Journal of Computer Languages 80 (2024) 101274

Contents lists available at ScienceDirect

Journal of Computer Languages

journal homepage: www.elsevier.com/locate/cola

Pharo: A reflective language—Analyzing the reflective API and its internal dependencies

Iona Thomas, Stéphane Ducasse, Pablo Tesone *, Guillermo Polito

Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille, 59650, France

ARTICLE INFO

Keywords:
Reflection
Meta-object protocols

ABSTRACT

Reflective operations are powerful APIs (Application Programming Interface) that let developers build advanced tools and architectures. Reflective operations are used for implementing tools and development environments (*e.g.,* compilers, debuggers, inspectors) or language features (*e.g.,* distributed systems, exceptions, proxies, aspect-oriented programming). In addition, languages are evolving, introducing better concepts, and revising practices and APIs. Since 2008 Pharo has evolved from Squeak and its reflective API has evolved accordingly, diverging consequently from the original Smalltalk reflective API. With more than 500 reflective methods identified, Pharo has one of the largest reflective feature sets ranging from structural reflection to on-demand stack reification. Those operations are often built on top of the other, creating different layers of reflective operations, from low-level to high-level ones.

There is a need to understand the current reflective APIs to understand their underlying use, potential dependencies, and whether some reflective features can be scoped and optional. Such an analysis is challenged by new metaobjects organically introduced in the system, such as first-class instance variables, and their mixture with the base-level API of objects and classes.

In this article, we analyze the reflective operations used in Pharo 12 and their interdependencies. We propose a classification based on their semantics and we identify a set of issues of the current implementation. Such an analysis of reflective operations in Pharo is important to support the revision of the reflective layer and its potential redesign.

1. Introduction

Reflective operations are powerful APIs that let developers build advanced tools or architectures that otherwise would have to be implemented in language implementation engines, would require complex infrastructure (such as code representation), or may simply not be possible. These reflective features support the implementation of tools (*e.g.,* compilers, debuggers, 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 is, however, also a burden. Reflective features defeat static analysis [1] and are usable as security exploits. For example, they allow malicious users to violate encapsulation or execute methods that were not intended to be executed [2–5].

Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both incomplete because some parts of the program may not be included in the application call graph, and unsound because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection [1].

While the quote above is about Java, this tension is exacerbated in the case of deeply reflective languages such as Smalltalk descendants. Pharo, for example, as a descendant of Smalltalk is the essence of a reflective language with advanced reflective operations such as bulk pointer swapping [6], on-demand stack reification, and first-class resumable exceptions. In addition, in Smalltalk-80 and many of its

* Corresponding author.
E-mail addresses: iona.thomas@inria.fr (I. Thomas), stephane.ducasse@inria.fr (S. Ducasse), pablo.tesones@inria.fr (P. Tesone), guillermo.polito@inria.fr (G. Polito).

https://doi.org/10.1016/j.cola.2024.101274
Received 15 November 2023; Received in revised form 7 May 2024; Accepted 15 May 2024
Available online 18 May 2024

derivatives, reflective facilities are mixed with the base-level API of objects and classes [7–9]. They are a key part of the kernel of the language and libraries. Finally, since 2008 the Pharo programming language continuously evolved: 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 12. We scope the analysis to runtime reflection to focus on the core reflective features of the language and its associated virtual machine. Pharo inherited the reflective operations and facilities originally present in Squeak and Smalltalk-80, and extended them over the years. Some reflective methods like Object≫instVarAt: are still present and used, some names have changed and new reflective facilities have appeared.

Extension. This paper extends the workshop paper *Pharo: a reflective language – A first systematic analysis of reflective APIs* [10] with two new analyses whose goal is to better understand how existing reflective operations could be redesigned into a modular reflective API.

We chose to use the term *reflective API* to talk about reflective methods, as it highlights the fact that this is a programming interface offered by Pharo to developers.

The contributions of this article are:

- an up-to-date catalogue of the reflective features in Pharo,
- a classification and an analysis of such operations,
- a discussion of potential re-designs of such reflective operations,
- **Extension:** a dependency analysis between the reflective categories,
- **Extension:** a classification of layers appearing between the reflective categories.

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 [11].

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 12. The technical report [12] 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 the dependencies of reflective categories. Section 7 presents the layered architecture of reflective operations. Section 8 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 MetaObject Protocol(MOP) for Pharo.

2. The need for reflective behavior

> *Reflection is the ability of a program to manipulate as data something representing the state of the program during its execution. There are two aspects of such manipulation: introspection and intercession [...] [13]*

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 [14]. In the middle of the 90s, reflection was heavily explored: structural [15,16], computational [17,18], message-based [17, 19], compile-time [20] and partial reflection [21,22].

Reflection is an important tool that enables many important features of modern languages [23]. 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 are 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 [24–27].
- 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 [28], Distributed Smalltalk [29], or [30] transparently introduce object distribution. Language features such as multiple inheritance [31], backtracking facilities [32], and instance-based programming [33,34] have been introduced. Futures [35,36] or atomic messages [18,37] 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 [15]) and synchronization between asynchronous messages (Concurrent Smalltalk[1] [38]). Other work proposes new object reflective models such as CodA which is a meta-object protocol that controls all the activities of distributed objects [18], meta helix [39] or submethod reflection using Abstract Syntax Tree (AST) annotation [23,40].

More elaborate schemes have been proposed (*e.g., partial behavioral reflection* [21,22]) that provide a more flexible and fine-grained way to specify both the location being reflected and the metaobject invoked. Context-oriented [41] or aspect-oriented programming implementations are often based on reflection [42,43].

The importance and need for reflective features are also illustrated by the effort to offer them in more static languages such as C++ [20], Ada [44], and Java [21,45–47].

Often virtual machine implementations impose restrictions on the changes that are possible [48]. For example, even if Pharo is one of the most advanced reflective languages due to its large spectrum of capabilities, some reflective features (typically intercession *e.g.,* tracing any instance variables accesses or any message sends) are not possible due to their inherent runtime cost. Indeed, virtual machines are engines highly optimized for speed.

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 on its turn saw many different contributions. To give an idea of the activity in Pharo, since 2019 and the versioning of Pharo on GitHub, Pharo has around 100 yearly 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 [8] 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 [49–52], first-class instance variables, and the introduction of new tools using reflection such as the new inspector framework [53], reflectivity [23], object-centric debugging [27], error handling infrastructure [54], and on the fly deprecated message rewritings [55] to name a few. Callau et al. [56] studied the use of dynamic features of programming languages and used 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 [57], but it is not related to a concrete Pharo implementation.

[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.

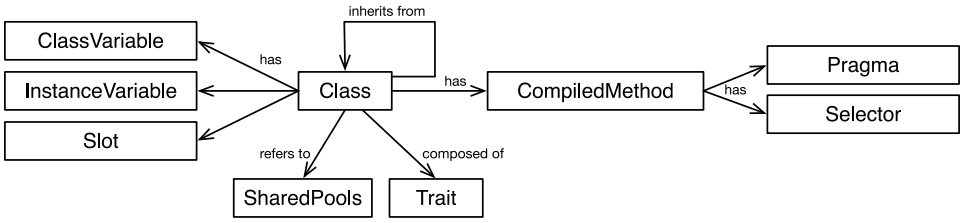

Fig. 1. The structural Pharo metamodel: Class aggregates variables, methods, constant management (SharedPools) and method annotation (Pragma) and exposes related APIs.

