Pharo: a reflective language - A first systematic analysis of reflective APIs

Iona Thomas, Stéphane Ducasse, Pablo Tesone, Guillermo Polito

▶ To cite this version:

Iona Thomas, Stéphane Ducasse, Pablo Tesone, Guillermo Polito. Pharo: a reflective language - A first systematic analysis of reflective APIs. IWST 23 - International Workshop on Smalltalk Technologies, Aug 2023, Lyon, France. hal-04217271v2

HAL Id: hal-04217271

https://inria.hal.science/hal-04217271v2

Submitted on 23 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Pharo: a reflective language – A first systematic analysis of reflective APIs

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

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

Abstract

Reflective operations are powerful APIs that let developers build advanced tools or architecture. Reflective operations are used for implementing tools and development environments (*e.g.,* compiler, debugger, inspector) or language features (*e.g.,* distributed systems, exception, proxy, aspect-oriented programming). In addition, languages are evolving, introducing better concepts, and revising practices and APIs. As such, since 2008 Pharo evolved and was built on Squeak which changed from the original Smalltalk reflective APIs. Pharo has one of the largest reflective feature sets ranging from structural reflection to on-demand stack reification. In addition, new metaobjects got integrated such as first-class instance variables. Finally, reflective facilities are mixed with the base-level API of objects and classes and reflective features are heavily used by the system tools.

Getting an understanding of such API is, however, tedious when the API is large and evolved over a decade. There is a need for a deep analysis of current reflective APIs to understand their underlying use, potential dependencies, and whether some reflective features can be scoped and optional.

In this article, we analyze the reflective operations used in Pharo 11. We classify the current reflective operations in different families. Also, we identify a set of issues raised by the use of reflective operations. Such an analysis of reflective operations in Pharo is important to support the revision of the reflective layer and its potential redesign.

Keywords

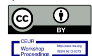
Reflection, Meta-object protocols

1. Introduction

Reflective operations are powerful APIs that let developers build advanced tools or architecture that otherwise would have to be implemented in language implementation engines, require complex infrastructure (such as code representation), or may simply not be possible. These reflective features support the implementation of tools (*e.g.,* compiler, debugger, inspectors), frameworks and libraries (*e.g.,* serialization, persistence, logging), and language infrastructure (*e.g.,* exceptions, distributed systems, continuations, green threads). Such a set of tools and frameworks are both used during the development and deployment of applications (See Section 2).

Giving too much power to developers can, however, also be a burden. Reflective features defeat static analysis [LWL05]. Reflective features may be used to violate encapsulation or execute methods that should not be executed [Duc99, RHBV11, MSL+08, HL07].

International Workshop on Smalltalk Technologies (IWST'23)
iona.thomas@inria.fr (I. Thomas); stephane.ducasse@inria.fr (S. Ducasse); pablo.tesones@inria.fr (P. Tesone); guillermo.polito@inria.fr (G. Polito)


© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings (CEUR-WS.org)

Pharo, for example, as a descendant of Smalltalk is the essence of a reflective language with advanced reflective operations such as bulk pointer swapping [MB15], on-demand stack reification, first-class resumable exceptions. In addition, in Smalltalk-80 and many of its derivative, reflective facilities are mixed with the base-level API of objects and classes [GR89, Riv96, BDN+09]. They are a key part of the kernel of the language and libraries. Finally, since 2008 Pharo evolved as a language: new concepts were added (slots, packages, pragmas...). There is a need for a deep analysis of reflective features.

In this paper, we present an analysis of existing reflective features in Pharo 11. We scope the analysis to runtime reflection to focus on the core reflective features of the language and its associated virtual machine. Pharo inherited Squeak and Smalltalk-80 reflective operations and facilities and extended them over the years. Some reflective methods like Object»instVarAt: are still present and used, some names changed and new reflective facilities appeared.

The contributions of this article are:

- an up-to-date and deep catalog of the reflective features and their home metaobjects in Pharo,
- a classification and an analysis of such operations, and
- a discussion of such reflective APIs and points to be considered to improve or design a new reflective runtime MOP for Pharo.

These contributions are of key importance since they set the foundation for a redesign of the reflective capabilities of Pharo for example to offer optional reflective capabilities and more controlled ones in the context of a more secure and modular version of the language [TDCD15].

The outline of the paper is as follows: first, we explain the need for reflective features in Section 2. In Section 3 we highlight why we need a classification. Section 4 presents an overview of the reflective APIs based on the classes supporting them and their interactions. In Section 5 we present with a high-level perspective the analysis of the runtime reflective APIs in Pharo 11. The Appendix A lists the detailed selectors. For each of the categories, we analyze the capabilities it provides and how they are used in Pharo. Section 6 presents a high-level discussion of considerations to be taken into account to improve such APIs. It also sketches some points for the design of a future MOP for Pharo.

2. Reflective behavior: A need

> *Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession [...] [BGW93]*

Reflective features in object-oriented languages are central to the development of advanced behavior ranging from enhanced development tools to new paradigm implementation such as Aspect-Oriented Programming [KLM+97]. In the middle of the 90s, reflection was heavily explored: structural [BC89, BLR98], computational [Fer89, McA95b], message-based [Fer89, Caz98], compile-time [Chi95] and partial reflection [TNCC03, RDT08].

Reflection is an important tool that enables many important features of modern languages [CAD20]. For example, message-passing control is one of the cornerstones of a broad range of applications and an important feature of reflective systems. Applications that use message-passing control can be roughly sorted into three main categories.

- The first category is *application analysis and introspection* that are based on tools that display interaction diagrams, class affinity graphs, and graphic traces [HDB90, PWG93, BFJR98, CKT+20].
- The second category is *language extension.* In such a case, message passing control allows one to define new features from within the language itself: Garf [GST95], Distributed Smalltalk [Ben87], or [McC87] introduce object distribution in a transparent manner. Language features such as multiple inheritance [BI82], backtracking facilities [LG88], and instance-based programming [Bec93b, Bec93a] have been introduced. Futures [Pas86, LP90] or atomic messages [FJ89, McA95a] are also based on message-passing control capabilities.
- The third category is the *definition of new object models*, introducing concurrent aspects such as active objects (Actalk [BC89]) and synchronization between asynchronous messages (Concurrent Smalltalk [1] [YT87]). Other work proposes new object reflective models such CodA that is a meta-object protocol that controls all the activities of distributed objects [McA95a], meta helix [CKL96] or submethod reflection using AST annotation [DDLM07, CAD20].

More elaborate schemes have been proposed (*e.g., partial behavioral reflection* [TNCC03, RDT08]) that provide a more flexible and fine-grained way to specify both the location being reflected and the metaobject invoked. Context-oriented [CH05] or aspect-oriented programming implementations are often based on reflection [BSL05], AspectBoxes [BHCC06]. Contexts and context history on the level of the pointcuts [HGC06, HGCD07]. Tanter [Tan06] provides a good overview and discussion on the many mechanisms for dynamically scoping crosscutting aspects.

