted.

accessing

actuateBlock
```
	^actuateBlock isNil
		ifTrue: [actuateBlock := [:aPlayer | ]]
		ifFalse: [actuateBlock]
```

actuateBlock: aBlock
```
	actuateBlock := aBlock
```

processTokenBlock
```
	^processTokenBlock isNil
		ifTrue: [processTokenBlock := [:aToken :aPlayer | aToken destroy]]
		ifFalse: [processTokenBlock]
```

processTokenBlock: aBlock
```
	processTokenBlock := aBlock
```

menu items

addOwnerDebugMenuItemsFor: aPlayer to: aCollection
```
	""
	"super addOwnerDebugMenuItemsFor: aPlayer to: aCollection"

	aCollection add: (AdventureMenuItem
				label: 'actuate mechanism'
				block: [self actuateFor: aPlayer]
				type: #debug).
	super addOwnerDebugMenuItemsFor: aPlayer to: aCollection
```

addPlugMenuItemsFor: aPlayer to: aCollection onBehalfOf: aToken
```
	"aToken has delegated menu building to me."

	aCollection add: (AdventureMenuItem label: 'deposit into ' , self owner name block: [self
			ifCompatibleWith: aToken
			execute:
				[self processToken: aToken for: aPlayer.
				self actuateFor: aPlayer]
```

for: aPlayer])

initialize-release

initialize

```
super initialize.
self description: 'token slot'
```

mechanism

actuateFor: aPlayer

```
self actuateBlock value: aPlayer
```

processToken: aToken for: aPlayer

```
self processTokenBlock value: aToken value: aPlayer
```

testing

plugType

```
"Answer the type that a compatible plug will present when asking for
menu items."

^#token
```

MetaClass: AdventureTokenOperatedMechanism class

instance creation

actuateBlock: aBlock

```
^self new actuateBlock: aBlock
```

shape: aSymbol actuateBlock: aBlock

```
^(self new) shape: aSymbol; actuateBlock: aBlock
```

Class: **AdventureTurnRequest**

Superclass: Object
Category: Adventure-Kernel
Instance variables: adventure turnQueued

I represent a request to begin a turn on my #adventure. I am polled to test if the turn has been granted. #endTurn must be sent to me at the end of the turn. I can be re-used repeatedly.

I request a turn in my name, so if you're debugging backwards from me, you'll need to send me #allOwners to find the original requestor.

Instance Variables:
adventure <Adventure> to which I apply.
turnQueued <Boolean> used to keep track of whether I have already queued a request for a turn to the adventure.

accessing

adventure
```
	^adventure
```

adventure: anAdventure
```
	adventure := anAdventure
```

turnQueued
```
	^turnQueued isNil
		ifTrue: [turnQueued := false]
		ifFalse: [turnQueued]
```

turnQueued: aBoolean
```
	turnQueued := aBoolean
```

initialize-release

on: anAdventure
```
	self adventure: anAdventure
```

turn synchronization

cancel
```
	self turnQueued
		ifTrue:
			[self adventure cancelTurnFor: self.
			self turnQueued: false]
```

endTurn
```
	self adventure endTurnFor: self.
	self turnQueued: false
```

pollForTurn

```
	"Answer true if I have been granted a turn by the adventure."

	self turnQueued
		ifFalse:
			[self adventure queueTurnFor: self.
			self turnQueued: true].
	^self adventure turnGrantedFor: self
```

MetaClass: AdventureTurnRequest class

instance creation

on: anAdventure

```
	^self new on: anAdventure
```

Class: **AdventureTurnstile**

Superclass: AdventureLockingDoor
Category: Adventure-Kernel

I’m a locking door that closes and locks itself after a player enters it. Just.. like... a... turnstile.

actions

inhale: aPlayer

```
"Make aPlayer enter me.. I’m a turnstile, so naturally, I close and lock
myself behind aPlayer."

super inhale: aPlayer.
self close.
self lock
```

MetaClass: AdventureTurnstile class

Class: **AdventureVolume**

Superclass: Object
Category: Adventure-Kernel
Instance variables: intensity
Class variables: AttenuationMap

I represent a simple simulation of sound volume. I can be attenuated. My intensity may be #silent, #low, #normal, or #high.

Instance Variables:
intensity <Symbol> representing my intensity.

Class Variables:
AttenuationMap <IdentityDictionary> of symbol-to-symbol associations for mapping an intensity to its attenuated equivalent.

formatting

formatString: aString

```
"Answer a formatted copy of aString. Format according to my intensity."

self intensity = #normal ifTrue: [^aString].
self intensity = #high ifTrue: [^aString asUppercase].
self intensity = #low ifTrue: [^aString asLowercase].
self intensity = #silent ifTrue: [^String new].
self error: 'Invalid intensity'
```

accessing

intensity

```
^intensity
```

intensity: aSymbol

```
intensity := aSymbol
```

attenuation

attenuated

```
^self copy intensity: (self class attenuationMap at: self intensity)
```

printing

printOn: aStream

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

MetaClass: AdventureVolume class

instance creation

high
```
^self new intensity: #high
```

low
```
^self new intensity: #low
```

normal
```
^self new intensity: #normal
```

silent
```
^self new intensity: #silent
```

private-accessing

attenuationMap
```
""
"self attenuationMap"
"self attenuationMap: nil"
^AttenuationMap isNil
    ifTrue: [AttenuationMap := (IdentityDictionary new)
        add: #high -> #normal;
        add: #normal -> #low;
        add: #low -> #silent;
        add: #silent -> #silent;
        yourself]
    ifFalse: [AttenuationMap]
```

attenuationMap: anIdentityDictionaryOrNil
```
AttenuationMap := anIdentityDictionaryOrNil
```