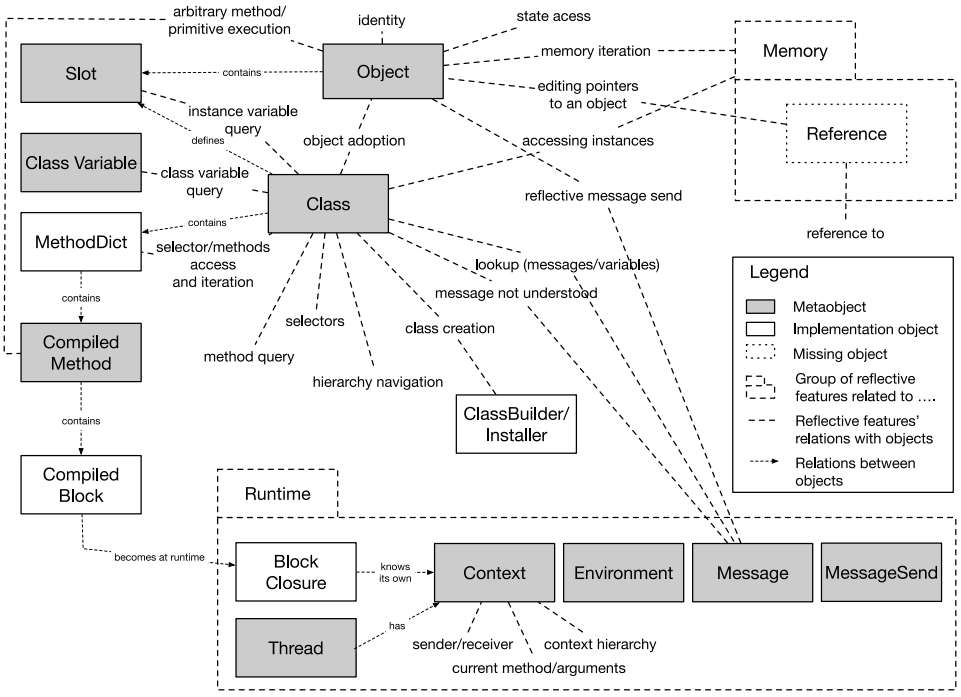

Fig. 2. Metaobjects controlling the reflective APIs of Pharo.

Importance of the analysis. Deploying an application with unneeded reflective facilities produces, however, potential safety issues: reflective operations might be used to bypass security measures or affect the stability of the executing application [11].

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 the technical report [12].

4. Metaobjects, classes and their related APIs

Before giving an overview of the API, we briefly present the structural metamodel of Pharo. The current version is Pharo 12.

4.1. Pharo structural meta model

A class is a central entity in Pharo's structural meta-model [9]. We briefly describe it, since a large part of the API is currently associated with classes.

- 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 uses zero or more shared pools which are collections of constants.
- A class inherits from another class and has zero or more subclasses. Since a couple of versions, a class is composed of traits (class fragments defining methods and state).
- A class contains methods. Methods have a selector and are annotated using zero or more Pragmas [58].

4.2. Overview of the reflective APIs

Fig. 2 shows an overview of the reflective API of Pharo, structured with the classes that expose such APIs in the Pharo 12 release.

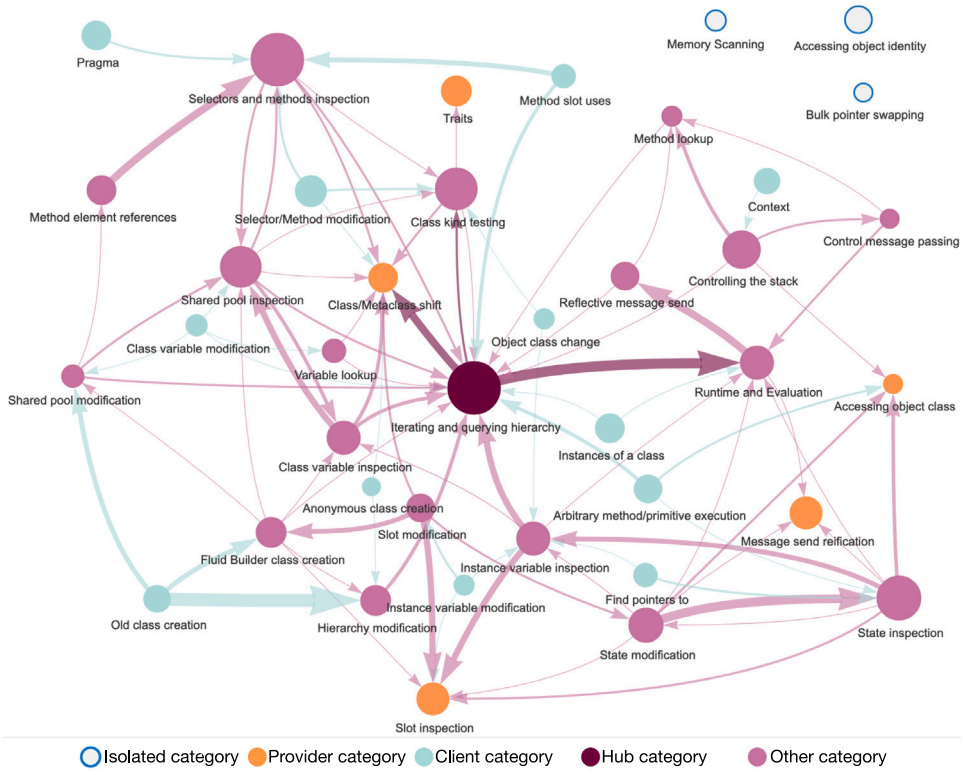

Fig. 3. Reflective category dependency graph. The size of the circle corresponds to the number of selectors in the category. Line thickness depends on the number of dependent selectors. For more details see Appendix B.

MetaObjects. Grey boxes represent first-class objects. Object, Slot, Class, ClassVariable, and CompiledMethod are structural metaobjects. The classes CompiledMethod and CompiledBlock are two entry points to AST nodes and sub-method reflective APIs. We decided not to add such a dimension since submethod reflection is optional and can be seen as compile-time reflection [54].

Implementation objects. We put the MethodDictionary, 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, 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 allows users to iterate memory with methods such as nextInstance and, on the other hand, Runtime which represents the execution aspect of the system with Context (stack reification), Messages, Thread, and Environment (keeping class and variable binding).

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 may override it.

5. A classification and analysis of runtime reflective operations

Rivard [8] 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 adding a new compiled method to a method dictionary. We complement and revisit this classification by adding *References*, and *Memory Scanning* (See Table 1).

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 modification has more impact in terms of state encapsulation. We are aware that systematically presenting lists may be tedious for the reader, this is why the technical report [12] describes systematically all the APIs.

Table 1 gives an overview of the classification: it groups reflective methods into APIs and APIs in high-level categories. The categories are sorted alphabetically. The letters are used to understand Fig. 3.

For each of the APIs we briefly describe it, list its key methods (we often group similar ones), the offered possibilities, and the areas of improvement when appropriate.

Finally, note that the existence of an API is more important than the fact that we classify it under a given heading. For example, asking an object to reflectively execute a method is listed together with other execution-oriented APIs and not directly in the object-centered API. In addition, the next subsections are organized to follow Rivard's classification order.

Sections 6.2 and 7 identify dependencies and layers among the categories.

5.1. Methodology

To analyze and classify the reflective API we focus on the base image of Pharo 12, build 636.[2] We manually identified reflective methods by reading the code of the base image, specifically code belonging to the explicit list of metaobjects and packages present in Figs. 1 and 2. We identified reflective methods using definitions of reflection from [13,59] and categorized them based on a categorization that

[2] Pharo-12.0.0+build.836.sha.8b241ecb87492515bbdd975557ecf8491a4af88b (64 Bit).

Table 1
Overview of the reflective categories and APIs alphabetically sorted. Leading letters are used for Fig. 3.