The importance and need for reflective features are also illustrated by the effort to offer them in more static languages such as C++ [Chi95], Ada [RW04], and Java [WS01, RC00, RC02, TNCC03].

Often virtual machine implementations impose restrictions on the changes that are possible [CGMD18]. For example, even if Pharo is one of the most advanced reflective languages due to its large spectrum of capabilities, some changes are not possible due to their inherent runtime cost. Indeed, virtual machines often define the way the objects are represented in memory, and how messages are handled.

3. The need for an up-to-date reflective feature classification

More than 25 years of evolution. Between 1996 and 2008, Squeak evolved from the original Smalltalk reflective API with many contributions. In 2008 Pharo was born from Squeak. Pharo

[1] Concurrent Smalltalk is based on the extension of the virtual machine and new byte-code definition. However, the synchronization of asynchronous messages uses the doesNotUnderstand: technique.

on its own turn saw many different contributions. To give an idea of the activity in Pharo, since 2019 and the versioning of Pharo on Git Hub, Pharo has around 100 contributors (with up to 30 regular ones). As of the writing of this article, its commit history counting only since 2019 is more 20 000 commits.

For an up-to-date analysis. Back in 1996, Rivard [Riv96] proposed the first-classification of Smalltalk reflective features. Such classification is, however, old, and includes aspects such as the compiler which are orthogonal to runtime reflective features. In addition, it is based on VisualWorks a proprietary Smalltalk that is not easily accessible nowadays. Finally, it does not take into account traits [DSW05, DNS+06, TDP+20, TPF+20], first-class instance variables, and the introduction of new tools using reflection such as the new inspector framework [CNSG15], reflectivity [CAD20], object-centric debugging [CKT+20], error handling infrastructure [CDP20], and on the fly deprecated message rewritings [DPZ+22] to name a few. Callau *et al.,* [CRRT11] studied the use of dynamic features of programming language and use Pharo as a case study. Their study is limited and focuses on the use of a limited set of elements. They do not embrace the full reflective APIs. Demers and Malenfant proposed to compare reflective capabilities in logic, functional and object-oriented programming [DM95], but it is not related to a concrete Pharo implementation.

Importance of the analysis. Deploying an application with unneeded reflective facilities produces, however, potential safety issues: as reflective operations might be used to bypass security measures or affect the stability of the executing application [TDCD15].

Removing unneeded reflective operations requires, however, a complete understanding of their usage and analysis to see if they can be separated from the language kernel and core libraries. The challenge is how to improve the modularity and security of the language core, without affecting the features used by tools, frameworks, and libraries. For example, serialization libraries such as STON highly use reflective operations to serialize and deserialize objects; removing those operations to improve the security of the application impedes the use of such a library. This is why in future work, we will analyze more precisely the potential issues that reflective operations may generate in productive applications and their impact on the safety and security of the application. As a first step in that direction, there is a need for a deep and up-to-date analysis that embraces the full spectrum of reflective features. This is what we develop from Section 4 and this is why we list the complete API in Appendix A.

4. Metaobjects, implementing classes and their related APIs

Before giving an overview of the API, we briefly present the structural metamodel of Pharo.

4.1. Pharo structural meta model

A class is a central entity in the structural meta-model [BDN+09]. We briefly describe it, since a large part of the API is currently associated with classes.

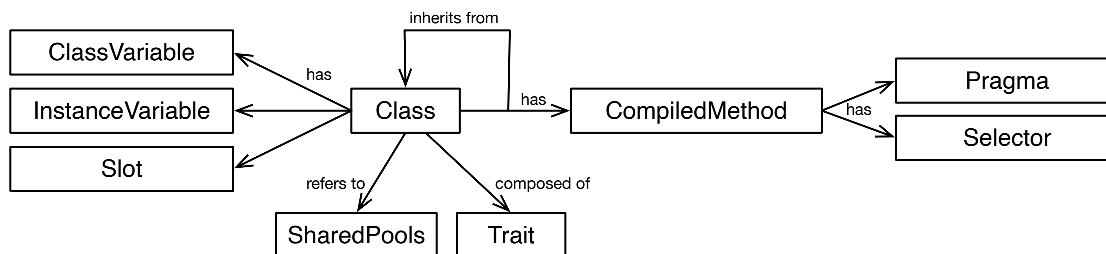


Figure 1: The structural Pharo metamodel: Class aggregates variables, methods, constant management (SharedPools) and method annotation (Pragma) and exposes related APIs.

- A class defines instance variables or slots. Since several versions of Pharo, slots (first-class instance variables) have been introduced and the fusion between instance variables and slots is under development. A class also defines class variables (a.k.a static variables) and it can also use shared pools which are a kind of collection of constants.
- A class inherits from another class and can have multiple subclasses. Since a couple of versions, a class can be composed out of traits (kind of class fragments having their own methods and state).
- A class contains methods. Methods have a selector and can be annotated using Pragma [DMP16].

4.2. Overview of the APIs

Figure 2 shows the classes and the reflective APIs they offer that are available in Pharo 11.

MetaObjects. Grey boxes represent first-class objects. Object, Slot, Class, ClassVariable, and CompiledMethod are structural metaobjects (see Section 6). CompiledMethod and CompiledBlock are potentially the way to access AST nodes and submethod reflective APIs. We decided not to add such a dimension since submethod reflection is optional and can be seen as compiled-time reflection [CDP20].

Implementation objects. We put the MethodDict, CompiledBlock, and BlockClosure in a white box because it is unclear whether we need or not a metaobject for them. Indeed in Pharo, the method dictionary is rather simple and does not offer a reflective entry point per se. It is more of an implementation object. Similarly, which BlockClosure can be introspected as an object it is unclear that it represents a metaobject.

Perspective. The dashed package-like packages represent two aspects of the system: on the one hand Memory which represents the fact that memory can be iterated with methods such as nextInstance and on the other hand Runtime which represents execution aspect of the system with Context (stack reification), Messages, Thread, and Environment (keeping class and variable binding).

