Technical report: A classification of runtime reflective operations in Pharo

Iona Thomas[1], Stéphane Ducasse[1], Pablo Tesone[1] and Guillermo Polito[1]

[1]*Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL.*

Abstract

This is a companion technical report of the article: *"Pharo: a reflective language – a first systematic analysis of reflective apis"* published at IWST 2023.

Reflective operations are powerful APIs that allow one to build new tools and alter the architecture of the language. They are used extensively in the Pharo tool ecosystem and new ones have been added when the need arose for new features. However, as the Pharo reflective API evolved organically with the language, we lacked information on which are the reflective operations inside Pharo. Pharo evolved from Squeak since 2008, and Squeak itself builds on the Smalltalk original API. At this point, the documentation of the reflective API was partial and outdated.

In this report, we propose a categorization and an up-to-date catalog of the available runtime reflective operations in Pharo. For each category, we provide a short description and a list of the corresponding reflective methods.

1. Overview of the reflective APIs and their categories

This is a companion technical report of the article: *"Pharo: a reflective language – a first systematic analysis of reflective apis"* published at IWST 2023 [TDTP23]. We extracted the catalog and the categories so that other articles can refer to such a long and dry analysis.

We present a catalog of the reflective features in Pharo. Figure 1 shows the categories we propose to classify these features. The following sections will each present a category briefly and list the corresponding methods. For a more in-depth presentation and analysis of these categories see [TDTP23].

2. Object inspection

The first family of reflective operations is centered around object inspection. Rivard describes these operations in the *Meta-Operations* category, but he groups together inspection and modification. Our category is composed of three subcategories:

State inspection. These operations allow one to access the number of indexable variables, and the values of instance variables, if they include a specific object

- Context»objectSize:

iona.thomas@inria.fr (I. Thomas); stephane.ducasse@inria.fr (S. Ducasse); pablo.tesones@inria.fr (P. Tesone); guillermo.polito@inria.fr (G. Polito)

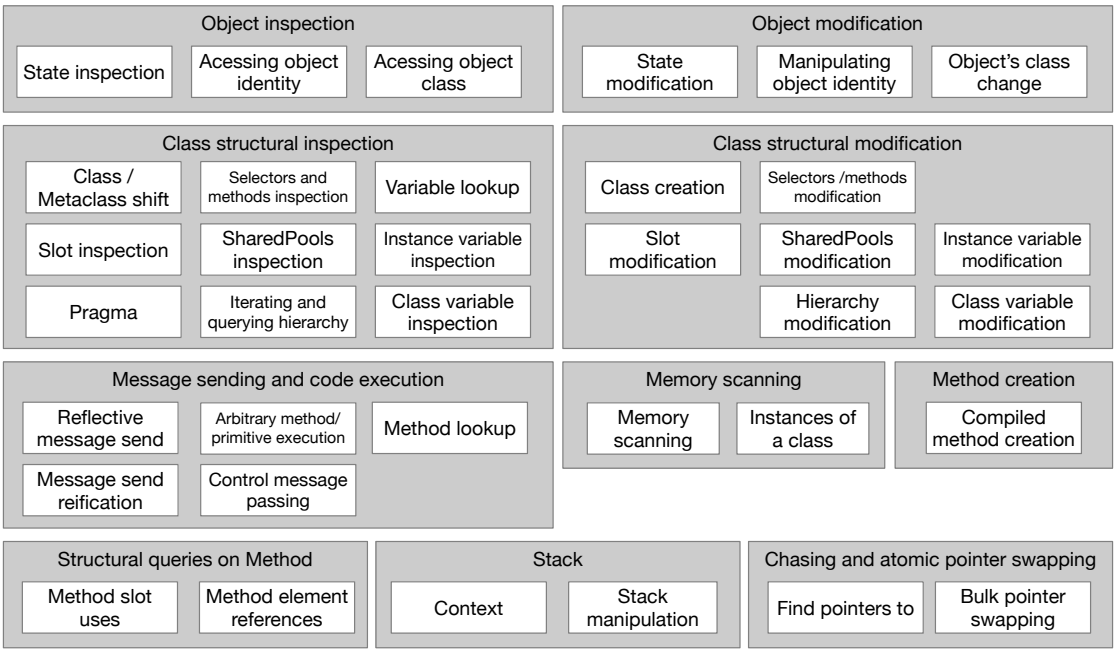

Figure 1: Overview of the reflective APIs.