Categories	APIs
A—Chasing and Swapping pointers	A1—Bulk pointer swapping A2—Find pointers to
B—Class structural Inspection	B1—Class kind testing B2—Class variable inspection B3—Class/Metaclass shift B4—Instance variable inspection B5—Iterating and querying hierarchy B6—Pragma B7—Selectors and methods inspection B8—Shared pool inspection B9—Slot inspection B10—Traits B11—Variable lookup
C—Class structural Modification	C1—Anonymous class creation C2—Class variable modification C3—Fluid Builder class creation C4—Hierarchy modification C5—Instance variable modification C6—Old class creation C7—Selector/Method modification C8—Shared pool modification C9—lot modification
D—Memory Scanning	D1—Memory Scanning D2—Instances of a class
E—Message sending & code execution	E1—Arbitrary method/primitive execution E2—Control message passing E3—Message send reification E4—Method lookup E5—Reflective message send E6—Runtime and Evaluation
F—Object Inspection	F1—Accessing object class F2—Accessing object identity F3—State inspection
G—Object Modification	G1—Object class change G2—State modification
H—Stack Manipulation	H1—Context H2—Controlling the stack
I—Structural queries On methods	I1—Class references I2—Method element references I3—Method slot uses

builds on top of Rivard's. We then tagged the identified reflective methods with a pragma[3] parametrized with its reflective category.

Using this methodology we identified and marked 532 methods with 344 unique selectors as reflective. This number shows the extent of the reflective API in Pharo [12]. The reflective method category tagging was proposed as a pull request to the Pharo repository and accepted by the community in September 2023.[4] In what follows, we refer as *reflective method* any method that is marked in our list. A *reflective operation* is an operation performing reflection and implemented through one or more methods.

In the remainder of this section, we present and analyze the reflective methods and operations we found, divided into categories. Later, in Section 6, we analyze the dependencies between these categories.

5.2. Object inspection reflective operations

The first category of reflective operations is centered around object inspection. Rivard [8] described these operations in the *Meta-Operations* category, but he grouped inspection and modification. Our category is composed of three subcategories:

- ***State inspection*** to read the values of the variables of an object.
- ***Accessing object identity*** to identify an object.
- ***Accessing object class*** to read the class of an object.

In Pharo, all instance variables are private, meaning they are not readable and writable by any other object. They are only 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. Key examples of this category are Object≫instVarAt: and Object≫instVarNamed:, which read an instance variable of an object from its index or name respectively. These operations combine well with those in the ***Accessing object class*** subcategory to work on object internal structure *e.g.,* 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... The ***Accessing object identity*** supports checking the identity of an object. basicIdentityHash is used for implementing identityHash variants, scanning for an object in a method dictionary, and testing.

Examples of uses.

- Serializing objects.
- Inspecting objects.
- Implementing hash methods.

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 improvement.

- In the analyzed Pharo version, 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 [60]. Such tools rely heavily on reflection, and loading several tools at the same time might lead to bugs and instabilities due to incompatibilities between them.

5.3. Object modification reflective operations

The second category 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 write the values of 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 allow one to bypass encapsulation and modifying variables of third-party objects. This could be used to write variables in an unanticipated way when they were originally designed to not 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 become an instance of another class, which is particularly important in Pharo's live environment when a

[3] A pragma is a method annotation in Pharo's parlance.
[4] https://github.com/pharo-project/pharo/pull/14821.

class has to be rebuilt. The ***Manipulating object identity*** category contains only one operation becomeForward:copyHash:[5]

Examples of uses.

- Copying objects.
- Deserializing objects.
- Modifying object on the fly in the debugger.
- Migrate instances between two class versions.

Areas of improvement. The API on object class change is weak and limited. The object state may be lost in the process and some constraints (the two classes should have the same format) make it difficult to change the actual class. Overriding the methods providing these operations provides a way to limit reflection at the cost of limiting services such as instance migration provided by the environment.

5.4. Class structural inspection reflective operations

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

- ***Class/metaclass shift*** to navigate between a class and its metaclass.
- ***Iterating and querying hierarchy*** to query class hierarchies.
- ***Instance variable inspection***, ***Class variable inspection***, ***Shared pool inspection***, ***Slot inspection*** to query variable definitions. ***Slot inspection*** provides a higher level view compared to ***Instance variable inspection***. Slots are either defined locally in a class or imported, for example from a trait. Thus the existence of the localSlots and slots operations.
- ***Selector and method inspection*** to query the set of methods/selectors implemented by a class.
- ***Variable lookup*** the access the binding of a variable.
- ***Pragmas*** to query pragmas.
- ***Class kind testing*** to query the state of a class (installed, obsolete, anonymous...).

Possibilities offered. The structural class introspection is large. It is mainly used by tools. It supports the interpretation of object inspection. ***Iterating and querying hierarchy*** methods support navigation of the graph with messages such as superclass and allSubclasses. ***Selector and method inspection*** methods allow one to check existing selectors and methods. All variable query operations allow one to list existing variables. Some methods such as isKindOf: or respondsTo: produce suboptimal designs when used at the base level. respondsTo: allows one to query if an object understands a given selector.

Examples of uses.

- Object Serialization and Deserialization.
- Code browsing.
- Object inspection.

Areas of improvement.

- 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 methods such as methodNamed:).

5.5. Class structural modification reflective operations

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) [2]. It helps to define instance-specific methods.

Examples of uses.

- Reflective code modification.
- Object-Centric Reflection.
- Proxy implementations.

Areas of improvement.

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

5.6. Method creation reflective operations

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.

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 code loader. It is an important asset that supports the bootstrap of Pharo [61] and ensures that the compiler is loadable into a system without having a compiler to compile and install code.

[5] Using becomeForward:copyHash: to swap a reference it is possible to change the object identity.

Areas of improvement. 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.

5.7. Structural queries on methods reflective operations

This category supports the cross-referencing between methods, instance variables, and classes. It is composed of two subcategories:

- ***Method slot uses*** to query usages of variable read/writes.
- ***Method element references*** to query internal implementation of methods.

Possibilities offered. These two subcategories are central for all the cross-referencing and code navigation in Pharo.

Examples of uses. These operations are important for the IDE and tools.

- query method senders and implementors
- query methods reading/writing a variable

Areas of improvement.

- The duality of selectors and methods could be handled better, *e.g.,* whichMethodsReferTo: vs. whichSelectorsReferTo:. We suggest not exposing the compiled method, since it is always possible to get the method out of a selector. It would lead to a more compact API.
- ***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.

5.8. Message sending and code execution reflective operations

This category of operations allows us to explicitly send messages, handle lookup failures, or execute compiled methods. It is composed of different subcategories:

- ***Reflective message send*** to lookup and execute methods.
- ***Arbitrary method/primitive execution*** to execute methods without lookup.
- ***Method lookup*** to simulate the method lookup.
- ***Control message passing*** to control message sends.
- ***Message send reification*** 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 allows 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 ***Reflective message send*** do a method lookup, methods of ***Arbitrary method/primitive execution*** allow us to execute directly a compiled method or a primitive operation.[6] While the methods of ***Reflective message send*** do a method lookup, methods of ***Arbitrary method/primitive execution*** allow us to execute directly a compiled method or a primitive.[7]

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

Examples of uses.

- Implementing frameworks with naming flexibility such as in SUnit.
- Decouple user interfaces from model objects.
- Proxy implementations.
- Debugging and profiling.

Areas of improvement.

- 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. Message represents 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:. MessageSend represents the concept of sending a message and holds in addition a receiver. Such a class is not used by the runtime and offers an API compatible with block closures: The messages value: and its variants allow one to execute a message. 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. There are, for example, three different reflective methods implementing similar behavior in ***Arbitrary method/primitive execution***. We suggest that only methods on CompiledMethod should be kept. The definition on ProtoObject would have the pernicious side effect of making domain developers think 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 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 expose its nature.