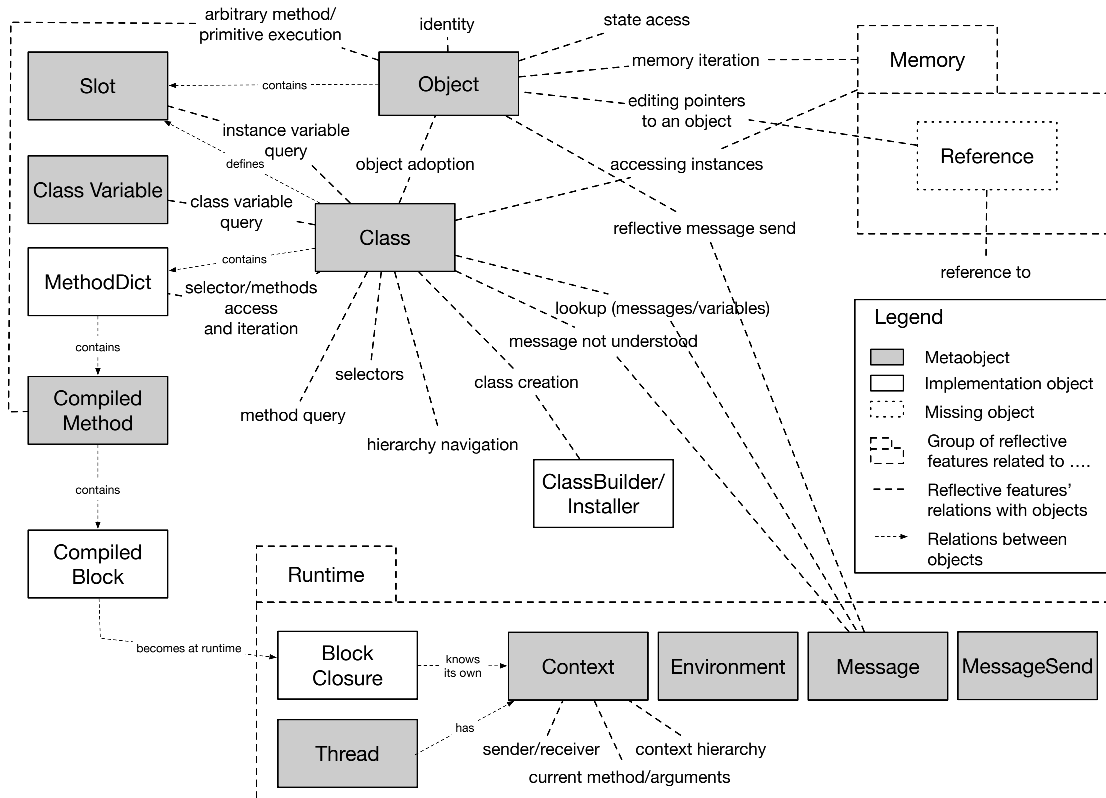


Figure 2: metaobjects controlling the reflective APIs of Pharo.

Note that some APIs are not controlled by metaobjects per se. For example, the Reference API is an API defined on Object as such every object could override it.

5. A classification and analysis of runtime reflective operations

Rivard classified reflective operations in the following categories: *Meta-Operation* (objects), *Structure* (class), *Semantics* (compiler), *Message Sending*, and *Control State* (thread). The *Semantics* part is just a description of the compilation process and involved classes - as such it is not relevant for our analysis since it boils down to the fact that we can add a new compiled method to a method dictionary. We complement and revisit this classification by adding *References*, and *Memory Scanning*.

We propose a detailed and systematic description of the APIs and runtime reflective behavior. Our classification subsumes the one of Rivard. In addition, we distinguish APIs supporting introspection from modification since the modification is more impacting in terms of state encapsulation. We are aware that presenting lists in a systematic manner may be tedious for the reader, this is why Appendix A describes systematically all the APIs. For each of the APIs we briefly describe it, list its key methods (we often group similar ones), and comment on it when appropriate. This is our future work to identify layers within such method lists. Finally,

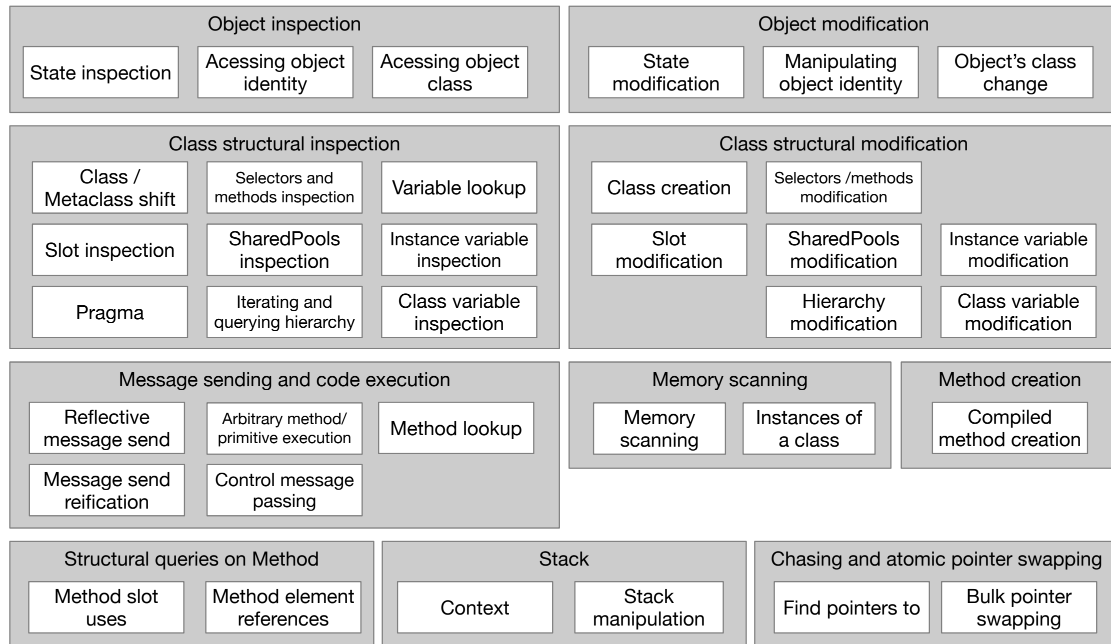


Figure 3: Overview of the reflective APIs.

note that the existence of an API is more important than the fact that we classify it under a given heading. For example, an object can be asked to reflectively execute a method, this API is a list with the other execution-oriented APIs and not directly in the object-centered API.

5.1. Object inspection

The first family of reflective operations is centered around object inspection. Rivard [Riv96] describes these operations in the *Meta-Operations* category, but he groups together inspection and modification. Our category is composed of three subcategories:

- *State inspection* to support inspecting the values of the variables of an object.
- *Accessing object identity* to support identification of an object.
- *Accessing object class* to access the class of an object.

In Pharo, all instance variables are private, meaning there are not readable and writable by any other object. They are only being accessible through getter or setter methods. Developers decide which instance variables are accessible by implementing or not methods to access them. Pharo also includes class instance variables and shared variables, these work in the same fashion as instance variables, and the analysis for instance variables is directly extensible to them. Using the *State inspection* operations is breaking the encapsulation and bypasses the decisions of the developer.

Several methods exist on the class Object allowing access to the state of internal variables. The main examples of this category are Object»instVarAt: and Object»instVarNamed:, which

read an instance variable of an object from respectively its index or name. These operations can be combined with those that can be called on the class of the object (accessible with self class which is part of *Accessing object class* subcategory) to better understand its internal structure like Behavior»allInstVarNames.

Possibilities offered: The *State inspection* operations give a uniform API to inspect all the instance variables of any object, including classes. They are particularly useful for designing tools addressing crosscutting needs, like debugging, inspecting an object, serializing it...

Examples of uses:

- Copying objects.
- Serializing objects.
- Printing objects.
- Inspecting objects.
- Checking references, for example, if an object points to another object.
- Testing, including for some fields without getter methods.

Possibilities offered: The *Accessing object identity* supports checking the identify of an object. basicIdentityHash is used for implementing identityHash variants, scanning for an object in a method dictionary, and testing.

