## Object subclass: #MethodCombinationContext

```
	instanceVariableNames: 'methodCombination genericFunction applicableMethods arguments '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!

'MethodCombinationContext objects house the per-call dynamic state for GenericFunction invocations.
They are created by MethodCombination objects when a GenericFunction is applied so that additional
context information can be recovered when Context>>callNextMethod or
Context>>callNextMethodWithArguments: is called.

Instance Variables:

methodCombination	<MethodCombination> The MethodCombination that created this
			MethodCombinationContext.
genericFunction		<GenericFunction> The GenericFunction that called the
			MethodCombination that created us.
applicableMethods	<OrderedCollection of: MultiMethod> The applicable MultiMethods
			that are still to be called.
arguments		<Array of: Obect> The arguments with which the next
			MultiMethod should be called.
'!
```

**!MethodCombinationContext methodsFor: 'instance initialization'!**

```
methodCombination: mc genericFunction: gf applicableMethods: am arguments: args
	"Stash the MethodCombination we were created for, the GenericFunction that summoned it, the
	applicable MultiMethods that will be grist for this mill, and the
	arguments the GenericFunction was
	called with."

	methodCombination := mc.
	genericFunction := gf.
	applicableMethods := am.
	arguments := args.
	^self! !
```

**!MethodCombinationContext methodsFor: 'evaluation'!**

```
applyNextWithArguments: args
	"Apply the next applicable method. If we were passed an
	argument list, use those instead of the ones
	we were created with. This will happen when callNextMethodWithArguments: is
	callNextMethod. Then, pick the next method off of the list
	of applicable methods, and prune it
	off the list. Finally, call the method, and return its result as
	our result. If there is no next applicable
	method, declare defeat."

	| method result |
	args isNil ifFalse: [arguments := args].
	applicableMethods isEmpty ifTrue: [self error: 'no applicable methods'].
	method := applicableMethods first.
	applicableMethods := applicableMethods rest.
	result := method valueWithReceiver: arguments first arguments: arguments rest asArray.
	Transcript show: ' -- applying ' , method printString , ' in MethodCombinationContext'; cr.
	^result! !
```

## CompiledMethod variableSubclass: #EffectiveMethod

```
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
```

**EffectiveMethod comment:**

'EffectiveMethods are not currently in use...'!

## Object subclass: #MethodCombination

```
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
```

**MethodCombination comment:**

*'MethodCombination objects take a collection of applicable methods and arguments, and run them in the manner prescribed by the qualifers and argument types.  In CLOS, MethodCombinations, in collusion with discriminating functions and the multimethods themselves, are used to produce optimized ''effective methods''.  Here, they often conduct the execution of the GenericFunction directly.  MethodCombination is an abstract class.'!*

**!MethodCombination methodsFor: 'evaluation'!**

**applyGenericFunction: gf withMethods: am andArguments: args**

```
	^self subclassResponsibility! !
```

## MethodCombination subclass: #SubStandardMethodCombination

```
	instanceVariableNames: ''
	classVariableNames: ''
	category: 'Generic-Functions'!
```

**SubStandardMethodCombination comment:**

*'SubStandardMethodCombination objects provide rudimentary support for Before, After, and Primary methods, with support for callNextMethod.  They use MethodCombinationContext objects to provide primary/callNextMethod support.  That hierarchy should be factored to parallel this one.'!*

```
!SubStandardMethodCombination methodsFor: 'evaluation'!

applyGenericFunction: gf withMethods: am andArguments: args
	"This is the meat of SubStandardMethodCombination. First, select all
	the #Before methods from the
	sorted applicable methods list, and exectue them in most-specifc to
	least-specific (most general)
	order. This is the way they come out of GenericFunction's sort. Next, create a
	MethodCombinationContext object that will house the context information for primary method
	execution, and ask it to start the first one up. If a next method is called,
	the stack is walked by callNextMethodXxx in order to locate this object, and it conducts
	the next invocation. Once we have a result from the primary chain, select and execute
	all the #After methods in least-specifc to
	most-specific order. Since they are in the sorted list
	Any results from #Before and #After methods are discarded,
	and the result from the primary chain
	is returned as the result of this GenericFunction invocation."

	| primaries v before mcc after |
	before := am select: [:method | method qualifier == #Before].
	before do: [:m | v := m valueWithReceiver: args first arguments: args rest asArray].
	primaries := am select: [:method | method qualifier isNil].
	mcc := MethodCombinationContext new
				methodCombination: self
				genericFunction: gf
				applicableMethods: primaries
				arguments: args.
	v := mcc applyNextWithArguments: nil.
	after := am select: [:method | method qualifier == #After].
	after reverse do: [:m | v := m valueWithReceiver: args first arguments: args rest asArray].
	Transcript show: ' -- applying GenericFunction in MethodCombination'; cr.
	^v!

callNextMethod
	"If you needs an example of a method where transformations that
	would normally be considered as
	behavior preserving might not be, consider
	#applyGenericFunction:withMethods:andArguments: in
	the presence of this method..."

	| applyContext gf args primaries |
	applyContext := self firstContextWithSelector:
#applyGenericFunction:withMethods:andArguments:.
	gf := applyContext tempAt: 1.
	args := applyContext tempAt: 3.
	primaries := applyContext tempAt: 4.
	primaries size <= 1 ifTrue: [^nil].
	self
		applyGenericFunction: gf
		withMethods: primaries rest
		andArguments: args!

callNextMethodWithArguments: arguments
	| applyContext gf args primaries |
	applyContext := thisContext firstContextWithSelector:
#applyGenericFunction:withMethods:andArguments:.
	gf := applyContext tempAt: 1.
	arguments isNil
		ifTrue: [applyContext tempAt: 3]
		ifFalse: [args := arguments].
	primaries := applyContext tempAt: 4.
	primaries size <= 1 ifTrue: [^nil].
	self
		applyGenericFunction: gf
		withMethods: primaries rest
		andArguments: args! !
```

## MethodCombination subclass: #SimpleMethodCombination

```
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
SimpleMethodCombination comment:
'SimpleMethodCombination objects specialize MethodCombination to provide a trivial scheme for
MultiMethod selection and exection: just execute ''em all in the order in which they come up.  It
does not currently support callNextMethod (though defaults should be factored into
MethodCombination for this soon).'!


!SimpleMethodCombination methodsFor: 'evaluation'!

applyGenericFunction: gf withMethods: am andArguments: args
	"An odd test of the method combination mechanism, that executes
	all the applicable methods, and returns the value of the last one..."

	| v |
	am do: [:m | v := m  valueWithReceiver: args first arguments: (args rest) asArray].
	Transcript show: 'applying gf in mc'; cr.
	^v! !
```