5.9. Chasing and atomic pointer swapping reflective operations

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 of identifying pointers to a given object (*e.g.,* ProtoObject≫pointersTo and ProtoObject≫pointsTo:).

Possibilities offered. ***Find pointers to*** is useful for building 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. Pharo's implementation implements this operation efficiently by using forwarders at the VM level [6]. 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.

[6] A primitive operation is a call to a virtual machine internal behavior.
[7] A primitive operation is a call to a virtual machine internal behavior.

Areas of improvement. 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 API such as the ones presented in the next section. Similarly, a slower version of ***Bulk pointer swapping*** could be possible.

5.10. Memory scanning reflective operations

This category contains two subcategories: *Memory scanning* that supports 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 [62]. The method nextObject and nextInstance are key to building other functionalities such as allInstances.

Examples of uses. The main use is the migration of instances between two versions of one class. All objects need to be obtained to be migrated to the new class and be potentially rebuilt if there are changes in their shape *e.g.,* if a variable is added/removed. 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 improvement. Understanding whether such a reflective API can be optional and only be loaded on demand would be a step toward building a more compact, tidier, and secure reflective MetaObject protocol.

5.11. Stack manipulation reflective operations

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 make the execution stack in a linked list.

Possibilities offered. Stack manipulation operations 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.

- Implementing exceptions.
- Debuggers.
- Bytecode interpretation.

Areas of improvement. 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. 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 [23].

Additionally, the fact that essential language features such as exceptions are built on top of it makes 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. Reflective API interdependencies

We extend the categorization of reflective operations shown in Section 5 with *inter-category* (Section 6.2) and *inter-method dependencies* (Section 7). Understanding the dependencies between reflective APIs will help in evolving the current API. This means, for example, restructuring parts of the API to fit better in the system or, identifying optional APIs that could be packaged separately. This would lead to a simpler, lighter base language, with potentially less threats to the stability of the system. In this section, we present an analysis of interdependencies between the categories presented in Section 5.

6.1. Methodology

We identify dependencies between reflective methods using static analysis based on their selectors, given that Pharo is a dynamically-typed language without type annotations. We say that reflective selector A depends on reflective selector B if any method with the selector A sends a message with selector B. Unless specified otherwise, we focus only on users who are reflective methods themselves. When relevant, we extend our analysis to base-level users.

Unless stated otherwise, when analyzing categories interdependencies we leave outside of our analysis some selectors that present both reflective implementors and not reflective implementors. This is the case of largely used selectors such as at:, at:put: and size that are implemented as reflective methods in Context but have non-reflective counterparts in collections. We also ignore from our analysis a dozen other selectors that have reflective implementors in different categories. (See Appendix A for the full list.)

Thus, the uses of these selectors do not necessarily imply dependencies on the reflective version of the method.

Section 6.2 presents a high-level view of the interdependencies in the reflective categories. For the sake of presentation, the dependency analysis removes, in addition to the previous list of selectors, the selector class as it is the one with the highest number of connections (28 other selectors have at least one method depending on class).

6.2. Reflective categories interdependencies overview

Fig. 3 presents a graph that illustrates how all reflective categories connect with their dependencies. The figure shows that most reflective categories are building on each other.

Based on this data we identified four types of categories based on their dependency topology, shown in Fig. 4:

- *Isolated categories*: These categories do not depend on any other categories and no other category depends on them.
- *Provider categories*: These categories do not depend on any other categories but they provide operations that are used by other categories.
- *Client categories*: These categories depend on other categories, but no other reflective category depends on them.
- *Hub categories*: These categories depend on a lot of other categories and many other categories depend on them.

Isolated and *Client* categories are two types of categories that could be the first candidates for being packaged in a library. This would also require analyzing the base-level uses of their APIs in the base image, which is outside the scope of this paper.

6.3. Isolated reflective categories

Memory Scanning - Memory Scanning, ***Object Inspection - Accessing object identity***, and ***Chasing and swapping pointers - Bulk pointer swapping*** are not relying on any other categories because they rely on primitive operations. There is also no other reflective category relying on them. To understand how they fit in Pharo, in this specific section

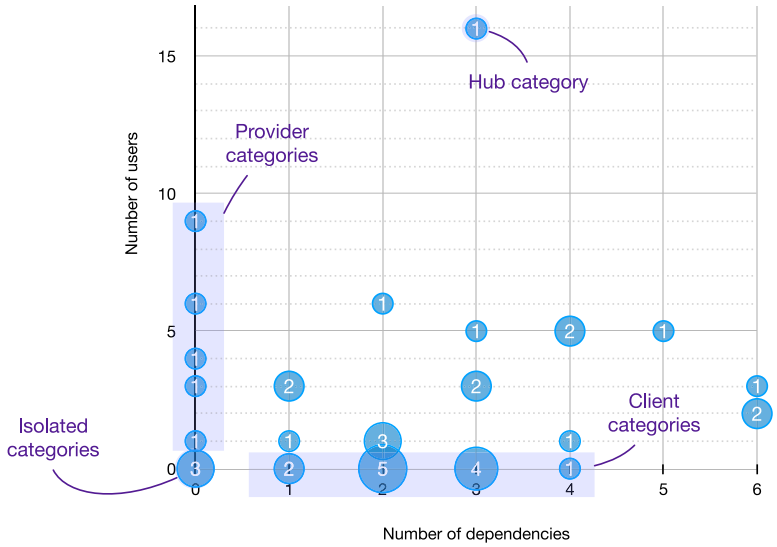

Fig. 4. Categories according to the number of incoming and outgoing dependencies.

analyze all users – reflective and not – in the base image, not only the ones that are reflective methods. As explained below, they provide low-level APIs that are either not used in the base image (*i.e.,* their selector appears neither in reflective nor non-reflective code) or used for low-level implementation details like defining equality and growing collections in memory.

For ***Memory Scanning—Memory Scanning***, implementations of both nextObject and someObject are directly calling primitive operations. Those are two low-level methods that allow us to iterate on the memory. The only sender of nextObject in the base image is a regression test on ProtoObject checking for a previous image crash. someObject is never used.

For ***Object Inspection—Accessing object identity***, 7 out of 11 methods are calling primitive operations, two have an internal dependency to basicIdentityHash. The two remaining methods are the overriding versions of basicIdentityHash and identityHash for SmallInteger which are based on the value of the integer itself. Those methods are used to implement hash functions and check for equality. They are not used by other reflective features.

The three members of ***Chasing and swapping pointers—Bulk pointer swapping*** rely on three methods in Array that call the primitive operations. While the method become: is only used for some tests, becomeForward:copyHash: is used in a proxy implementation in the Iceberg package, a version control package. becomeForward: is the most used of the three, but for non-reflective purposes: during the bootstrap to handle undeclared variables, to edit the specialObjectsArray, to manage internal representations of collections and method dictionaries, and to convert weak announcements into strong ones.

6.4. Provider categories

We have four categories that are not relying on any other categories, but are providing APIs used by others :

- ***Class structural inspection—Class/Metaclass shift***. Its implementations of classSide, instanceSide, and variants rely on the metamodel and its hierarchy. In terms of selectors that have been removed for clarity, it is only using class once. Its messages are however used in a dozen of other categories.
- ***Class structural inspection—Slot inspection***. Slots are stored in the metaobjects of Pharo in a collection. Their implementation does not rely on primitive operations. Accessing the slots and querying them are used to access the state of an object or class and modify it, but also to create or modify the instance variables of a class. This covers seven categories of reflective operations.
- ***Object Inspection—Accessing object class*** Its four methods rely on the same primitive operation, either directly or indirectly. It is usually used either for comparisons or to access the API of the class. In terms of reflective API Context≫objectClass: and MirrorPrimitives class≫classOf are used to access instance variables, send messages, and execute arbitrary methods/primitive operations. With more than 4800 senders in both the base-level and reflective methods, class is one of the selectors that is the most used in the Pharo image. This is why in the rest of this article, ProtoObject≫class is excluded when looking at the connections between categories. The class method is used directly by 15 out of 40 categories, from which 13 use only class from this category.
- ***Class structural inspection—Traits***. Like slots, traits do not rely on primitive operations. They are implemented in the metaobjects of Pharo. In the reflective API, they are only used in the ***Class structural inspection—Class kind testing*** to test for users of a class defining a trait.