Examples of uses:

- Scanning for objects in a collection
- Implementing hash methods
- Testing
- Printing unique id for some objects (Process, Workers, AdditionalMethodState)

This use raises the question of whether accessing object identity is a reflective operation and not just part of the base-level object API in a language where references are ubiquitous.

Areas of improvements.

- In Pharo base image, there is currently no provided solution for intercession on state read or write on a class or even on a specific object. This requires using additional libraries or implementing ad hoc solutions [CAD22]. Such a tool relies heavily on reflection, and loading several tools at the same time might lead to bugs and instability as each may modify some parts of the Pharo language, its compilation and/or its execution.

5.2. 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* to support modifying the values of the variables of an object.
- *Manipulating object identity* to manipulate the identity of an object.
- *Object's class change* to change the class of an object.

Possibilities offered: The *State modification* operations allows one to bypass the encapsulation for modifying variables, which allows one to set variables that cannot be changed via base-level message passing. This is for example useful for deserialization. They allow one to build tools that will modify objects. The *Object's class change* operations allow one object to transform in an instance of another class, which is particularly important in Pharo's live environment when a class has to be rebuilt following changes.

Examples of uses:

- Copying objects.
- Deserializing objects.
- Modifying object on the fly in the debugger
- Testing, including for some fields without setter.
- To make proxy harder to detect
- Updating instances of classes to migrate them to the new version of a class

The *Manipulating object identity* category is mostly empty: One can only copy the hash of an existing object as a side effect of reference swapping with #becomeForward:copyHash.

Areas of improvements. The API on object class change is weak and limited. The object state can be lost in the process and some constraints (the two classes should have the same format) make it difficult to change the actual class [Riv97]. Access and modification methods can potentially be overridden. Overriding could provide a way for a class to limit reflection on its instances but at the cost of limiting the API.

5.3. 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...).

Possibilities offered: The structural class introspection is large. It is mainly used by tools. It supports the interpretation of object inspection. Methods related to the class hierarchy support navigation of the graph with messages such as superclass and allSubclasses. Methods related to the method dictionary allow one to check all existing selectors and methods, as well as adding and removing them. Methods related to instance variables allow one to list their names, remove some of them and add new ones.

Some methods such as isKindOf: or respondsTo: tend to be used at the domain level and produce a suboptimal design.

Examples of uses:

- Looking up selectors.
- Accessing environment, class side information, check for slot/instance variable.
- Querying about memory format (isImmediate object / isVariable / isFixed) or how much is used.
- Querying if an object is an instance of an anonymous class, a specific class, or part of a specific class hierarchy.
- Copying/Converting to create new instances.
- For the Code Browser, the change recorder, and the code coverage calculation.
- Printing objects and error messages.
- Asserting that an object is of a given class in testing.
- Checking for types in arithmetic operations/comparisons for collections.
- Type checking to determine the visibility of graphical elements.
- Safeguards raising errors if the type does not match what is expected.
- Equality implementations.

An example of the use of environment information is for the doOnlyOnce checking for a value stored in the environment. We also would like to point out the existence of species and respondsTo:. The first is meant to be overridden if necessary and allow classes to specify as instances of which class their instances should be perceived. respondsTo: allows one to query if an object understands a given selector. These two messages can be used to not put as much of a burden on polymorphism.

Areas of improvements. The following improvements would benefit the system.

- There is spurious redundancy between isClassSide and isMeta. Such double methods should be corrected.
- As a general remark, the question of the systematic application of the Law of Demeter should be discussed because it bloats the API. For example, messages such as selectSuperclasses: / selectSubclasses: do not seem to be necessary. In addition, withAllSuperAndSubclasses and includesBehavior: look superfluous.
- We see the old protocol with cryptic names such as instSize to mean instanceVariableSize.
- The duality of instance variables and slots is an artifact of the current evolution of Pharo. Nevertheless, this is important that in the future, instance variables get fully replaced by slots and that the corresponding reflective APIs get merged.
- The duality of selectors versus methods should be evaluated. Since a method dictionary always has the selector of the method as a key, the API could favor selectors for most of the queries and only favor one access to compiled methods (via a method such as methodNamed: and one iterator).

5.4. 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.

Possibilities offered: Structural modification operations allow the user to modify the current structure/shape of classes. They include operations to add/remove subclasses, instance variables, and class variables. We distinguish introspection and modification APIs because we want to stress that modification is destructively modifying the executed system and that as such they represent more powerful operations.

In addition to the traditional class creation API (kept for backward compatibility) and the fluid API, Pharo introduced the notion of anonymous classes (message newAnonymousSubclass) [Duc99]. It helps to define instance-specific methods.

We do not want to give too much importance to such classification. Appendixes present the details.

One use case: Fluid class creation. The presented APIs are the basis for the dynamic recompilation and runtime mutation of the system. There is one point that we want to stress: the **class creation**.

For a couple of years, Pharo is in the process of revisiting fundamentally the way classes are created.

- First, a new implementation of a class builder supports better instance migration logic.

- Second, the creation of a class is separated from its effective installation in the system. This supports the creation of class definitions without side effects on the system. The messages classBuilder and classInstaller correspond to these two different actions.
- Third, since Pharo 10 a fluid API has been introduced to control and limit the combinatorial an explosion of arguments in the traditional class creation API (subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:) as shown in the Appendix. Pharo limited the exposure to the users of infrequent parameters such as pool dictionaries. Still, the API was extremely large with the introduction of new kinds of subclasses such as ephemerons. *Fluid class creation* API has been designed to reduce drastically such large API and to not expose directly the API via a class object: indeed the definition of the class is done via messages configuring a class builder. The message « is creating a class builder object as shown in the following example.

```
Object << #Point
  slots: { #x . #y };
  tag: 'BasicObjects';
  package: 'Kernel'
```

Finally, in addition to the traditional class creation API (kept for backward compatibility) and the fluid API, the API *Anonymous class creation* supports the creation of anonymous classes (message newAnonymousSubclass) [Duc99]. It helps to define instance-specific methods.

Areas of improvements.

- The unification of Slots and variables should be continued to avoid duplication at the reflective APIs level.
- About *Selector/Method modification*. The 'silently' prefix raises the question of the management of the notification of modification. Indeed, some tools need to get notified to react to new elements. Nevertheless, this duality suggests a layered API where low-level API elements are clearly identified.
- The API *Anonymous class creation* can be packaged separately from the *Fluid class creation*.

5.5. 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.

Possibilities offered: *Compiled method creation* offers the possibility to create a compiled method and this even without the need for the compiler.

Examples of uses: This API supports the modularization of the system core such as making the compiler optional in the system bootstrap. It is used by Hermes a binary binary code loader. It is an important asset that supports the bootstrap of Pharo [Pol15] and ensures that the compiler can be loaded in a system not having a compiler to compile and install code.

Areas of improvements. Since the code for creating compiled methods is just the code of the CompiledMethod class. It has not been explicitly designed to be a MOP API. Revisiting this central API in the context of the *Selector/Method modification* and the interplay between the two could lead to a stronger MOP.