## CompiledMethod variableSubclass: #MultiMethod

```
	instanceVariableNames: 'specializers genericFunctionSelector multiMethodSelector qualifier '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
MultiMethod comment:
'MultiMethod objects are like regular methods, except that any and all of their arguments can
participate in their selection, and they can be qualified as being of particular types like #Before
or #After, that are treated differently from normal #Primary methods.

The are currently defined by using a special type syntax in the browser.  For instance,

moveThrough: medium <Land>
...

in class <Mammal> defines a MultiMethod the first argument of which is specialized on (in CLOS
parlance) Mammal (as usual) and the second argument of which is specialized on class Land.  The
definition of a MultiMethod in a class for which unspecialized method by that name exists
automatically creates a stub for that method.  If a method already exists, it is hidden when the
DiscriminatingMethod for the class is installed, which occurs when a MultiMethod is accepted.

Our specializer syntax depends on an extendedLanguage flag in the Parser>>initScanner being turned
on.

Currently, no effort is made to "promote" existing unspecialized methods to MultiMethods when a
GenericFunction on their selector is established.  This would be a useful thing to do, and the
policies and mechanisms for this are under investigation.  Also, DiscriminatingMethods are not
currently removed when all the MultiMethods on a particular first argument class are removed.  They
can be removed ''by hand'', and by using the MethodWrapper uninstall protocol.

Currently, #Before and #After qualifications can be made by including the words Before or After
anywhere in a method name.  This is a peculiar, stopgap mechanism that will be removed when a
better syntax or mechanism for qualifiers is defined.

Instance Variables:

specializers		<OrderedCollection> The Specializers (currently all ClassSpecializers)
			for this MultiMethod, in left-to-right order (for now).
genericFunctionSelector <Symbol> The name of the GenericFunction we specialize.
multiMethodSelector <Symbol> The bizarre selector used to identify us in
			MethodDictionaries, and in the Browser.
qualifier		<Symbol> One of #Before, #After, or nil (for primary methods)
			#Around will follow shortly.  User-defined qualifiers will be okay.
'!

!MultiMethod methodsFor: 'removing'!

removeFromGenericFunction
	"Disentangle us from our GenericFunction. It is in charge of the DiscimінatingMethods too."

	self genericFunction remove: self! !
```

```
!MultiMethod methodsFor: 'testing'!

= anotherMultiMethod
	"Say two MultiMethods are equal if they are on the same selector,
	and all the Specializers are equal..."

	^self genericFunctionSelector = anotherMultiMethod genericFunctionSelector and: [self
specializers = anotherMultiMethod specializers]!

isMultiMethod
	"I most certainly is.  CompiledCode denies this.  Everything else is
	agnostic on the question."

	^true!
```

```

lessThan: rhs withArguments: args
	"We compare our specializer with the right argument's specializers on at a time, using the
	corresponding argument to arbitrate relative priority (which is (will be?)
	relevant with multiple inheritance). The order method returns an interval or
	collection that allows the order in which the
	arguments are checked to be changed. CLOS GenericFunctions allow user
	control of the argument precedence. We shall too, soon..."

	| left right |
	left := self specializers.
	right := rhs specializers.
	self order do: [:i | ((left at: i)
				lessThan: (right at: i)
				withArgument: (args at: i))
				ifTrue: [^true]].
	^false!

matches: args
	"Check each of our Specializers against each respective argument.
	If they all match, we match. If
	any does not, we don't either."

	self specializers with: args do: [:s :a | (s matches: a)
				ifFalse: [^false]].
	^true!

order
	"Return a stock left to right interval. We'll override this somehow
	to do argument permutations. An
	instance variable with a default could do this now, but one
	must beware of subtle complications
	first..."

	^1 to: self specializers size! !

!MultiMethod methodsFor: 'instance initialization'!

on: gfSelector using: mmSelector withArgumentSpecializers: argumentSpecializers
	"The other version is the one at the bottom. We just add a Specializer
	for our left-hand argument, and
	pass the buck..."

	^self
		on: gfSelector
		using: mmSelector
		withSpecializers: (Array with: (ClassSpecializer on: mclass name))
					, argumentSpecializers!

on: gfSelector using: mmSelector withSpecializers: anArray
	"Initialize this Multimethod, and add it to its GenericFunction..."

	self specializers: anArray copy.
	self  genericFunctionSelector: gfSelector.
	self  multiMethodSelector: mmSelector.
	self  qualifier: (MultiMethod qualifierFor: mmSelector).

	"As with CLOS's ensure-generic-function, we create the GF
	if it one doesn't already exist.  This means users won't normally need
	to declare GFs explicitly.  This is good, because, unlike CLOS, we
	have no syntactic sugar to make this more palatable..."
	(GenericFunction on: genericFunctionSelector) add: self .

	"Flush the ENTIRE cache..."
	self class flushVMmethodCache! !

!MultiMethod methodsFor: 'accessing'!

```

```

genericFunction
	"Always go through the registry..."

	^GenericFunction on: self genericFunctionSelector!

genericFunctionSelector
	^genericFunctionSelector!

genericFunctionSelector: anObject
	genericFunctionSelector := anObject!

multiMethodSelector
	^multiMethodSelector!

multiMethodSelector: anObject
	multiMethodSelector := anObject!

qualifier
	^qualifier!

qualifier: aSymbol
	^qualifier := aSymbol!

specializerAt: anIndex
	^specializers at: anIndex!

specializerAt: anIndex put: specializer
	^specializers at: anIndex put: specializer!

specializers
	^specializers!

specializers: anArray
	^specializers := anArray! !

!MultiMethod methodsFor: 'discriminating method'!

discriminatingMethod
	"Let our GenericFunction keep track of where these are planted..."

	^self genericFunction discriminatingMethodFor: self!

install
	"If our DiscriminatingMethod is not already installed, ask it to do so now..."

	self discriminatingMethod isInstalled ifFalse: [self discriminatingMethod install]! !

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

MultiMethod class
	instanceVariableNames: ''!


!MultiMethod class methodsFor: 'instance creation'!

new
	"Don't allow empty constructors for multimethods..."

	self shouldNotImplement!

new: anObject
	"Don't allow empty constructors for multimethods..."

	self shouldNotImplement! !

```

```
!MultiMethod class methodsFor: 'qualifiers'!

qualifierFor: aSelector
	"MultiMethod qualifierFor: #dogFood: nil"
	"MultiMethod qualifierFor: #AroundDogFood: #Around"
	"MultiMethod qualifierFor: #BeforeDogFood: #Before"
	"Any occurrence works for the moment.  This is an expedient until
	we get real qualifier syntax..."

	#(#Before #After #Around) do: [:q | (aSelector findString: q startingAt: 1)
			> 0 ifTrue: [^q]].
	^nil!

unqualifiedSelectorFor: aSelector
	"MultiMethod unqualifiedSelectorFor: #BeforeSomething: #something:"
	"MultiMethod unqualifiedSelectorFor: #AfterSomething: #something:"
	"MultiMethod unqualifiedSelectorFor: #something: #something:"

	| q s |
	q := MultiMethod qualifierFor: aSelector.
	q isNil ifTrue: [^aSelector].
	s := aSelector copyFrom: q size + 1 to: aSelector size.
	s := s asString.
	s isEmpty ifTrue: [self error: 'Bad Selector: ' , aSelector].
	s at: 1 put: (s at: 1) asLowercase.
	^s asSymbol! !
```

## MethodWrapper variableSubclass: #DiscriminatingMethod

```
	instanceVariableNames: 'genericFunctionSelector receiverSpecializer installedFlag '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
DiscriminatingMethod comment:
'DiscriminatingMethods are MethodWrappers that intercept the invocations of methods for which
GenericFunctions are defined, and pass control to these GenericFunctions.  (Future) subclasses may
exploit the partial knowlege of the dispatch outcome that DiscriminatingMethods have, as a result
of their placement, and use this narrow and expedite this process.

Instance Variables:

genericFunctionSelector		<Symbol> The selector for the GenericFunction we specialize, (and for
the MethodDictionary slot we occupy).
receiverSpecializer		<ClassSpecializer> A specializer for the
				receiver position (not currently in use).
installedFlag			<Boolean> Are we currently installed?
'!
```

**!DiscriminatingMethod methodsFor: 'evaluating'!**

```
valueWithReceiver: object arguments: args
	"When an instance of MethodWrapper is called, this method is
	given control. DiscriminatingMethods
	pass control to their GenericFunctions..."


	| v |
	v := (GenericFunction on: self genericFunctionSelector)
				applyReceiver: object withArguments: args.
	^v! !
```

**!DiscriminatingMethod methodsFor: 'accessing'!**

```
genericFunctionSelector
	^genericFunctionSelector!

genericFunctionSelector: aSymbol
	genericFunctionSelector := aSymbol!

receiverClass
	^receiverClass!

receiverClass: anObject
	receiverClass := anObject!

receiverSpecializer
	^receiverSpecializer!

receiverSpecializer: anObject
	receiverSpecializer := anObject! !
```

```
!DiscriminatingMethod methodsFor: 'installation'!

install
	"Let's not let these nest for now. This is unnecessarily restrictive,
	but keeps things simple for now..."

	| targetMethod |
	self receiverSpecializer: (ClassSpecializer on: mclass name).
	targetMethod := mclass compiledMethodAt: selector.

	"Is there a wrapper there? If so, yank it out first..."
	(targetMethod isKindOf: MethodWrapper)
		ifTrue:
			[Transcript show: 'uninstalling a '; print: targetMethod class; show: ' at ';
print: selector; cr; endEntry.
			targetMethod uninstall].

	"Our parent knows how to handle this..."
	super install.

	"Show something other than the hidden method in the Browsers..."
	sourceCode := (DiscriminatingMethod class compiledMethodAt:
#installedDiscriminatingMethodSource) sourcePointer.
	installedFlag := true!

uninstall
	"Do the real uninstall, and then say we don't know our code either,
	and mark us as uninstalled..."

	super uninstall.
	sourceCode := (DiscriminatingMethod class compiledMethodAt:
#uninstalledDiscriminatingMethodSource) sourcePointer.
	installedFlag := false! !

!DiscriminatingMethod methodsFor: 'testing'!

isDiscriminatingMethod
	"CompiledCode denies this..."

	^true!

isInstalled
	"Is this DiscriminatingFunction currently installed in a MethodDictionary, or not?"

	^installedFlag! !

!DiscriminatingMethod methodsFor: 'initialize-release'!

class: aClass selector: sel
	"This is a stock MethodWrapper set up method. Before letting MethodWrapper
	finish up, we say we
	aren't installed, and set our sourceCode pointer to a dummy chunk of
	source code so that Browsers
	will have something to look at that indicates
	our status..."

	installedFlag := false.
	sourceCode := (DiscriminatingMethod class compiledMethodAt:
#uninstalledDiscriminatingMethodSource) sourcePointer.
	^super class: aClass selector: sel! !
```

```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

DiscriminatingMethod class
	instanceVariableNames: ''!


!DiscriminatingMethod class methodsFor: 'installation'!

on: selector inClass: class
	"Do pretty much what our super does, but don't forget our genericFunction selector..."
	"Why selector AND genericFunctionSelector?  History.  I probably don't need both."

	| d |
	d := super on: selector inClass: class.
	d genericFunctionSelector: selector.
	^d! !

!DiscriminatingMethod class methodsFor: 'accessing'!

canWrap: aSelector inClass: aClass
	"Test if a method can be wrapped without causing infinite recursion."

	| class method |
	(aClass includesBehavior: MethodWrapper) ifTrue: [^false].
	aClass == BlockClosure ifTrue:
				[(#(#valueAsUnwindBlockFrom: #valueNowOrOnUnwindDo:) includes: aSelector)
					ifTrue: [^false]].
	^true
	"I decided to not be picky about whether there is a method to wrap..."
	"class := aClass whichClassIncludesSelector: aSelector.
	class isNil ifTrue: [^false].
	method := class compiledMethodAt: aSelector ifAbsent: [nil].
	^method notNil and: [(self primitives includes: method primitiveNumber) not]"! !

!DiscriminatingMethod class methodsFor: 'source templates'!

installedDiscriminatingMethodSource
	"* * * The method you are looking at is an INSTALLED DiscriminatingMethod for the current
selector. It's actual
	code is hidden. * * *"
	"Don't change and accept this code unless you want a method named
#installedDiscriminatingMethodSource..."
	"The source pointer for this method is copied to that of each DiscriminatingMethod when it
is installed..."

	^self!

uninstalledDiscriminatingMethodSource
	"* * * You are looking at an UNINSTALLED Discriminating Method * * *"
	"Don't change and accept this code unless you want a method named
#uninstalledDiscriminatingMethodSource..."
	"The source pointer for this method is copied to that of each DiscriminatingMethod when it
is created and uninstalled..."

	^self! !

```

```
!DiscriminatingMethod class methodsFor: 'nothing methods'!

createEmptyMethodFor: selector
	"Create a stub method for the indicated selector. First use
	emptyMethodFor: to create some source,
	and then compile it. We return a method node..."

	^(self compilerClass new
		compile: (self emptyMethodFor: selector)
		in: self
		notifying: nil
		ifFail: []) generate!

createNothingMethodFor: numArgs
	"Depricated..."

	^(self compilerClass new
		compile: (self doNothingStringFor: numArgs)
		in: self
		notifying: nil
		ifFail: []) generate!

doNothingStringFor: numArgs
	"(0 to: 3) collect: [:i | DiscriminatingMethod doNothingStringFor: i]"

	| nameString methodComment |
	methodComment := '"* * * DiscriminatingMethod stub method * * *"'.
	nameString := numArgs = 0
				ifTrue: ['nothing']
				ifFalse: [''].
	1 to: numArgs do: [:i | nameString := nameString , 'nothing: t' , i printString , ' '].
	^nameString , ' ' , methodComment , '  ^self'!

emptyMethodFor: selector
	"DiscriminatingMethod emptyMethodFor: #to:do:"

	| methodComment |
	methodComment := '"* * * DiscriminatingMethod stub method... * * *"'.
	^(self methodHeaderFor: selector)
		, methodComment , '  ^self'!

methodHeaderFor: selector
	"DiscriminatingMethod methodHeaderFor: #size 'size '"
	"DiscriminatingMethod methodHeaderFor: #+ '+ rhs '"
	"DiscriminatingMethod methodHeaderFor: #to:do: 'to: a1 do: a2 '"

	| s |
	selector numArgs == 0 ifTrue: [^selector asString , ' '].
	selector isInfix ifTrue: [^selector asString , ' ' , 'rhs' , ' '].
	s := ''.
	(1 to: selector numArgs)
		with: selector keywords do: [:i :k | s := s , k , ' a' , i printString , ' '].
	^s! !
```

## Object subclass: #Specializer

```
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
```

**Specializer comment:**
*'Specializer is an abstract class for CLOS-style argument specializers.  It currently houses default denials for the isXxx queries defined in its subclasses.  The class side houses some default queries.'!*

**!Specializer methodsFor: 'testing'!**

```
isClassSpecializer
	"No I'm not one of these..."

	^false!

isEqualSpecializer
	"Deny this heritage..."

	^false! !
```

"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

## Specializer class

```
	instanceVariableNames: ''!
```

**!Specializer class methodsFor: 'instance creation'!**

```
default
	"The default specializer shall be whatever ClassSpecializer thinks a good default is..."

	^ClassSpecializer default! !
```

## Specializer subclass: #EqualSpecializer

```
	instanceVariableNames: 'specializerInstance specializerBlock '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
```

**EqualSpecializer comment:**
*'EqualSpecializers (when they are implemented, which they currently are not) will mimic CLOS-style per-instance MultiMethod specializers*

*Instance Variables:*

```
specializerInstance	<Object>	The value this EQL-specializer must match.
specializerBlock		<BlockClosure> A one argument block that returns true if an object matches us.'!
```