6.5. Client categories

We have eleven categories which are relying on other categories and are not used by others:

- ***Chasing and swapping pointers—Find pointers to***
- ***Class structural inspection—Pragma***
- ***Class structural modification—Anonymous class creation***
- ***Class structural modification—Class variable Modification***
- ***Class structural modification—Instance variable modification***
- ***Class structural modification—Old class creation***
- ***Class structural modification—Selector/Method modification***
- ***Memory Scanning—Instances of a class***
- ***Message sending and code execution—Arbitrary method/primitive execution***
- ***Object Modification—Object class change***
- ***Stack Manipulation—Context***
- ***Structural queries on methods—Method slot uses***

As there are eleven client categories, we focus on highlighting commonalities instead of detailing each of them. Five of them are from the bigger category: ***Class structural modification***. Client categories

usually provide higher-level APIs, like ***Structural queries on methods—Method slot uses***. For example, in ***Class structural modification—Instance variable modification***, the methods relying on other categories are the ones offering to add or remove instance variables by their names. Those methods hide the complexity of the implementation with slots, which is powerful but more complicated to understand. Another case is the old class creation API. These API methods have been rewritten to rely on the new fluid class builder. We believe that it has no users because of the migration to the new API. This API is only present for retro-compatibility and class creation required by other reflective APIs like slot modification using the new API.

6.6. Iterating and querying hierarchy, a hub category

In Fig. 3, we identify one hub category in the top left that presents many more connections than the others. In Fig. 4 we see that this category appears to rely on three other categories for its implementation and has 16 categories using it directly. The ***Class structural inspection—Iterating and querying hierarchy*** is a hub category. In particular, its user with the strongest connection is ***Class structural inspection—Instance variable inspection***. The operations to iterate and query the class hierarchy are used to look up for instance variable definitions. The three categories it relies on are:

- ***Class structural inspection—Class kind testing***. The two messages isTrait and isMetaclassOfClassOrNil are used respectively in the implementations of includesBehavior: and subclassesDo: to check for specific cases in the Metaclass class.
- ***Class structural inspection—Class/Metaclass shift***. Both instanceSide and classSide are used by three Metaclass methods: subclasses, subclassesDo: and obsoleteSubclasses. Those three methods rely on the instance side to get the subclasses: instances of Metaclass have a parallel hierarchy to the instances of class. The subclasses of the class side are the same as the class side of the instance side's subclasses.
- ***Message sending and code execution—Runtime and Evaluation***. The messages value: and value:value: are used to evaluate blocks in eight methods, including five enumeration and iteration methods, and a method looking for a superclass verifying a criteria passed as a block. These are false positives due to polymorphism between reified messages and blocks.

7. Layers and internal dependencies in reflective categories

To analyze the dependencies within a single category, we build visualizations showing each selector's dependencies (See Section 6.1 for the dependency heuristic). All selectors belonging to the studied categories are in black while selectors from other categories have other colors. To get a more detailed view in this analysis, we keep all selectors including the one excluded previously.

The graph is laid out to show the hierarchy of dependencies with dependent selectors placed below the ones they depend on. Two selectors with a dependency relationship are placed as close as possible while respecting the vertical positioning. (For example in Fig. 5 sender is just above pointersToExcept:among: instead of being higher on the graph). This leads to the apparition of some visual layers (see the blue annotations on the following figures).

7.1. Chasing and swapping pointers—Find pointers to

The category ***Chasing and swapping pointers—Find pointers to*** is an example of the layers that emerge in some reflective APIs. When looking at the graph in Fig. 5 we see that the method pointsTo: is the core one of this category, with all others except pointOnlyWeaklyTo: relying on it. This pointsTo: method tests for the presence of the parameter either as the class or in the instance variables of the receiver.

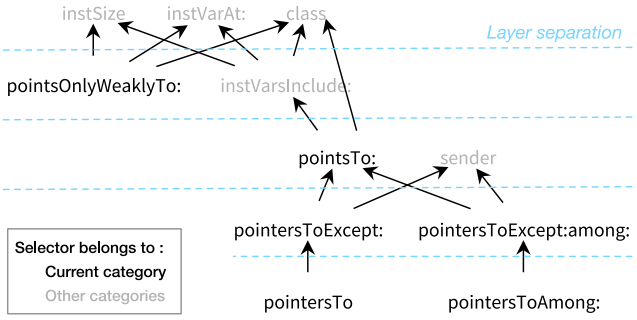

Fig. 5. Subgraph of Chasing and swapping pointers—Find pointers to. The black selectors belong to the studied category. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

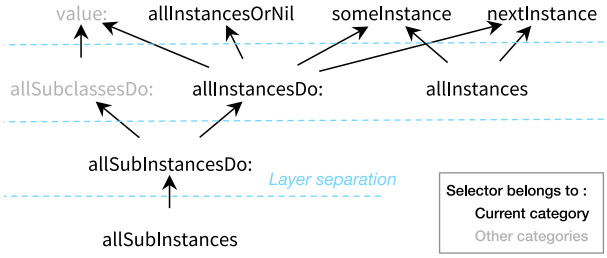

Fig. 6. Subgraph of Memory Scanning—Instances of a class. Black selectors belong to the studied category.

By building on top of this method, we get more complex operations that allow one to get all pointers to an object.

We also notice that pointersTo: and pointersToAmong: rely respectively on pointersToExcept: and pointersToExcept:among:. The simpler APIs rely on the ones with more parameters for their implementations, therefore avoiding code duplication and facilitating code evolution. The presence of those simpler APIs in addition to the ones with more parameters reflects the absence of default parameters in Pharo.

pointsOnlyWeaklyto: is isolated in the category because it provides an independent operation: when calling it, a precondition is that the receiver is pointing to the parameter. This method tests if the references are weak or not. To do so it is using a lower level API, as information on the strength of references is not available at the abstraction level of pointsTo:.

7.2. Memory Scanning—Instances of a class

The category **Memory Scanning—Instances of a class** is another example of a layered API. Here we got three root methods in the category allowing to access the instances of a class: allInstancesOrNil, someInstance, and nextInstance. More complicated iteration methods and methods accessing sub-instances are built on top of those.

We notice in Fig. 6 that allInstancesOrNil and allInstances look similar but do not depend on each other. This is due to both of them relying on the same primitive operation. While allInstancesOrNil fail when running out of memory, allInstances has a backup implementation, which relies on someInstance to get the first instance of a class and nextInstance to iterate. The presence of both operations could be due to historical and retro-compatibility reasons.