5.6. 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).

Possibilities offered: These two APIs are central for all the cross-referencing and navigation such as senders and which methods use a given variable. As such these APIs are really important for the IDE and the tools.

Areas of improvements. The duality of selectors and methods (exemplified by Behavior»whichMethodsReferTo: and Behavior»whichSelectorsReferTo:) could be again better handled. We suggest not exposing the compiled method, since it is always possible to get the selector of a method. It would lead to a more compact API.

In addition *Method slot uses* looks like an optional API that can be packaged with the tools. A unification between the two APIs would produce a more coherent API.

5.7. 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).

In Pharo, when sending a message to an object, the first step is to search the message selector in the class hierarchy of the message's receiver. This is the lookup. Once a compiled method is found in the receiver's class or one of its superclasses, the method is applied to the receiver. When the lookup does not find any corresponding method, doesNotUnderstand: is sent to the receiver with the message reified as an instance of MessageSend. This gives the opportunity to the receiver to specialize message error. The APIs are central to bringing flexibility to applications. In particular *Reflective message send* with its perform: methods is important for pluggable UI logic.

While the methods of the APIs *Reflective message send* do a method lookup, methods of the *Arbitrary method/primitive execution* API allow us to execute directly a compiled method or a primitive. The main methods in this category are ProtoObject»(withArgs:) executeMethod:, CompiledMethod»valueWithReceiver:arguments: and ProtoObject»tryPrimitive:withArgs:

Possibilities offered: These operations allow us to explicitly send a message and handle failure cases. The selector sent can be determined dynamically from an input, a string, or a symbol. Run a specific primitive or version of a compiled method without needing to install a method in the class hierarchy.

Examples of uses for perform: message.

- Iterating over a list of messages and sending them. Sometimes coming from a list of pragmas, using methodSelector.
- Sending a message received as a String or Symbol parameter.
- In testing, including when compiling the method corresponding to the selector in the test.
- Sometimes used when a non-reflective message-send could be used

Many uses are related to user input and are found especially in user interfaces (*e.g.,* executing the result of a selection). They are also used for methods on reified messages, system settings, class parser, and the Finder tool. One more example, not included in the Pharo 11 base image, is the TeaPot library and how it handles HTTP requests.

Examples of uses for 'message not understood'. The method doesNotUnderstand: has 23 senders and 27 implementors in the base Pharo image. Actually, from these senders:

- 2 come from the lookup implementation,
- 13 from doesNotUnderstand: reimplementations,
- 1 from the implementation of debugging to test if the current context is a DNU,
- 1 corresponds to the inclusion of the selector in the specialObjectsArray,
- and the 4 remaining from tests and mocks.

Among doesNotUnderstand: implementors, many try to delegate using the message sentTo: answering the message to another object before raising the exception. This has been used for example to implement proxies [Pas86].

Examples of uses for *Arbitrary method/primitive execution.* Many uses are key to supporting debugging and advanced tool facilities.

- In Context class for Debugging and primitive simulation
- UnifiedFFI/Fuel/OpalCompiler/Decompiler Tests and test coverage
- Debugging support.
- Evaluating (*e.g.,* for RBNode, OpalCompiler)
- Profiling
- FFI methods

Areas of improvements.

- Pharo offers two ways of representing a message via the class MessageSend and Message. This situation shows that the addition of concepts was not done carefully to avoid duplication. Pharo defines also a class Message (selector and arguments). This class is used to represent a message when an error occurs (doesNotUnderstand:). It supports the possibility to perform a lookup via the message sendTo: which is the counterpart of doesNotUnderstand:. The class MessageSend represents the concept of sending a message. It holds a receiver, a selector, and arguments. As such the class MessageSend offers corresponding getters and setters. Such a class is not used by the runtime and it is used only to represent and execute a message. For its execution, the class offers an API compatible with block closures: The messages value: and its variants allow one to execute a message. In addition, cull: and variants provide more flexibility in the number of arguments. We suggest merging MessageSend into Message since this last one is used to reify messages on error.
- Having several ways to express the same behavior can be improved. CompiledMethod»valueWithReceiver:arguments:, ProtoObject»executeMethod:, and ProtoObject»withArgs:executeMethod: are similar. We suggest that only methods on CompiledMethod s should be kept. The definition on ProtoObject would have the pernicious side effect to make domain developers that it is safe to use such messages.
- The direct execution of a compiled method as offered by the *Arbitrary method/primitive execution* API is really dangerous because the system does not check that the executed method can be executed on the receiver. This is usually ensured by the lookup. Therefore while it makes sense to use methods of the *Reflective message send* API, we believe domain developers should not be exposed to the *Arbitrary method/primitive execution* API. In addition, this API should be packaged separately to clearly expose its nature.

While the previous category allowed one to query the identity and type of an object, this one is related to modifying those.

5.8. 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:).

Possibilities offered: *Find pointers to* is really useful to build tools to identify a memory leak; it is optional and its use is well-scoped.

The second one *Bulk pointer swapping* is one of the hallmarks of Smalltalk reflective APIs. Indeed in the original Smalltalk implementation, the use of an object table made the implementation of swapping references very cheap and used for example for growing collections. However, such an old implementation did not scale for the modern amount of objects a runtime contains. For modern implementations, become: conceptually requires scanning the complete memory and is a costly operation. Modern implementations lowered the cost of this operation by using forwarders at the VM level [MB15]. It should be noted that Pharo offers two semantics

for swapping: become: which symmetrically swaps the pointers and becomeForward: which is one way. In addition, one cannot be used to express the other at the language level.

Examples of uses:

- Memory leak analyzers.
- Dynamically updating existing instances to new class shapes.

Areas of improvements. Such API while useful during development sessions should be limited during deployment. A precise analysis of the use of *Bulk pointer swapping* should be done to differentiate the places where it is mandatory from the places where it is a convenient optimized solution. It should be noted that *Find pointers to* could be implemented on top of a full memory scan APIs such as the ones presented in the next section. Similarly, a slower version of *Bulk pointer swapping* could be possible.

5.9. 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.

Possibilities offered: This API is usually not mentioned in the literature but it is at the core of live programming [Tes18]. The method nextObject and nextInstance are key to building other functionalities such as allInstances.

Examples of uses: The main use is the class build that needs to access all the instances of a given class to reflect potential class shape changes. Other uses such as collecting all instances of a class are more anecdotal and reflect the lack of an explicit registration mechanism in the domain.

Areas of improvements. Understanding whether such a reflective API can be optional and only be loaded on demand would be a step to build a more compact, tidier, and secure reflective MetaObject protocol.

5.10. 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.

Possibilities offered: The Stack Manipulation APIs provide on the one hand low-level access to the call stack (*e.g.,* sender, programCounter), context meta-data (*e.g.,* method), and context operand stack (*e.g.,* push:, pop and at:), and on the other hand support for continuations built on top of the previous APIs (*e.g.,* return:, resume:).