- Object»instVarAt:
- Object>instVarNamed:
- ProtoObject»instVarsInclude:

Accessing object identity. These operations allow one to identify one object

- ProtoObject»basicIdentityHash
- ProtoObject»identityHash

Accessing object class. This operation allows one to access the class of an object :

- ProtoObject»class
- Context»objectClass:

3. Object modification

The second family of reflective operations is centered around object modification. It is the counterpart of the first one and it is composed of *State modification*, *Manipulating object identity*, and *Object's class change*.

State modification. These operations allow one to set the value of an instance variable.

- Object»instVarAt:put: Write an instance variable using its index
- Object»instVarNamed:put: Write an instance variable using its name.

Manipulating object identity. As a side effect of reference swapping, the reflective operation ProtoObject»becomeForward:copyHash: manipulates the hash of an object.

Object's class change. These operations allow one to change the class of an object to another one:

- Behavior»adoptInstance:
- Metaclass»adoptInstance:from:
- Object»primitiveChangeClassTo:

4. Class structural inspection

This category groups reflective APIs on the query over the class structure and its constituents: methods, variables (instance/class/slots). It is composed of the following APIs:

- *Class/metaclass shift* to support the navigation from a class to its metaclass and the inverse,
- *Iterating and querying hierarchy* to support hierarchy analysis with, in particular, testing for belonging to a specific class or class hierarchy (*e.g.,* isKindOf:),
- *Instance variable inspection*, *Class variable inspection*, *Shared pool inspection*, *Slot inspection* all deal about variables. *Slot inspection* provides a higher level view compared to instance variables. Some slots can come from traits. Therefore there is a difference between localSlots (not coming from traits) and slots (taking into account that they may come from traits),
- *Selectors and method inspection* supports if a class has (abstract) methods, an object responds to a specific selector, and all the classes implementing a set of selectors.
- *Variable lookup* supports the access to the binding of a variable.
- *Pragmas* supports the query and iteration of class annotations (named pragmas in Pharo).
- *Class kind testing* supports the testing of the state of a class from a system perspective (obsolete, anonymous...).

Class/metaclass shift. Those operations deal with testing and navigating between the class and instance side of classes.

- ClassDescription»classSide / Class»classSide / Metaclass»classSide
- ClassDescription»instanceSide / Class»instanceSide / Metaclass»instanceSide
- ClassDescription»isClassSide / CompiledMethod»isClassSide
- ClassDescription»isInstanceSide
- Metaclass»isMetaclassOfClassOrNil
- ClassDescription»hasClassSide / Class»hasClassSide / Metaclass»hasClassSide
- Object»isClass / Class»isClass / Trait»isClass / Metaclass»isClass
- Behavior»isMeta / ClassDescription»isMeta / Metaclass»isMeta

Iterating and querying hierarchy. These operations allow one to query the hierarchy of a class and iterate over those. They allow one to access superclasses and subclasses, testing for common ancestry in the class hierarchy, and testing that an object is an instance or subinstance of a class.