Another noticeable point is the fact that while allInstancesDo: relies on allInstances, allSubinstances relies on allSubinstancesDo:. This is because allSubinstances requires iterating over instances of subclasses, and therefore if one wants to call a method on all subinstances, it is better to apply it directly rather than creating a new

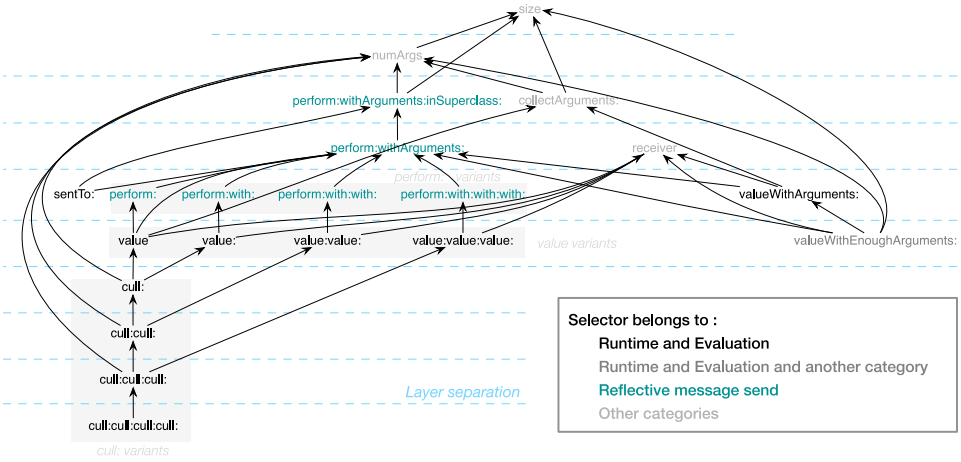

Fig. 7. Subgraph of ***Message sending and code execution - Runtime and Evaluation*** and ***Message sending and code execution - Reflective message send***.

collection and then reiterate over it. While unintuitive, this is an optimization for subinstances.

7.3. Message sending and code execution - Runtime and Evaluation & message sending and code execution - Reflective message send

Fig. 7 shows in black the ***Runtime and Evaluation*** subcategory, and in teal operations of the ***Reflective message send*** subcategory. The former relies on the latter. Moreover, these categories are organized in different layers. In **Reflective message send**, the simpler APIs, with few arguments each as a different parameter, rely on the perform:withArguments: that takes an array of arguments in its second parameter. Finally perform:withArguments: itself relies on perform:withArguments:inSuperclass:. This is a similar structure as ***hashing and swapping pointers—Find pointers to*** category, seen in Section 7.1.

Runtime and Evaluation contains value, cull: and their variants with one or more arguments. Variants of value assume that they are given the appropriate amount of arguments, while variants of cull ignore exceeding arguments. This leads to a horizontal layer in the API with all the variants of value relying on the perform: variant with the corresponding number of arguments. Variants of cull: rely both on the value variant with the same number of arguments if there is the right number of arguments, and the cull: version with one less argument in case there are too many arguments. The cull: method with only one argument relies instead on the value message if no arguments are expected. This highlights the cost of using cull: variants when the number of arguments is known.

Once again here we observe that the absence of default arguments in Pharo leads to more methods being created to compensate. With empty default parameters, variants of cull:, value:, and perform: could be summarized by three methods with the maximum number of parameters.

8. 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 [63], 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.

8.1. Threats to validity

By construction, this study is susceptible to false negatives on reflective methods, both false positives and false negatives on dependencies.

False negatives in reflective methods. As the identification of reflective methods was done manually, we might have missed some packages and methods as we could not read through the whole image. Therefore some reflective methods might not be identified and tagged as such. However, with 532 identified methods, we believe that the presented study is representative of the Pharo reflective API. Moreover, those tags have been submitted to Pharo, reviewed, and integrated for later versions.

False positives in dependency identification. Having base-level methods being polymorphic with reflective ones might lead to false positives during the static analysis. While we identified a few selectors (at:, at:put:, size and value) for which identified dependencies are not reliable, some others might have slipped through unidentified. In the dependency graph of selectors, selectors belonging to more than one category are shown. However, we do not differentiate between the dependencies of methods belonging to different categories. This may lead to some false positives. For example, if a single method implementing the selector has a dependency, the categories of the other methods implementing this selector will show the dependency even if it is not their version of the method.

False negatives in dependency identification. As we removed selectors belonging to several categories when drawing the category dependencies graph, some dependencies relationships are missing. However, only 13 selectors on 344 are removed and they belong to varied categories. This leads us to the conclusion that no strong dependencies are going by unidentified.

One of the limits of the study is that we only look at direct dependencies between reflective methods. Therefore if a reflective method calls a non-reflective method, which itself calls a second reflective method, the dependency between both reflective methods will not be identified.

Non-formal definition of layers. While visual layers used in Section 7 offer a way to understand the hierarchy of dependencies, they do not have a formal definition. Slightly different visualizations of dependencies could generate other layers. However, as the organization of the selectors by the visualizations is specified, this makes those more reproducible.

8.2. General concerns

About dual entities. On several occasions, the MOP proposes a kind of duplicated API: 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 forces the developer to have an object when it may not be possible. It means that using a name is a good approach. In particular, such metaobjects (compiled method, slot) expose their name. We suggest reducing the API spectrum by not proposing two APIs but instead favoring one based on the name for query and access. For modifications, the developer will query first based on the name to access an object and then perform the corresponding operation. In that regard, the question of the application of the Law of Demeter should be evaluated since it tends to produce larger APIs.

Absence of clear layers between base-level and meta-level. On several occasions, we see the need to 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 variables that feel outdated in a modern language). Finally, some APIs are large because basic functionality is augmented with helper behavior built on top of such basic functionality. While this is a good practice to promote code reuse and offer developers stronger APIs, we suggest laying off 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 modify the internals of the system and the causal connection makes sure that the modifications get in effect. We see this approach as an organic one and from this perspective we say that the metaobject Protocol of Smalltalk has been less designed than the one of CLOS [64].

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 executions of the methods. In Pharo, the hook to specify the class of a method is not clear. More important, when a method is executed no identified method is called before the method execution. Frameworks such as Method Proxy, MethodWrappers [26] 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 [18]. 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 executed during code execution. 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 [8] 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 installs 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 for the Code Browser to be notified.

A clear notification API should guarantee that the tools and libraries 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 [65]. In this scenario, there is a need to represent a method with a source code that is not one of the currently installed compiled methods. 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 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.

8.3. Mirror architecture

At the language level, Mirrors [59,66] 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 [66], StrongTalk [67] and Newspeak [68]. Mirrors offer a mirror implementation in Pharo. 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 [69] 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.

8.4. 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 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. [70] proposes a model for the reification of the control of reflection. Indeed controlling the runtime 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 the implementation of the language from the reflective API. Lorenz proposes a pluggable reflection [71], 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.

9. Conclusion

This article acknowledged that the runtime reflection offered by Pharo needed a deep and systematic analysis after the evolution of Squeak and the subsequent evolution of Pharo since 2008. The analysis of Rivard [8] while interesting is simply dated. Indeed Pharo metaobjects evolved and got [49–52], first-class instance variables, and the introduction of new tools using reflection such as the new inspector framework [53], reflectivity [23], object-centric debugging [27], error handling infrastructure [54] and on the fly deprecated message rewritings [55]. This is not counting the full rewrite of the compiler.

In this article, we presented a new systematic and deep analysis of the reflective APIs revealing 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.

CRediT authorship contribution statement

Iona Thomas: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Stéphane Ducasse:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Pablo Tesone:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Guillermo Polito:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Ignored selectors

In our analysis, we have ignored some selectors. Some of the ignored selectors are left out of the diagrams and figures to make them more readable (like class), others are highly polymorphic selectors used in non-reflective libraries and frameworks, and they introduce noise in the whole analysis especially producing false positives in the dependency analysis (like at:). In this section, we explain the reasons for ignoring them and the complete list of ignored selectors.

Selector class is removed by default for presentation's sake. An analysis of its uses is in Section 6.4, in paragraph ***Object Inspection—Accessing object class***.

The following selectors are removed due to polymorphism with non-reflective selectors:

- at:
- at:put:
- value
- size

The following selectors are removed due to polymorphism across multiple reflective categories:

- valueWithEnoughArguments:
- outerContext
- usingMethods
- receiver
- numArgs
- arguments
- arguments:
- instVarAt:put:
- selector
- selector:
- receiver:
- isClass
- sender

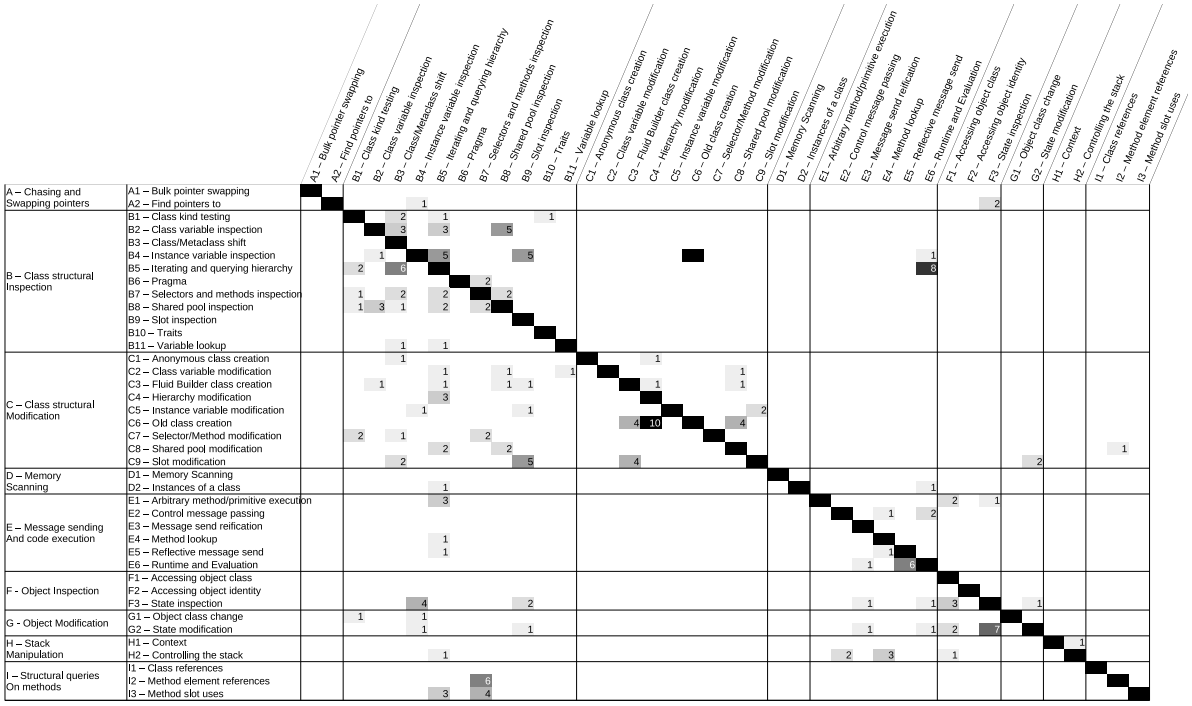

Fig. B.8. Matrix of dependencies between categories. The category in row X depends on the category in Column Y if there is a number at the intersection. The number corresponds to the number of different selectors depending on the other category.

Appendix B. Dependency matrix

Fig. B.8 presents a table with all the categories and the interdependency of them. The intersection of categories shows the number of selectors used by the category on top of the category on the right. A higher number shows a higher level of dependency between the two categories.

References

[1] B. Livshits, J. Whaley, M.S. Lam, Reflection analysis for java, in: Proceedings of Asian Symposium on Programming Languages and Systems, 2005.

[2] S. Ducasse, Evaluating message passing control techniques in Smalltalk, J. Object-Oriented Program. (JOOP) 12 (6) (1999) 39–44.

[3] G. Richards, C. Hammer, B. Burg, J. Vitek, The eval that men do: A large-scale study of the use of eval in JavaScript applications, in: Proceedings of Ecoop 2011, 2011.

[4] M.S. Miller, M. Samuel, B. Laurie, I. Awad, M. Stay, Caja Safe Active Content in Sanitized Javascript, Technical Report, Google Inc., 2008.

[5] G.C. Hunt, J.R. Larus, Singularity: rethinking the software stack, SIGOPS Oper. Syst. Rev. 41 (2) (2007) 37–49.

[6] E. Miranda, C. Béra, A partial read barrier for efficient support of live object-oriented programming, in: International Symposium on Memory Management, ISMM'15, Portland, United States, 2015, pp. 93–104.

[7] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, 1989.

[8] F. Rivard, Smalltalk: a reflective language, in: Proceedings of REFLECTION'96, 1996, pp. 21–38.

[9] A.P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, Kehrsatz, Switzerland, 2009, p. 333.

[10] I. Thomas, S. Ducasse, P. Tesone, G. Polito, Pharo: a reflective language - a first systematic analysis of reflective APIs, in: IWST 23 - International Workshop on Smalltalk Technologies, Lyon, France, 2023.

[11] C. Teruel, S. Ducasse, D. Cassou, M. Denker, Access control to reflection with object ownership, in: Dynamic Languages Symposium, DLS'2015, 2015.

[12] I. Thomas, S. Ducasse, P. Tesone, G. Polito, A Classification of Runtime Reflective Operations in Pharo, Technical Report, Inria - Evref, 2023.

[13] D.G. Bobrow, R.P. Gabriel, J. White, CLOS in context — The shape of the design, in: A. Paepcke (Ed.), Object-Oriented Programming: The CLOS Perspective, MIT Press, 1993, pp. 29–61.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Aksit, S. Matsuoka (Eds.), European Conference on Object-ORiented Programming, ECOOP'97, Springer-Verlag, 1997, pp. 220–242.

[15] J.-P. Briot, P. Cointe, Programming with explicit metaclasses in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 419–432.

[16] N. Bouraqadi, T. Ledoux, F. Rivard, Safe metaclass programming, in: Proceedings OOPSLA '98, 1998, pp. 84–96.

[17] J. Ferber, Computational reflection in class-based object-oriented languages, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 317–326.

[18] J. McAffer, Meta-level programming with coda, in: W. Olthoff (Ed.), Proceedings ECOOP '95, in: LNCS, vol. 952, Springer-Verlag, Aarhus, Denmark, 1995, pp. 190–214.

[19] W. Cazzola, Evaluation of object-oriented reflective models, 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, 1998, pp. 3–540.

[20] S. Chiba, A metaobject protocol for C++, in: Proceedings of OOPSLA '95, in: ACM SIGPLAN Notices, vol. 30, 1995, pp. 285–299.

[21] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in: Proceedings of OOPSLA '03, ACM SIGPLAN Notices, 2003, pp. 27–46.

[22] D. Röthlisberger, M. Denker, É. Tanter, Unanticipated partial behavioral reflection: Adapting applications at runtime, J. Comput. Lang. Syst. Struct. 34 (2–3) (2008) 46–65.

[23] S. Costiou, V. Aranega, M. Denker, Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use, Art Sci. Eng. Program. 4 (3) (2020).

[24] J.H. Heinz-Dieter Bocker, What tracers are made of, in: Proceedings of OOPSLA/ECOOP '90, 1990, pp. 89–99.

[25] F. Pachet, F. Wolinski, S. Giroux, Spying as an object-oriented programming paradigm, in: Proceedings of TOOLS EUROPE '93, 1993, pp. 109–118.

[26] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: Proceedings European Conference on Object Oriented Programming, ECOOP'98, in: LNCS, vol. 1445, Springer-Verlag, 1998, pp. 396–417.

[27] S. Costiou, M. Kerboeuf, C. Toullec, A. Plantec, S. Ducasse, Object miners: Acquire, capture and replay objects to track elusive bugs, J. Object Technol. 19 (1) (2020) 1:1–32.

[28] B. Garbinato, R. Guerraoui, K.R. Mazouni, Distributed programming in GARF, in: R. Guerraoui, O. Nierstrasz, M. Riveill (Eds.), Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, in: LNCS, vol. 791, Springer-Verlag, 1994, pp. 225–239.

[29] J.K. Bennett, The design and implementation of distributed Smalltalk, in: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '87, ACM, New York, NY, USA, 1987, pp. 318–330.

[30] P.L. McCullough, Transparent forwarding: First steps, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 331–341.

[31] A.H. Borning, D.H. Ingalls, Multiple inheritance in Smalltalk-80, in: Proceedings at the National Conference on AI, Pittsburgh, PA, 1982, pp. 234–237.
[32] W.R. LaLonde, M.V. Gulik, Building a backtracking facility in Smalltalk without kernel support, in: Proceedings OOPSLA '88, ACM SIGPLAN Notices, Vol. 23, 1988, pp. 105–122.
[33] K. Beck, Instance specific behavior: How and why, Smalltalk Rep. 2 (7) (1993).
[34] K. Beck, Instance specific behavior: Digitalk implementation and the deep meaning of it all, Smalltalk Rep. 2 (7) (1993).
[35] G.A. Pascoe, Encapsulators: A new software paradigm in Smalltalk-80, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, 1986, pp. 341–346.
[36] W. LaLonde, J. Pugh, Inside Smalltalk: Volume 1, Prentice Hall, 1990.
[37] B. Foote, R.E. Johnson, Reflective facilities in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 327–336.
[38] Y. Yokote, M. Tokoro, Experience and evolution of ConcurrentSmalltalk, in: Proceedings OOPSLA '87, Vol. 22, 1987, pp. 406–415.
[39] S. Chiba, G. Kiczales, J. Lamping, Avoiding confusion in metacircularity: The meta-helix, in: K. Futatsugi, S. Matsuoka (Eds.), Proceedings of ISOTAS '96, Vol. 1049, Springer, 1996, pp. 157–172.
[40] M. Denker, S. Ducasse, A. Lienhard, P. Marschall, Sub-method reflection, in: Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007, Vol. 6/9, ETH, 2007, pp. 231–251.
[41] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: An overview of ContextL, in: Proceedings of the Dynamic Languages Symposium, DLS'05, ACM, New York, NY, USA, 2005, pp. 1–10.
[42] N. Bouraqadi, A. Seriai, G. Leblanc, Towards unified aspect-oriented programming, in: Proceedings of 13th International Smalltalk Conference, ISC'05, 2005.
[43] A. Bergel, R. Hirschfeld, S. Clarke, P. Costanza, Aspectboxes — Controlling the visibility of aspects, in: Joaquim Filipe, Boris Shiskov, Markus Helfert (Eds.), Proceedings of the International Conference on Software and Data Technologies, ICSOFT 2006, 2006, pp. 29–38.
[44] P. Rogers, A. Wellings, OpenAda: Compile-time reflection for ada 95, in: Reliable Software Technologies - Ada Europe, in: LNCS, vol. 3063, Springer, 2004.
[45] I. Welch, R.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, San Antonio, Texas, USA, 2001, pp. 119–130.
[46] B. Redmond, V. 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.
[47] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: Proceedings of European Conference on Object-Oriented Programming, Vol. 2374, Springer-Verlag, 2002, pp. 205–230.
[48] G. Chari, D. Garbervetsky, S. Marr, S. Ducasse, Fully reflective execution environments: Virtual machines for more flexible software, Trans. Softw. Eng. 45 (2018) 858–876.
[49] S. Ducasse, N. Schärli, R. Wuyts, Uniform and safe metaclass composition, J. Comput. Lang. Syst. Struct. 31 (3–4) (2005) 143–164.
[50] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, A.P. Black, Traits: A mechanism for fine-grained reuse, ACM Trans. Program. Lang. Syst. (TOPLAS) 28 (2) (2006) 331–388.
[51] P. Tesone, S. Ducasse, G. Polito, L. Fabresse, N. Bouraqadi, A new modular implementation for stateful traits, Sci. Comput. Program. 195 (2020) 1–37.
[52] P. Tesone, G. Polito, L. Fabresse, N. Bouraqadi, S. Ducasse, Preserving instance state during refactorings in live environments, Future Gener. Comput. Syst. 110 (2020) 1–17.
[53] A. Chiş, O. Nierstrasz, A. Syrel, T. Gîrba, The moldable inspector, in: 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!), in: Onward! 2015, ACM, New York, NY, USA, 2015, pp. 44–60.
[54] S. Costiou, T. Dupriez, D. Pollet, Handling error-handling errors: dealing with debugger bugs in pharo, in: International Workshop on Smalltalk Technologies - IWST 2020, 2020.
[55] S. Ducasse, G. Polito, O. Zaitsev, M. Denker, P. Tesone, Deprewriter: On the fly rewriting method deprecations, J. Object Technol. (JOT) 21 (1) (2022).
[56] O. Callau, R. Robbes, D. Rothlisberger, E. Tanter, How developers use the dynamic features of programming languages: the case of smalltalk, in: Mining Software Repositories International Conference, MSR'11, 2011.
[57] F.-N. Demers, J. 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.
[58] S. Ducasse, E. Miranda, A. Plantec, Pragmas: Literal messages as powerful method annotations, in: International Workshop on Smalltalk Technologies, IWST'16, Prague, Czech Republic, 2016.
[59] G. Bracha, D. 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, ACM Press, New York, NY, USA, 2004, pp. 331–344.
[60] S. Costiou, V. Aranega, M. Denker, Reflection as a tool to debug objects, in: International Conference on Software Language Engineering, SLE, 2022, pp. 55–60.
[61] G. Polito, Virtualization Support for Application Runtime Specialization and Extension (Ph.D. thesis), University Lille 1 - Sciences et Technologies - France, 2015.
[62] P. Tesone, Dynamic Software Update for Production and Live Programming Environments (Ph.D. thesis), Université de Lille - IMT Lille Douai, 2018.
[63] J. Kubelka, R. Robbes, A. Bergel, The road to live programming: Insights from the practice, in: Proceedings of the 40th International Conference on Software Engineering, ICSE '18, ACM, New York, NY, USA, 2018, pp. 1090–1101.
[64] G. Kiczales, J. des Rivières, D.G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.
[65] V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Meta-models and infrastructure for smalltalk omnipresent history, in: Smalltalks'2010, 2010.
[66] D. Ungar, R.B. Smith, Self: The power of simplicity, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 227–242.
[67] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, U. Hölzle, Mixins in Strongtalk, in: ECOOP '02 Workshop on Inheritance, 2002.
[68] P. Tesone, G. Polito, L. Fabresse, N. Bouraqadi, S. Ducasse, Instance migration in dynamic software update, in: Meta'16, Amsterdam, Netherlands, 2016.
[69] N. Papoulias, N. Bouraqadi, M. Denker, S. Ducasse, L. Fabresse, Towards structural decomposition of reflection with mirrors, in: Proceedings of International Workshop on Smalltalk Technologies, IWST'11, Edingburgh, United Kingdom, 2011.
[70] N. Papoulias, M. Denker, S. Ducasse, L. Fabresse, Reifying the reflectogram, in: 30th ACM/SIGAPP Symposium on Applied Computing, Salamanca, Spain, 2015.
[71] D.H. Lorenz, J. Vlissides, Pluggable reflection: decoupling meta-interface and implementation, in: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2003, pp. 3–13.