Examples of uses: These APIs make possible essential features in the programming language and environment such as *exceptions*, *bytecode simulation*, and *debugging*.

Areas of improvements. Stack manipulation support for stack modification and intercession is, at the moment of this writing, limited. A single class Context is allowed, and its instances are reified on-demand by the execution engine by the Virtual Machine implementation [Mir99]. This means that the APIs described above cannot be refined in subclasses to modify the behavior of method execution. Currently, such fine-grained intercession is achieved by bytecode rewriting using frameworks such as reflectivity [CAD20].

Additionally, the fact that essential language features such as exceptions are built on top of stack manipulation support turns this support mandatory. Optional stack manipulation requires a major redesign such as a re-implementation of such essential features on the Virtual Machine, or at the extreme considering exceptions as optional reflective features too.

6. Discussions

As we have seen in the previous sections, if reflective operations are extremely powerful and useful to implement tools to navigate Pharo's live environment [KRB18], they are also impeding application safety in multiple ways. Analyzing and designing a new modular MOP is a clear challenge. In this section we discuss various aspects ranging from the analysis we presented to the consideration to be taken into account.

6.1. General concerns

About dual entities. On several occasions, the MOP proposes a kind of duplicated APIs: one for selectors and the other for compiled methods, or one for a variable and its name. Having only one API based on the object is not good because it would force the developer to have an object while it may not be possible. It means that using a name is a good approach. In particular, since when one has an object (compiled method, slot) we can get its name. We suggest reducing the API spectrum by not proposing two APIs but favoring one based on the name for query or access, and for the modification, the developer can query first based on name to access an object and then perform the corresponding operation. In that regards the question of the application of the Law of Demeter should be evaluated since it tends to produce larger APIs.

Absence of layers. On several occasions we see the need to clearly identify the level of API. Indeed some methods require mere index (instVarAt:) while some others require names (instVarName:). While the first one is needed, we suggest (1) a clear naming convention that helps understand the level of the functionality, (2) a better naming (methods named instSize

that returns the number of instance variable feels outdated in a modern language. Finally, some APIs are large due to the fact that basic functionality is augmented with helper behavior built on top for such basic functionality. While this is a good practice to promote code reuse and offer developers stronger APIs, we suggest to layer such APIs and making sure that high-level APIs are optional with clearly identified users.

About metaobject Protocol and piecemeal growth. In Smalltalk, reflection is exposed as methods of objects that can modify the internals of the system and the causal connection makes sure that the modifications get in effect. We can see this approach as an organic one and from this perspective we can say that the metaobject Protocol of Smalltalk has been less designed than the one of CLOS [KR90].

We suggest that the design of a new MOP should consider how certain objects represent customization points and avoid piecemeal and accidental MOP growth. For example, in CLOS it is possible to specify at the metaclass level, the class of the executed method. The Method class is then a natural metaobject exposing a method that can be specialized to for example count the number of execution of the methods. In Pharo, the hook to specify the class of a method is not clear. More important, when a method is executed there is no identified method that is called prior to the method execution. Frameworks such as Method Proxy, MethodWrappers [BFJR98] build such functionality using VM hooks such the possibility to place any object in a system dictionary and that such an object receives the message run:with:in:.

A MOP may decide to expose customization points as dedicated objects and not necessarily objects that are currently been executed [McA95a]. For example in CodA, different lifetime aspects of objects (message send, message received, state access, execution,...) are reified via specific metaobjects.

Execution-Time Reflection. In our analysis, we have centered on the reflective API during the execution time. We analyzed the operations that are able to be done during the execution of our code. In this sense, we have left outside operations performed outside the execution of the code. Operations such as static code analysis and rewriting, memory dump inspection and modification, refactoring, and on-load code rewriting or instrumentation are not performed during execution time. Those operations are outside our definition of reflective operations.

Compiler. In contrast to Rivard [Riv96] who considers the compiler as part of the reflective API of the system, in our analysis we keep it out. We have taken this decision as the reflective API provides ways of creating and installing methods. In Pharo reflective API a method is created from its bytecode, literals, and header. The complexity of generating such a set of bytecode, literals, and header for the method is outside the reflective API. The compiler has as input the source code of the method. Through a series of complex transformations (such as parsing, AST building, AST rewriting, AST optimizations, Intermediate Representation Building, and bytecode generation) the compiler generates the bytecode, literals, and headers. A compiler is just one possible source of these elements. For example, in Pharo, we have a binary code loader that generates and install methods. This binary code loader is used without the compiler to load code during the bootstrap process of the image. The compiler and the binary code

loader both use the same reflective API, that allows them to create and install new methods in the running environment. Moreover, it is possible to have more alternative tools to generate methods profiting Pharo reflective API.

Package Loading / Unloading Missing. The existing reflective API does not present a clear metamodel to handle the concept of Packages. Even though this concept is used outside the execution of code. It is a key element in the metamodel of Pharo. It is used to load, unload and version classes, methods, and extensions existing in Pharo. Moreover, it is the key element to support method extensions.

A clear reflective API is required to handle the loading, unloading, versioning, and modification of packages in Pharo. Also, clear modeling of the package allows for additional points of extension to the metamodel and the ability to improve existing tools (*e.g.,* scoping extensions, dependencies).

We have left outside of this paper the analysis of the features and a possible design of such Package API, but we recognize the importance of such reflective API.

Architecture for notification. In our analysis, we have found that there is no clear API for handling the notifications of changes. Tools working on the metamodel of Pharo require a good integration to be notified of changes. For example, a Code Browser requires a clear way of getting notifications when a new method or class is added to the system. Also, there are scenarios where the tools modify the system but this modification should not be notified. For example, when instrumenting a method, if the original method is replaced there is no need that the Code Browser to be notified.

A clear notification API should guarantee that the tools and libraries are able to scope the notifications they want to produce and consume.

An extensive analysis of this notification architecture is outside the scope of this paper, however, we realize that such a notification architecture is required.

About definition and method reification. In early versions of Squeak, a compiled method did not know its class or its selector. It was then expensive to ask a compiled for its selector since it required scanning all the methods installed in a class. Over the years compiled methods saw their API and representation improved. At the same time, there is a need to be able to represent methods that are not installed in a class, for example, to browse multiple versions of a method or perform branch analysis [UGDD10]. In this scenario, there is a need to represent a method with a source code that is not the one of the currently installed compiled method. Similarly, several meta-models such as Ring and Ring2 have been designed to support the analysis of code not loaded in an image (browsing, crossreferencing, remote browser...). There is a need to have method definitions as well as compiled methods. This raises the question of whether the tools should not manipulate method definitions and not compiled methods. Such method definitions could be connected to a compiled method when the compiled method is actually installed in the system. From a reflective API, compiled methods could be more executable objects and expose only information related to their execution and for all the other needs the tools could request the associated method definition. By making sure that the tools and reflective API always go

from a method definition to the compiled method, we could restrain the compiled method API. Such architecture, however, should be built and validated because in an image for development, it would double the number of objects representing methods or special caches should be done to support method cross-referencing.