**!EqualSpecializer methodsFor: 'testing'!**

```
= aSpecializer
	"If we are both EqualSpecializers, and our specializerInstance
	variables match, we're satisfied..."

	aSpecializer isEqualSpecializer ifFalse: [^false].
	^self specializerInstance = aSpecializer specializerInstance!

isEqualSpecializer
	^true! !
```

```

!EqualSpecializer methodsFor: 'accessing'!

specializerBlock
	^specializerBlock!

specializerBlock: anObject
	^specializerBlock := anObject!

specializerInstance
	^specializerInstance!

specializerInstance: anObject
	^specializerInstance := anObject! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

EqualSpecializer class
	instanceVariableNames: ''!


!EqualSpecializer class methodsFor: 'instance creation'!

on: aBlock
	"Store the block we were given..."

	| s |
	s := super new.
	s specializerBlock: aBlock.
	^s! !


Specializer subclass: #ClassSpecializer
	instanceVariableNames: 'specializerClassName '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Generic-Functions'!
ClassSpecializer comment:
'ClassSpecializers implement CLOS-style MultiMethod argument specializers.  They store the identity
of the class the specializer must match.

Instance Variables:

specializerClassName <Symbol> The name of the Class we must match.  Storing the name avoids forward
reference and zombie reference problems.'!


!ClassSpecializer methodsFor: 'testing'!

= aSpecializer
	"If we are both ClassSpecializers, and the Classes match, we're happy..."

	aSpecializer isClassSpecializer ifFalse: [^false].
	^self specializerClass = aSpecializer specializerClass!

isClassSpecializer
	^true!

lessThan: rhs withArgument: arg
	"We determine order by our relative postions given the CPL for a <particular argument>..."

	| cpl |
	cpl := self computeClassPrecedenceListFor: arg.
	^(cpl indexOf: self specializerClassName)
		< (cpl indexOf: rhs specializerClassName)! !
```

```
!ClassSpecializer methodsFor: 'precedence'!

computeClassPrecedenceListFor: anObject
	"Without multiple inheritance, a Class Precedence List (CPL in CLOS parlance) comes
	down to a straight patrilinear chain all the way to
	nil (the only superclass worth having)..."

	| cpl class |
	cpl := OrderedCollection new: 5.	"What depth covers most of 'em?"
	class := anObject class.
	[class isNil]
		whileFalse:
			[cpl add: class name.
			class := class superclass].
	^cpl!

matches: anObject
	"In CLOS, you specify the precendence of your (perhaps multiple) direct superclass(es).
	This means different arguments may have a particular class in different positions
	in their CPLs, which is why they use CPLs for this stuff."

	"Yes, I know that for single inhertitance the commented line of code below
	really is equivalent, but CPLs may matter later..."
	"result := anObject isKindOf: self specializerClass."
	| cpl result |
	cpl := self computeClassPrecedenceListFor: anObject.
	result := cpl includes: self specializerClassName.
	"The temporary result leaves us somewhere to put the self halt during testing..."
	^result! !

!ClassSpecializer methodsFor: 'printing'!

printOn: aStream
	"ClassSpecializers print as <Junk>..."

	aStream nextPut: $<.
	aStream print: self specializerClass.
	aStream nextPut: $>! !

!ClassSpecializer methodsFor: 'accessing'!

specializerClass
	"Dig the actual class for our specializer out of Smalltalk..."

	^Smalltalk at: self specializerClassName!

specializerClass: aClass
	"Store the name, not the class..."

	self specializerClassName: aClass name!

specializerClassName
	^specializerClassName!

specializerClassName: aSymbol
	^specializerClassName := aSymbol! !
```

```
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

ClassSpecializer class
	instanceVariableNames: ''!


!ClassSpecializer class methodsFor: 'instance creation'!

default
	"We could probably cache this..."
	"Our default is Object (CLOS 'T'). nil superclass shenanegans
	will probably merit some thought doen
	the road..."

	^super new; specializerClass: Object; yourself!

on: aClassOrSymbol
	"Create a ClassSpecializer. Let callers send us Symbols, or the real McCoy.
	This may make a useful
	MultiMethod demonstration one of these days..."

	| s |
	s := super new.
	(aClassOrSymbol isKindOf: Symbol)
		ifTrue: [s specializerClassName: aClassOrSymbol]
		ifFalse: [s specializerClass: aClassOrSymbol].
	^s! !
```

## Object subclass: #GenericFunction

```
	instanceVariableNames: 'selector multiMethods discriminatingMethods methodCombination '
	classVariableNames: 'GenericFunctionRegistry '
	poolDictionaries: ''
	category: 'Generic-Functions'!
GenericFunction comment:
'GenericFunctions implement CLOS-style families of MultiMethods, where dispatching takes not only
the left-hand (receiver) argument into account, but all the other arguments to a method invocation
as well.  A GenericFunction knows all the MultiMethods that ''specialize'' it.  It also has a
MethodCombination object, which determines the manner in which applicable MultiMethods are
executed.

Instance Variables:

selector		<Symbol>	The selector, or operator that this GenericFunction oversees.
multiMethod <Dictionary> A mapping from MultiMethod selectors to MultiMethods
discriminatingMethods <Dictionary> A mapping from Class names to DiscriminatingMethods.
methodCombination <MethodCombination> The MethodCombination used for this GenericFunction.

Class Variables:

GenericFunctionRegistry <Dictionary> A mapping from each Symbol for which a GenericFunction is
defined to that GenericFunction.  This global registry ensures that one and only one
GenericFunction is defined at any given time for a particular selector/operator.
'!
```

**!GenericFunction methodsFor: 'private'!**

```
allImplementors
	"Return a SortedCollection of all the methods in the system that implement our selector..."
	"This is based on Browser>>allImplementorsOf:."
	"(GenericFunction on: #at:put:) allImplementors"

	| aCollection |
	aCollection := OrderedCollection new.
	Smalltalk allBehaviorsDo: [:class | (class includesSelector: selector)
				ifTrue: [aCollection add: (class compiledMethodAt: selector)]].
	^aCollection!

allImplementorsOld
	"Return a SortedCollection of all the methods in the system that implement our selector..."

	"This is based on Browser>>allImplementorsOf:."
	"(GenericFunction on: #at:put:) allImplementors"

	| aCollection |
	aCollection := SortedCollection new.
	Smalltalk allBehaviorsDo: [:class | (class includesSelector: selector)
				ifTrue: [aCollection add: class name , ' ' , selector]].
	^aCollection! !
```

**!GenericFunction methodsFor: 'printing'!**

```
printOn: aStream
	"We might use the temporary for vowel testing one of these days..."

	| title |
	title := self class name.
	aStream nextPutAll: title.
	aStream nextPutAll: ' on: '.
	aStream print: selector! !
```

```
!GenericFunction methodsFor: 'evaluation'!

apply: args
	"To apply a GenericFunction, first, determine the applicable methods
	(those for which the arguments
	satisfy the specializers). These are then sorted in the order in which
	they are applicable. Then, they
	are passed to the MethodCombination, which actually executes the methods..."

	| am v s |
	am := self computeApplicableMethods: args.
	am isEmpty ifTrue: [self error: 'no-applicable-methods'].
	s := SortedCollection sortBlock: [:l :r | l lessThan: r withArguments: args].
	s addAll: am.
	am := s asOrderedCollection.
	v := self methodCombination
				applyGenericFunction: self
				withMethods: am
				andArguments: args.
	Transcript show: 'after mc apply'; cr.
	^v!

applyReceiver: receiver withArguments: args
	"Concatentate the receiver and the given arguments and pass 'em all to apply:..."
	"(Yes, this is a peculiar idiom for this, but I froze the code for now.)"

	| a |
	a := OrderedCollection new: args size + 1.
	a add: receiver.
	a addAll: args.
	Transcript show: 'applying gf'; cr.
	^self apply: a!

computeApplicableMethods: args
	"Return a collection of all the applicable methods, regardless of
	qualifiers, given the current
	arguments..."

	| am |
	am := multiMethods values select: [:m | m matches: args].
	^am! !

!GenericFunction methodsFor: 'multimethod add/remove'!

add: aMultiMethod
	"Put this MultiMethod in our Dictionary of potentially applicable methods..."
	"Do some goaltending first..."

	self selector = aMultiMethod genericFunctionSelector unqualified ifFalse: [self error: 'Bad
multimethod selector'].
	multiMethods at: aMultiMethod multiMethodSelector put: aMultiMethod.
	"Note that this DM isn't installed yet. This is because we may not have ensured it has a
		selector to wrap yet."
	discriminatingMethods at: aMultiMethod mclass name ifAbsentPut: [
		DiscriminatingMethod on: selector unqualified inClass: aMultiMethod mclass]!

remove: aMultiMethod
	"If someone is trying to remove MultiMethod from the wrong GF, catch 'em here.
	Test the unqualified
	name so as to allow the before/after qualifier gambit."

	self selector = aMultiMethod genericFunctionSelector unqualified ifFalse: [self error:
'MultiMethod and GenericFunction selector mismatch'].
	multiMethods removeKey: aMultiMethod multiMethodSelector ifAbsent: [self error: 'Bad
multimethod selector'].
	"We need to reference count these, or otherwise correctly manage them..."
	"discriminatingMethods at: aMultiMethod mclass"
	Transcript show: '-- Removing: ' , self printString; cr; endEntry! !
```

```
!GenericFunction methodsFor: 'accessing '!

arity
	"Return the number of arguments our selector takes, including the receiver..."

	^self numArgs + 1!

discriminatingMethodFor: aMultiMethod
	"Return the DiscriminatingMethod that catches invocations a particular
	MultiMethod on a particular
	first argument."

	^discriminatingMethods at: aMultiMethod mclass name ifAbsent: [nil]!

discriminatingMethods
	"Return the discriminating methods for this GenericFunction..."

	^discriminatingMethods!

discriminatingMethods: aCollection
	^discriminatingMethods := aCollection!

methodCombination
	^methodCombination!

methodCombination: aMethodCombination
	^methodCombination := aMethodCombination!

multiMethods
	"Return the multimethods for this GenericFunction..."

	^multiMethods!

multiMethods: aDictionary
	"Store the multimethods for this GenericFunction..."

	^multiMethods := aDictionary!

numArgs
	"Return the number of arguments our selector takes, not including
	the receiver.  To include the receiver, use #arity..."

	^selector numArgs!

selector
	"Return the selector for this GenericFunction..."

	^selector!

selector: aSelector
	"Return the RHS to allow cascaded assigments."

	^selector := aSelector! !

!GenericFunction methodsFor: 'test'!

trash! !
```

```
!GenericFunction methodsFor: 'initialization'!

initialize
	"Start with the SimpleMethodCombination scheme for now.
	Standard will be a better default once it is working correctly..."

	self methodCombination: SimpleMethodCombination new.
	multiMethods := Dictionary new: 10.
	self discriminatingMethods: (Dictionary new: 10)! !
"-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- "!

GenericFunction class
	instanceVariableNames: ''!


!GenericFunction class methodsFor: 'class initialization'!

initialize
	"Initialize the GenericFunction class by creating an empty registry..."
	"GenericFunction initialize"

	GenericFunctionRegistry := IdentityDictionary new: 10! !

!GenericFunction class methodsFor: 'accessing'!

genericFunctions
	"Return the registry..."
	"GenericFunction genericFunctions"

	^GenericFunctionRegistry! !

!GenericFunction class methodsFor: 'instance creation'!

new
	"Don't allow uncommited GFs to be minted..."

	self shouldNotImplement!

new: anObject
	"Don't allow uncommited GFs to be minted for now..."

	self shouldNotImplement!

on: aSelector
	"Create a new GenericFunction for the indicated selector..."
	"If we've already created a generic function for this selector, return it.
	Otherwise, create a new generic function object, and register it.
	In this way, we use GenericFunction class as both a GenericFunction factory
	and repository..."
	"GenericFunction on: #at:put:"
	"An empty registry means we haven't been initialized, or someone
	is up to mischief. In either case, the reinitialization is indicated..."

	| selector |
	GenericFunctionRegistry isNil ifTrue: [self initialize].
	selector := MultiMethod unqualifiedSelectorFor: aSelector.
	^GenericFunctionRegistry at: selector
		ifAbsent:
			["If a GF already exists for this selector, return it, otherwise, create
one..."
			| gf |
			gf := super new.
			gf initialize.
			gf selector: selector.
			GenericFunctionRegistry at: selector put: gf.
			gf]! !
```

```
!GenericFunction class methodsFor: 'testing'!

isGenericFunction: aSelector
	"Let the caller know whether we've got this selector in the registry..."
	"We might want to hide the registry mechanism behind some protocol at some point..."

	| selector |
	selector := MultiMethod unqualifiedSelectorFor: aSelector.
	^GenericFunctionRegistry includesKey: selector! !
GenericFunction initialize!

Object subclass: #Medium
	instanceVariableNames: ''
	classVariableNames: ''
	poolDictionaries: ''
	category: 'MultiMethod-Menagerie'!

```

## Medium subclass: #Land

```
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MultiMethod-Menagerie'!
```

## Object subclass: #Animal

```
        instanceVariableNames: 'legs '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MultiMethod-Menagerie'!
```

```
!Animal methodsFor: 'ethology'!

AfterMoveThrough: medium <Medium>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'AFTER <Animal>AfterMoveThrough<Medium>...'; cr; endEntry!

BeforeMoveThrough: medium <Medium>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'BEFORE <Animal>BeforeMoveThrough<Medium>...'; cr; endEntry!

AfterMoveThrough: a1 "* * * DiscriminatingMethod stub method... * * *"  ^self!

BeforeMoveThrough: a1 "* * * DiscriminatingMethod stub method... * * *"  ^self!

installedDiscriminatingMethodSource
        "* * * The method you are looking at is an INSTALLED DiscriminatingMethod for the current
selector. It's actual
        code is hidden. * * *"
        "Don't change and accept this code unless you want a method named
#installedDiscriminatingMethodSource..."
        "The source pointer for this method is copied to that of each DiscriminatingMethod when it
is installed..."

        ^self!

speak
        Transcript show: 'I am an animal';cr;endEntry.
        ^66! !
```

```
!Animal methodsFor: 'accessing'!

legs
        ^legs!

legs: n
        ^legs := n! !
```

## Animal subclass: #Mammal

```
        instanceVariableNames: 'hasHair aquatic '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'MultiMethod-Menagerie'!


!Mammal methodsFor: 'accessing'!

aquatic
        ^aquatic!

aquatic: aBoolean
        ^hasHair := aquatic!

hasHair
        ^hasHair!

hasHair: aBoolean
        ^hasHair := aBoolean! !

!Mammal methodsFor: 'ethology'!

AfterMoveThrough: medium <Medium>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'AFTER <Mammal>AfterMoveThrough<Medium>...'; cr; endEntry!

BeforeMoveThrough: medium <Medium>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'BEFORE <Mammal>BeforeMoveThrough<Medium>...'; cr; endEntry!

moveThrough: medium <Land>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'Mammals can usually move over land'; cr; endEntry!

moveThrough: medium <Medium>
        "(Mammal new) moveThrough: (Water new)"
        "(Mouse new) moveThrough: (Land new)"
        "(Mammal new) speak"

        Transcript show: 'Mammals can usually move adequately through any Medium for at least brief
periods of time...'!

AfterMoveThrough: a1 "* * * DiscriminatingMethod stub method... * * *"  ^self!

BeforeMoveThrough: a1 "* * * DiscriminatingMethod stub method... * * *"  ^self!

speak
        Transcript show: 'I am am a Mammal';cr;endEntry.
        ^super speak! !
```

## Mammal subclass: #Mouse

```
	instanceVariableNames: 'favoriteCheese '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'MultiMethod-Menagerie'!


!Mouse methodsFor: 'accessing'!

favoriteCheese
	^favoriteCheese!

favoriteCheese: cheese
	^favoriteCheese := cheese! !

!Mouse methodsFor: 'ethology'!

moveThrough: medium <Land>
	"GenericFunction initialize"
	"(Mammal new) moveThrough: (Water new)"
	"(Mouse new) moveThrough: (Land new)"
	"(Mammal new) speak"
	"(GenericFunction on: #moveThrough:) inspect"
	"(GenericFunction on: #moveThrough:) methodCombination: StandardMethodCombination new"
	"(GenericFunction on: #moveThrough:) methodCombination: SimpleMethodCombination new"
	"(GenericFunction on: #moveThrough:) methodCombination: SubStandardMethodCombination new"

	thisContext callNextMethodWithArguments: nil.
	Transcript show: 'Mice move quite well over land'; cr; endEntry!

installedDiscriminatingMethodSource
	"* * * The method you are looking at is an INSTALLED DiscriminatingMethod for the current
selector. It's actual
	code is hidden. * * *"
	"Don't change and accept this code unless you want a method named
#installedDiscriminatingMethodSource..."
	"The source pointer for this method is copied to that of each DiscriminatingMethod when it
is installed..."

	^self!

speak
	"Mouse new speak"

	Transcript show: 'I am am a Mouse'; cr; endEntry.
	^super speak! !
```

```
!Compiler class methodsFor: 'simulating'!

simulateAgain: expr
	"Simulate the evaluation of expr,
	without flushing the cache"
	| method ctx root dummy |
	method := Compiler compileClass: Object selector: #doIt source: 'doIt ^(', expr, ')'.
	dummy := CompiledMethod bytes: #() literals: #() numArgs: 0 numTemps: 0 maxDepth: 1
needsFrame: true hybrid: false forContext: false.
	dummy mclass: Object.
	root := MethodContext sender: nil receiver: nil method: dummy arguments: #().
	ctx := MethodContext sender: root receiver: nil method: method arguments: #().
	[ctx := ctx interpretNextInstructionFor: ctx.
	(ctx isKindOf: Array) ifTrue: [ctx := ctx at: 1]. "<BF>"
	ctx == root]
		whileFalse: [].
	^ctx top! !

!Symbol methodsFor: 'converting'!

unqualified
	"Pass this to where it doesn't really belong, for now..."

	^MultiMethod unqualifiedSelectorFor: self! !

!SequenceableCollection methodsFor: 'copying'!

rest
	"This goes well with first and last..."
	"For frustrated Lisp'ers who miss CDR..."
	"#() rest #()"
	"#(1) rest #()"
	"#(1 2) rest #(2)"
	"#(1 2 3) rest #(2 3)"
	"'brat' rest 'rat'"

	self size < 2 ifTrue: [^self copyEmpty: 0].
	^self copyFrom: 2 to: self size! !


!Context methodsFor: 'simulation-debugger'!

sendFlagSet: s
	"Sneak a value in for this flag..."

	Send := s! !
```

```
!Context methodsFor: 'multimethod support'!

callNextMethod
	"If you needs an example of a method where transformations that would
	normally be considered as
	behavior preserving might not be, consider
	#applyGenericFunction:withMethods:andArguments: in
	the presence of this method..."

	| applyContext gf am args mc |
	applyContext := self firstContextWithSelector:
#applyGenericFunction:withMethods:andArguments:.
	gf := applyContext tempAt: 1.
	am := applyContext tempAt: 2.
	args := applyContext tempAt: 3.
	"mc := applyContext receiver outerContext receiver."
	mc := applyContext receiver.
	am size <= 1 ifTrue: [^nil].
	mc
		applyGenericFunction: gf
		withMethods: am rest
		andArguments: args!

callNextMethodWith: a1
	^self callNextMethodWithArguments: (Array with: a1)!

callNextMethodWith: a1 with: a2
	^self callNextMethodWithArguments: (Array with: a1 with: a2)!

callNextMethodWith: a1 with: a2 with: a3
	^self callNextMethodWithArguments: (Array
			with: a1
			with: a2
			with: a3)!

callNextMethodWithArguments: arguments
	"Find a MethodCombinationContext on the stack, and hand off to it. When
	it's done, return its
	result..."

	applyContext := self firstContextForInstanceOf: MethodCombinationContext.
	methodCombinationContext := applyContext receiver.
	result := methodCombinationContext applyNextWithArguments: arguments.
	^result!
```

```
firstContextForInstanceOf: aClass
	| c |
	c := self.
	[c isNil ifTrue: [^nil].
	(c receiver isKindOf: aClass)
		ifTrue: [^c].
	Transcript show: c selector; cr; endEntry.
	c == self sender ifTrue: [^nil].
	c := self sender] repeat!

firstContextWithSelector: aSelector
	"With a doIt, you should find the context below..."
	"thisContext firstContextFor: #performMethod:arguments:"

	| c |
	c := self.

	[c isNil ifTrue: [^nil].
	c selector == aSelector ifTrue: [^c].
	Transcript show: c selector; cr; endEntry.
	c == self sender ifTrue: [^nil].
	c := self sender] repeat!

hasNextMethod
	"ERROR: This should be handed off to the MethodCombinationContext!!!!!!..."

	| applyContext am |
	applyContext := self firstContextWithSelector: 
#applyGenericFunction:withMethods:andArguments:.
	am := applyContext tempAt: 2.
	"If there are any methods left, say 'right this way'..."
	^am rest isEmpty not!

!Context class methodsFor: 'class initialization'!

sendFlagSet: s
	"Sneak a value in for this flag..."

	Send := s! !

!ParameterNode methodsFor: 'multimethod support'!

getTypeInfo
	"Since we have some test methods with type expressions, we don't flag them for now,
	but arbitrarily treat them as default specializers..."

	type isNil
		ifTrue: [^ClassSpecializer on: Object name]
		ifFalse: [(type isKindOf: VariableNode)
				ifTrue: [^ClassSpecializer on:  type name asSymbol]
				ifFalse: ["self error: 'Bad Type Expression'"
					^Specializer default]]!

hasTypeInfo
	"For a ParameterNode, just see if a type expression was given at all..."

	^type ~= nil! !
```

```
!MethodNode methodsFor: 'multimethod support'!

getExperimentalSelector
	"(MultiTest parseTreeForMethod: #Junk:with:) getExperimentalSelector
#'Junk:<MultiTest>with:<Junk>'"
	"(MultiTest parseTreeForMethod: #'Junk:<MultiTest>with:<Junk>') getExperimentalSelector"

	self hasTypeInfo ifFalse: [^selector]
		ifTrue:
			[| specializers newSelector words |
			newSelector := ''.
			specializers := self getTypeInfo.
			words := selector isKeyword ifFalse: [Array with: selector]
						ifTrue: [selector keywords].
			words with: specializers do: [:w :s | newSelector := newSelector , w , s
printString].
			^newSelector asSymbol]!

getOriginalSelector
	"Since we intercept the selector accessor and return a
	MultiMethod selector (using getSelector) in
	response to the selector accessor, we provide this method
	for cases where the original selector is
	needed..."

	^selector!

getSelector
	"If we are not a multimethod, return the usual selector, otherwise
	make a dotted selector with the
	types at the beginning..."

	self hasTypeInfo ifFalse: [^selector]
		ifTrue:
			[| specializers newSelector |
			newSelector := ''.
			specializers := self getTypeInfo.
			specializers do: [:s | newSelector := newSelector , s specializerClass
printString].
			newSelector := newSelector , '.' , selector.
			^newSelector asSymbol]!

getTypeInfo
	^block getTypeInfo!

hasTypeInfo
	"Ask our code block, which is a BlockNode, about this..."

	^block hasTypeInfo! !

!MethodNode methodsFor: 'accessing'!

selector
	"Call into the multimethod code for this..."
	"^selector"

	^self getSelector! !
```

```
!BlockNode methodsFor: 'multimethod support'!

getTypeInfo
	"Glean the type information each of our arguments..."

	^arguments collect: [:arg | arg getTypeInfo]!

hasTypeInfo
	"If any of our arguments had a type specifier, flag us as a potential MultiMethod..."

	arguments do: [:arg | arg hasTypeInfo ifTrue: [^true]].
	^false! !

!Behavior methodsFor: 'creating method dictionary'!

removeSelector: selector
	"Assuming that the message selector is in the receiver's method dictionary,
	remove it.  If the selector is not in the method dictionary, create an error
	notification."

	| method |
	method := methodDict at: selector ifAbsent: [self error: 'Removing non-existant method'].
	(method isKindOf: MultiMethod) ifTrue: [method removeFromGenericFunction].
	"Resume stock code here..."
	methodDict removeKey: selector.
	self flushVMmethodCacheEntriesFor: selector! !

!Behavior methodsFor: 'accessing method dictionary'!

parseTreeForMethod: aSymbol
	^self parserClass new
		parse: (self sourceCodeAt: aSymbol) readStream
		class: self
		noPattern: false
		context: nil
		notifying: (SilentCompilerErrorHandler new failBlock: [^nil])
		builder: ProgramNodeBuilder new
		saveComments: true
		ifFail: [^nil]! !

!Behavior methodsFor: 'compiling'!

compile: code notifying: requestor ifFail: failBlock
	"Compile the argument, code, as source code in the context of the receiver and
	install the result in the receiver's method dictionary. The argument requestor is to
	be notified if an error occurs. The argument code is either a string or an
	object that converts to a string or a PositionableStream on an object that
	converts to a string. This method does not save the source code.
	Evaluate the failBlock if the compilation does not succeed."

	| methodNode selector |
	methodNode := self compilerClass new
				compile: code
				in: self
				notifying: requestor
				ifFail: failBlock.
	methodNode hasTypeInfo ifTrue: [methodNode := (MethodWrapperCompiler new methodClass:
MultiMethod)
					compile: code
					in: self
					notifying: requestor
					ifFail: failBlock].
	selector := methodNode selector.
	self addSelector: selector withMethod: methodNode generate.
	^selector!

```

```
!CompiledCode methodsFor: 'testing'!

isDiscriminatingMethod
	"Part of the GenericFunction code..."

	^false!

isMultiMethod
	"Part of the GenericFunction code..."

	^false!

isWrapper
	"Part of the method wrapper code..."

	^false! !


!MethodNodeHolder methodsFor: 'multimethod support'!

getExperimentalSelector
	"Duct tape..."

	^methodNode getExperimentalSelector!

getOriginalSelector
	"More duct tape..."

	^methodNode getOriginalSelector!

getTypeInfo
	"Forward this rascal too..."

	^methodNode getTypeInfo!

hasTypeInfo
	"Forward this rascal..."

	^methodNode hasTypeInfo! !
```

```
!ClassDescription methodsFor: 'compiling'!

compile: code classified: heading notifying: requestor
	"Compile the argument, code, as source code in the context of the receiver
	and install the result in the receiver's method dictionary under the
	classification indicated by the second argument, heading. The third
	argument, requestor, is to be notified if an error occurs. The argument code
	is either a string or an object that converts to a string or a
	PositionableStream on an object that converts to a string."

	| selector method |
	selector := self
				compile: code
				notifying: requestor
				ifFail: [^nil].
	(methodDict at: selector)
		sourcePointer: (SourceFileManager default
				storeMethodSource: code asString
				class: self
				category: heading
				safely: true).
	self organization classify: selector under: heading.

	"Begin the multimethod alterations here..."
	"We could almost call the current method recursively, if we generated a better method
string..."
	method := methodDict at: selector.
	((method isKindOf: MultiMethod)
		and: [(methodDict at: method genericFunctionSelector ifAbsent: [nil]) isNil])
		ifTrue:
			[| sel met |
			sel := method genericFunctionSelector.
			met := DiscriminatingMethod createEmptyMethodFor: sel.
			self addSelector: sel withMethod: met.
			(methodDict at: sel) sourcePointer: (SourceFileManager default
					storeMethodSource: (DiscriminatingMethod emptyMethodFor: sel)
					class: self
					category: heading
					safely: true).
			self organization classify: sel under: heading.
			"Give the MM a change to install the DM here..."
			method install].

	"Resume normal code here..."
	^selector!
```

```
compile: code notifying: requestor ifFail: failBlock
	"Intercept this message in order to remember system changes."

	| methodNode compiledMethod selector |
	methodNode := self compilerClass new
				compile: code
				in: self
				notifying: requestor
				ifFail: failBlock.
	selector := methodNode selector.
	"If this method has type information, compile it again as a MultiMethod..."
	methodNode hasTypeInfo ifTrue: [methodNode := (MethodWrapperCompiler new methodClass:
MultiMethod)
					compile: code
					in: self
					notifying: requestor
					ifFail: failBlock].
	"Resume old code here..."
	(methodDict includesKey: selector)
		ifTrue: [ChangeSet current changeSelector: selector class: self]
		ifFalse: [ChangeSet current addSelector: selector class: self].
	compiledMethod := methodNode generate.
	"If this is a MultiMethod, find it's funny selector, and complete
	it's initialization..."
	(compiledMethod isKindOf: MultiMethod)
		ifTrue:
			[selector := methodNode getExperimentalSelector.
			selector := ('<' , self printString , '>' , selector) asSymbol.
			compiledMethod
				on: methodNode getOriginalSelector
				using: selector
				withArgumentSpecializers: methodNode getTypeInfo].
	"Resume old code here..."
	self addSelector: selector withMethod: compiledMethod.
	^selector!


!Parser methodsFor: 'initialize-release'!

initScanner
	"Turning on extendedLanguage enables <Xxx> type expressions in method defintions..."

	super initScanner.
	typeTable := TypeTable.		"Default language choice:"
	oldLanguage := true.
	newLanguage := true.
	extendedLanguage := true! !

```

```
!Parser methodsFor: 'expression types-^value/error'!

argument
	| arg argType oldPos oldToken oldTokenType |
	tokenType == #word ifFalse: [^self expected: 'Argument name'].
	arg := builder declareVariableName: token.
	arg sourcePosition: mark.
	self scanToken.
	"Save these elements of the parser's state in case we see a pragma..."
	oldPos := source position.
	oldToken := token.
	oldTokenType := tokenType.
	"If this turns out to be a pragma, try to back up and say there was no type <gulp>..."
	(extendedLanguage and: [token == #<])
		ifTrue:
			[self scanToken.
			(tokenType == #keyword and: [self pragmaKeywords includes: token asSymbol])
				ifTrue:
					[source position: oldPos - 1.
					token := oldToken.
					tokenType := oldTokenType.
					argType := nil]
				ifFalse:
					[self typeExpression ifFalse: [self expected: 'Type 
expression'].
					argType := parseNode.
					(self matchToken: #>)
						ifFalse: [self expected: '>']]]
		ifFalse: [argType := nil].
	^builder newParameterVariable: arg type: argType!
```

```
;;;
;;; animal -- A rather prosaic taxonomic example class hierarchy...
;;;
(defclass animal ()
  ;; Define the slots for generic beasts here...
  ((species :accessor species :initarg :species :initform "An Animal")
   (legs :accessor legs :initarg :legs :initform 0 :type fixnum)
   ;; Define a slot with the :class allocation-type.  This is
   ;; similar to a Smalltalk class variable.  For fun, try giving it
   ;; a type.  (Vishnu is the second member of the three member Hindu
   ;; trinity...)
   (creator :accessor creator :initform "Brahma" :type (string 255)
            :allocation :class :documentation "Shiva is the destroyer..."))
  ;; Use the usual metaclass.  STANDARD-CLASS is the default, so
  ;; the line below is, in effect, a no-op...
  (:metaclass standard-class) ;An inline (single) comment (FRED does nothing)...
  ;; Give no useful information here...
  (:documentation "I am a generic beast..."))

;;;
;;; mammal -- Animals that give milk...
;;;
(defclass mammal (animal)
  ((breasts :accessor breasts :initform 2 :initarg :breasts))
  (:documentation "I give milk...")
  ;; Try out the :default-initargs feature here...
  (:default-initargs :species "Some sort of Mammal" :legs 2))

;;;
;;; mouse -- A rodent...
;;;
(defclass mouse (mammal)
  ((favorite-cheese :accessor favorite-cheese :initform "Swiss"
                    :initarg :favorite-cheese))
  (:documentation "I'm a mouse...")
  ;;:defaut-initargs are inherited.  This is the only class option
  ;;that is, says Keene...
  (:default-initargs :species "Mouse" :legs 4)
)

;;;
;;; speak -- A generic speak function...
;;;
(defgeneric speak (who) ;Takes just a dummy lambda list...
  (:documentation "Will the animals talk back?")
  (:argument-precedence-order who))

(defmethod speak ((who animal))
  (format t "I'm an animal:  ~A~%" who))

(defmethod speak :before ((who animal))
  (format t "Before specializing on animal:  ~A~%" who))

(defmethod speak :after ((who animal))
  (format t "After specializing on animal:  ~A~%" who))
```

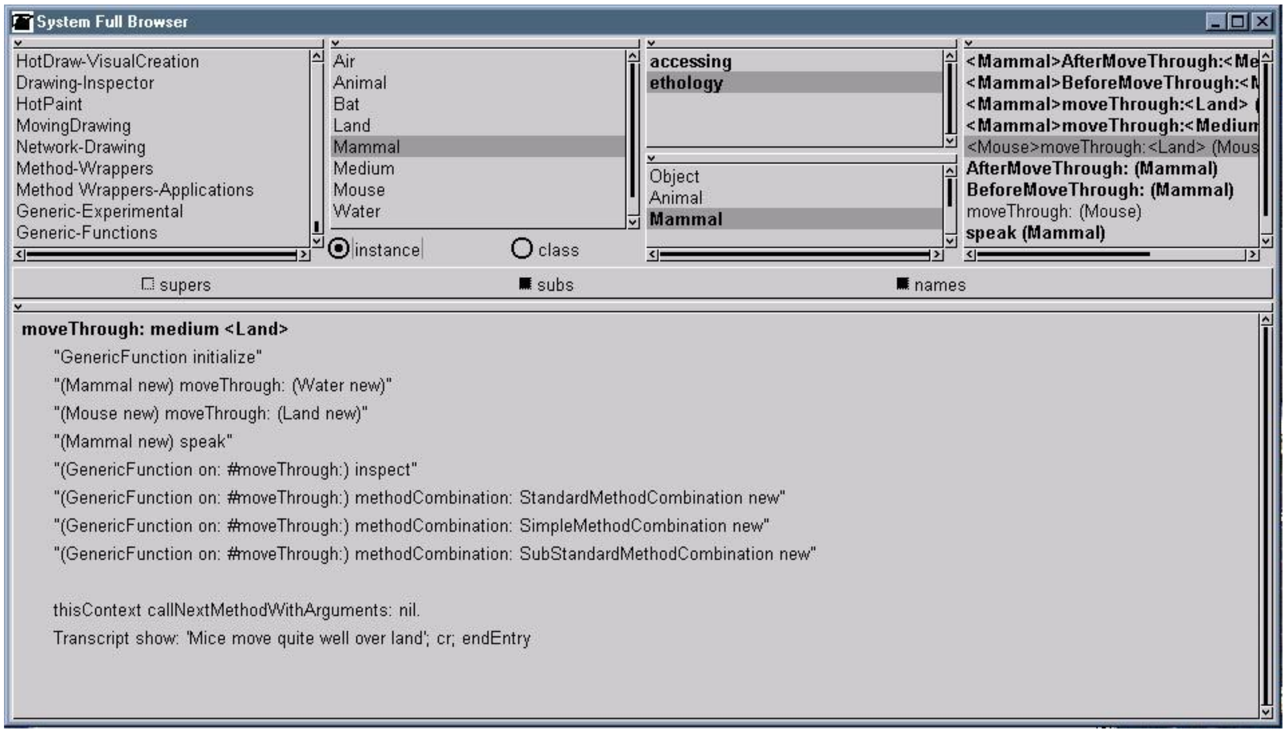
```
;;;
;;; speak mouse -- primary method (This returns the value)...
;;;
(defmethod speak ((who mouse))
  (format t "I am but a mouse...~%")
  (format t "call-next-method:  ")
  ;;This code illustrates the use of next-method-p.  If call-next-method
  ;;is called in a context where no next most specific method is
  ;;applicable, CLOS will signal an array.  (Contrast this with the
  ;;treatment of usual-xxx in Object Lisp.)  The next-method-p predicate
  ;;can be used to avoid such mishaps.  In the event that no next method
  ;;is defined, we call the CL error function...
  (if (next-method-p)
    (call-next-method)
    (error "No next method (Should never occur...), who:  ~A~%" who))
  ;;Return an atypical, easy to identify result...
  99)

;;;
;;; Called for side effects, doesn't affect the value.  In the core framework,
;;; any values of before- or after-methods are ignored, and the generic function
;;; returns the value of the primary method.  If there is no applicable
;;; primary method, an error is signaled...
;;;
(defmethod speak :before ((who mouse))
  (format t "Before:  Eek Eek...~%"))

;;;
;;; Called for side effects, doesn't affect the value...
;;;
(defmethod speak :after ((who mouse))
  ;;I can't directly modify the returned value, but I can cause side effects...
  (format t "After:  Eek Eek...~%"))

;;;
;;; :around methods use call-next-method to explicitly control
;;; the sequencing of methods.  The :around methods are called
;;; from most to least specific, and then the entire "core framework"
;;; is called.  Before-methods, primary-methods, and after-methods comprise
;;; CLOS's core framework.  The call-next-method mechanism is also
;;; used in primary methods to invoke "shadowed" methods.  Before-methods
;;; are called in most-specific to least-specific (most general) order.
;;; After-methods are called in least-specific to most-specific order.
;;; The core framework is said to employ a "declarative" approach.
;;; The call-next-method scheme is thought of as an "imperative" approach.
;;; If call-next-method is invoked with no arguments, the original arguments
;;; are used...
;;;
;;; (An additional novelty:  The &aux variable...)
;;;
(defmethod speak :around ((who mouse) &aux result)
  (format t "In mouse :around method, before call-next-method...~%")
  (setf result (call-next-method))
  (format t "In mouse :around method, after call-next-method:  ~A~%" result)
  result)
```

1

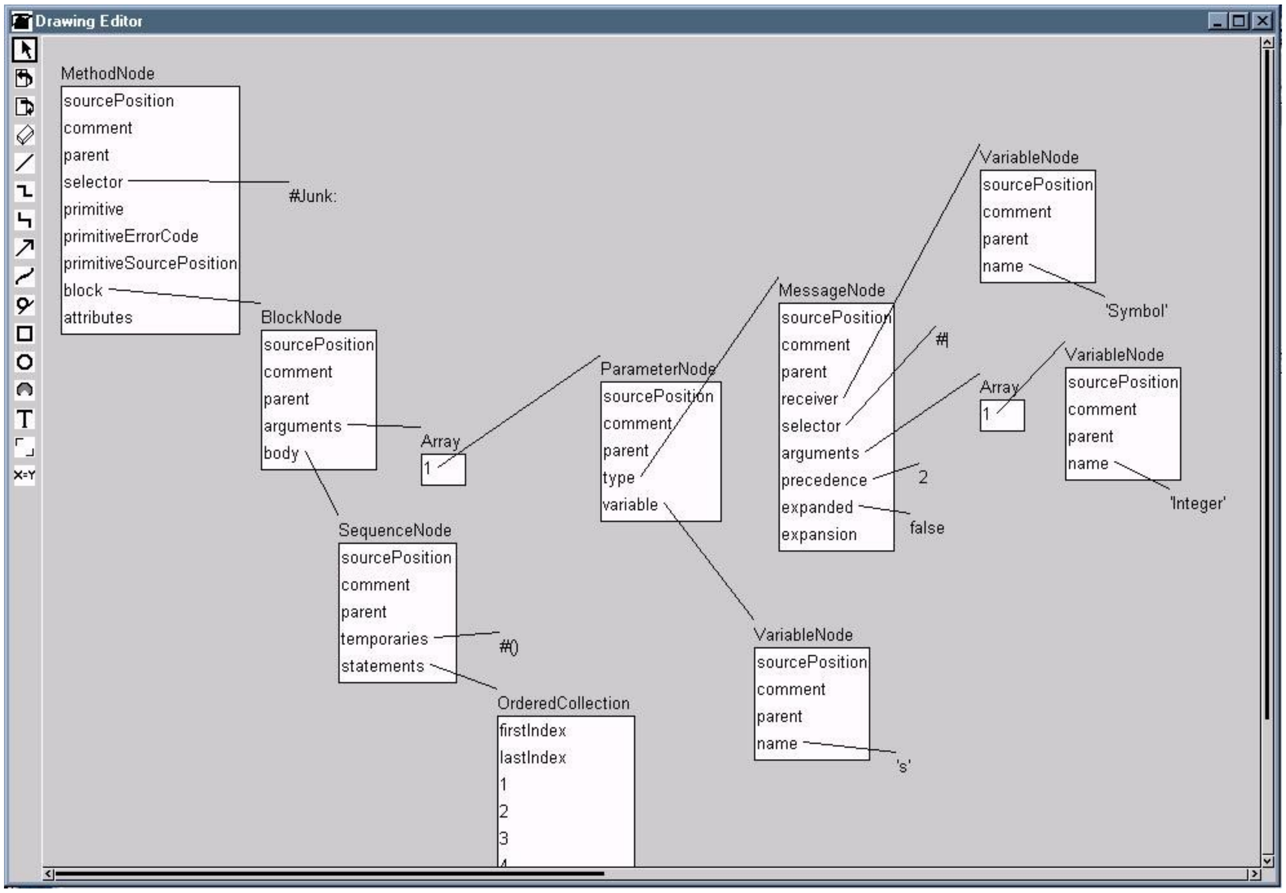


```
moveThrough: medium <Land>
	"GenericFunction initialize"
	"(Mammal new) moveThrough: (Water new)"
	"(Mouse new) moveThrough: (Land new)"
	"(Mammal new) speak"
	"(GenericFunction on: #moveThrough:) inspect"
	"(GenericFunction on: #moveThrough:) methodCombination: StandardMethodCombination new"
	"(GenericFunction on: #moveThrough:) methodCombination: SimpleMethodCombination new"
	"(GenericFunction on: #moveThrough:) methodCombination: SubStandardMethodCombination new"

	thisContext callNextMethodWithArguments: nil.
	Transcript show: 'Mice move quite well over land'; cr; endEntry
```

2

1


Drawing Editor
MethodNode
sourcePosition
comment
parent
selector
primitive
primitiveErrorCode
primitiveSourcePosition
block
attributes
#Junk:
BlockNode
sourcePosition
comment
parent
arguments
body
Array
1
SequenceNode
sourcePosition
comment
parent
temporaries
statements
#()
OrderedCollection
firstIndex
lastIndex
1
2
3
ParameterNode
sourcePosition
comment
parent
type
variable
VariableNode
sourcePosition
comment
parent
name
's'
MessageNode
sourcePosition
comment
parent
receiver
selector
arguments
precedence
expanded
expansion
#|
2
false
Array
1
VariableNode
sourcePosition
comment
parent
name
'Symbol'
VariableNode
sourcePosition
comment
parent
name
'Integer'

2