- Class»subclasses / MetaClass»subclasses / UndefinedObject»subclasses
- Behavior»allSubclasses
- Behavior»withAllSubclasses
- Behavior»obsoleteSubclasses / Metaclass»obsoleteSubclasses
- Class»hasSubclasses
- Behavior»superclass
- Behavior»allSuperclasses
- Behavior»withAllSuperclasses
- Behavior»allSuperclassesIncluding:
- Behavior»allSubclassesWithLevelDo:startingLevel:
- Class»subclassesDo: / Metaclass»subclassesDo: / UndefinedObject»subclassesDo:
- Behavior»allSubclassesDo:
- Behavior»withAllSubclassesDo:
- Behavior»allSuperclassesDo: / UndefinedObject»allSuperclassesDo:
- Behavior»withAllSuperclassesDo:
- Behavior»withAllSuperAndSubclasses
- Behavior»withAllSubAndSuperclassesDo:
- Object»isKindOf:
- Object»isMemberOf:
- Behavior»kindOfSubclass
- Class»commonSuperclassWith: / UndefinedObject»commonSuperclassWith:
- Behavior»whichSuperclassSatisfies:
- Behavior»inheritsFrom:
- Behavior»includesBehavior:
- Behavior»isRootInEnvironment
- Behavior»selectSuperclasses:
- Behavior»selectSubclasses:

Instance variable inspection. These operations allow one to query and get instance variable names, iterate over those, and get their index and the class defining a specific instance variable.

- Behavior»instVarNames / ClassDescription»instVarNames
- Behavior»allInstVarNames / ClassDescription»allInstVarNames
- ClassDescription»instanceVariableNamesDo:
- ClassDescription»hasInstVarNamed:
- Behavior»definedVariables / Class»definedVariables

- ClassDescription»allInstVarNamesEverywhere
- ClassDescription»classThatDefinesInstVarNamed:
- Behavior»whichClassDefinesInstVar:
- Behavior»instSize
- Object»basicSize / Context»basicSize
- Behavior»instVarNames

Class variable inspection. These operations allow one to query and test class variables, get class variable names, get their values, iterate over those, class defining a specific instance variable and who is using them.

- Class»classVariables / Metaclass»classVariables
- Behavior»allClassVarNames
- Behavior»classVarNames / Class»classVarNames / Metaclass»classVarNames
- Class»hasClassVariable:
- Class»hasClassVarNamed: / Metaclass»hasClassVarNamed:
- Class»classVariableNamed:ifAbsent:
- Class»definesClassVariable:
- Class»definesClassVariableNamed:
- Class»readClassVariableNamed:
- ClassDescription»classThatDefinesClassVariable:
- Behavior»whichClassDefinesClassVar:
- Class»usesClassVarNamed:

The methods ClassDescription»classThatDefinesClassVariable: and Behavior»whichClassDefinesClassVar: looks the same and the mix between variable and var is prejudicial.

Shared pool inspection. These operations allow one to query and test on sharedPools and where there are used. They are important to navigate with the IDE.

- ClassDescription»sharedPools / Class»sharedPools
- Behavior»allSharedPools / ctClassDescription»allSharedPools / ctClass»allSharedPools
- Class»sharedPoolNames / Metaclass»sharedPoolNames
- ClassDescription»hasSharedPools / Class»hasSharedPools
- ClassDescription»sharedPoolOfVarNamed / Class»sharedPoolOfVarNamed
- Class»sharedPoolsDo:
- Class»classPool / Metaclass»classPool
- ClassDescription»usesLocalPoolVarNamed: / Class»usesLocalPoolVarNamed:
- ClassDescription»usesPoolVarNamed: / Class»usesPoolVarNamed:
- ClassDescription»includesSharedPoolNamed:

Slot inspection. These operations allow one to query and test on slots in a class, or read those in an object. This is a higher-level view compared to instance variables. Some slots can come from traits. Therefore there is a difference between localSlots (without traits) and slots (with).

- Behavior»instanceVariables (returns a collection of slots)
- Behavior»slots / ClassDescription»slots / TraitedMetaclass»slots
- ClassDescription»localSlots
- Behavior»allSlots / ClassDescription»allSlots
- ClassDescription»slotNames
- ClassDescription»hasSlotNamed:
- ClassDescription»slotNamed:
- ClassDescription»slotNamed:ifFound:
- ClassDescription»slotNamed:ifFound:ifNone:
- Object»readSlot:
- Object»readSlotNamed:
- ClassDescription»definesSlot:
- ClassDescription»definesSlotNamed:

Selectors and method inspection. These operations allow one to test if a class has (abstract) methods, an object responds to a specific selector, and all the classes that implement a set of selectors.

- Behavior»hasMethods / Class»hasMethods
- Behavior»hasAbstractMethods / Class»hasAbstractMethods
- Object»respondsTo:
- ClassDescription»classesThatImplementAllOf:

Getting the selectors and local selectors, iterating over them, and querying them :

- Behavior»includesSelector:
- Behavior»includesLocalSelector: / TraitedClass»includesLocalSelector: / TraitedMetaclass»includesLocalSelector:
- Behavior»isDisabledSelector:
- Behavior»isLocalSelector: / TraitedClass»isLocalSelector: / TraitedMetaclass»isLocalSelector:
- Behavior»selectors
- Behavior»selectorsDo:
- Behavior»selectorsWithArgs:
- Behavior»whichClassIncludesSelector:

Getting the methods, iterating over them, and querying them :

- Behavior»methods
- Behavior»methodNamed:
- Behavior»includesMethod:
- Behavior»methodsDo:
- Behavior»selectorsAndMethodsDo:

Variable lookup. These operations allow one to lookup a variable in the scope of the receiver.

- Behavior»classBindingOf: / SharedPool class»classBindingOf:
- Object»bindingOf: / Behavior»bindingOf: / Class»bindingOf: / MetaClass»bindingOf:
- Behavior»lookupVar: / ctContext»lookupVar: / ctSystemDictionnary»lookupVar:
- Behavior»lookupVar:declare: / ctContext»lookupVar:declare: / ctSystemDictionnary»lookupVar:declare:
- Behvior»lookupVarForDeclaration:

Pragmas. Pragmas are method annotations that got introduced both in VisualWorks and Pharo over the year. These operations (Class»pragmasClass»pragmasDo:)allow one to get all the pragmas of a class and iterate over them.

Class kind testing. Test various properties of classes: usage, anonymity, obsolescence.

- Behvior»isAnonymous / Class»isAnonymous /MetaClass»isAnonymous
- Object»isClassOrTrait / Class»isClassOrTrait
- Behavior»isUsed / Class»isUsed / Trait»isUsed / Metaclass»isUsed
- Behavior»isObsolete / Class»isObsolete / Metaclass»isObsolete

5. Class structural modification

This category is the counterpart of the previous one. It is composed of the following APIs whose objectives are clear: *Hierarchy modification*, *Instance variable modification*, *Shared pool modification*, *Slot modification*, *Selector/Method modification*, *Old class creation*, *Fluid class creation*, and *Anonymous class creation*. It focuses on the modification of the structural relation a class has with its constituents.

Hierarchy modification. These operations allow one to edit the class hierarchy either by adding/removing/changing subclasses, or changing the superclass.

- Class»subclasses:
- Behavior»superclass:
- Behavior»basicSuperclass:
- Class»addSubclass: / Metaclass»addSubclass: / UndefinedObject»addSubclass:
- Class»removeSubclass: / Metaclass»removeSubclass: / UndefinedObject»removeSubclass:
- Behavior»removeAllObsoleteSubclasses
- Behavior»addObsoleteSubclass: / Metaclass»addObsoleteSubclass:

Instance variable modification. It allows one to add or remove an instance variable (by name).

- ClassDescription»addInstVarNamed: / Class»addInstVarNamed: / Metaclass»addInstVarNamed:
- ClassDescription»removeInstVarNamed:

Class variable modification. These operations allow one to add or remove a class variable.

- Class»addClassVariable:
- Class»addClassVarNamed:
- Class»removeClassVariable:
- Class»removeClassVarNamed:

Shared pool modification. These operations allow one to add, change, or remove shared pools.

- Class»sharedPools:
- Class»addSharedPool:
- Class»addSharedPoolNamed:
- Class»removeSharedPool:
- Class»classPool:

Slot modification.

- ClassDescription»addSlot: / Class»addSlot: / Metaclass»addSlot:
- ClassDescription»removeSlot: / Class»removeSlot: / Metaclass»removeSlot:
- Class»addClassSlot:
- Class»removeClassSlot:
- Object»writeSlot:value:
- Object»writeSlotNamed:value
- Class»writeClassVariableNamed:value:

Selector/Method modification. These operations allow one to add or remove selectors, or query the class including one or more selectors.

- Behavior»removeSelector: / ClassDescription»removeSelector: / TraitedMetaclass»removeSelector: / TraitedClass»removeSelector:
- Behavior»removeSelectorSilently:
- Behavior»addSelectorSilently:withMethod: / ClassDescription»addSelectorSilently:withMethod:
- Behavior»addSelector:withMethod: / ClassDescription»addSelector:withMethod: / TraitedMetaclass»addSelector:withMethod:

- addSelector:withMethod: / TraitedClass»>addSelector:withMethod: / TraitedClass class»>addSelector:withMethod:
- Behavior»addSelector:withRecompiledMethod: / TraitedMetaclass»addSelector:withRecompiledMethod: / TraitedClass»addSelector:withRecompiledMethod:

Old class creation. The following lists present the partial old API of Class for class creations. It ignores the message supporting optional arguments. It shows clearly why it is about to be deprecated.

- subclass:instanceVariableNames:classVariableNames:category:
- subclass:instanceVariableNames:classVariableNames:package:
- subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- subclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- subclass:layout:slots:classVariables:package:
- subclass:layout:slots:classVariables:poolDictionaries:package:
- subclass:slots:classVariables:package:
- subclass:slots:classVariables:poolDictionaries:package:
- variableByteSubclass:instanceVariableNames:classVariableNames:category:
- variableByteSubclass:instanceVariableNames:classVariableNames:package:
- variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- variableDoubleByteSubclass:instanceVariableNames:classVariableNames:package:
- variableDoubleWordSubclass:instanceVariableNames:classVariableNames:package:
- variableSubclass:instanceVariableNames:classVariableNames:category:
- variableSubclass:instanceVariableNames:classVariableNames:package:
- variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- variableWordSubclass:instanceVariableNames:classVariableNames:category:
- variableWordSubclass:instanceVariableNames:classVariableNames:package:
- variableWordSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableWordSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- weakSubclass:instanceVariableNames:classVariableNames:category:
- weakSubclass:instanceVariableNames:classVariableNames:package:
- weakSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- weakSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- ephemeronSubclass:instanceVariableNames:classVariableNames:package:
- ephemeronSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- ephemeronSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- immediateSubclass:instanceVariableNames:classVariableNames:package:
- immediateSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- immediateSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:

Fluid class creation. The following list presents the equivalent of the previous API in the new fluid class creation API:

- Behavior»« / Metaclass»« / Trait class»«
- FluidClassBuilder»layout:
- FluidBuilder»traits:
- FluidClassBuilder»sharedVariables:
- FluidClassBuilder»superclass:
- FluidClassBuilder»sharedPools:
- FluidBuilder»slots:
- FluidBuilder»package:
- FluidBuilder»tag:

Anonymous class creation. Pharo introduced the notion of anonymous classes that support instance-specific behavior.

- Class»newAnonymousSubclass / Metaclass»newAnonymousSubclass

6. Method creation

The API *Compiled method creation* is a low-level API that supports the definition of compiled methods. Such an API is often ignored but it is central because it is responsible for the creation of new compiled methods that can then be further installed and executed.

Compiled method creation These operations support the creation of compile methods.

- CompiledMethod class»newMethod:header:
- CompileMethod»literalAt:put:
- CompiledMethod»classBinding:
- CompiledMethod»at:put:

7. Structural queries on methods

This category supports the cross-referencing between methods, instance variables, and classes. It is composed of two APIs: *Method slot uses* (which supports the access to a variable and modification of a variable) and *Method element references* (which is a low-level API querying the internal implementation of methods).

Method slot uses. Query on which method is using some specific slot and how.

- Behavior»allMethodsAccessingSlot:
- Behavior»allMethodsReadingSlot:
- Behavior»allMethodsWritingSlot:
- Behavior»methodsAccessingSlot:
- Behavior»methodsReadingSlot:
- Behavior»methodsWritingSlot:
- Behavior»hasMethodAccessingVariable:

Method element references. Query which method/selector is using a given literal.

- Behavior»whichMethodsReferTo:
- Behavior»whichSelectorsReferTo:
- Behavior»thoroughHasSelectorReferringTo:
- Behavior»thoroughWhichMethodsReferTo:
- Behavior»thoroughWhichMethodsReferTo:specialIndex:
- Behavior»thoroughWhichSelectorsReferTo:
- Behavior»hasSelectorReferringTo:

Class references. Behavior»usingMethods returns methods is referencing a given class.

8. Message sending and code execution

This category of operations allows us to explicitly send messages, handle lookup failures or execute compiled methods. It is composed of different APIs: *Reflective message send* (which supports the lookup and execution of a method), *Arbitrary method/primitive execution* (which supports the execution of methods without lookup), *Method lookup* (which simulate the method lookup), *Control message passing* (which supports the possibility to control message sent), and *Message send reification* (which provides ways to access message information).

Reflective message send

- Object»perform:
- Object»perform:orSendTo:
- Object»perform:with:
- Object»perform:with:with:
- Object»perform:with:with:with:
- Object»perform:with:with:with:with:
- Object»perform:withArguments:
- Object»perform:withArguments:inSuperclass:
- Object»perform:withEnoughArguments:

Arbitrary method/primitive execution. While the message perform: supports message sending, the following messages bypass the method lookup and execute a given compiler method.

- ProtoObject»withArgs:executeMethod:
- ProtoObject»tryPrimitive:withArgs:
- CompiledMethod»valueWithReceiver:arguments:
- ProtoObject»executeMethod:
- CompiledMethod»receiver:withArguments:executeMethod:

Method lookup. These operations look up for selectors, and query about this lookup.

- Behavior»lookupSelector:
- Behavior»canPerform:
- Behavior»canUnderstand:

Control message passing.

- ProtoObject»cannotInterpret: Is sent to the receiver if the lookup finds a nil method dictionary. Use a special lookup starting above the class with the nil method dictionary. Has been implemented for proxy implementation.
- ProtoObject»doesNotUnderstand: / Object»doesNotUnderstand: Is sent to the receiver of the message if the lookup reaches the root of the class hierarchy without finding the receiver.
- ReflectiveMethod»run:with:in:

Message send reification. These operations allow one to query a message send to know if it is its arguments, receiver and selectors, and to get a corresponding message.

- MessageSend»isValid
- MessageSend»arguments / Message»arguments
- MessageSend»numArgs / Message»numArgs
- MessageSend»collectArguments:
- MessageSend»receiver
- MessageSend»selector / Message»selector
- MessageSend»message
- Message»lookupClass

These operations allow one to modify a message.

- MessageSend»arguments:
- Message»argument:
- MessageSend»receiver:
- MessageSend»selector:
- Message»setSelector:
- Message»lookupClass:
- Message»setSelector:arguments:

Runtime and Evaluation These operations allow one to evaluate a messageSend or convert it to a messageSend.

- Message»asSendTo:
- Message»sends:
- Message»sentTo:
- MessageSend»value BlockClosure»value
- MessageSend»value: BlockClosure»value:
- MessageSend»value:value: BlockClosure»value:value:
- MessageSend»value:value:value: BlockClosure»value:value:value:
- MessageSend»valueWithArguments: BlockClosure»valueWithArguments:
- MessageSend»valueWithEnoughArguments: BlockClosure»valueWithEnoughArguments:
- MessageSend»cull: BlockClosure»cull:
- MessageSend»cull:cull: BlockClosure»cull:cull:
- MessageSend»cull:cull:cull: BlockClosure»cull:cull:cull:

The following methods perform explicit message sending (do the lookup and then apply a compiled method if any is found):

- Object»perform:(with: with: with: with:)
- Object»perform: orSendTo:
- Object»perform:withArguments:
- Object»perform:withArguments: inSuperclass:
- Object»perform:withEnoughArguments:

9. Chasing and atomic pointer swapping

The APIs in this category are *Find pointers to* and *Bulk pointer swapping* (supports the atomic swapping to references). The first one is rarely mentioned but Pharo supports the possibility to identify pointers to a given object (*e.g.,* ProtoObject»pointersTo and ProtoObject»pointsTo:).

Find pointers to.

- ProtoObject»pointersTo
- ProtoObject»pointersToAmong:
- ProtoObject»pointersToExcept:
- ProtoObject»pointersToExcept:among:
- Object»pointsOnlyWeaklyTo:
- ProtoObject»pointsTo:

Bulk pointer swapping.

- ProtoObject»become: swaps the references in both ways,
- ProtoObject»becomeForward: swaps only towards a given object.
- ProtoObject»becomeForward:copyHash:

10. Memory scanning

This category contains two API *Memory scanning* that support the traversal of the complete heap and *Instances of a class* that gives access to all the instances of a class.

Bulk pointer swapping. The operations allow one to traverse the heap.

- Object»someObject
- ProtoObject»nextObject

Instances of a class. These operations allow one to get or iterate over the instances and subinstances of a class.

- Behavior»someInstance
- ProtoObject»nextInstance
- Behavior»allInstances
- Behavior»allInstancesOrNil
- Behavior»allInstancesDo:
- Behavior»allSubInstancesDo:
- Behavior»allSubInstances

11. Stack Manipulation

This category groups together all APIs that support traversing and modifying the execution stack. These APIs are accessible from two main entry points: the Process class that provides access to the existing processes and their suspended execution stack, and the thisContext pseudo-variable that provides access to the current method execution. Both these entry points provide instances of Context that represent a method execution and that represent the execution stack as a linked list.

Context These operations allow one to get information on the execution context (stack, sender, receiver, ...). The thisContext pseudo-variable supports access to the current execution context. This supports the creation of continuations and co-routines.

- Context»selector
- Context»sender
- Context»activeOuterContext

- Context»arguments
- Context»at:
- Context»at:put:
- Context»method
- Context»methodNode
- Context»outerContext
- Context»outerMostContext
- Context»receiver
- Context»tempAt:
- Context»tempAt:put:

Controlling the stack These operations allow one to control the execution of the current program.

- Context»top
- Context»stepUntilSomethingOnStack
- Context»runUntilErrorOrReturnFrom:
- Context»resume:through:
- Context»activateMethod:withArgs:receiver:class:
- Context»terminate
- Context»send:to:with:lookupIn:
- Context»resumeEvaluating:
- Context»jump
- Context»terminateTo:
- Context»send:to:with:super:
- Context»return
- Context»pop
- Context»shortDebugStack
- Context»return:
- Context»push:
- Context»step
- Context»return:from:
- Context»resume
- Context»stepToCallee
- Context»return:through:
- Context»resume:

References

[TDTP23] Iona Thomas, Stéphane Ducasse, Pablo Tesone, and Guillermo Polito. Pharo: a reflective language - A first systematic analysis of reflective APIs. In *IWST 23 - International Workshop on Smalltalk Technologies*, Lyon, France, August 2023.