6.2. Mirror architecture

At the language level, Mirrors [BU04, US87] aims for stratification of meta-level facilities and gives access to reflective capability based on a reference to a mirror factory. Mirrors were implemented in several languages, for example, Self [US87], StrongTalk [BBG+02] and Newspeak [TPF+16]. We have seen in Section **??** that MirrorPrimitives offer a mirror implementation. However, given that MirrorPrimitives is a class, it is registered in the global environment, making access to this facility possible from anywhere. This defeats the idea of restricting access to reflection through the use of references to mirror factories. In [PBD+11] the authors advocate that mirrors should also address structural decomposition. Mirrors should not only be the entry points of reflective behavior but also be the storage entities of meta-information. Pharo offers a first implementation of Mirrors (with APIs to read/write fields, check the class and the identity of an object, and execution of a method with another receiver) but it is not systematically used and adds another mechanism.

6.3. Controlling reflection

Optional reflective features. Our analysis identified some APIs of reflective behavior that are optional. This is important and we suggest continuing this effort with the objective to obtain a minimal core with modular and optional extensions. This is particularly interesting because the expectations of a development session should be automatically the same as the one of a deployed application. Of course, this is handy to be able to do fancy reflective actions on deployed applications to debug them, but this is important that MOP designers consider other scenarios such as more constrained application deployment setup.

Controlling reflection. N. Papoulias *et al.,* [PDDF15] proposes a model for the reification of the control of reflection. Indeed controlling the run-time behavior of reflective facilities introduces several challenges, such as computational overhead, the possibility of meta-recursion, and an unclean separation of concerns between the base and the meta-levels. They present five dimensions of meta-level control from related literature that try to remedy these problems. These dimensions are namely: temporal and spatial control, placement control, level control, and identity control. Making a reflective feature optional and identifying its dependencies is one way to control it.

External reflection. Another approach is to separate completely the implementation of the language from the reflective API. Lorenz proposes a pluggable reflection [LV03], in which the reflective API should be external to the language. This solution allows tools using reflectivity to process information coming from different sources (source code, live environment) as long as the same external API is available. This solution aims at flexibility and interoperability.

While this might seem to not fit the model of the Smalltalk/Pharo live environment with all its embedded tools, having such an external reflective API could remote debugging while removing the reflectivity inside the image.

7. Conclusion

This article acknowledged that the runtime reflection offered by Pharo was in need of a deep and systematic analysis after the evolution of Squeak and the subsequent evolution of Pharo since 2008. The analysis of Rivard [Riv96] while interesting is simply dated. Indeed Pharo metaobjects evolved and got [DSW05, DNS+06, TDP+20, TPF+20], first-class instance variables and the introduction of new tools using reflection such as the new inspector framework [CNSG15], reflectivity [CAD20], object-centric debugging [CKT+20], error handling infrastructure [CDP20] and on the fly deprecated message rewritings [DPZ+22]. This is not counting the full rewrite of the compiler.

In this article, we present a new systematic analysis of the reflective APIs unrevealing some often undocumented aspects such as memory scanning or method installation. In addition, the analysis proposes some potential improvements to the existing MOP. The discussion raises the bar of the analysis because MOP designers should challenge the monolithic perception that a MOP should be omnipotent and cover all the aspects all the time. We believe that a faceted MOP where on-demand reflective operations can be made available is the way to create a more versatile system that has different varieties of flavors depending on the kind of applications that one wants to deploy and their companion security properties.

Acknowledgments. We gratefully acknowledge the support of the "Région Hauts-de-France" for the financial support of the PhD of I. Thomas.

References

[BBG+02] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, June 2002.

[BC89] Jean-Pierre Briot and Pierre Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 419–432, October 1989.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Bec93a] Kent Beck. Instance specific behavior: Digitalk implementation and the deep meaning of it all. *Smalltalk Report*, 2(7), May 1993.

[Bec93b] Kent Beck. Instance specific behavior: How and Why. *Smalltalk Report*, 2(7), May 1993.

[Ben87] John K. Bennett. The design and implementation of distributed Smalltalk. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 318–330, New York, NY, USA, 1987. ACM.

[BFJR98] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and J.L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[BHCC06] Alexandre Bergel, Robert Hirschfeld, Siobhàn Clarke, and Pascal Costanza. Aspectboxes — controlling the visibility of aspects. In Markus Helfert Joaquim Filipe, Boris Shiskov, editor, *In Proceedings of the International Conference on Software and Data Technologies (ICSOFT 2006)*, pages 29–38, September 2006.

[BI82] Alan H. Borning and Daniel H.H. Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, pages 234–237, Pittsburgh, PA, 1982.

[BLR98] Noury Bouraqadi, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings OOPSLA '98*, pages 84–96, 1998.

[BSL05] Noury Bouraqadi, Abdelhak Seriai, and Gabriel Leblanc. Towards unified aspect-oriented programming. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, 2005.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CAD20] Steven Costiou, Vincent Aranega, and Marcus Denker. Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use. *The Art, Science, and Engineering of Programming*, 4(3), February 2020.

[CAD22] Steven Costiou, Vincent Aranega, and Marcus Denker. Reflection as a tool to debug objects. In *International Conference on Software Language Engineering (SLE)*, pages 55–60, 2022.

[Caz98] Walter Cazzola. Evaluation of object-oriented reflective models. In *In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS 98), in 12th European Conference on Object-Oriented Programming (ECOOP 98), Brussels, Belgium, on 20th-24th*, pages 3–540, 1998.

[CDP20] Steven Costiou, Thomas Dupriez, and Damien Pollet. Handling Error-Handling Errors: dealing with debugger bugs in Pharo. In *International Workshop on Smalltalk Technologies - IWST 2020*, November 2020.

[CGMD18] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. Fully reflective execution environments: Virtual machines for more flexible software. *Transaction on Software Engineering*, 45:858–876, September 2018.

[CH05] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS'05)*, pages 1–10, New York, NY, USA, October 2005. ACM.

[Chi95] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, October 1995.

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacir-

cularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049, pages 157–172. Springer, 1996.

[CKT+20] Steven Costiou, Mickaël Kerboeuf, Clotilde Toullec, Alain Plantec, and Stéphane Ducasse. Object miners: Acquire, capture and replay objects to track elusive bugs. *Journal of Object Technology*, 19(1):1:1–32, July 2020.

[CNSG15] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The moldable inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 44–60, New York, NY, USA, 2015. ACM.

[CRRT11] Oscar Callau, Romain Robbes, David Rothlisberger, and Eric Tanter. How developers use the dynamic features of programming languages: the case of smalltalk. In *Mining Software Repositories International Conference (MSR'11)*, 2011.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.

[DM95] Francois-Nicola Demers and Jacques Malenfant. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*, 1995.

[DMP16] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. Pragmas: Literal messages as powerful method annotations. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.

[DPZ+22] Stéphane Ducasse, Guillermo Polito, Oleksandr Zaitsev, Marcus Denker, and Pablo Tesone. Deprewriter: On the fly rewriting method deprecations. *Journal of Object Technologies (JOT)*, 21(1), 2022.

[DSW05] Stéphane Ducasse, Nathanael Schärli, and Roel Wuyts. Uniform and safe metaclass composition. *Journal of Computer Languages, Systems and Structures*, 31(3-4):143–164, December 2005.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[Fer89] Jacques Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[GST95] Jeff Garbus, David Salomon, and Brian Tretter. *SYBASE DBA: Survival Guide*. Sams Publishing, 1995.

[HDB90] Jurgen Herczeg Heinz-Dieter Bocker. What tracers are made of. In *Proceedings of OOPSLA/ECOOP '90*, pages 89–99, October 1990.

[HGC06] Charlotte Herzeel, Kris Gybels, and Pascal Costanza. A temporal logic language

for context awareness in pointcuts. In *Proceeding of the Workshop on Revival of Dynamic Languages*, 2006.

[HGCD07] Charlotte Herzeel, Kris Gybels, Pascal Costanza, and Theo D'Hondt. Modularizing crosscuts in an e-commerce application in lisp using halo. In *Proceeding of the International Lisp Conference (ILC) 2007*, 2007.

[HL07] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242. Springer-Verlag, June 1997.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. In *Proceedings of ACM conference on Lisp and Functional Programming*, pages 99–105, Nice, 1990.

[KRB18] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: Insights from the practice. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1090–1101, New York, NY, USA, 2018. ACM.

[LG88] Wilf R. LaLonde and Mark Van Gulik. Building a backtracking facility in Smalltalk without kernel support. In *Proceedings OOPSLA '88, ACM SIGPLAN Notices*, volume 23, pages 105–122, November 1988.

[LP90] Wilf LaLonde and John Pugh. *Inside Smalltalk: Volume 1.* Prentice Hall, 1990.

[LV03] David H. Lorenz and John Vlissides. Pluggable reflection: decoupling meta-interface and implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 3–13, Washington, DC, USA, 2003. IEEE Computer Society.

[LWL05] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of Asian Symposium on Programming Languages and Systems*, 2005.

[MB15] Eliot Miranda and Clément Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management (ISMM '15)*, pages 93–104, Portland, United States, June 2015.

[McA95a] Jeff McAffer. *A Meta-level Architecture for Prototyping Object Systems.* Ph.D. thesis, University of Tokyo, September 1995.

[McA95b] Jeff McAffer. Meta-level programming with coda. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[McC87] Paul L. McCullough. Transparent forwarding: First steps. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 331–341, December 1987.

[Mir99] Eliot Miranda. Context management in visualworks 5i, 1999.

[MSL+08] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja safe active content in sanitized javascript. Technical report, Google Inc., 2008.

[Pas86] Geoffrey A. Pascoe. Encapsulators: A new software paradigm in Smalltalk-80. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 341–346, November 1986.

[PBD+11] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Towards structural decomposition of reflection with mirrors. In *Proceed-*

ings of International Workshop on Smalltalk Technologies (IWST'11), Edingburgh, United Kingdom, 2011.

[PDDF15] Nick Papoulias, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Reifying the reflectogram. In *30th ACM/SIGAPP Symposium On Applied Computing*, Salamanca, Spain, apr 2015.

[Pol15] Guillermo Polito. *Virtualization Support for Application Runtime Specialization and Extension*. PhD thesis, University Lille 1 - Sciences et Technologies - France, April 2015.

[PWG93] Francois Pachet, Francis Wolinski, and Sylvain Giroux. Spying as an Object-Oriented Programming Paradigm. In *Proceedings of TOOLS EUROPE '93*, pages 109–118, 1993.

[RC00] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in javascript applications. In *Proceedings of Ecoop 2011*, 2011.

[Riv96] Fred Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION'96*, pages 21–38, April 1996.

[Riv97] Fred Rivard. *Évolution du comportement des objets dans les langages à classes réflexifs*. PhD thesis, Ecole des Mines de Nantes, Université de Nantes, France, 1997.

[RW04] P. Rogers and A.J. Wellings. Openada: Compile-time reflection for ada 95. In *Reliable Software Technologies - Ada Europe*, volume 3063 of *LNCS*. Springer, 2004.

[Tan06] Éric Tanter. On dynamically-scoped crosscutting mechanisms. In Günter Kniesel, editor, *Proceedings of the European Workshop on Aspects in Software (EWAS 2006)*, pages 18–22, Twente, The Netherlands, September 2006. Technical Report IAI-TR-2006-6, University of Bonn, Germany.

[TDCD15] Camille Teruel, Stéphane Ducasse, Damien Cassou, and Marcus Denker. Access control to reflection with object ownership. In *Dynamic Languages Symposium (DLS'2015)*, 2015.

[TDP+20] Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, and Noury Bouraqadi. A new modular implementation for stateful traits. *Science of Computer Programming*, 195:1–37, 2020.

[Tes18] Pablo Tesone. *Dynamic Software Update for Production and Live Programming Environments*. PhD thesis, Université de Lille - IMT Lille Douai, 2018.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[TPF+16] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane

Ducasse. Instance migration in dynamic software update. In *Meta'16*, Amsterdam, Netherlands, October 2016.

[TPF+20] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Preserving instance state during refactorings in live environments. *Future Generation Computer Systems*, 110:1–17, 2020.

[UGDD10] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Meta-models and infrastructure for smalltalk omnipresent history. In *Smalltalks'2010*, 2010.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

[WS01] Ian Welch and Robert J. Stroud. Kava — using bytecode rewriting to add behavioural reflection to Java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology (COOTS'2001)*, pages 119–130, San Antonio, Texas, USA, February 2001.

[YT87] Yasuhiko Yokote and Mario Tokoro. Experience and evolution of ConcurrentSmalltalk. In *Proceedings OOPSLA '87*, volume 22, pages 406–415, December 1987.

A. Details of the systematic classification of reflective operations

We propose a detailed and systematic description of the APIs and reflective behavior. Our classification subsumes the one of Rivard.

We are aware that presenting lists in a systematic manner may be tedious for the reader. We consciously decided to do so to give a full view of the current situation in this appendix. For each of the APIs, we briefly describe it, list its methods (we often group similar ones together), and provide when adequate a little comment. This is our future work to identify layers within such method lists.

B. Object Inspection

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

- ProtoObject»class
- Context»objectClass:
- Context»objectSize:
- 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

C. Object Modification

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:

D. Class structural inspection

Structural inspection operations allow the user to get information about the structure of the current Pharo world. This includes information about the classes, their variables names and hierarchy, as well as object, their variables names, and if they are capable of answering some messages.

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

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

- 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 [DMP16]. 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

E. Class structural modification

Structural modification operations allow the user to modify the current structure/shape of classes. They include operations to add/remove subclasses, instance variables, class variables, ...

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:

- Beharior»« / 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

F. Compiled method creation

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

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

G. Structural queries on 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.

H. Message Sending and code execution

H.1. 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:

I. Chasing and swapping pointers

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:

J. Memory Scanning

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

K. Stack manipulation

K.1. 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:

K.2. 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: