Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Fuel: A Fast General Purpose Object Graph Serializer

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Martín Dias

Director: Hernán Wilkinson
Codirector: Carlos E. Ferro
Buenos Aires, 2012

FUEL: A FAST GENERAL PURPOSE OBJECT GRAPH SERIALIZER

Since objects need to be stored and reloaded on different environments, serializing object graphs is a very important activity. There is a plethora of serialization frameworks with different requirements and design trade-offs. Most of them are based on recursive parsing of the object graphs, which is often a too slow approach. In addition, most of them prioritize a language-agnostic format instead of speed and language-specific object serialization. For the same reason, such serializers usually do not support features like class-shape changes, global references or executing pre and post load actions. Looking for speed, some frameworks are partially implemented at *Virtual Machine* (VM) level, hampering code portability and making them difficult to understand, maintain and extend.

In this work we present Fuel, a general-purpose object serializer based on these principles: (1) speed, through a compact binary format and a pickling algorithm which invests time on serialization for obtaining the best performance on materialization; (2) good object-oriented design, without special help at VM; (3) serialize any object, thus have a full-featured language-specific format.

We implement and validate this approach in Pharo, where we demonstrate that Fuel is faster than other serializers, even those ones with special VM support. The extensibility of Fuel made possible to successfully serialize various objects: classes in Newspeak, debugger stacks, and full CMS object graphs.

Keywords: Object-Oriented Programming, Serialization, Object Graphs, Pickle Format, Smalltalk

AGRADECIMIENTOS

Nuestros méritos son secretos,
aun para nosotros mismos.
Nunca sabremos lo que hemos merecido
porque no hay un jardín
para los buenos versos
ni un fuego eterno para los malos.
–Alejandro Dolina

A la educación pública, que me formó desde las primeras letras. A docentes excelentes, que además de enseñar transmiten otros valores, como la búsqueda de la calidad del conocimiento. A Hernán, a Stéphane, y al INRIA, que posibilitaron que hiciera la pasantía en Lille. A Carlos, que me ayudó con gran dedicación en correcciones y sugerencias, en algunas partes de este trabajo. A Mariano, que colaboró mucho, tanto en la escritura de Fuel, como de este trabajo.

A mi primo Abel, por entusiasmarme con la programación y acercarme a las Ciencias de la Computación. A Nataly, por su amor y comprensión. A mis hermanos, Nicolás, Sol y Juan, por su afecto; a mamá, por todo su cariño, su aliento para que termine la carrera, y las noches y noches esperándome con la cena calentita; a papá, por el apoyo, por enseñarme el valor del trabajo y la curiosidad de inventor inquieto.

CONTENIDOS

1. Introduction 1
   1.1 State of the Art 1
      1.1.1 Base-Level Object Persistence 2
      1.1.2 Meta-Level Object Persistence 4
   1.2 Our Choice 6

2. Objects in Pharo 7
   2.1 Important Concepts 7
      2.1.1 Identity and Equality 7
      2.1.2 Special Identity Management 7
      2.1.3 Hash 8
      2.1.4 Shared Variables 8
   2.2 Kinds of Objects 12
      2.2.1 SmallInteger 12
      2.2.2 Pointer Objects 12
      2.2.3 Variable Bits Objects 13
      2.2.4 Compiled Methods 14
   2.3 Meta-Level 15
      2.3.1 Classes and Metaclasses 15
      2.3.2 Traits 17
      2.3.3 Execution Stack 18

3. Fuel Fundamentals 21
   3.1 Pickle format 21
   3.2 Grouping objects in clusters 22
   3.3 Analysis phase 24
   3.4 Two phases for writing instances and references. 24
   3.5 Iterative graph recreation 24
   3.6 Iterative depth-first traversal 25

4. Serializer Required Concerns and Challenges 27
   4.1 Serializer concerns 27
   4.2 Serializer challenges 28

5. Fuel Features 31
   5.1 Fuel serializer concerns 31
   5.2 Fuel serializer challenges 32
   5.3 Discussion 33

6. Fuel Design and Infrastructure 35

7. Benchmarks . . . 37
7.1 Benchmarks constraints and characteristics . . . 37
7.2 Benchmarks serializing primitive and large objects . . . 38
7.3 ImageSegment results explained . . . 40
7.4 Different graph sizes . . . 41
7.5 Differences while using CogVM . . . 42
7.6 General Benchmarks Conclusions . . . 42

8. Real Cases Using Fuel . . . 45

9. Related work . . . 47

10. Conclusions and Future Work . . . 49

Apéndice 51

A. Resumen en castellano . . . 53
A.1 Introducción . . . 53
A.1.1 Estado del arte . . . 54
A.1.2 Nuestra elección . . . 58
A.2 Objetos en Pharo . . . 59
A.2.1 Conceptos importantes . . . 59
A.2.2 Tipos de objetos . . . 64
A.2.3 Metanivel . . . 67
A.3 Desafíos del Dominio . . . 71
A.3.1 Identidad . . . 71
A.3.2 Flexibilidad . . . 74
A.3.3 Grafos Cíclicos . . . 75
A.3.4 Cambios en el Metanivel . . . 76
A.3.5 Especializaciones . . . 76
A.3.6 Integridad de la representación secuencial . . . 78
A.3.7 Modalidades Alternativas de Lectura . . . 79
A.4 Fuel . . . 79
A.4.1 Algoritmo . . . 79
A.4.2 Diseño . . . 80

1. INTRODUCTION

A running **object environment** has thousands of objects sending messages, mutating their status, being born and dying in the volatile memory of the system. There are several situations where it is required to **capture** the state of certain objects in order to **reproduce** them, either in the original execution environment, or in another one.

Let us introduce into this idea of *capture*. The state of an object is given by its **internal collaborators**. Let's have, for example, an object that represents a bank transfer, whose internal collaborators are a timestamp. an origin bank account, a target bank account, and a sum of money. Then, to capture the state of the transfer, we will also need to capture its internal collaborators, and so on. Thus, to capture the state of an object is necessary to capture a directed graph whose edges are the relations of knowledge, and whose nodes are the objects reachable through those relationships.

To capture and/or persist these states, we use serialization. We define the **serialization** of an object as the generation of a **sequential representation** of the graph of its collaborators. The purpose of serialization is to be able to reproduce the original object at a later stage, from its sequential representation. We call this complementary process either **deserialization** or **materialization**.

The sequential representation above mentioned is usually a **byte string**, which is written in a more persistent memory performance as a file or database.

An optional requirement is that the serialized string has a simple **textual format** that allows humans to be understood and manipulated it. The purpose may be to facilitate the implementation of the serializer and deserializer in any technology, and thus facilitate the exchange of data with a wide range of technologies. The main disadvantage of textual format is that it may be inefficient in space, and therefore in time, when objects are large and complex. In **binary format**, however, it seeks the compactness of serialized chains, giving better performance at the expense of losing the readability by human.

The main purpose of some serializers is sharing objects between **different technologies**, for example in the case of *Web Services*. In such cases, it is central to have a common interchange format to understand all together, even if in the implied abstraction we miss certain features of each technology. It is usual to resort to a simplification of the objects as simple data container units, in pursuit of achieving compatibility between different technologies. Instead, focusing on a **specific technology** can have very different results, and it is the case we are interested in studying in the present work. The knowledge of the specific technology where the object will be reproduced, opens the possibility of capturing and reproducing objects in the environment with more precision and detail.

1.1 State of the Art

In this section, we will review the main usage scenarios for serialization, listing concrete solutions. We have investigated and analyzed them to make an overview of

the state of the art, and to identify problems in the domain of serialization with real solutions.

We decided to not consider in our review the object persistence via mappings to relational databases, because representation in either *SQL* or *tables* is not exactly a byte stream representation, and thus does not contribute to our work.

The main classification of serializers depends on the level to which the persisted objects belong. Usually, a serializer focuses only on base-level objects, or just on the meta-level ones, but not both.

1.1.1 Base-Level Object Persistence

Studying these scenarios, we find two main requirements, which are in tension and therefore determine a compromise solution based on the value assigned to each:

1. Interoperability between heterogeneous technologies. It is often necessary to capture the state of objects in a simplified way, abstracting the concepts they represent in data structures that other technologies can understand and materialize.

2. Efficiency. The larger the size of the graph to be transported, and the greater the volume of transactions per second, the more necessary that the format is compact, and mechanism of encoding and decoding is fast.

Textual Format

They are usually handled when data structures are small, and interoperability is privileged, since it will be relatively easy to implement serialization and materialization in each technology. When choosing a format from the range of possibilities, the popularity in the market is decisive, far more than the good qualities compared. The intuition behind the idea would be that a non popular format make communication difficult between developers to implement, because they will need to learn and study it, and between different technologies, because it may be not implemented in some of them, or to have an implementation with poor maintenance.

XML[1] (*Extensible Markup Language*) is a format designed for the representation of structured documents, but due to having become a *de facto* standard in the industry, it is used as a general purpose format. An example is SOAP[2] that, leaning on his popularity, defines a protocol that is an extension of XML whose objective is the universal remote execution, with wide interoperability between technologies.

JSON[3] (*JavaScript Object Notation*) is a widely used format, imposed by the popularity of *JavaScript* as a language and platform for web applications. The format is very simple and can be easily implemented in any technology, but has an advantage in their original language because the materialization is very efficient and straightforward, because the evaluation of the sequence is done directly on the virtual machine. It has as counterpart the security risk that unexpectedly the sequence could contain arbitrary executable code, so precautions should be taken.

[1] http://en.wikipedia.org/wiki/XML
[2] http://www.w3.org/TR/2007/REC-soap12-part0-20070427/
[3] http://www.json.org

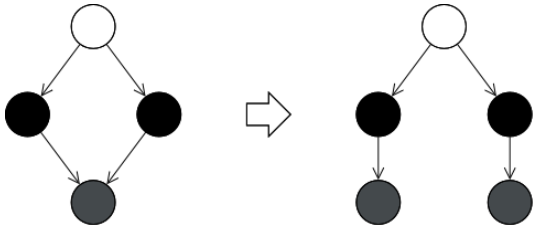

Fig. 1.1: Serializing the object in the upper side (in white), using Object»storeOn:, and then materializing (via Object»readFrom:) the object in the bottom side (in grey), which is recreated by duplicate.

YAML[4] (*YAML Ain't Markup Language*), is similar to *JSON*, but favors interoperability and ease to represent complex data structures. This election comes at the expense of efficiency for serialization and materialization.

The traditional mechanism of *Smalltalk-80* is implemented with the protocol storeOn:/readFrom: that every object can answer, and any class can specialize. The format is textual, consisting of code to be compiled and evaluated in the environment for materialization. So it do not store purely descriptive information about the serialized objects, as most of the formats, but stores a script that tells the system how to recreate them operationally. This solution is simple, but perhaps its biggest disadvantage is that it is not prepared to circle graphs, which cause *stack overflow*. Furthermore, in the case that an object is pointed more than once within the serialized graph, it would be recreated by duplicate, losing their identity in the graph materialized, as illustrated in Figure 1.1.

Binary Format

CORBA[5], *Thrift*[6] y *Protocol Buffers*[7] have their own binary formats, and its strategy to ensure compatibility between technologies is to generate code automatically, both server and client, in a wide variety of languages. This generation is made from a specification of data types and services in a *domain specific language* (*DSL*). In this way they provide a support similar to that of a textual format, but with the efficiency of a binary format.

The purpose of *MessagePack*[8] is to replace *JSON*, but with a fast and compact binary format.

In *Java*, *Serialization API*[9] supports native binary format. The classes of the objects to be serialized must implement the java.io.Serializable interface. It has proved efficiency for distributed computing with *RMI* (*Remote Method Invocation*). However, it is known to be deficient in its adaptation to changes in the metamodel, since it is not prepared to tolerate the addition, deletion or change in the order of the instance variables of a class.

[4] http://yaml.org/
[5] http://www.omg.org/spec/CORBA/3.2/
[6] http://thrift.apache.org/
[7] http://code.google.com/apis/protocolbuffers/
[8] http://msgpack.org/
[9] http://java.sun.com/developer/technicalArticles/Programming/serialization/

In *Ruby*, the native serializer is *Marshal*[10], who, despite of being dedicated to a specific technology, has the disadvantage of not supporting serialization of closures.

In *Python*, *Pickle*[11] is dedicated to the transport of arbitrary objects. It has a version called *cPickle*, implemented in *C*, *i.e.,* outside the object paradigm, which is said to be up to 1000 times faster, while imposing some restrictions on use.

BOSS (*Binary Object Streaming Service*), original also from *Smalltalk-80*, has evolved in the several dialects. It is currently present in *VisualWorks Smalltalk*, *Smalltalk/X*, and *VisualSmalltalk*. It has binary format, and also overcomes the mentioned disadvantages of the readFrom:/storeOn: mechanism about circular graphs and shared objects.

In *Squeak* and *Pharo* dialects, the main binary serializer is *ImageSegment*[12], which has some serious design problems that make it difficult to maintain. In order to achieve good performance, the central part of its mechanism is implemented in the virtual machine. This is an undesirable coupling, which makes it difficult to understand and extend. A few classes implement complex functionality with very extensive methods.

In *Squeak* and *Pharo* we also count with the binary serializers *DataStream*, *ReferenceStream*, and *SmartRefStream*. The first is ready neither for cyclic graphs nor shared objects; the second, solves this problem; the third, addresses an issue that the two previous have: they do not tolerate changes in the instance variables in the classes of serialized object.

There are several serializers whose objective is to exchange objects between the different dialects of *Smalltalk*: *SIXX*[13] is based on the popularity of *XML* to propose a universal format; *StOMP*[14] is an implementation of *MessagePack*, and therefore is fast, binary and compact; *SRP*[15] is also binary, compact, very flexible and a good design.

1.1.2 Meta-Level Object Persistence

Versioning and sharing classes and other meta objects of the environment is a necessity in most object-oriented language. We can say that there are two main approaches, the textual and the binary, although not exclusive. With the textual approach we refer to store the source code of the class and its methods, and then compile it at the time of materialization. In the binary approach, the result of the compilation is serialized, so that the implementation is faster, to avoid compile time.

Persisting the original source code facilitates the collaborative development, because it is easier to compare versions or merge changes. For the distribution to customers this can be inconvenient, for copyright reasons. On the other hand, share compiled binary code is not always desirable or possible. Some technologies compile the source code to an intermediate code, called bytecode, which is a good candidate to be shared as binary because it is hardware independent. However, currently there is a tendency for interpreted languages to do not compile to bytecodes, but directly

[10] http://www.ruby-doc.org/core-1.9.3/Marshal.html
[11] http://docs.python.org/library/pickle.html
[12] http://wiki.squeak.org/squeak/2316
[13] http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html
[14] http://www.squeaksource.com/STOMP.html
[15] http://sourceforge.net/projects/srp/

to machine code. In such cases it does not make sense to share those binaries. In scripting languages like *Ruby* and *JavaScript*, it might be a strategic disadvantage to distribute binaries: it is preferable that each language implementation is free to solve their own way.

Here we discuss some real examples:

In *Java*, the compiler generates a class file from source code, which then a ClassLoader is able to materialize. The source code is attached as text files. The packet distribution is done in compressed files, including the class files and optionally java files. This enables the efficient distribution of packets in binary files, and share source code for development.

In *JavaScript*, the code is shared in textual form, even though in its main use case (*web applications*), this implies some significant scalability restrictions: an application with too much code slows down the loading of the web pages.

Ruby is a scripting language that has different implementations, and they do not have an unique one to share binary code. Although some implementations compile to bytecodes, like *Rubinius*, *YAML*, and *SmallRuby* and *MagLev*. It uses a scripting textual format for persistence.

In *Python*, *Marshal*[16] is the native mechanism for binary persistence of compiled code. It is put in files with extension pyc, whose content is evaluated at materialization time.

In *Oberon*, researchers have experimented with *slim binaries* [10], based on the idea of persisting *abstract syntax tree* (*AST*), an intermediate product of compilation.

Within the world of *Smalltalk* there is much to list, but we will highlight some formats and serializers.

In the various implementations of *Smalltalk* there are *FileOut* variants. It is a format for sharing text-based scripting code, where the materialization is done with the compiler evaluating the contents of the serialized file. The format consists of a series of scripts separated by a ! character. In *Squeak* and *Pharo* there are very popular code versioning tools using this format, like *ChangeSet*[17] and *Monticello*[18].

Parcels [18] is the main tool for code versioning in *VisualWorks Smalltalk*. It uses binary format files to persist methods with their bytecode, optionally attaching source files in *XML* format. In this way, it allows the usage scenarios mentioned in the section of *Java*: efficient distribution of packages in binary files, and share source code for development.

SMIX[19] intended to use *XML* format for sharing code between dialects, but was never adopted by the community. *Cypress*[20] pursues the same objective, using a textual format, but instead of structuring the classes within a single file, it takes advantage of the directory structure of *GIT* repositories, a popular *source content management* tool.

[16] http://docs.python.org/release/2.5/lib/module-marshal.html
[17] http://wiki.squeak.org/squeak/674
[18] http://wiki.squeak.org/squeak/1287
[19] http://www.mars.dti.ne.jp/~umejava/smalltalk/smix/
[20] https://github.com/CampSmalltalk/Cypress

1.2 Our Choice

Smalltalk environments differ from other object-oriented technologies with classification in that either classes, or methods, or execution contexts, are represented as ordinary objects within the environment. This enables us to perform experiments in the domain of serialization that would be unthinkable in other technologies.

One of the most promising environments today is *Pharo*. It emerged in 2008 as an branch of *Squeak*. It is developed with an *open-source* license, and has a very active community of contributors.

In the present work, we intend to describe the domain of Serialization, based on the study of their problems and on acquired knowledge, develop a new serialization framework in *Pharo*, with the following objectives:

Flexibility. In an environment of pure objects, where classes, methods, closures and execution contexts are represented as ordinary objects, a single serializer could be used for the persistence of any object, whether from the base-level or from the meta-level. We want to build a new multipurpose framework with the capacity to be adapted to very different use cases.

Concreteness. In order to achieve a rich and precise object recreation, we will focus on a specific technology rather than exchanging with other technologies. We intend to persist any object of the chosen technology, whether from the base-level or the meta-level. Accordingly, it will be necessary to carefully study our technology implementation details.

Efficiency. The serializer should be useful in real use cases, where performance is essential. Therefore we will give relevance to make an efficient implementation, and we will make a comparative evaluation that allows us to arrive at solid conclusions.

Design. The developed software must meet the criteria of good design of objects, and will be implemented entirely within the object-oriented paradigm, without resorting to modifications to the virtual machine. In addition, the functionality will be supported by a good battery of test cases.

We understand that none of the tools we reviewed (most of which we mentioned in the State of the Art section) satisfactorily meets all these objectives.

2. OBJECTS IN PHARO

Since serialization is trying to capture the state of an object and then recreate it, and since we want to be able to do so with complete detail and with any object in the environment, it is worth then starting to divide which types of objects exist in the technology we have chosen and what modalities or problems they present to a serializer.

We have divided this chapter into three parts. In the first part, we introduce and develop concepts and important properties of the environment; in the second part, we explain the object types available in this technology; in the third part, we investigate the meta-level objects, which have greater complexity.

2.1 Important Concepts

2.1.1 Identity and Equality

Two fundamental relations of both the paradigm and the domain under study, are the identity and the equality. Both are implemented as messages in *Smalltalk*. The **identity** (==) occurs when the argument and the receiver of the message are the same object. The **equality** (=) occurs when the argument and the receiver of the message represent the same concept. Equality should be redefined in subclasses, while the identity is implemented as a special message, which can not be redefined.

2.1.2 Special Identity Management

Some objects conceptually represent unique entities of the environment. Except for instances of SmallInteger, which will be discussed later in this chapter, this property is implemented in *Pharo* in a pure manner, within the *Object Paradigm*. This means that, when recreating serialized objects, the materialization process should be careful to not duplicate the instances that should be unique, *i.e.,* it should be careful that the property is not violated. Next, we list concrete well known cases in *Pharo*.

An elementary case is nil, who is the only instance of UndefinedObject. Another case is true and false, which are single instances of True and False, respectively. This uniqueness property can be lost using the message basicNew, which we discuss later in this chapter, which is used to instantiate objects in primitive way. It is enough to show that the following expressions are false:

```
UndefinedObject basicNew == nil
True basicNew == true
```

As noted in Figure 2.1, an instance of Character has a numeric code as a collaborator, indicating its *Unicode* value. The characters corresponding to the first 256 codes are unique in the environment, held in a dictionary by the Character class. However, other characters do allow duplicates, and therefore, the first line is true while the second is false:

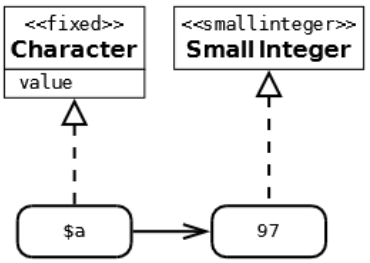

Fig. 2.1: The class Character keeps an unique instance representing the letter a.

```
(Character value: 97) == (Character value: 97)
(Character value: 257) == (Character value: 257)
```

An instance of Symbol is a unique string in the environment, so it is true that:

```
'unSímbolo' asSymbol == 'unSímbolo' asSymbol
```

To implement this, the class Symbol maintains a set of *weak* instances. We talk about weak references later, in this chapter.

2.1.3 Hash

The **hash** (hash) of an object is an integer, which is related to what such object conceptually represents, defined so that any pair of equal objects necessarily have the same hash value. The **identity hash** (identityHash) of an object, however, is an integer related to the identity of that object, which remains constant throughout its existence. Unless it has been redefined, the hash of an object is the same as its identityHash.

Preferably, these integers to be calculated very fast and well distributed, since they are used, for example, as an index to locate items in the HashedCollection. In Figure 2.2 we show the hierarchy of these classes. Serializing any of these collections with maximum detail, and then recreate it accurately, presents a difficulty: we must assume that the indices are outdated to materialize, as they had been calculated based on the hash or identityHash. Prepared for this situation, these collections implement the rehash message, which is responsible to update them with new values.

2.1.4 Shared Variables

With the idea that certain objects must be accessed easily from the methods of the environment through names, there are several types of **shared variables**, each with a different scope. **Global variables** are accessible from any method; **class variables**, from the methods of the class that declares the variable, or from its subclasses; the **pool variables**, from the methods of the classes that declared to use the shared pool that contains the variable.

Shared variables are represented in **associations** which form part of the **shared dictionaries**. The name of the variable (by convention, a capitalized symbol) is the **key** (key) of the association, while the **value** (value) is the object pointed by the variable. When the compiler finds the name of a shared variable in a method,

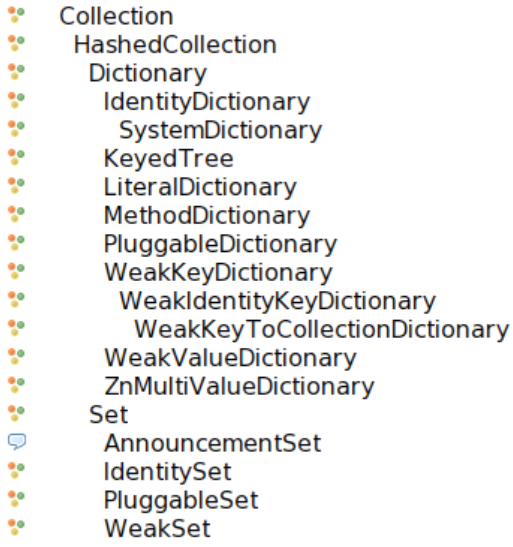

Fig. 2.2: Hierarchy of HashedCollection

the corresponding association is collected among the literals. Thus, associations are referenced from both the shared dictionaries and the methods linked to them.

In Figure 2.3, we illustrate the full range of shared variables, showing the method tabWidth of the class TextStyle, which declares TextConstants as shared pool, as shown in the bottom. There, we also show that TextConstants is a subclass of SharedPool. The method answers the value of DefaultTab, which is a variable shared by TextConstants. Because of this association is shared, if the value of DefaultTab is updated to point to another object, then this change would apply to all methods that use it.

The **system dictionary** contains the associations that represent the global variables of the environment. In *Pharo*, it can be accessed via the Smalltalk globals expression. Each class which is recorded in the environment has a global variable defined in this dictionary. At the top of Figure 2.3, we emphasize two global associations: those of the classes TextConstants and TextStyle. In addition to classes, other unique and critical objects of the system are registered as global variables, like Smalltalk, Transcript, Processor, Sensor, World, ActiveHand, Undeclared, and SystemOrganization. User defined global variables can be added, although it is considered bad practice in *Pharo*[1], recommending the use of class variables, when possible.

The **shared pool of a class** is a dictionary whose associations represent the class variables that are shared in the whole hierarchy of subclasses. In Figure 2.3, DefaultTab is a *class variable* of TextConstants, and its value is assigned in the class method initialize, a common practice with this kind of variables.

A **shared pool** in *Pharo* is a subclass of SharedPool whose associations represent shared variables of such pool. In Figure 2.3, DefaultTab is a variable that TextConstants shares, and that the tabWidth method uses. Neither is it considered a good practice to use shared pools in *Pharo* since, although its scope is more restricted than that

[1] Pharo by Example [4]: "Current practice is to strictly limit the use of global variables; it is usually better to use class instance variables or class variables, and to provide class methods to access them."

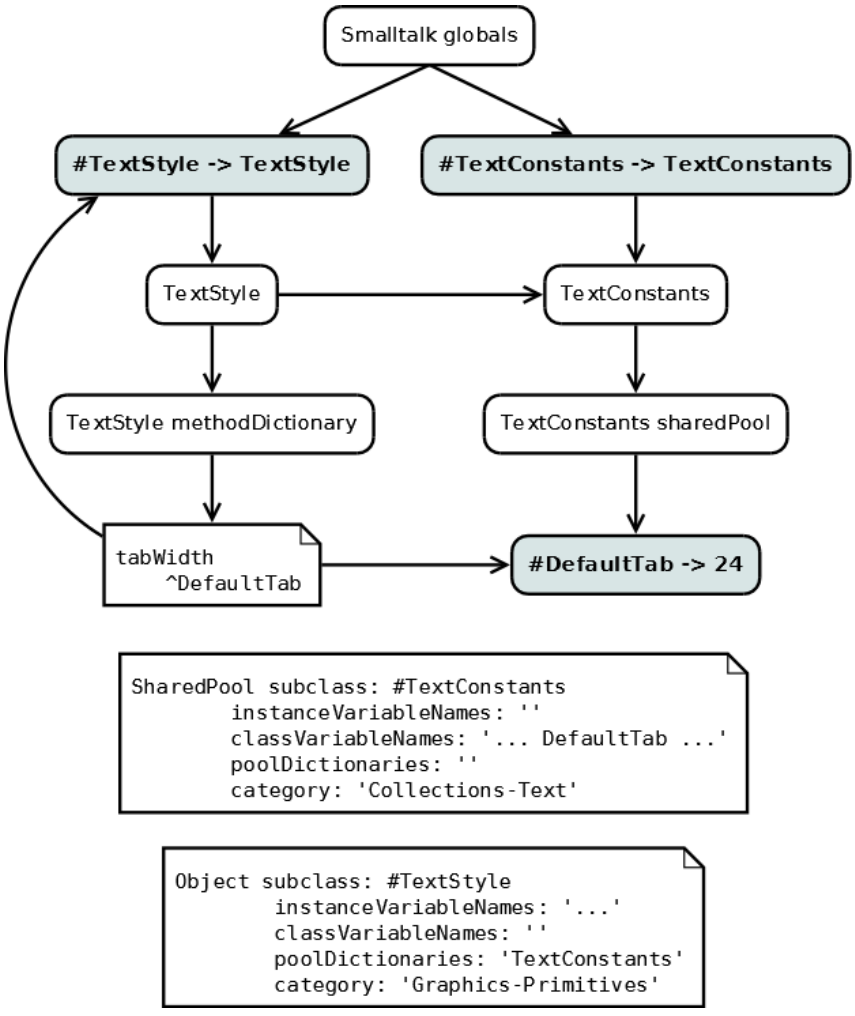

Fig. 2.3: Shared variables

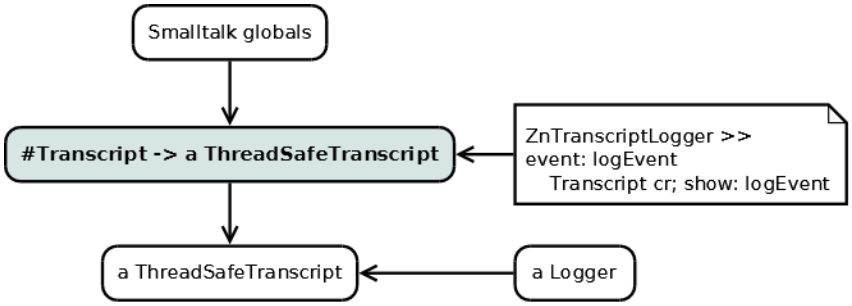

Fig. 2.4: Shared variables and global value

of global variables, it is still preferable to use class variables (or a class' instance variable).

For the purposes of serialization, it must be borne in mind that, for this shared variable mechanism, it is essential the identity of the associations. We can think of different usage scenarios in which this fact has different consequences. Suppose that a binary code version tool loads into the environment the method tabWidth. If TextConstants exists and defines the variable DefaultTab, then the same association should be used as literal in materialization. An exact replica of tabWidth works the same way as the original as long as associations are not copies. But if TextConstants does not define DefaultTab, it could be more appropriate to define ourselves with some default value. In another scenario, it could be that TextConstants is absent, and perhaps it would be appropriate to give the user an error indicating the required dependency.

Another example is the system global variables mentioned above (Transcript, Processor, etc.), which we can assume that any *Pharo* environment will define. In Figure 2.4, the shared association Transcript acts as literal of the method event: of ZnTranscriptLoader class. Regardless of that fact, a hypothetical object called aLogger knows the value of the association, *i.e.,* the unique-instance in the environment of ThreadSafeTranscript. It is clear that in the case of being materializing the method event:, the used association must be the one currently existing in the environment, and never a copy of it.

In Figure 2.4, we incorporate another case: aLogger is an object that has Transcript as a collaborator. Let's imagine two serialization scenarios for this object. In the first one, probably the most typical, we are interested in recreating aLogger state, but we want the Transcript to be treated as a global, such that when materialized, aLogger has the unique instance of Transcript as collaborator, and in that way the logger will continue working normally, displaying information on the screen. In the second scenario, in contrary, there was a system critical error and, therefore, we want to serialize the Transcript in full detail, to be able to analyze later what happened, carefully reproducing the last system state.

In summary, we have shown by examples that in some cases, but not always, a serializer should be able to detect in the graph either shared associations or them values, such that later, when materializing them, they must not be recreated in detail but, instead, they should embody existing shared objects in the environment.

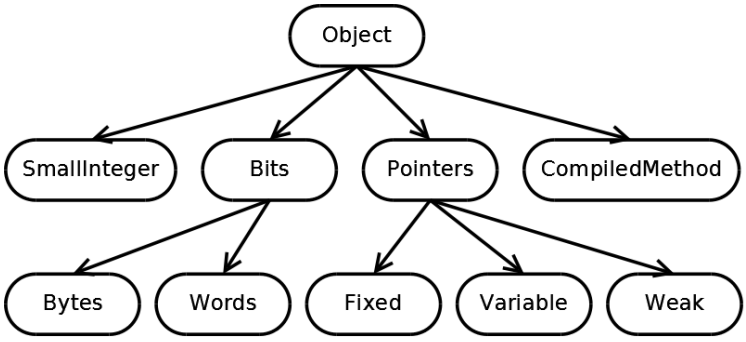

Fig. 2.5: Kinds of object

2.2 Kinds of Objects

We present in this section a taxonomy of objects given from the implementation of the virtual machine. We summarize it in Figure 2.5, as an introduction, and we explain it below.

2.2.1 SmallInteger

The integers between -1073741824 and 1073741823 have a special representation: each of these integers is encoded within the 32 bits of a regular object pointer. This means that these integers do not really occupy additional space in memory, just like any other object.

This special treatment by the virtual machine has a consequence in which we are interested for the matters of serialization: for these integers, it is impossible to distiguish between the concepts of equality and identity. There is no way to build a SmallInteger representing the number 5 which is non-identical to another SmallInteger also representing the number 5. In *Pharo* there are no other objects that have this property: even basic objects such as nil, true and false are allowed to be duplicated, in violation of the property of being unique instances in the environment.

2.2.2 Pointer Objects

They are objects that can reference other objects, also called internal collaborators, through **instance variables**. These variables can be **named** or **indexed**.

The named variables can not be more than 254, and are defined at the class level. Therefore, all instances of a class necessarily have the same number of named variables. **Fixed pointers** objects only have variables of this kind. Capturing the state of one of these objects usually requires capturing the state of the contributors it points. These variables can be accessed using the introspection protocol for reading (instVarAt:) and for writing (instVarAt:put:), available in the Object class. For materialization, these objects can be instantiated by sending the basicNew message to their class.

We call **variable pointers** objects to those which ones, besides of named variables, have indexed variables. While an object's named variables are defined in its class, the number of indexed variables is defined at the instantiation time. For this,

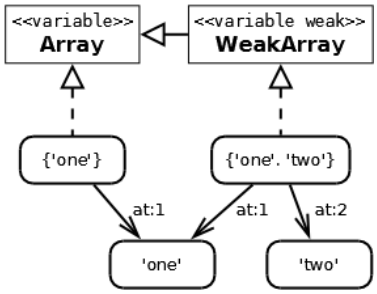

Fig. 2.6: Variable pointers objects

we have basicNew: message, which we will find useful for materializing this kind of objects. With basicSize we can get the number of indexed variables of an instance.

Foreseeably, to serialize this kind of objects, we extend what was said for fixed pointers, now also capturing the state of the collaborators pointed by their indexed variables. Object class provides specific introspection protocol to access indexed variables: basicAt: and basicAt:put:.

Within the kind of variable pointers objects, there are **weak variable pointers** objects, which differ in that their indexed variables are **weak references**, in the sense that they do not prevent its collaboratos from being collected by the *garbage collector*. In this case, the indexed variables pointing to collected collaborators, change automatically to point to nil. Beyond implementatitive details, the semantic is that weak variable objects know their collaborators just as they exist, but without preventing them from ceasing to exist.

The introspection protocol for these objects is the same as for the variable pointers ones, *i.e.,* they are polymorphic, in this sense. If we think about how to serialize, is not as obvious as in the previous cases how to capture their state. One possibility is to do it as if it was an ordinary variable pointers object. Although, it may be more appropriate that, for each indexed variable, we capture a pointer to nil (as if it had been collected), unless the weak collaborator is pointed in a non-weak way by another object in the graph. To illustrate this last idea, we look at Figure 2.6, where on the right side, we show an instance of WeakArray pointing to 'one' and 'two'. If we serialize this instance, we will do it like if it was {nil. nil}. In contrast, if we were also serializing another object which also includes in its graph a non-weak object, like the instance of Array (which points to 'one'), then we would capture the weak array as {'one'. nil}.

2.2.3 Variable Bits Objects

These objects represent sequences of simple objects, whose status can be encoded in a few bits: **variable bytes** objects in 8 bits; **variable words**, in 32 bits. The purpose of their existence is to save memory space and to optimize bulk operations. For example, ByteString instances are sequences of 8-bit characters, and ColorArray instances are sequences of 32-bit color depth. This kind of objects can not have named variables.

Despite their special representation, these objects are polymorphic with the variable pointers ones, except in that they restrict the types of objects allowed to be

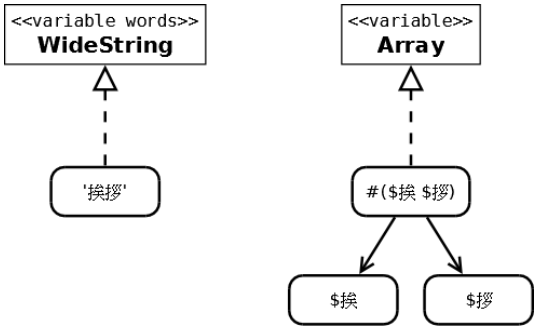

Fig. 2.7: A WideString saves memory space, representing the same sequence of *Unicode* characters as the Array, but without pointing to real instances of Character.

assigned as collaborators. Figure 2.7 shows the difference between a WideString and an array representing the same sequence of *Unicode* characters.

For serialization, it will be beneficial to us to take advantage of read and write methods already optimized for this kind of objects, as virtual machine level primitives. For variable words objects, we have to be careful because *Pharo* performs these primitive operations with the current system *endianness*[2], and therefore it is serializer responsibility to fix eventual incompatibilities. For example, if a Bitmap is serialized in a little-endian computer and it is materialized in a big-endian computer, then some *endianness* correction must be done.

2.2.4 Compiled Methods

A **compiled method** (CompiledMethod) is a byte variable object that, however, knows other objects as if it were a pointer: their **literals**. In Figure 2.8, we illustrate the method and: in True class, whose literals are its selector and the shared association that tells to which class it belongs. The virtual machine interprets the byte sequence in a method like this:

- header (4 bytes). It encodes data such as the number of literals, the number of temporary variables, the number of arguments, or if it is a primitive method.
- literals (4 bytes each). The virtual machine interprets them as pointers to objects.
- bytecodes (variable). They are usually generated by the compiler, and interpreted by the virtual machine at runtime.
- trailer (variable). It is used to encode information about the source code of the method. Usually indicates in which file and at what position is the source code.

[2] Endianness is the order of the bytes in the word. Big-endian systems encode first its most significant byte, while the little-endian encode the least significant at first.

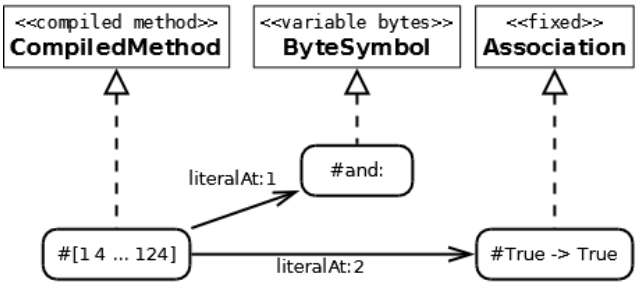

Fig. 2.8: A compiled method is a variable bytes object which, nevertheless, points to other objects: its literals.

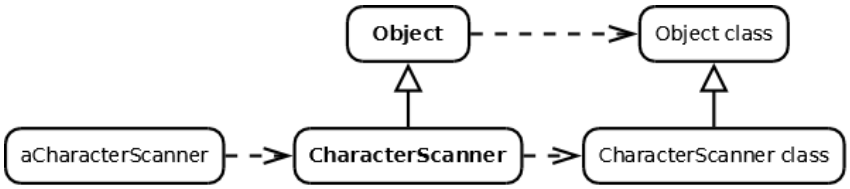

Fig. 2.9: Every object is an instance of a class, which is also an object. The arrow of dotted lines means "is instance of" while the other arrow means "is subclass of".

2.3 Meta-Level

2.3.1 Classes and Metaclasses

In *Smalltalk*, every object is an instance of a class, which is also an object. We illustrate these statements in Figure 2.9, CharacterScanner is subclass of Object, and instance of its metaclass.

In Figure 2.10, we show the complexity of a class in *Pharo*. It is CharacterScanner class together with its collaborators. This does not mean that it is an object that can not be serialized and materialized as any other. Let's consider the scenario in which we persist in detail this class and its metaclass, but without persisting in detail the other shared objects in the environment.

The currently available mechanism in *Pharo* for serialization and materialization, is to fileOut and to fileIn, discussed in the previous chapter. The first, writes in a file a textual definition of both class and metaclass of CharacterScanner, as well as the source code of their methods. The second, reads the file, running the class definitions, and compiling the method's source code. An instance of ClassBuilder is responsible for validating the class definition, building the class, installing it in the *system dictionary*, and sending relevant *system notifications* about the operation. Then the building process takes care of interpreting the source code, compiling it to bytecodes, finding the corresponding literals, building instances of CompiledMethod, and installing them on the newly installed CharacterScanner class.

However, our intention is different from that mechanism: we want to be able to realize a class (if the necessary conditions are met), without using the ClassBuilder or compiler, but do it carefully reproducing the class, recreating the graph collaborators. The alluded conditions relate to the environment where it is intended to materialize, which should have the shared objects that the serialized graph requires, and particu-

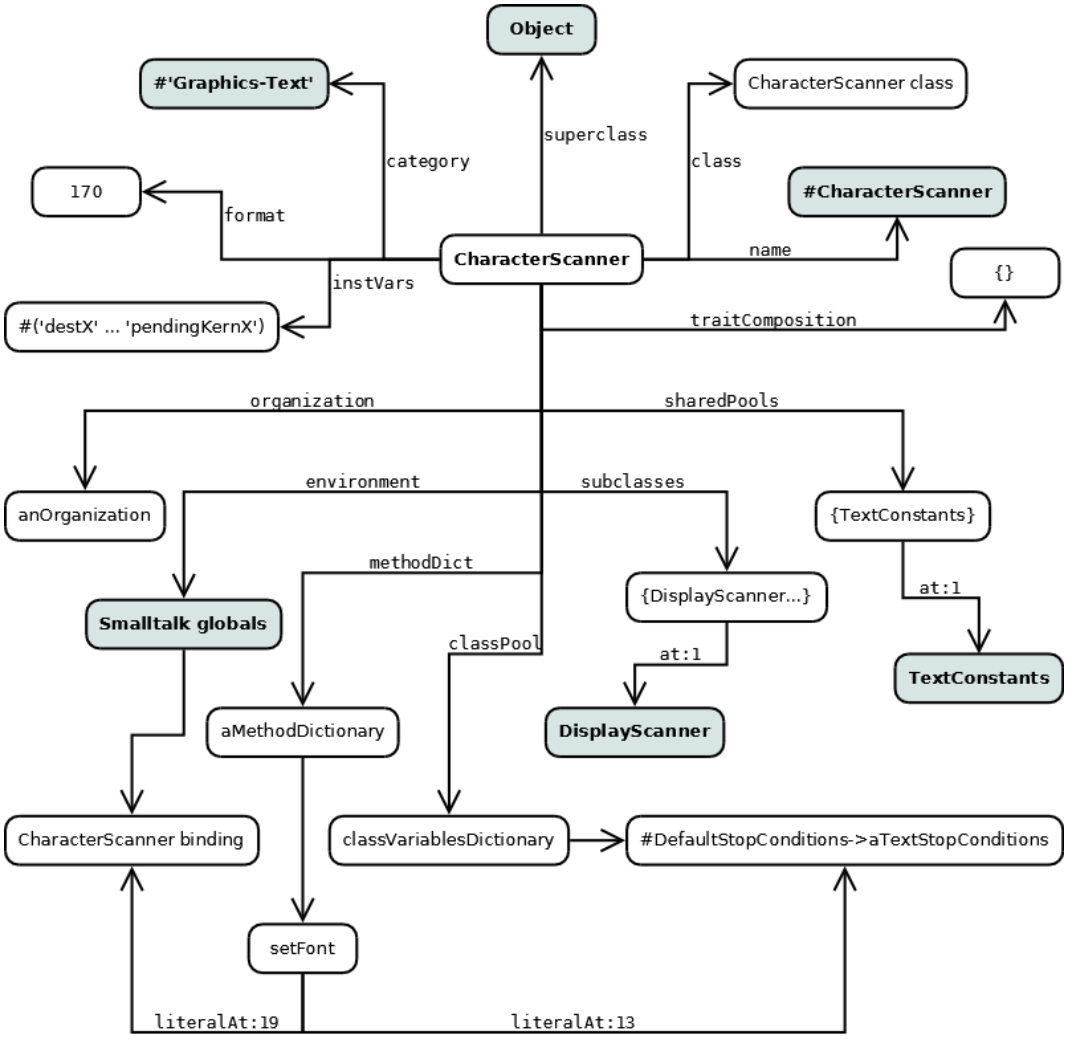

Fig. 2.10: CharacterScanner class, with his collaborators. To serialize it, we may consider the highlighted objects as shared, and thus do not capute its state in detail.

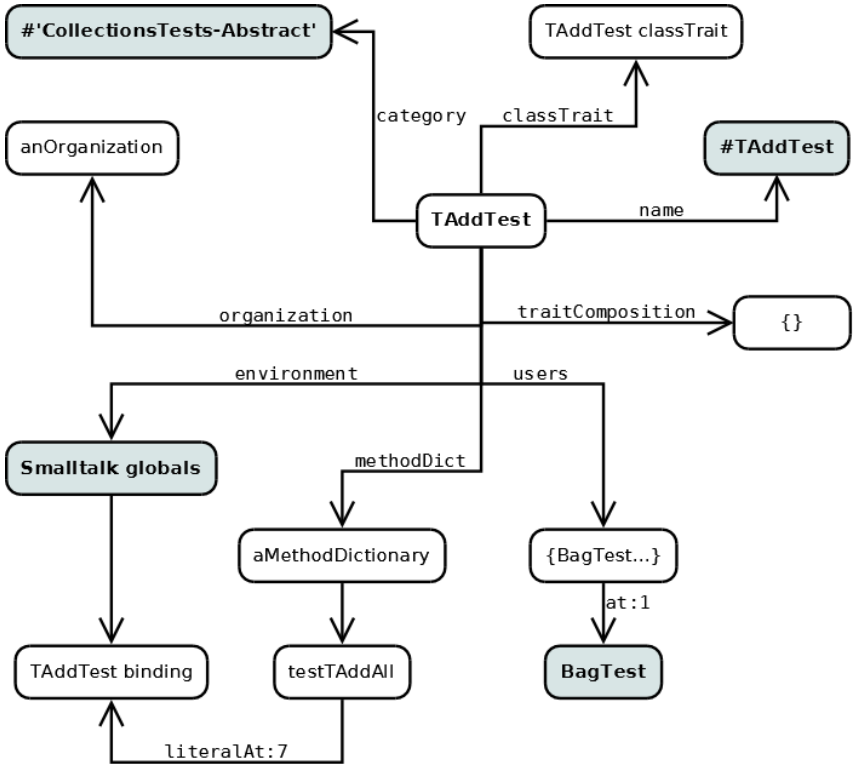

Fig. 2.11: The trait TAddTest, with its collaborators. To serialize it, we may consider the highlighted objects as shared.

larly, the classes should not have changed its format and its list of variables, as these facts could invalidate the original compiled methods' bytecodes.

2.3.2 Traits

A **trait** is a collection of methods that can be included in the behavior of a class without using inheritance. In other words, they enable classes to share methods, transversally to their hierarchies. In Figure 2.11 we observe TAddTest, a trait containing tests to prove functionality of adding items to a collection.

A class can use not only a trait, but a composition of them. In Figure 2.12 we see the example of BagTest, whose definition specifies a trait composition expression, which is constructed by special operations (+ and -, and @, but not in this example). Such composition is represented by an object structure that follows the composite design pattern.

However, the difficulty to serialize a trait composition is centered on the complexity of traits, which is similar to serialize classes. The fileIn/fileOut mechanism works with traits in a very similar fashion as with classes. The same difference of purpose we have here: we want to be able to materialize a trait without using neither the ClassBuilder nor the compiler. We want to carefully recreate the graph collaborators.

```
CollectionRootTest subclass: #BagTest
 uses: TAddTest + TIncludesWithIdentityCheckTest + TCloneTest + TCopyTest +
  TSetArithmetic + TConvertTest + TAsStringCommaAndDelimiterTest +
  TRemoveForMultiplenessTest + TPrintTest + TConvertAsSortedTest +
  TConvertAsSetForMultiplinessTest + TConcatenationTest +
  TStructuralEqualityTest + TCreationWithTest - #testOfSize +
  TOccurrencesForMultiplinessTest
 instanceVariableNames: '...'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'CollectionsTests-Unordered'
```

Fig. 2.12: An example of class definition with a traits composition expression.

2.3.3 Execution Stack

An **execution context** (MethodContext) represents a running state, in two situations: in the method activation (CompiledMethod) after a message send, or the activation of a block closure (BlockClosure) on its evaluation. It is a *variable pointers* object, with some important peculiarities. First, let's describe what the variables represent:

- receiver is the object that embodies the *pseudo-variable*[3] self during the execution.
- method is which owns the *bytecodes* in execution. Depending on the case, it is the activated method, or the method that defines the activated block closure.
- pc is the index, in the method, of the next *bytecode* to execute.
- stackp is the position of the top in the execution stack.
- closureOrNil, depending on the case, could be either the activated block, or nil.
- The *indexed variables* contain the context arguments and the temporary variables.

The size in which MethodContext instances are created is restricted to two values: 16 or 56 indexed variables. This number corresponds to the method framesize, that depends on how many arguments and temporary variables the context needs, and it is encoded in the compiled method's header. The virtual machine hides to the environment the true size of MethodContext instances, which varies along the execution, according to stackp.

In Figure 2.13 we show a a snapshot of a fragment of the execution stack, when evaluating with *"do it"* in a *Workspace* the following expression:

```
#(3 4 5) select: [:n| n > 4]
```

[3] The *pseudo-variables* in Pharo are self, super, thisContext, nil, true, and false.

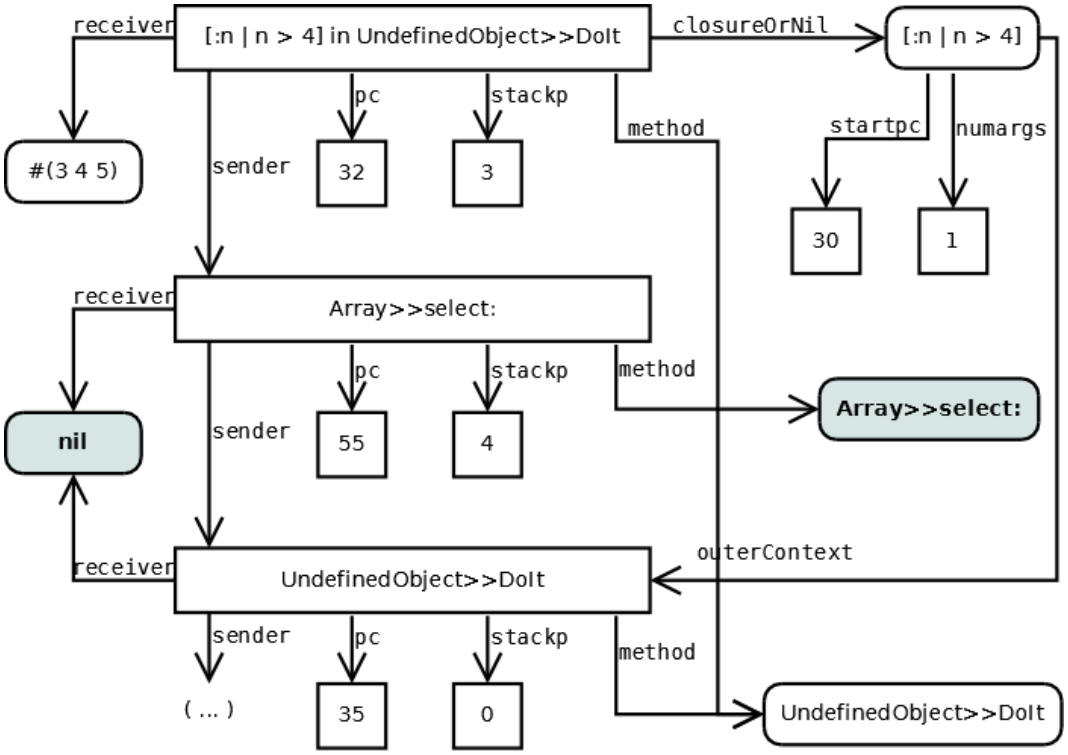

Fig. 2.13: Evaluating an expression in a *Workspace*. The highlighted objects, that is, nil and Array»select: method, could be considered as shared, but not so the UndefinedObject»DoIt method, because it has been temporarily compiled in that class by the *"do it"* command, which will remove it at the end of execution.

The mentioned command temporarily compiles the expression in the UndefinedObject class, under the DoIt selector, and removes it when the execution is completed. For serialization, we could consider shared objects both nil and the method Array»select:, but not so UndefinedObject»DoIt method, since we should assume that it will not be present at materialization time. Consequently, when serializing an execution stack, this method must be included in the graph with full detail. Moreover, if Array»select: were coded as a shared object, we run the risk that when we materialize it has changed, and thus the bytecodes pointed by pc is obsolete. This could cause serious problems in the environment, therefore precautions should be taken, like including a verification code for bytecodes.

3. FUEL FUNDAMENTALS

In this section we explain the most important characteristics of *Fuel* implementation that make a difference with traditional serializers.

3.1 Pickle format

Pickle formats are efficient formats to support *transport*, *marshalling* or *serialization* of objects [24]. Riggs defines: "*Pickling* is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. *Unpickling* is the complementary process of recreating objects from the serialized representation." (extracted from [24])

Fuel's pickle format works this way: during serialization, it first performs an analysis phase, which is a first traversal of the graph. During such traversal, each object is associated to a specific *cluster*. Then *Fuel* first writes the instances (vertexes in the object graph) and after that, the references (edges). While materializing, *Fuel* first materializes the instances. Since all the objects of a cluster have the same type, *Fuel* stores and reads that information in the stream only once. The materialization can be done in a bulk way (it can just iterate and instantiate the objects). Finally, *Fuel* iterates and sets the references for each of the materialized objects.

Even if the main goal of *Fuel* is materialization speed, the benchmarks of Section 7 show we also have almost the best speed on serialization too.

Serializing a rectangle. To present the pickling format and algorithm in an intuitive way, we show below an example of how *Fuel* stores a rectangle.

In the following snippet, we create a rectangle with two points that define the origin and the corner. A rectangle is created and then passed to the serializer as an argument. In this case, the rectangle is the *root* of the graph which also includes the points that the rectangle references. The first step analyzes the graph starting from the root. Objects are mapped to *clusters* following some criteria. For this example we only use the criterium *by class*. In reality *Fuel* defines a set of other clusters such as *global objects* (it is at Smalltalk dictionary) or small integer range (*i.e.,* an integer is between 0 and $2^{32} - 1$) or *key literals* (nil true or false), etc.

```
| aRectangle anOrigin aCorner |
anOrigin := Point x: 10 y: 20.
aCorner := Point x: 30 y: 40.
aRectangle := Rectangle origin: anOrigin corner: aCorner.
FLSerializer newDefault serialize: aRectangle on: aFileStream.
```

Figure 3.1 illustrates how the rectangle is stored in the stream. The graph is encoded in four main sections: header, vertexes, edges and trailer. The Vertexes section

collects the instances of the graph. The Edges section contains indexes to recreate the references between the instances. The trailer encodes the root: a reference to the rectangle.

At load time, the serializer processes all the clusters: it creates instances of rectangles, points, small integers in a batch way and then set the references between the created objects.

3.2 Grouping objects in clusters

Typically, serializers do not group objects. Thus, each object has to encode its type at serialization and decode it at deserialization. This is an overhead in time and space. In addition, to recreate each instance the serializer may need to fetch the instance's class.

The purpose of grouping similar objects is not only to reduce the overhead on the byte representation that is necessary to encode the *type* of the objects, but more importantly because the materialization can be done *iteratively*. The idea is that the type is encoded and decoded only once for all the objects of that type. Moreover, if recreation is needed, the operations can be grouped.

The type of an object is sometimes directly mapped to its class but the relation is not always one to one. For example, if the object being serialized is Transcript, the type that will be assigned is the one that represents global objects. For speed reasons, we distinguish between positive SmallInteger and negative one. From *Fuel*'s perspective, they are from different types.

In *Fuel* we have a class hierarchy of Clusters. Each one knows how to encode and decode the objects they group. Here are some examples: PositiveSmallIntegerCluster groups positive instances of SmallInteger; NegativeSmallIntegerCluster groups negative instances of SmallInteger; FloatCluster groups Float instances. FixedObjectCluster is the cluster for regular classes with indexable instance variables that do not require any special serialization or materialization. One instance of this cluster is created for each class.

In Figure 3.1, there is one instance of PositiveSmallIntegerCluster and two instances of FixedObjectCluster, one for each class (Rectangle and Point). Such clusters will contain all the respective instances of the classes they represent.

Clusters decide not only *what* is encoded and decoded but also *how*. For example, FixedObjectCluster writes into the stream a reference to the class whose instances it groups and, then, it writes the instance variable names. In contrast, FloatCluster, PositiveSmallIntegerCluster or NegativeSmallIntegerCluster do not store such information because it is implicit in the cluster implementation.

In Figure 3.1, for small integers, the cluster directly writes the numbers 10, 20, 30 and 40 in the *Vertexes* part of the stream. However, the clusters for Rectangle and Point do not write the objects in the stream. This is because such objects are no more than just references to other objects. Hence, only their references are written in the *Edges* part. In contrast, there are objects that contain self contained state, *i.e.,* objects that do not have references to other objects. Examples are Float, SmallInteger, String, ByteArray, LargePositiveInteger, etc. In those cases, the cluster associated to them have to write those values in the *Vertexes* part of the stream.

The way to specify custom serialization or materialization of objects is by creating

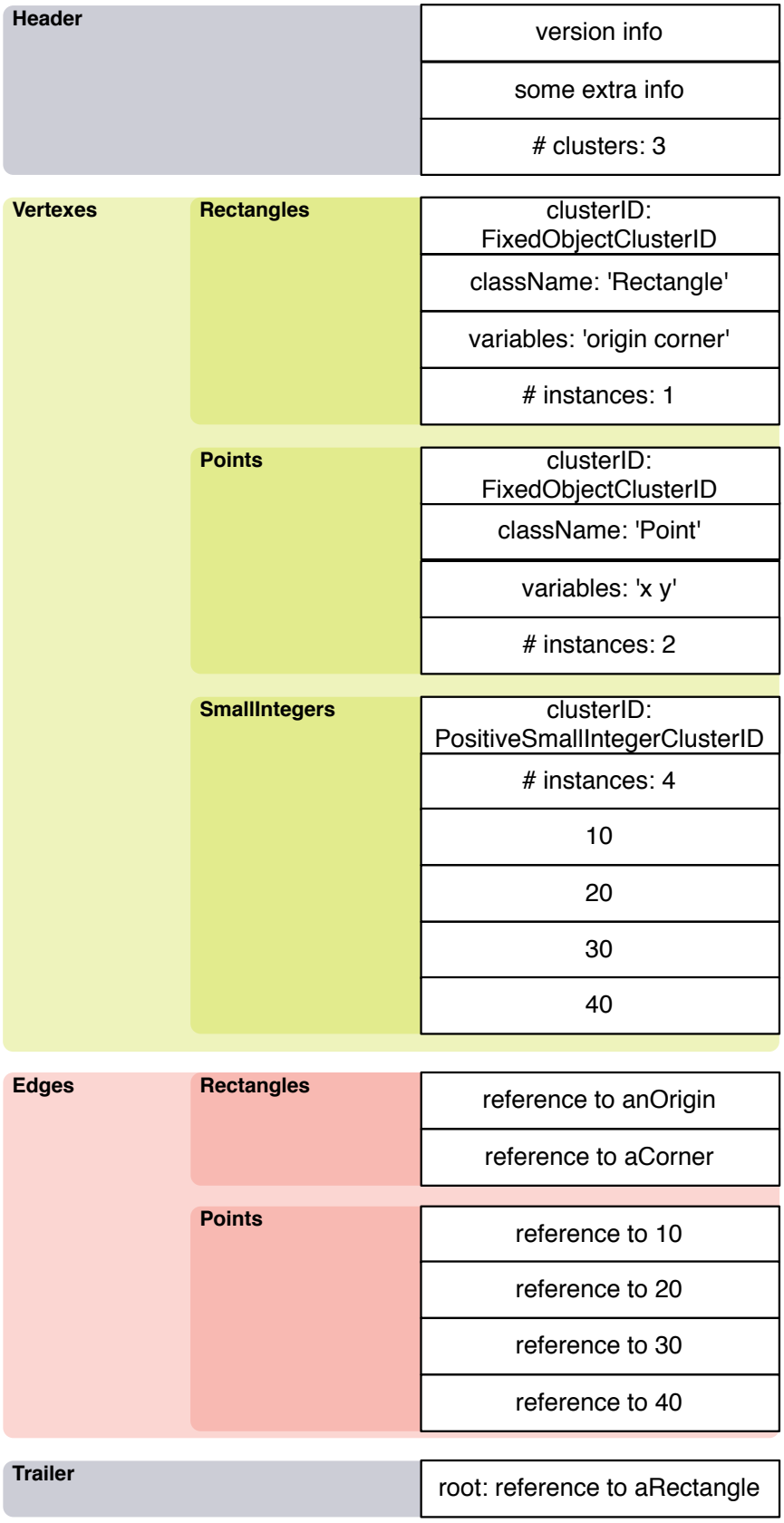

Fig. 3.1: A graph example encoded with the pickle format.

specific clusters.

3.3 Analysis phase

The common approach to serialize a graph is to traverse it and, while doing so, to encode the objects into a stream. Since *Fuel* groups similar objects in clusters, it needs to traverse the graph and associate each object to its correct cluster. As explained, that fact significantly improves the materialization performance. Hence, *Fuel* does not have one single phase of traversing and writing, but instead two phases: analysis and writing. The analysis phase has several responsibilities:

- It takes care of traversing the object graph and it associates each object to its cluster. Each cluster has a corresponding list of objects which are added there while they are analyzed.
- It checks whether an object has already been analyzed or not. *Fuel* supports cycles (an object is only written once even if it is referenced from several objects in the graph).
- It gives support for global objects, *i.e.,* objects which are considered global are not written into the stream. Instead the serializer stores the minimal needed information to get the reference back at materialization time. Consider as an example the objects that are in Smalltalk globals. If there are objects in the graph referencing *e.g.,* the instance Transcript, we do not want to serialize that instance. Instead, we just store its global name to get the reference back during materialization. The same happens with the Smalltalk class pools.

Once the analysis phase is over, the writing follows: it iterates over the clusters and, for each it writes its objects.

3.4 Two phases for writing instances and references.

The encoding of objects is divided in two parts: (1) instances writing and (2) references writing. The first phase includes just the minimal information needed to recreate the instances *i.e.,* the vertexes of the graph. The second phase has the information to recreate references that connect the instances *i.e.,* the edges of the graph.

Notice that these two phases are mandatory to be able to perform the bulk materialization. If this division does not exist, the serializer cannot do a bulk materialization because to materialize an object it needs to materialize its instance variables, which of course can be of a different type.

In the materialization, there are two phases as well, the first one for materializing the instances and the second one to set the references between the materialized objects.

3.5 Iterative graph recreation

This is the most important characteristic of *Fuel*'s pickle format. Other characteristics such as the analysis phase, grouping instances in clusters, and having two phases

for serializing/materializing instances and references, are all necessary to achieve iterative graph recreation.

During *Fuel* serialization, when a cluster is serialized, the amount of objects of such cluster is stored as well as the total amount of objects of the whole graph. This means that, at materialization time, *Fuel* knows exactly the number of allocations (new objects) needed for each cluster. For example, one *Fuel* file can contain 17 large integers, 5 floats, 5 symbols, etc. In addition, for variable objects, *Fuel* also stores the size of such objects. So, for example, it does not only know that there are 5 symbols but also that the first symbol is size 4, the second one is 20, the third is 6, etc.

Therefore, the materialization populates an object table with indices from 1 to N where N is the number of objects in the file. Most serializers determine which object to create as they walk a (flattened) input graph. In the case of *Fuel*, it does so in batch (spinning in a loop creating N instances of each class in turn).

Once that is done, the objects have been materialized but updating the references is pending, *i.e.,* which fields refer to which objects. Again, the materializer can spin filling in fields from the reference data instead of determining whether to instantiate an object or dereference an object ID as it walks the input graph.

This is the main reason why materializing is much faster in *Fuel* than in other approaches.

3.6 Iterative depth-first traversal

Most of the serializers use a depth-first traversal mechanism to serialize the object graph. Such mechanism consists of a simple recursion:

1. Take an object and look it up in a table.

2. If the object is in the table, it means that it has already been serialized. Then, we take a reference from the table and write it down. If it is not present in the table, it means that the object has not been serialized and that its contents needs to be written. After that, the object is serialized and a reference representation is written into the table.

3. While writing the contents, *e.g.,* instance variables of an object, the serializer can encounter simple objects such as instances of String, SmallInteger, LargePositiveInteger, ByteArray or complex objects (objects which have instance variables that refer to other objects). In the latter case, we start over from the first step.

This mechanism can consume too much memory depending on the graph, *e.g.,* its depth, the memory to hold all the call stack of the recursion can be too much.

In *Fuel*, we use depth-first traversal but not in the straightforward recursive implementation. We used a technique often known as *stack externalization.* The difference is mainly in the last step of the algorithm. When an object has references to other objects, instead of following the recursion to analyze these objects, we just push such objects on a custom stack. Then, we pop objects from the stack and analyze them. The routine is to pop and analyze elements until the stack is empty. In addition, to improve even more speed, *Fuel* has its own SimpleStack class

implementation. With this approach, the resulting stack size is much smaller and the memory footprint is smaller as well. At the same time, we decrease serialization time by 10%.

This is possible because *Fuel* has two phases, for writing instances and references, as explained above.

4. SERIALIZER REQUIRED CONCERNS AND CHALLENGES

Before presenting *Fuel*'s features in more detail, we present some useful elements of comparison between serializers. This list is not exhaustive.

4.1 Serializer concerns

Below we list general aspects to analyze in a serializer.

Performance. In almost every software component, time and space efficiency is a wish or sometimes even a requirement. It does become a need when the serialization or materialization is frequent or when working with large graphs. We can measure both speed and memory usage, either serializing and materializing, as well as the size of the obtained stream. We should also take into account the initialization time, which is important when doing frequent small serializations.

Completeness. It refers to what kind of objects the serializer can handle. It is clear that it does not make sense to transport instances of some classes, like FileStream or Socket. Nevertheless, serializers often have limitations that restrict use cases. For example, an apparently simple object like a SortedCollection usually represents a challenging graph to store: it references a block closure which refers to a method context and most serializers do not support transporting them, often due to portability reasons. In view of this difficulty, it is common that serializers simplify collections storing them just as a list of elements.

In addition, in comparison with other popular environments, the object graphs that one can serialize in Smalltalk are much more complex because of the reification of metalevel elements such as methods, block closures, and even the execution stack. Usual serializers are specialized for plain objects or metalevel entities (usually when their goal is code management), but not both at the same time.

Portability. Two aspects related to portability. One is related to the ability to use the same serializer in different dialects of the same language or even a different language. The second aspect is related to the ability of being able to materialize in a dialect or language a stream which was serialized in another language. This aspect brings even more problems and challenges to the first one.

As every language and environment has its own particularities, there is a trade-off between portability and completeness. Float and BlockClosure instances often have incompatibility problems.

For example, Action Message Format [1], Google Protocol Buffers [22], Oracle Coherence*Web [20], Hessian [11], have low-level language-independent formats oriented to exchange structured data between many languages. In contrast, SmartRefStream in Pharo and Pickle [21] in Python choose to be language-dependent but enabling serialization of more complex object graphs.

Security. Materializing from an untrusted stream is a possible security problem. When loading a graph, some kind of dangerous objects can enter to the environment. The user may want to control in some way what is being materialized.

Atomicity. We have this concern expressed in two parts: for saving and for loading. As we know, the environment is full of mutable objects *i.e.,* that change their state over the time. So, while the serialization process is running, it is desired that such mutable graph is written in an atomic snapshot and not a potential inconsistent one. On the other hand, if we load from a broken stream, it will not successfully complete the process. In such case, no secondary effects should affect the environment. For example, there can be an error in the middle of the materialization which means that certain objects have already been materialized.

Versatility. Let us assume a class is referenced from the graph to serialize. Sometimes we may be interested in storing just the name of the class because we know it will be present when materializing the graph. However, sometimes we want to really store the class with full detail, including its method dictionary, methods, class variables, etc. When serializing a package, we are interested in a mixture of both: for external classes, just the name but, for the internal ones, full detail.

This means that given an object graph, there is not an unique way of serializing it. A serializer may offer the user dynamic or static mechanisms to customize this behavior.

4.2 Serializer challenges

The following is a list of concrete issues and features that users can require from a serializer.

Cyclic object graphs and duplicates. Since the object graph to serialize usually has cycles, it is important to detect them and to preserve the objects' identity. Supporting this means decreasing the performance and increasing the memory usage, because for each object in the graph it is necessary to check whether it has been already processed or not, and if it has not, it must be temporally stored.

Maintaining identity. There are objects in the environment we do not want to replicate on deserialization because they represent well-known instances.

We can illustrate with the example of Transcript, which in Pharo environment is a global variable that binds to an instance of ThreadSafeStream. Since every enviroment has its own unique-instance of Transcript, the materialization of it should respect this characteristic and thus not create another instance of ThreadSafeStream but use the already present one.

Transient values. Sometimes, objects have a temporal state that we do not want to store and we want also an initial value when loading. A typical case is serializing an object that has an instance variable with a lazy-initialized value. Suppose we prefer not to store the actual value. In this sense, declaring a variable as *transient* is a way of delimiting the graph to serialize.

There are different levels of transient values:

- Instance level: When only one particular object is transient. All objects in the graph that are referencing to such object will be serialized with a nil in their instance variable that points to the transient object.
- Class level: Imagine we can define that a certain class is transient in which case all its instances are considered transient.
- Instance variable names: the user can define that certain instance variables of a class have to be transient. This means that all instances of such class will consider those instance variables as transient. This type of transient value is the most common.
- List of objects: the ability to consider an object to be transient only if it is found in a specific list of objects. The user should be able to add and remove elements from that list.

Class shape change tolerance. Often, we need to load instances of a class in an environment where its definition has changed. The expected behavior may be to adapt the old-shaped instances automatically when possible. We can see some examples of this in Figure 4.1. For instance variable position change, the adaptation is straightforward. For example, version v2 of Point changes the order between the instance variables x and y. For the variable addition, an easy solution is to fill with nil. Version v3 adds instance variable distanceToZero. If the serializer also lets one write custom messages to be sent by the serializer once the materialization is finished, the user can benefit from this hook to initialize the new instance variables to something different that nil.

In contrast to the previous examples, for variable renaming, the user must specify what to do. This can be done via hook methods or, more dynamically, via materialization settings.

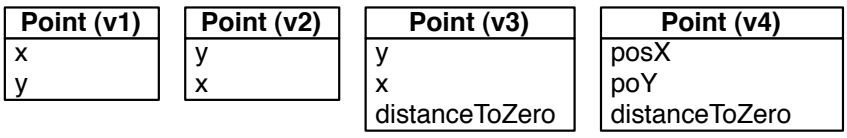

Fig. 4.1: Several kinds of class shape changing.

There are even more kinds of changes such as adding, removing or renaming a class or an instance variable, changing the superclass, etc. As far as we know, no serializer fully manages all these kinds of changes. Actually, most of them have a limited number of supported change types. For example, the Java Serializer [12] does not support changing an object's hierarchy or removing the implementation of the Serializable interface.

Custom reading. When working with large graphs or when there is a large number of stored streams, it makes sense to read the serialized bytes in customized ways, not necessarily materializing all the objects as we usually do. For example, if there

are methods written in the streams, we may want to look for references to certain message selectors. Maybe we want to count how many instances of certain class we have stored. We may also want to list the classes or packages referenced from a stream or even extract any kind of statistics about the stored objects.

Partial loading. In some scenarios, especially when working with large graphs, it may be necessary to materialize only a part of the graph from the stream instead of the whole graph. Therefore, it is a good feature to simply get a subgraph with some holes filled with nil or even with proxy objects to support some kind of lazy loading.

Versioning. The user may need to load an object graph stored with a different version of the serializer. This feature enables version checking so that future versions can detect that a stream was stored using another version and act consequently: when possible, migrating it and, when not, throwing an error message. This feature brings the point of backward compatibility and migration between versions.

5. FUEL FEATURES

In this section, we analyze *Fuel* in accordance with the concerns and features defined in Section 4.

5.1 Fuel serializer concerns

Performance. We achieved an excellent time performance. The main reason behind *Fuel*'s performance in materialization is the ability to perform the materialization *iteratively* rather than *recursively*. That is possible thanks to the clustered pickle format. Nevertheless, there are more reasons behind *Fuel*'s performance:

- We have implemented special collections to take advantage of the characteristics of algorithms.
- Since *Fuel* algorithms are iterative, we know *in advance* the size of each loop. Hence, we can always use optimized methods like to:do: for the loops.
- For each basic type of object such as Bitmap, ByteString, ByteSymbol, Character, Date, DateAndTime, Duration, Float, Time, etc. , we optimize the way they are encoded and decoded.
- *Fuel* takes benefits of being platform-specific (Pharo), while other serializers sacrifice speed in pursuit of portability.

Performance is extensively studied and compared in Section 7.

Completeness. We are close to say that *Fuel* deals with all kinds of objects available in a Smalltalk runtime. Note the difference between being able to serialize and getting something meaningful while materializing. For example, *Fuel* can serialize and materialize instances of Socket, Process or FileStream but it is not sure they will still be valid once they are materialized. For example, the operating system may have given the socket address to another process, the file associated to the file stream may have been removed, etc. There is no magic. *Fuel* provides hooks to solve the mentioned problems. For example, there is a hook so that a message is sent once the materialization is done. One can implement the necessary behavior to get a meaningful object. For instance, a new socket may be assigned. Nevertheless sometimes there is nothing to do, *e.g.,* if the file of the file stream was removed by the operating system. Note that some well known special objects are treated as external references because that is the expected behavior for a serializer. Some examples are, Smalltalk, Transcript and Processor.

Portability. As we explained in other sections, the portability of *Fuel*'s source code is not our main focus. However, *Fuel* has already been successfully ported to Squeak, to Newspeak programming language and, at the moment of this writing, half ported to VisualWorks. What *Fuel* does not support is the ability to materialize in a dialect

or language a stream which was serialized in another language. We do not plan to communicate with another technology.

Even if *Fuel*'s code is not portable to other programming languages, the algorithms and principles are general enough for being reused in other object environments. In fact, we have not invented this type of pickle format that groups similar objects together and that does a iterative materialization. There are already several serializers that are based on this principle such as Parcels serializer from VisualWorks Smalltalk.

Versatility. Our default behavior is to reproduce the serialized object as exact as possible. Nonetheless, for customizing that behavior we provide what we call *substitutions* in the graph. The user has two alternatives: at class-level or at instance-level. In the former case, the class implements hook methods that specify that its instances will be serialized as another object. In the latter, the user can tell the serializer that when an object (independently of its class) satisfies certain condition, then it will be serialized as another object.

Security. Our goal is to give the user the possibility to configure validation rules to be applied over the graph (ideally) before having any secondary effect on the environment. This has not been implemented yet.

Atomicity. *Fuel* can have problems if the graph changes during the analysis or serialization phase. Not only *Fuel* suffers this problem, but also the rest of the serializers we analyzed. From our point of view, the solution always lies at a higher level than the serializer. For example, if one has a domain model that is changing and wants to implement save/restore, one needs to provide synchronization so that the snapshots are taken at valid times and that the restore actions work correctly.

5.2 Fuel serializer challenges

In this section, we explain how *Fuel* implements some of the features previously commented. There are some mentioned challenges such as *partial loading* or *custom reading* that we do not include here because *Fuel* does not support them at the moment.

Cyclic object graphs and duplicates. *Fuel* checks that every object of the graph is visited once supporting both cycle and duplicate detection.

Maintaining identity. The default behavior when traversing a graph is to recognize some objects as *external references*: Classes registered in Smalltalk, global objects (referenced by global variables), global bindings (included in Smalltalk globals associations) and class variable bindings (included in the classPool of a registered class).

This mapping is done at object granularity, *e.g.,* not every class will be recognized as external. If a class is not in Smalltalk globals or if it has been specified as an *internal class*, it will be traversed and serialized in full detail.

Transient values. There are two main ways of declaring transient values in *Fuel*. On the one hand, through the hook method fuelIgnoredInstanceVariableNames, where the user specifies variable names whose values will not be traversed nor serialized. On materialization, they will be restored as nil. On the other hand, as we provide the possibility to substitute an object in the actual graph by another one, then an object with transient values can substitute itself by a copy but with such values set to nil. This technique gives a great flexibility to the user for supporting different forms of transient values.

Class shape change tolerance. *Fuel* stores the list of variable names of the classes that have instances in the graph being written. While recreating an object from the stream, if its class has changed, then this meta information serves to automatically adapt the stored instances. When an instance variable does not exist anymore, its value is ignored. If an instance variable is new, it is restored as nil. This is true not only for changes in the class but also for changes in any class of the hierarchy. Nevertheless, there are much more kinds of changes a class can suffer that we are not yet able to handle correctly. This is a topic we have to improve.

Versioning. We sign the stream with a well-known string prefix, and then we write the version number of the serializer. Then, when loading the signature and the version has to match with current materializer. Otherwise, we signal an appropriate error. At the moment, we do not support backward compatibility.

5.3 Discussion

Since performance is an important goal for us, we could question why to develop a new serializer instead of optimizing an existing one. For instance, the benefits of using a buffered stream for writing could apply to any serializer. Traditional serializers based on a recursive format commonly implement a technique of caching classes to avoid decoding and fetching the classes on each new instance. The advantages of our clustering format for fast materialization may look similar to such optimization.

Despite of that, we believe that our solution is necessary to get the best performance. The reasons are:

- The caching technique is not as fast as our clustered pickle format. Even if there is cache, the type is *always* written and read per object. Depending of the type of stream, for example, network-based streams, the time spent to read or write the type can be bigger than the time to decode the type and fetch the associated class.
- Since with the cache technique the type is written per object, the resulted stream is much bigger than *Fuel*'s one (since we write the type once per cluster). Having larger streams can be a problem in some scenarios.
- *Fuel*'s performance is not only due to the pickle format. As we explained at the beginning of this section, there are more reasons.

Apart from the performance point of view, there is a set of other facts that makes *Fuel* valuable in comparison with other serializers:

- It has an object-oriented design, making it easy to adapt and extend to custom user needs. For example, as explained in Section 8, *Fuel* was successfully customized to correctly support Newspeak modules or proxies in Marea's object graph swapper.
- It can serialize and materialize objects that are usually unsupported in other serializers such as global associations, block closures, contexts, compiled methods, classes and traits. This is hard to implement without a clear design.
- It is modular and extensible. For example, the core functionality to serialize plain objects is at the Fuel package, while another named FuelMetalevel is built of top of it, adding the possibility to serialize classes, methods, traits, etc. Likewise, on top of FuelMetalevel, FuelPackageLoader supports saving and loading complete packages without making use of the compiler.
- It does not need any special support from the VM.
- It is covered by tests and benchmarks.

6. FUEL DESIGN AND INFRASTRUCTURE

Figure 6.1 shows a simplified class diagram of *Fuel*'s design. It is challenging to explain a design in a couple of lines and a small diagram. However, the following are the most important characteristics about *Fuel*'s design.

- Serializer, Materializer and Analyzer, marked in bold boxes, are the *API* for the whole framework. They are facade classes that provide what most users need to serialize and materialize. In addition they act as builders, creating on each run an instance of Serialization, Materialization and Analysis, respectively, which implement the algorithms. Through *extension methods* we modularly add functionalities to the protocol. For example, the optional package FuelProgressUpdate adds the message showProgress to the mentioned facade classes, which activates a progress bar when processing. We have also experimented with a package named FuelGzip, which adds the message writeGzipped to Serializer, providing the possibility to compress the serialization output.

- The hierarchy of Mapper is an important characteristic of our design: Classes in this hierarchy make possible to completely customize how the graph is traversed and serialized. They implement the *Chain of Responsibility* design pattern for determining what cluster corresponds to an object [2]. An Analyzer creates a chain of mappers each time we serialize.

- The hierarchy of Cluster has 44 subclasses, where 10 of them are optional optimizations.

- *Fuel* has 1861 lines of code, split in 70 classes. Average number of methods per class is 7 and the average lines per method is 3.8. We made this measurements on the *core* package Fuel. *Fuel* is covered by 192 unit tests that covers all use cases. Its test coverage is more than 90% and the lines of code of tests is 1830, almost the same as the core.

- *Fuel* has a complete benchmark framework which contains samples for all necessary primitive objects apart from samples for large object graphs. We can easily compare serialization and materialization with other serializers. It has been essential for optimizing our performance and verifying how much each change impacts during development. In addition, the tool can export results to csv files which ease the immediate build of charts.

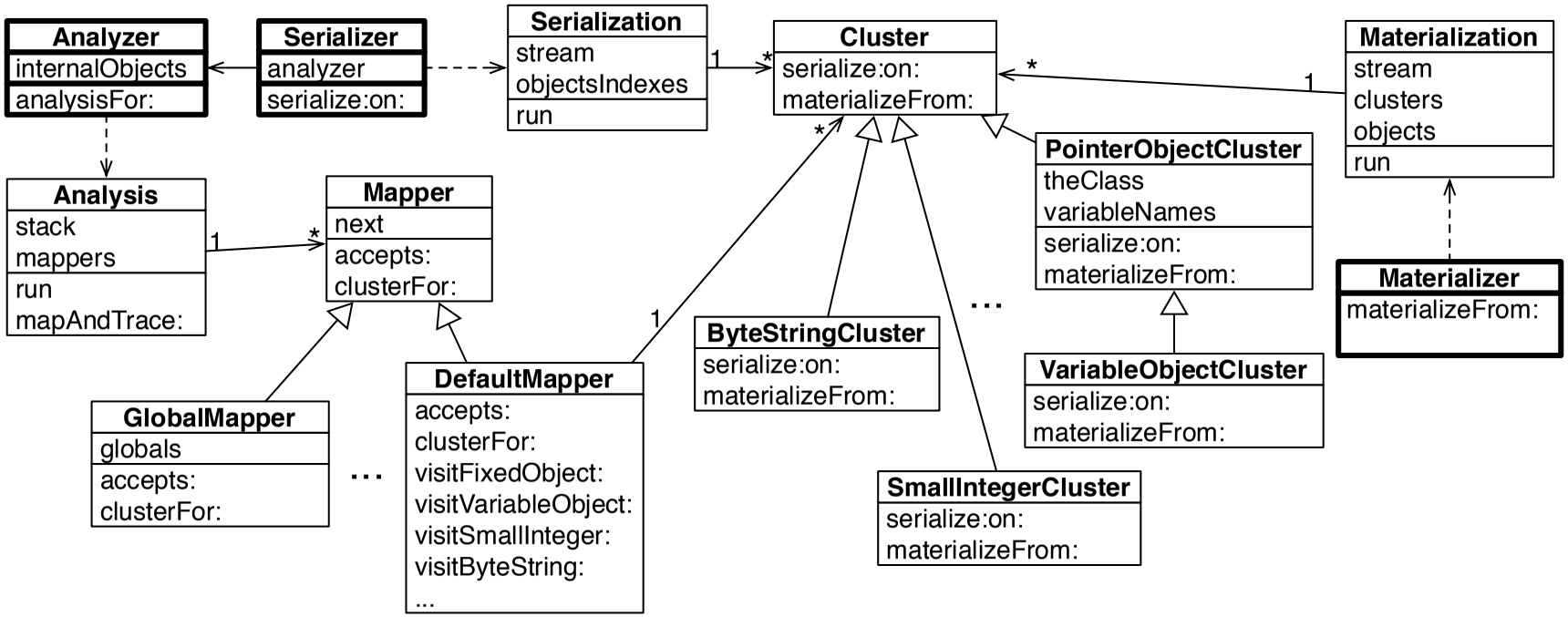

Fig. 6.1: Fuel's design

7. BENCHMARKS

We have developed several benchmarks to compare different serializers. To get meaningful results all benchmarks have to be run in the same environment. Since *Fuel* is developed in Pharo, we run all the benchmarks with Pharo-1.3 and Cog Virtual Machine version "VMMaker.oscog-eem.56". The operating system was Mac OS 10.6.7.

In the following benchmarks, we have analyzed the serializers: *Fuel* (version 1.7), SIXX (version 0.3.6), SmartRefStream (the version in Pharo 1.3), ImageSegment (the version in Pharo 1.3), Magma object database serializer (version 1.2), StOMP (version 1.8) and SRP (version SRP-mu.11). Such serializers are explained in Section 9.

7.1 Benchmarks constraints and characteristics

Benchmarking software as complex as a serializer is difficult because there are multiple functions to measure which are used independently in various real-world use-cases. Moreover, measuring only the speed of a serializer, is not complete and it may not even be fair if we do not mention the provided features of each serializer. For example, providing a hook for user-defined reinitialization action after materialization or supporting class shape changes slows down serializers. Here is a list of constraints and characteristics we used to get meaningful benchmarks:

All serializers in the same environment. We are not interested in comparing speed with serializers that run in a different environment than Pharo because the results would be meaningless.

Use default configuration for all serializers. Some serializers provide customizations to improve performance, *i.e.,* some parameters or settings that the user can set for serializing a particular object graph. Those settings would make the serialization or materialization faster or slower, depending on the customization. For example, a serializer can provide a way to do *not* detect cycles. Detecting cycles takes time and memory hence, not detecting them is faster. Consequently, if there is a cycle in the object graph to serialize, there will be a loop and finally a system crash. Nevertheless, in certain scenarios, the user may have a graph where he knows that there are no cycles.

Streams. Another important point while measuring serializers performance is which stream will be used. Usually, one can use memory-based streams and file-based streams. There can be significant differences between them and all the serializers must be benchmarked with the same type of stream.

Distinguish serialization from materialization. It makes sense to consider different benchmarks for the serialization and for the materialization.

Different kinds of samples. Benchmark samples are split in two kinds: primitive and large. Samples of primitive objects are samples with lots of objects which are instances of the same class and that class is "primitive". Examples of those classes are Bitmap, Float, SmallInteger, LargePositiveInteger, LargeNegativeInteger, String, Symbol, WideString, Character, ByteArray, etc. Large objects are objects which are composed by other objects which are instances of different classes, generating a large object graph.

Primitive samples are useful to detect whether one serializer is better than the rest while serializing or materializing certain type of object. Large samples are more similar to the expected user provided graphs to serialize and they try to benchmark examples of real life object graphs.

Avoid JIT side effects. In Cog (the VM we used for benchmarks), the first time a method is used, it is executed in the standard way and added to the method cache. The second time the method is executed (when it is found in the cache), Cog converts that method to machine code. However, extra time is needed for such task. Only the third time, the method will be executed as machine code and without extra effort.

It is not fair to run with methods that have been converted to machine code together with methods that have not. Therefore, for the samples, we first run twice the same sample without taking into account its execution time to be sure we are always in the same condition. Then, the sample is finally run and its execution time is computed. We run several times the same sample and take the average of it.

7.2 Benchmarks serializing primitive and large objects

Primitive objects serialization. Figure 7.1 shows the results of primitive objects serialization and materialization using memory-based streams. The conclusions are:

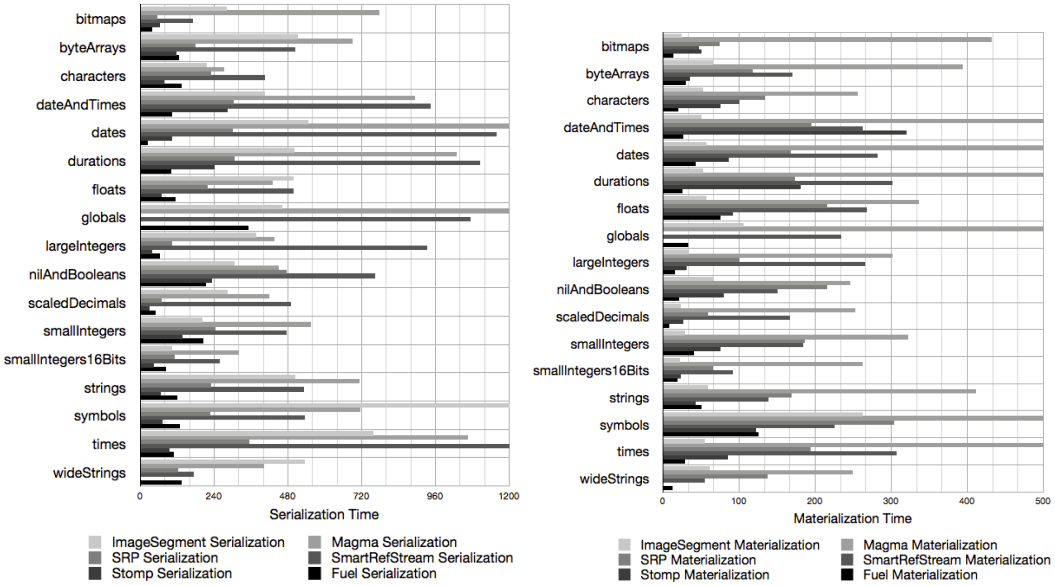

Fig. 7.1: Time (in ms) for primitive objects serialization and materialization (the smaller the better).

- We did not include SIXX in the charts because it was so slow that we were not able to show the differences between the rest of the serializers. This result is expected since SIXX is a text based serializer which is far slower than a binary one. However, SIXX can be opened and modified by any text editor. This is an usual trade-off between text and binary formats.
- Magma and SmartRefStream serializers seem to be the slowest ones in most cases.
- StOMP is the fastest one in serialization nearly followed *Fuel*, SRP and ImageSegment.
- Magma serializer is slow with "raw bytes" objects such as Bitmap and ByteArray, etc.
- Most of the times, *Fuel* is faster than ImageSegment, which is even implemented in the Virtual Machine.
- ImageSegment is really slow with Symbol instances. We explain the reason later in Section 7.3.
- StOMP has a zero (its color does not even appear) in the WideString sample. That means that it cannot serialize those objects.

For materialization, *Fuel is the fastest one followed by StOMP and ImageSegment.* In this case and in the following benchmarks, we use memory-based streams instead of file or network ones. This is to be fair with the other serializers. Nonetheless, *Fuel* does certain optimizations to deal with slow streams like file or network. Basically, it uses an internal buffer that flushes when it is full. This is only necessary because the streams in Pharo are not very performant. This means that, if we run these same benchmarks but using *e.g.,* a file-based stream, *Fuel* is at least 3 times faster than the second one in serialization. This is important because, in the real uses of a serializer, we do not usually serialize to memory, but to disk.

Large objects serialization. As explained, these samples contain objects which are composed by other objects that are instances of different classes, generating a large object graph. Figure 7.2 shows the results of large objects serialization and materialization. The conclusions are:

- The differences in speed are similar to the previous benchmarks. This means that, whether we serialize graphs of all primitive objects or objects instances of all different classes, *Fuel is the fastest one in materialization and one of the best ones in serialization.*
- StOMP cannot serialize the samples for *associations* and *bag*. This is because those samples contain different kind of objects, which StOMP cannot serialize. This demonstrates that the mentioned serializers do not support serialization and materialization of all kind of objects. At least, not out-of-the-box. Notice also that all these large objects samples were built so that most serializers do not fail. To have a rich benchmark, we have already excluded different types of objects that some serializers do not support.

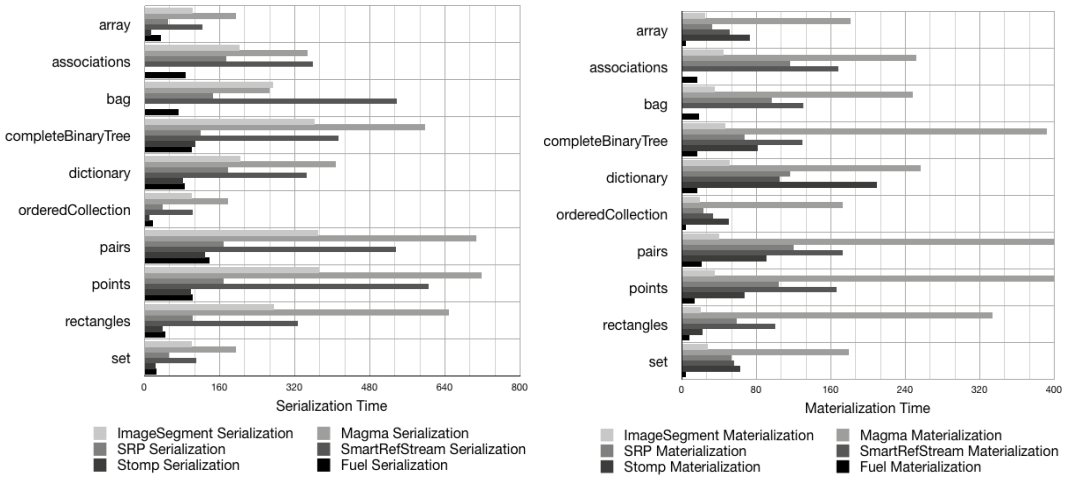

Fig. 7.2: Time (in ms) for large objects serialization and materialization (the smaller the better).

7.3 ImageSegment results explained

ImageSegment seems to be really fast in certain scenarios. However, it deserves some explanations of how ImageSegment works [15]. Basically, ImageSegment gets a user defined graph and needs to distinguish between *shared objects* and *inner objects*. Inner objects are those inside the subgraph which are *only* referenced from objects inside the subgraph. *Shared objects* are those which are not only referenced from objects inside the subgraph, but also from objects outside.

All *inner objects* are put into a byte array which is finally written into the stream using a primitive implemented in the virtual machine. Afterwards, ImageSegment uses SmartRefStream to serialize the *shared objects*. ImageSegment is fast mostly because it is implemented in the virtual machine. However, as we saw in our benchmarks, SmartRefStream is not really fast. The real problem is that it is difficult to control which objects in the system are pointing to objects inside the subgraph. Hence, there are frequently several *shared objects* in the graph. The result is that, the more *shared objects* there are, the slower ImageSegment is because those *shared objects* will be serialized by SmartRefStream.

All the benchmarks we did with primitive objects (all but Symbol) create graphs with zero or few shared objects. This means that we are measuring the fastest possible case ever for ImageSegment. Nevertheless, in the sample of Symbol, one can see in Figure 7.1 that ImageSegment is really slow in serialization and the same happens with materialization. The reason is that, in Smalltalk, all instances of Symbol are unique and referenced by a global table. Hence, all Symbol instances are shared and, therefore, serialized with SmartRefStream.

Figure 7.3 shows an experiment we did where we build an object graph and we increase the percentage of *shared objects*. Axis X represents the percentage of shared objects inside the graph and the axis Y represents the time of the serialization or materialization.

Conclusions for ImageSegment results

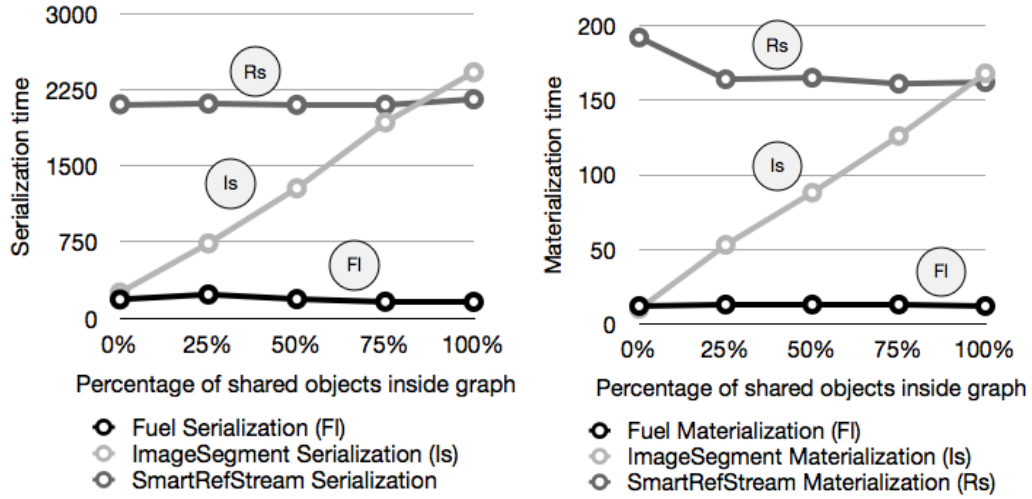

Fig. 7.3: ImageSegment serialization and materialization in presence of shared objects.

- The more shared objects there are, the more similar is ImageSegment speed compared to SmartRefStream.
- For materialization, when all objects are shared, ImageSegment and SmartRefStream have almost the same speed.
- For serialization, when all objects are share, ImageSegment is even slower than SmartRefStream. This is because ImageSegment needs to do the whole memory traverse anyway to discover shared objects.
- ImageSegment is unique in the sense that its performance depends on both: 1) the amount of references from outside the subgraph to objects inside; 2) the total amount of objects in the system since the time to traverse the whole memory depends on that.

7.4 Different graph sizes

Another important analysis is to determine if there are differences between the serializers depending on the size of the graph to serialize. We created different subgraphs of different sizes. To simplify the charts we express the results in terms of the largest subgraph size which is 50.000 objects. The scale is expressed as percentage of this size.

Figure 7.4 shows the results of the experiment. Axis X represents the size of the graph, which in this case is represented as a percentage of the largest graph. Axis Y represents the time of the serialization or materialization.

Conclusions for different graph sizes. There are not any special conclusions for this benchmark since the performance differences between the serializers are almost the same with different graph sizes. In general the serializers have a linear dependency with the number of objects of the graph. For materialization, *Fuel* is the fastest and

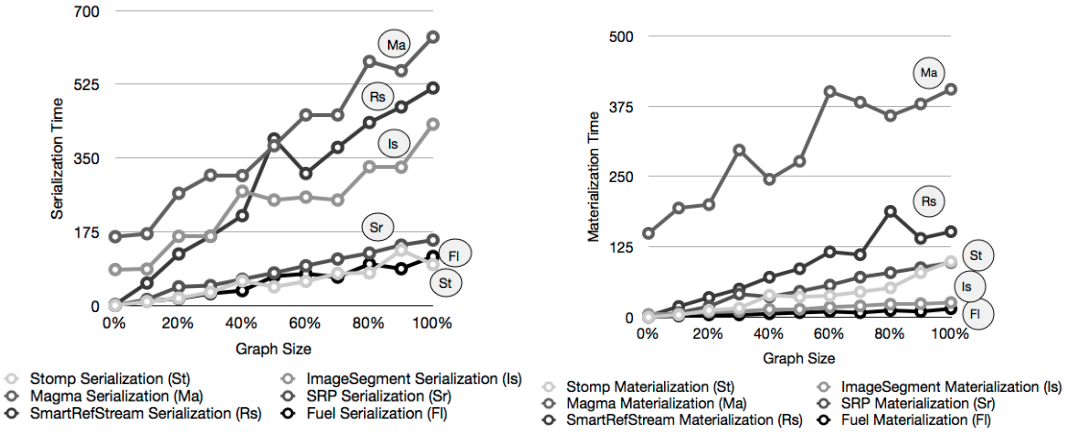

Fig. 7.4: Serialization and materialization of different graph sizes.

for serialization is similar than StOMP. *Fuel* performs then well with small graphs as well as large ones.

7.5 Differences while using CogVM

At the beginning of this section, we explained that all benchmarks are run with the Cog Virtual Machine. Such a virtual machine introduces several significant improvements such as PIC (polymorphic inline cache) and JIT (just in time compiling) which generates machine code from interpreted methods. All those improvements impact, mainly, in regular methods implemented in the language side but not in VM inside itself, VM primitives or plugins.

Cog is at least 4 times faster than the interpreter VM (the previous VM). It is a common belief that ImageSegment was the fastest serialization approach. However, along this section, we showed that *Fuel* is most of the times as fast as ImageSegment and sometimes even faster.

One of the reasons is that, since ImageSegment is implemented as VM primitives and *Fuel* is implemented in the language side, with Cog *Fuel*, the speed increases four times while ImageSegment speed remains almost the same. This speed increase takes place not only with *Fuel*, but also with the rest of the serializers implemented in the language side.

To demonstrate these differences, we did an experiment: we ran the same benchmarks with ImageSegment and *Fuel* with both virtual machines, Cog and Interpreter VM. For each serializer, we calculated the difference in time of running both virtual machines. Figure 7.5 shows the results of the experiment. Axis X represents the difference in time between running the benchmarks with Cog and non Cog VMs.

As we can see in both operations (serialization and materialization), the difference in *Fuel* is much bigger that the difference in ImageSegment.

7.6 General Benchmarks Conclusions

Magma serializer seems slow but it is acceptable taking into account that this serializer is designed for a particular database. Hence, the Magma serializer does an

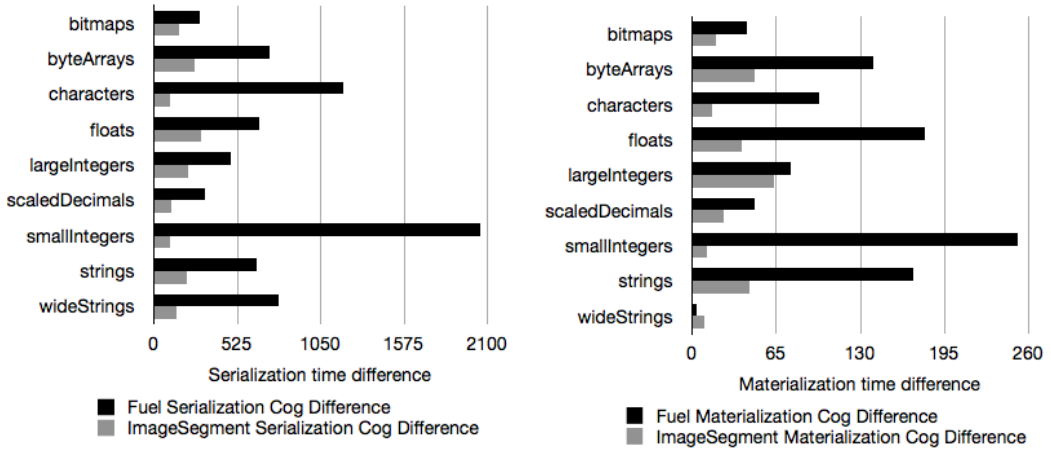

Fig. 7.5: Serialization and materialization differences when using CogVM

extra effort and stores extra information that is needed in a database scenario but may not be necessary for any other usage.

SmartRefSteam provides a good set of hook methods for customizing serialization and materialization. However, it is slow and its code is complex and difficult to maintain from our point of view. ImageSegment is known to be really fast because it is implemented inside the virtual machine. Such fact, together with the problem of *shared objects*, brings a large number of limitations and drawbacks as it has been already explained. Furthermore, with Cog, we demonstrate that *Fuel* is even faster in both, materialization and serialization. Hence, the limitations of ImageSegment are not worth it.

SRP and StOMP are both aimed for portability across Smalltalk dialects. Their performance is good, mostly at writing time, but they are not as fast as they could be because of the need of being portable across platforms. In addition, for the same reason, they do not support serialization for all kind of objects.

This work demonstrates that *Fuel* is the fastest in materialization and one of the fastest ones in serialization. In fact, when serializing to files, which is what usually happens, *Fuel* is the fastest. *Fuel* can also serialize any kind of object. *Fuel* aim is not portability but performance. Hence, all the results make sense from the goals point of view.

8. REAL CASES USING FUEL

Even if *Fuel* is still in development, it is already being used in real applications. Here we report the first ones we are aware of. *Fuel* is also ported by external developers to other Smalltalk dialects.

Moose Models. Moose is an open-source platform for software and data analysis [19]. It uses large data models that can be exported and imported from files. The models produced by Moose represent source code and information produced by analyzers. A Moose model can easily contain 500,000 entities. Moose is also implemented on top of Pharo. The *Fuel* Moose extension is eight times faster in exporting and four times faster in importing than its competitor MSE. We have developed a model export/import extension[1] which has been integrated into Moose Suite 4.4.

Pier CMS persistency. Pier is a content management system that is light, flexible and free [23]. It is implemented on top of the Seaside web framework [25]. In Pier all the documents and applications are represented as objects. A simple persistency strategy has been implemented using Fuel[2]. This tool persists Pier CMS kernels (large object graphs), *i.e.,* Pier CMS websites. This way you can backup your system and load it back later on if desired.

SandstoneDB with Fuel backend. SandstoneDB[3] is a lightweight Prevayler style embedded object database with an ActiveRecord API that does not require a command pattern and works for small applications that a single Pharo image can handle. The idea is to make a Pharo image durable, crash proof and suitable for use in small office applications. By default, SandtoneDB used the SmartRefStream serializer. Now there is a *Fuel* backend which accelerates SandstoneDB 300% approximately.

Newspeak port. Newspeak[4] is a language derived from Smalltalk. Its developers have successfully finished a port of *Fuel* to Newspeak and they are using it to save and restore their data sets. They had to implement one extension to save and restore Newspeak classes, which is complex because these are instantiated classes inside instantiated Newspeak modules [6] and not static Smalltalk classes in the Smalltalk dictionary. *Fuel* proved to be flexible enough to make such port successful taking only few hours of work.

Marea. Marea is a transparent object graph swapper whose goal is to use less memory by only leaving in primary memory what is needed and used serializing and swapping out the unused objects to secondary memory [16, 17]. When one of these unused objects is needed, Marea brings it back into primary memory. To achieve this, the system replaces original objects with proxies. Whenever a proxy receives

[1] http://www.moosetechnology.org/tools/fuel
[2] http://ss3.gemstone.com/ss/pierfuel.html
[3] http://onsmalltalk.com/sandstonedb-simple-activerecord-style-persistence-in-squeak
[4] http://newspeaklanguage.org/

a message, it loads back and materializes the swapped out object from secondary memory. Marea needs to correctly serialize and materialize any type of object such as classes, methods, contexts, closures, etc.

When *Fuel* is processing a graph, it sends messages to each object, such as asking its state or asking its hash (to temporally store it into a hashed collection). But in presence of proxies inside the graph, that means that the proxy intercepts those messages and swaps in the swapped out graph. To solve this problem, Marea extends *Fuel* so that it takes special care when sending messages to proxies. As a result, Marea can serialize graphs that contain proxies without causing them to swap in.

9. RELATED WORK

We faced a general problem to write a decent related work: serializers are not clearly described. At the best, we could execute them, sometimes after porting effort. Most of the times there was not even documentation. The rare work on serializers that we could find in the literature was done to advocate that using C support was important [24]. But since this work is implemented in Java we could compare and draw any scientific conclusion.

The most common example of a serializer is one based on XML like SIXX [26] or JSON [13]. In this case, the object graph is exported into a portable text file. The main problem with text-based serialization is encountered with big graphs as it does not have a good performance and it generates huge files. Other alternatives are ReferenceStream or SmartReferenceStream. ReferenceStream is a way of serializing a tree of objects into a binary file. A ReferenceStream can store one or more objects in a persistent form including sharing and cycles. The main problem of ReferenceStream is that it is slow for large graphs.

A much more elaborated approach is Parcel [18] developed in VisualWorks Smalltalk. *Fuel* is based on Parcel's pickling ideas. Parcel is an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement and partial loading. The key to making this deployment mechanism feasible and fast is the pickling algorithm. Although Parcel supports code and objects, it is more intended to source code than normal objects. It defines a custom format and generates binary files. Parcel has a good performance and the assumption is that the user may not have a problem if saving code takes more time as long as loading is really fast. The main difference with Parcels is that such project was mainly for managing code: classes, methods, and source code. Their focus was that, and not to be a general-purpose serializer. Hence, they deal with problems such as source code in methods, or what happens if we install a parcel and then we want to uninstall it, what happened with the code, and the classes, etc. Parcel is implemented in Cincom Smalltalk so we could not measure their performance to compare with *Fuel*.

The recent StOMP [28] (Smalltalk Objects on MessagePack[1]) and the mature SRP (State Replication Protocol) [27] are binary serializers with similar goals: Smalltalk-dialect portability and space efficiency. They are quite fast and configurable but they are limited with dialect-dependent objects like BlockClosure and MethodContext.

Object serializers are needed and used not only by final users, but also for specific type of applications or tools. What is interesting is that they can be used outside the scope of their project. Some examples are the object serializers of Monticello2 (a source code version system), Magma object database, Hessian binary web service protocol [11] or Oracle Coherence*Web HTTP session management [20].

Martinez-Peck et al. [15] performed an analysis of ImageSegment (a virtual machine serialization algorithm) and they found that the speed increase in ImageSegment is mainly because it is written in C compared to other frameworks written in

[1] http://msgpack.org

Smalltalk. However, ImageSegment is slower when objects in the subgraph to be serialized are externally referenced.

10. CONCLUSIONS AND FUTURE WORK

In this work, we have looked into the problem of serializing object graphs in object-oriented systems. We have analyzed its problems and challenges, which are general and independent of the technology.

These object graphs operations are important to support virtual memory, backups, migrations, exportations, etc. Speed is the biggest constraint in these kind of graph operations. Any possible solution has to be fast enough to be actually useful. In addition, the problem of performance is the most common one among the different solutions. Most of them do not deal properly with it.

We presented *Fuel*, a general purpose object graph serializer based on a pickling format and algorithm different from typical serializers. The advantage is that the unpickling process is optimized. On the one hand, the objects of a particular class are instantiated in bulk since they were carefully sorted when pickling. This is done in an iterative instead of a recursive way, which is what most serializers do. The disadvantage is that the pickling process takes extra time in comparison with other approaches. However, we show in detailed benchmarks that we have the best performance in most of the scenarios.

We implement and validate this approach in Pharo. We demonstrate that it is possible to build a fast serializer without specific VM support with a clean object-oriented design and providing the most possible required features for a serializer.

Instead of throwing an error, it is our plan to analyze the possibility of creating light-weight shadow classes when materializing instances of an inexistent class. Another important issue we would like to work on is in supporting backward compatibility and migration between different *Fuel* versions. Partial loading as well as the possibility of being able to query a serialized graph, are two concepts that are in our roadmap.

To conclude, *Fuel* is a fast object serializer built with a clean design, easy to extend and customize. New features will be added in the future and several tools will be build on top of it.

APÉNDICE

A. RESUMEN EN CASTELLANO

A.1 Introducción

Un **ambiente de objetos** en ejecución tiene miles de objetos enviando mensajes, mutando su estado, naciendo y muriendo en la memoria volátil del sistema. Hay diversos casos en donde se requiere **capturar** el estado de ciertos objetos con la finalidad de **reproducirlo** luego, ya sea en el mismo ambiente en ejecución original o en otro.

Adentrémonos en esta idea de captura. El estado de un objeto está dado por sus **colaboradores internos**. Tomemos como ejemplo a un objeto que representa a una transferencia bancaria, que tiene como colaboradores internos a una fecha de emisión, a una cuenta de banco origen y una destino, y a una suma de dinero. Entonces, para capturar el estado de la transferencia será necesario capturar también el de sus colaboradores internos, y así sucesivamente. En conclusión, afirmamos que para capturar el estado de un objeto es necesario capturar un grafo dirigido cuyas aristas son las relaciones de conocimiento, y cuyos nodos son los objetos alcanzables a través de esas relaciones.

Para capturar y/o persistir estos estados, utilizamos la serialización. Definimos la **serialización** de un objeto como la generación de una **representación secuencial** del grafo de sus colaboradores internos. El objetivo de la serialización es poder reproducir en una etapa posterior al objeto original a partir de esa representación secuencial. A dicho proceso complementario lo llamamos **deserialización** o **materialización**.

La representación secuencial antes mencionada es normalmente una **cadena de bytes**, que se escribe en un medio más persistente que la memoria de ejecución, como un archivo o una base de datos.

Un requerimiento opcional es que la cadena serializada tenga un **formato textual** sencillo tal que permita ser comprendido y manipulado por el ser humano. La finalidad puede ser la de facilitar la implementación del serializador y deserializador en cualquier tecnología, y por lo tanto facilitar el intercambio de datos con una gama amplia de tecnologías. La desventaja principal del formato textual es que puede resultar poco eficiente en espacio, y por lo tanto en tiempo, cuando los objetos intercambiados son grandes y complejos. En el **formato binario**, por el contrario, se busca la compacidad de las cadenas serializadas, obteniendo mejor performance a costa de perder la legibilidad por parte humana.

La finalidad principal de algunos serializadores es la de compartir objetos entre **tecnologías diferentes**, por ejemplo en el caso de *Servicios Web*. En tales casos es central tener un formato de intercambio común para entenderse, aunque en la abstracción se pierdan ciertas particularidades de cada tecnología. Es usual recurrir a una simplificación de los objetos a sencillas unidades contenedoras de datos, en pos de lograr compatibilidad entre tecnologías diferentes. Enfocarse en una **tecnología específica** puede tener resultados muy diferentes, y es el caso que más nos interesa estudiar. El conocimiento de la tecnología concreta en la que se reproducirá el objeto, abre la posibilidad de capturar y reproducir con más precisión y detalle los objetos

del ambiente.

A.1.1 Estado del arte

Haremos un repaso de los principales escenarios de uso para la serialización, enumerando soluciones concretas, que hemos investigado y analizado para tener un panorama general del estado del arte, y para determinar qué problemáticas del dominio de la serialización cuentan con soluciones reales.

Decidimos no considerar en nuestro repaso la persistencia de objetos con mapeos a bases de datos relacionales, porque la representación, ya sea en SQL o en tablas, no es precisamente una representación en secuencia de bytes, y no aporta a nuestro trabajo.

La principal clasificación que hacemos de los serializadores que observamos se refiere al nivel al que pertenecen los objetos que pretende persistir. Generalmente un serializador se enfoca sólo a los objetos de nivel base, o sólo a los de metanivel, pero no a ambos.

Persistencia de objetos de nivel base

Estudiando estos escenarios, nos encontramos con dos requerimientos principales, que están en tensión y por lo tanto determinan una solución de compromiso según el valor que se le asigne a cada uno:

- Interoperabilidad entre tecnologías heterogéneas. Suele ser necesario capturar el estado de los objetos de una manera simplificada, abstrayendo los conceptos representados por dichos objetos en estructuras de datos que la otra tecnología eventualmente pueda comprender y materializar.

- Eficiencia. Cuanto mayor sea el tamaño del grafo a transportar, y mayor sea el volumen de operaciones por segundo, más necesario será que el formato sea compacto, y el mecanismo de codificación y decodificación sea veloz.

Formato textual Suelen manejarse cuando las estructuras de datos son pequeñas, y se privilegia la interoperabilidad, dado que será relativamente fácil implementar la serialización y materialización en cada tecnología. A la hora de elegir un formato del abanico de posibilidades, la popularidad en el mercado es determinante, mucho más que las buenas cualidades comparadas. La intuición detrás de esto sería que un formato no popular dificulta la comunicación entre desarrolladores para implementarlo, porque necesitarán aprenderlo y estudiarlo, y entre las distintas tecnologías, porque puede no estar implementado en algunas de ellas, o tener una implementación con menor mantenimiento.

XML[1] (*Extensible Markup Language*) es un formato pensado para la representación de documentación estructurada, aunque debido a haberse convertido en un estándar de facto en la industria, se utiliza como un formato de propósitos generales. Un ejemplo es *SOAP*[2] que, apoyado en su popularidad, define un protocolo que es

[1] http://en.wikipedia.org/wiki/XML
[2] http://www.w3.org/TR/2007/REC-soap12-part0-20070427/

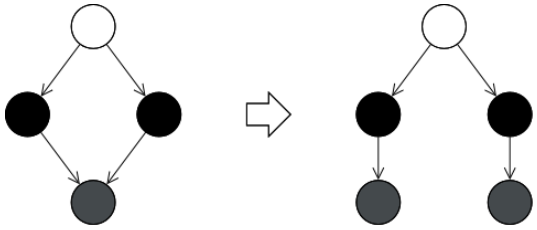

Fig. A.1: Serializando al objeto superior (en blanco) mediante Object»storeOn:, y luego materializando (con Object»readFrom:), el objeto inferior (en gris), se recrea por duplicado.

una extensión de *XML* cuyo objetivo es la ejecución remota universal, de amplia interoperabilidad entre tecnologías.

JSON[3] (*JavaScript Object Notation*) es un formato también muy difundido, impuesto por la popularidad de *JavaScript* como lenguaje y plataforma para aplicaciones web. El formato es muy sencillo y puede ser implementado fácilmente en cualquier otra tecnología, aunque tiene ventaja en su lenguaje original porque la materialización es muy eficiente y directa, gracias a que la evaluación de la secuencia se hace directamente en la máquina virtual. Tiene como contracara el riesgo de seguridad de que la secuencia contenga código ejecutable inesperado, por lo cual deben tomarse precauciones.

YAML[4] (*YAML Ain't Markup Language*), es similar a *JSON*, pero privilegia la interoperabilidad y la facilidad para representar estructuras complejas de datos. Esta elección va en desmedro de la eficiencia para la serialización y materialización.

El mecanismo tradicional de *Smalltalk-80* está implementado con el protocolo storeOn:/readFrom:, que todo objeto sabe responder, y cualquier clase puede especializar. El formato es textual, consistiendo en código que para materializarse será compilado y evaluado en el ambiente. O sea que no guarda información puramente descriptiva de los objetos serializados, como hacen la mayoría de los formatos, sino que guarda un script que indica cómo recrearlos operacionalmente. La solución es simple, pero quizás su mayor desventaja es que no está preparado para grafos circulares, los cuales provocan *Stack Overflow*. Además, en el caso de que un objeto fuera apuntado más de una vez dentro del grafo serializado, el mismo sería recreado por duplicado, perdiendo su identidad en el grafo materializado, como ilustramos en la Figura A.1.

Formato binario *CORBA*[5], *Thrift*[6] y *Protocol Buffers*[7] poseen formatos binarios propios, y su estrategia para asegurar la compatibilidad entre tecnologías es generar código automáticamente, tanto de servidor como de cliente, en una amplia variedad de lenguajes. Dicha generación se hace a partir de una especificación de tipos de datos y de servicios en un lenguaje específico de dominio (DSL). De esta forma brindan una compatibilidad semejante a la de un formato textual, pero con la eficiencia de un formato binario.

[3] http://www.json.org
[4] http://yaml.org/
[5] http://www.omg.org/spec/CORBA/3.2/
[6] http://thrift.apache.org/
[7] http://code.google.com/apis/protocolbuffers/

La intención de *MessagePack*[8] es reemplazar a *JSON* pero con un formato binario rápido y compacto.

En *Java*, la *Serialization API*[9] brinda soporte nativo con formato binario. Las clases de los objetos a serializar deben implementar la interfaz java.io.Serializable. Ha resultado eficiente para computación distribuida con *RMI* (*Remote Method Invocation*). Sin embargo, es sabido que resulta deficiente en su adaptación a la evolución del metamodelo, ya que no está preparado para tolerar el agregado, eliminación o cambio en el orden de las variables de instancia de una clase.

En *Ruby*, el serializador nativo es *Marshal*[10], que a pesar de estar dedicado a una tecnología específica, tiene el defecto de no soportar serialización de closures.

En *Python*, *Pickle*[11] está dedicado al transporte de objetos arbitrarios. Posee una versión llamada cPickle, implementada en lenguaje *C*, es decir, por fuera del paradigma de objetos, de la que se dice que es hasta 1000 veces más rápido, aunque impone algunas restricciones al uso.

BOSS (Binary Object Streaming Service), originario también de *Smalltalk-80*, ha evolucionado en distintos dialectos. Está presente actualmente en *VisualWorks Smalltalk,* en *Smalltalk/X*, y en *VisualSmalltalk*. Tiene formato binario, y además resuelve los inconvenientes mencionados con el mecanismo storeOn:/readFrom: acerca de los grafos circulares y los objetos compartidos.

En los dialectos *Squeak* y *Pharo*, el principal serializador binario es ImageSegment[12], que tiene algunos problemas serios de diseño que lo hacen difícil de mantener. Con la intención de lograr buena performance, la parte central de su mecanismo está implementada en la máquina virtual. Esto es un acoplamiento indeseable, que lo hace difícil de entender y extender. Unas pocas clases implementan una funcionalidad muy compleja con métodos muy extensos.

En *Squeak* y *Pharo* también contamos con los serializadores binarios DataStream, ReferenceStream y SmartRefStream. El primero no está preparado para grafos cíclicos ni objetos compartidos; el segundo, soluciona este problema; el tercero, soluciona un problema que tienen los dos anteriores, que es el de no tolerar cambios en las variables de instancia de las clases de objetos serializados.

Hay varios serializadores cuyo objetivo es intercambiar objetos entre los distintos dialectos de *Smalltalk: SIXX*[13] se apoya en la popularidad de XML para proponer un formato universal; *StOMP*[14] es una implementación de *MessagePack*, y por lo tanto rápido, binario y compacto; *SRP*[15] también es binario, compacto, muy flexible y con un diseño muy prolijo.

Persistencia de objetos del metanivel

Versionar y compartir clases y demás metaobjetos del ambiente es una necesidad en todo lenguaje orientado a objetos. Podemos decir que existen dos acercamientos

[8] http://msgpack.org/
[9] http://java.sun.com/developer/technicalArticles/Programming/serialization/
[10] http://www.ruby-doc.org/core-1.9.3/Marshal.html
[11] http://docs.python.org/library/pickle.html
[12] http://wiki.squeak.org/squeak/2316
[13] http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html
[14] http://www.squeaksource.com/STOMP.html
[15] http://sourceforge.net/projects/srp/

principales, el textual y el binario, aunque no son excluyentes. Con el acercamiento textual nos referimos a guardar el código fuente de la clase y sus métodos, para luego compilarlo en el momento de la materialización. En el acercamiento binario, el resultado de la compilación es serializado, por lo que la materialización es más rápida, al evitar el tiempo de compilación.

Persistir el código fuente original facilita el desarrollo en colaboración, poruqe es más sencillo comparar versiones o combinar cambios. Para la distribución a clientes puede ser inconveniente, por razones de copyright. Por otro lado, compartir código binario compilado no siempre es deseable o posible. Algunas tecnologías compilan el código fuente a un código intermedio de bytecodes, el cual es buen candidato a ser compartido como binario porque es independiente del hardware. De todas formas, hay una tendencia a que los lenguajes interpretados no compilen a bytecodes, sino directamente a código de máquina. En esos casos no tiene tanto sentido compartir los binarios. En lenguajes de scripting como *Ruby* o *JavaScript*, podría ser una desventaja estratégica distribuir binarios: es preferible que cada implementación del lenguaje tenga libertad para resolverlo a su manera.

A continuación comentaremos algunos ejemplos concretos:

En *Java*, el el compilador genera un archivo class a partir del código fuente, que luego un ClassLoader es capaz de materializar. El código fuente se adjunta en archivos textuales java. La distribución de paquetes se realiza en archivos comprimidos, que incluyen los archivos class y, opcionalmente, los archivos java. Esto permite la distribución eficiente de paquetes en archivos binarios, y también compartir código fuente para el desarrollo.

En *JavaScript* el código se comparte de forma textual, a pesar de que en su escenario principal de uso –aplicaciones web– esto implica algunas restricciones importantes en cuanto a la escalabilidad del código: una aplicación con mucho código hace lenta la carga de las páginas web.

Ruby es un lenguaje de scripting que cuenta con diversas implementaciones, pero no tiene un formato binario único para compartir código, a pesar de que algunas implementaciones compilan a bytecodes, como por ejemplo *Rubinius*, *YAML*, *MagLev* y *SmallRuby*. Utiliza formato textual de script para la persistencia.

En *Python*, *Marshal*[16] es el mecanismo nativo para la persistencia binaria de código compilado, en archivos con extensión pyc, cuyo contenido es evaluado en la materialización.

En *Oberon* se ha experimentado con *slim binaries* [10], que se basan en la idea de persistir los árboles de sintaxis abstracta (AST), producto intermedio de la compilación.

Dentro del mundo de *Smalltalk* también hay mucho para enumerar, pero nos limitaremos a destacar algunos formatos y serializadores.

En las distintas implementaciones de *Smalltalk* hay variantes de *FileOut*, un formato textual para compartir código basado en scripting, donde la materialización se realiza evaluando con el compilador al contenido del archivo serializado. El formato consiste en una serie de scripts separados por el signo "!". En *Squeak* y *Pharo*, este formato es muy popular, con herramientas de versionado de código como *ChangeSet*[17]

[16] http://docs.python.org/release/2.5/lib/module-marshal.html
[17] http://wiki.squeak.org/squeak/674

y *Monticello*[18].

Parcels [18], versionador de código en *VisualWorks Smalltalk*, utiliza archivos con formato binario para persistir métodos con bytecodes, opcionalmente acompañados con archivos de código fuente, en formato *XML*. De esta manera permite los escenarios de uso que mencionamos en la sección de *Java*: distribución eficiente de paquetes en archivos binarios y compartir código fuente para el desarrollo.

SMIX[19] se propuso utilizar el formato *XML* para compartir código entre dialectos. *Cypress*[20] persigue el mismo objetivo y también tiene formato textual, pero en lugar de estructurar los paquetes y clases dentro de un sólo archivo, lo hace en la estructura de directorio de repositorios de *GIT*, un software de control de versiones.

A.1.2 Nuestra elección

Los ambientes *Smalltalk* se diferencian de otras tecnologías orientadas a objetos con clasificación en que tanto las clases como los métodos o los contextos de ejecución son representados como objetos corrientes dentro del ambiente. Esto hace posible realizar experimentos en el dominio de la serialización que serían impensables en otras tecnologías.

Uno de los ambientes más prometedores hoy en día es *Pharo*. Surgió en 2008 como una ramificación de *Squeak*. Es desarrollado con licencia open-source, y cuenta con una comunidad de colaboradores muy activa.

En el presente trabajo nos proponemos describir el dominio de la serialización en base al estudio de su problemática, y en base al conocimiento adquirido, desarrollar un nuevo framework de serialización en *Pharo* con los siguientes objetivos principales:

- Flexibilidad. En un ambiente de objetos puro, donde clases, métodos, closures y contextos de ejecución están representados como objetos comunes y corrientes, un único serializador debe servir para la persistencia de cualquier objeto, sea del nivel base o del nivel meta. Se busca lograr un framework multipropósito con la capacidad de adecuarse a muy diversos casos de uso.

- Concretitud. Con el objetivo de lograr una reproducción de objetos rica y precisa, nos centraremos en una tecnología específica y no en el intercambio con otras tecnologías. Nos proponemos persistir objetos propios de la tecnología elegida, del nivel base o meta. Para ello, será necesario estudiar con total minuciosidad los detalles de la tecnología que hemos elegido.

- Eficiencia. El serializador debe ser útil en casos de uso reales, en donde la performance es esencial. Por lo tanto le daremos mucha importancia a hacer una implementación eficiente, y nos ocuparemos de hacer una evaluación comparativa que nos permita arribar a conclusiones sólidas.

- Diseño. El software creado debe cumplir con los criterios de buen diseño de objetos, y será implementado enteramente dentro del paradigma de objetos, sin recurrir a modificaciones a la máquina virtual. Además, la funcionalidad estará apoyada por una buena batería de casos de tests.

[18] http://wiki.squeak.org/squeak/1287
[19] http://www.mars.dti.ne.jp/~umejava/smalltalk/smix/
[20] https://github.com/CampSmalltalk/Cypress

Entendemos que ninguna de las herramientas que revisamos (la mayoría de las cuales mencionamos en la sección de Estado del Arte) cumple satisfactoriamente con todos estos objetivos.

A.2 Objetos en Pharo

Dado que la serialización se trata de capturar el estado de un objeto para luego recrearlo, y dado que nos interesa ser capaces de hacerlo con total detalle y con cualquier objeto del ambiente, vale la pena entonces empezar analizando minuciosamente qué tipos de objeto existen en la tecnología que hemos elegido y qué modalidades o problemas presentan para un serializador.

Hemos dividido a este capítulo en tres partes. En la primera, introducimos y desarrollamos algunos conceptos y propiedades importantes del ambiente; en la segunda, explicamos los tipos de objeto existentes en esta tecnología; en la tercera, indagamos en objetos del metanivel, de mayor complejidad.

A.2.1 Conceptos importantes

Identidad e igualdad

Dos relaciones fundamentales del paradigma y del dominio que estudiamos, son la identidad y la igualdad. Ambas están implementadas como mensajes en *Smalltalk*. La **identidad** (==) se da cuando el argumento y el receptor del mensaje son el mismo objeto. La **igualdad** (=) se da cuando el argumento y el receptor del mensaje representan al mismo concepto. La igualdad debería ser redefinida en las subclases, mientras que la identidad está implementada como un mensaje especial, que no puede ser redefinido.

Manejo especial de identidad

Algunos objetos representan entidades conceptualmente únicas en el ambiente. Salvo en el caso de las instancias de SmallInteger, de las que hablaremos más adelante en este capítulo, esta propiedad está implementada en *Pharo* de manera pura, desde el paradigma de objetos. Esto significa que al recrear objetos serializados, el proceso de materialización debe tener cuidado de no violar estos contratos, duplicando instancias que deberían ser únicas.

A continuación enumeraremos casos bien conocidos en *Pharo*, aunque trataremos en general este tema más adelante, entre los desafíos de la sección anterior.

Un caso elemental es el de nil, que representa a La Nada, y es la única instancia de UndefinedObject. Otro caso es el de true y false, que son instancias únicas de True y False, respectivamente. Esta propiedad de unicidad puede perderse utilizando el mensaje basicNew, que explicamos más adelante en este capítulo, y sirve para instanciar objetos de forma primitiva. Basta mostrar que las siguientes expresiones son falsas:

```
UndefinedObject basicNew == nil
True basicNew == true
```

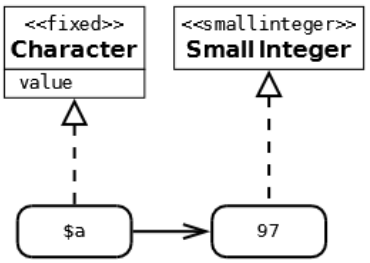

Fig. A.2: La clase Character mantiene una instancia única representando la letra a.

Como observamos en la Figura A.2, una instancia de Character tiene un código numérico como colaborador, que indica su valor *Unicode*. Los caracteres correspondientes a los primeros 256 códigos son únicos en el ambiente, mantenidos en un diccionario por la clase Character. En cambio, los demás caracteres sí permiten duplicados, y por lo tanto, la primera línea es verdadera y la segunda es falsa:

```
(Character value: 97) == (Character value: 97)
(Character value: 257) == (Character value: 257)
```

Una instancia de Symbol es una secuencia de caracteres única en el ambiente, de modo que es verdadero lo siguiente:

```
'unSímbolo' asSymbol == 'unSímbolo' asSymbol
```

Para implementarlo, la clase Symbol mantiene un conjunto débil de instancias.

Hash

El **hash** (hash) de un objeto es un número entero, relacionado con lo que representa conceptualmente ese objeto, definido de manera tal que cualquier par de objetos iguales necesariamente tendrán el mismo valor de hash. El **hash de identidad** (identityHash) de un objeto, en cambio, es un número entero relacionado con la identidad de ese objeto, que permanece constante a lo largo de su existencia. A menos que haya sido redefinido, el hash de un objeto es igual a su identityHash.

Preferentemente, estos enteros deben calcularse muy rápidamente y estar bien distribuidos, ya que son usados, por ejemplo, como índice para ubicar elementos en las HashedCollection. En la Figura A.3 mostramos la jerarquía de esas clases, que es voluminosa. Serializar cualquiera de estas colecciones con máximo detalle, y luego recrearla con precisión presenta una dificultad: Debemos suponer que los índices están desactualizados al materializar, ya que habían sido calculados en base al hash o identityHash. Preparadas para esta situación, estas colecciones implementan el mensaje rehash, que se ocupa de actualizarlas con los nuevos valores.

Variables compartidas

Con el motivo de que ciertos objetos se puedan acceder cómodamente desde los métodos del ambiente a través de nombres, hay varios tipos de variables compartidas, cada uno con distinto alcance. Las variables **globales** son accesibles desde cualquier

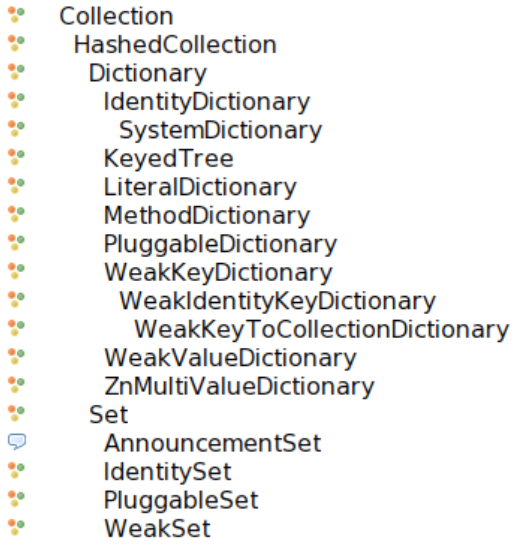

Fig. A.3: Jerarquía de HashedCollection

método; las de **clase**, desde los métodos de la clase que declara la variable o de sus subclases; las de **pool**, desde los métodos de las clases que declaren usar el *pool compartido* que contiene la variable.

Las variables compartidas están representadas en **asociaciones** que forman parte de los **diccionarios compartidos**. El nombre de la variable (por convención, un símbolo que comienza en mayúscula) es la **clave** (key) de la asociación, mientras que el **valor** (value) es el objeto al que apunta la variable. Cuando el compilador encuentra el nombre de una variable compartida en un método, la asociación correspondiente es incluida entre los literales. O sea que las asociaciones son referenciadas tanto en los diccionarios compartidos como de los métodos vinculados a ellos.

En la Figura A.4 ilustramos la gama completa de variables compartidas, mostrando al método tabWidth de la clase TextStyle, la cual declara a TextConstants como pool compartido, como puede apreciarse en la parte inferior de la figura. Ahí también mostramos que TextConstants es subclase de SharedPool. El método responde el valor de DefaultTab, que es una de las variables compartidas por TextConstants. Al estar centralizado en una única asociación compartida el valor de DefaultTab, si su valor cambiara para apuntar a otro objeto, el cambio se aplicaría en todos los métodos que la usan.

El **diccionario del sistema** contiene las asociaciones que representan a las variables globales del ambiente. En *Pharo*, se lo puede acceder mediante la expresión Smalltalk globals. Cada clase registrada en el ambiente tiene una variable global definida en este diccionario. En la parte superior de la figura remarcamos dos asociaciones globales: las de las clases TextStyle y TextConstants. Además de las clases, otros objetos únicos y vitales del sistema son registrados como variables globales, como Transcript, Smalltalk, Processor, Sensor, World, ActiveHand, Undeclared, y SystemOrganization. Pueden agregarse variables globales, aunque se considera una mala práctica en *Pharo*[21], recomendándose, en lo posible, el uso de variables de clase.

[21] Pharo by Example [4]: "Current practice is to strictly limit the use of global variables; it is

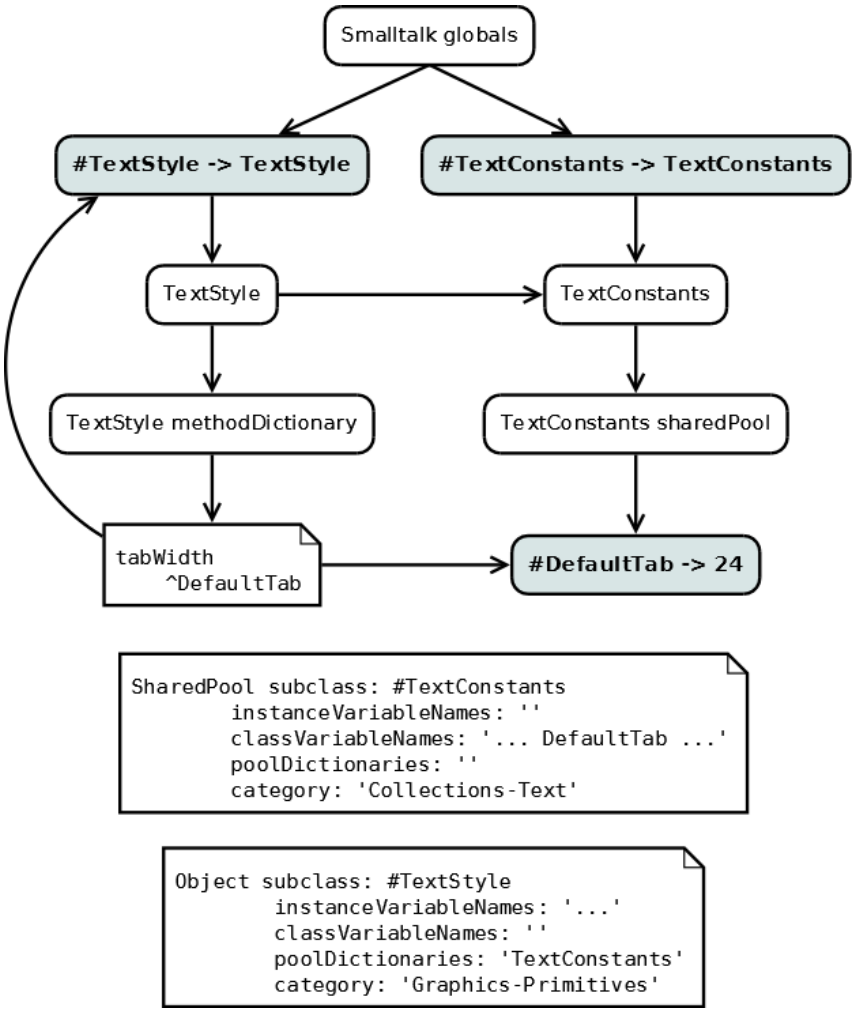

Fig. A.4: Variables compartidas

El **pool compartido de una clase** es un diccionario cuyas asociaciones representan a las variables *de clase*, que son compartidas en la jerarquía de subclases. En la figura, DefaultTab es una variable de clase de TextConstants, y su valor se asigna en el método de clase initialize, práctica habitual con este tipo de variables.

Un **pool compartido** en *Pharo* es una subclase de SharedPool cuyas asociaciones representan a las variables compartidas de ese pool. En la figura, DefaultTab es una variable que TextConstants comparte, y el método tabWidth utiliza. Tampoco se considera una buena práctica usar pools compartidos en *Pharo* ya que, aunque su *scope* es más restringido que el de las variables globales, sigue siendo preferible usar las variables de clase (o de instancia de clase).

A los efectos de la serialización, hay que tener en cuenta que en este mecanismo de variables compartidas es esencial la identidad de las asociaciones. Podemos pensar distintos escenarios de uso en los cuales este hecho tiene diferentes consecuencias. Supongamos que desde una herramienta de versionado binario de código estamos cargando en el ambiente al método tabWidth. Si TextConstants existe y define a la variable DefaultTab, debería usarse esa misma asociación como literal en la materialización. Una réplica exacta de tabWidth funciona de la misma manera que el original siempre y cuando las asociaciones compartidas no sean copias. En cambio, si TextConstants no define DefaultTab, lo más adecuado podría ser definirla nosotros mismos con algún valor por defecto. En otro escenario, podría ser TextConstants quien está ausente, y quizás lo adecuado sería dar un error al usuario, indicando la dependencia requerida.

Otro ejemplo es el de las variables globales de sistema que mencionamos más arriba (Transcript, Processor, etc.), que se puede dar por sentado que están definidas en toda imagen *Pharo*. En la Figura A.5, la asociación compartida de Transcript es literal del método event: de la clase ZnTranscriptLoader; independientemente de ese hecho, un objeto hipotético que denominamos aLogger conoce el valor de dicha asociación, es decir, a la instancia única en el ambiente de la clase ThreadSafeTranscript. Es claro que en el caso de materializar el método event:, la asociación debe ser la instancia ya existente en el ambiente y no una copia. Acabamos de dar un ejemplo muy similar en párrafos anteriores.

En esta figura incorporamos otro caso: aLogger es un objeto que tiene al Transcript como colaborador. Vamos a imaginar dos escenarios de serialización de este objeto. En el primero, probablemente el más típico, estamos interesados en recrear el estado de aLogger, pero queremos que el Transcript sea tratado como una global para que, en la materialización, aLogger tenga como colaborador al Transcript único de la imagen destino, y de esa manera pueda seguir funcionando, mostrando información en pantalla. En el segundo escenario, en cambio, hubo algún error grave en el sistema y por lo tanto nos interesa serializar al Transcript con total detalle, para analizar posteriormente lo ocurrido, reproduciéndolo minuciosamente.

En resumen, hemos mostrado con ejemplos que en algunos casos, pero no siempre, un serializador debería ser capaz de detectar en el grafo asociaciones compartidas y a los valores de las mismas, para que luego, al materializar, no sean recreadas en detalle, sino que deben encarnar objetos compartidos ya existentes en el ambiente.

---

usually better to use class instance variables or class variables, and to provide class methods to access them."

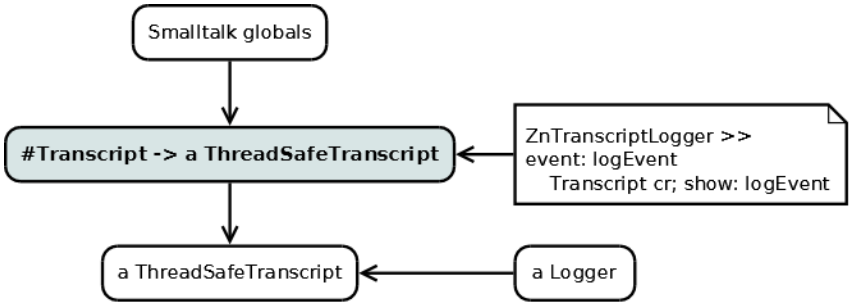

Fig. A.5: Asociación global y valor global

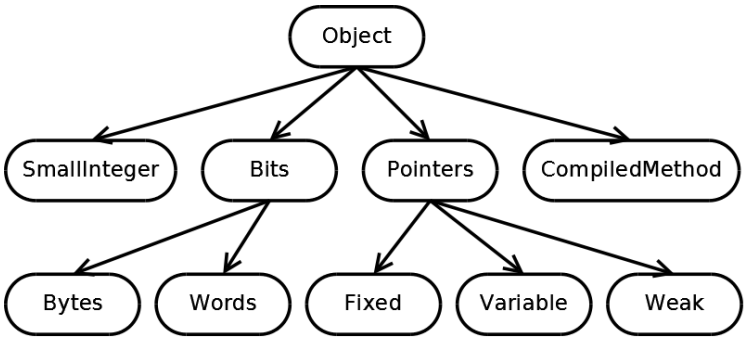

Fig. A.6: Formatos de objeto

A.2.2 Tipos de objetos

Presentamos en esta sección una taxonomía de objetos dada desde la implementación de la máquina virtual. La resumimos a modo introductorio en la Figura A.6, y la desarrollamos luego.

SmallInteger

Los números enteros entre *-1073741824* y *1073741823* tienen una representación especial: cada uno de estos enteros se codifica dentro de los 32 bits de un puntero a objeto normal. Esto significa que dichos enteros no ocupan realmente un espacio adicional en memoria, tal como cualquier otro objeto.

Este trato especial por parte de la máquina virtual, tiene una consecuencia que nos interesa a los efectos de la serialización: para estos enteros es imposible hacer diferencia entre los conceptos de igualdad e identidad. No hay manera de construir un SmallInteger que represente al 5 y que sea no-idéntico a otro SmallInteger también representando al 5. No hay en *Pharo* otros objetos que tengan esta propiedad: hasta objetos tan elementales como nil, true y false admiten ser duplicados, violando el contrato de ser instancias únicas en el ambiente.

Objetos de punteros

Son los objetos que pueden conocer a otros objetos, también llamados colaboradores internos, a través de **variables de instancia**. Estas variables pueden ser **de nombre** o **indexadas**.

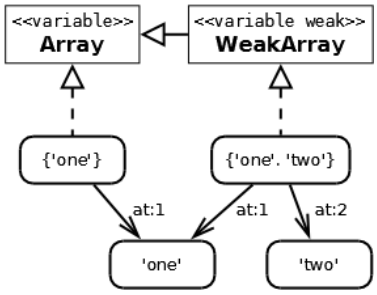

Fig. A.7: Objetos variable pointers

Las variables de nombre no pueden ser más de *254*, y se definen a nivel de clase. Por lo tanto, todas las instancias de una misma clase necesariamente tendrán la misma cantidad de variables de nombre. Los objetos de tipo **fixed pointers** sólo tienen variables de este tipo. Capturar el estado para serializar estos objetos generalmente requerirá capturar el estado de los colaboradores a los que éste apunta. Podemos acceder a estas variables mediante el protocolo de introspección para leerlas (instVarAt:) y para escribirlas (instVarAt:put:), disponibles en la clase Object. Para la materialización, estos objetos pueden ser instanciados enviando el mensaje basicNew a su clase.

Denominamos de tipo **variable pointers** a los objetos que, además de variables de nombre, tienen variables indexadas. Mientras que las variables de nombre de un objeto se definen en su clase, la cantidad de variables indexadas se define en el momento de la instanciación. Para ello, disponemos del mensaje basicNew:, que nos va a ser útil en la materialización. La cantidad de variables indexadas de una instancia la obtenemos con el mensaje basicSize.

Previsiblemente, para serializar este tipo de objetos se extiende lo dicho para los fixed pointers, capturando además el estado de los colaboradores apuntados por sus variables indexadas. La clase Object brinda protocolo de introspección específico para acceder a las variables indexadas: basicAt: y basicAt:put:.

Dentro del tipo de objetos variable pointers existen los **weak variable pointers**, que se diferencian en que sus variables indexadas son *referencias débiles*, en el sentido de que no impiden que sus colaboradores sean recolectados por el mecanismo de *garbage collection*. En tal caso, las variables indexadas apuntando a colaboradores recolectados cambian automáticamente para apuntar a nil. Más allá de detalles implementativos, la semántica es que estos objetos conocen a sus colaboradores sólo mientras estos existan, pero sin impedirles dejar de existir.

El protocolo de introspección para estos objetos es el mismo que para los variable pointers, es decir, son polimórficos en este sentido. Si pensamos en cómo serializarlos , ya no es tan evidente como en los casos anteriores cómo capturar su estado. Una posibilidad es hacerlo tal como si fuera un objeto variable común y corriente. Aunque quizás sea más adecuado que, para cada variable indexada, se capture un puntero a nil (como si hubiese sido recolectado) a menos que el colaborador correspondiente sea apuntado de forma no-weak por otro objeto del grafo. Para ilustrar esta última idea, observemos la Figura A.7, donde a la derecha mostramos una instancia de WeakArray que apunta a 'one' y a 'two'. Si serializáramos esta instancia, lo haríamos como {nil. nil}. En cambio, si serializáramos un objeto que también incluyese en su grafo a la

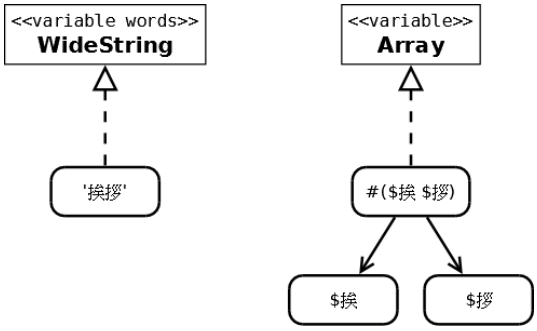

Fig. A.8: Un WideString economiza espacio en memoria para representar la misma secuencia de caracteres *Unicode* pero sin referenciar instancias de Character.

instancia de Array (que apunta a 'one'), entonces el arreglo débil se capturaría como {'one'. nil}.

Objetos variables de bits

Estos objetos representan secuencias de objetos simples, tales que su estado puede codificarse en unos pocos bits: los objetos **variable bytes**, en 8 bits; los **variable words**, en 32 bits. La finalidad de su existencia es la de economizar espacio en memoria y optimizar operaciones masivas. Por ejemplo, las instancias de ByteString son secuencias de caracteres de 8 bits y las instancias de ColorArray son secuencias de colores de 32 bits de profundidad. Este tipo de objetos no puede tener variables de nombre.

A pesar de su representación especial, estos objetos son polimórficos con los de tipo variable pointers, excepto en que restringen qué tipos de objeto se les permite asignar como colaboradores. En la Figura A.8 mostramos la diferencia entre un WideString y un Array representando la misma secuencia de caracteres *Unicode*.

Para la serialización, nos será beneficioso aprovechar los métodos disponibles ya optimizados mediante primitivas para leer y escribir de manera optimizada este tipo de objetos. Para los objetos de words hay que tener en cuenta que estas operaciones se realizan en *Pharo* con el *endianness*[22] del hardware corriente, y por lo tanto queda a cargo del serializador la responsabilidad de corregir posibles incompatibilidades.

Métodos compilados

Un **método compilado** (CompiledMethod) es un objeto variable de bytes que, sin embargo, conoce a otros objetos como si fuera un objeto de punteros: sus literales. En la Figura A.9 ilustramos con el método and: de la clase True, cuyos literales son el símbolo selector del método, y la asociación de diccionario que indica a qué clase pertenece. La máquina virtual interpreta la secuencia de bytes de un método de esta manera:

1. **header** (4 bytes). Codifica datos tales como la cantidad de literales, la cantidad de variables temporales, la cantidad de argumentos, o si es un método primitivo.

[22] El orden de los bytes en el word. Los sistemas *big-endian* codifican primero a su byte más significativo, mientras que los *little-endian* codifican primero al menos significativo.

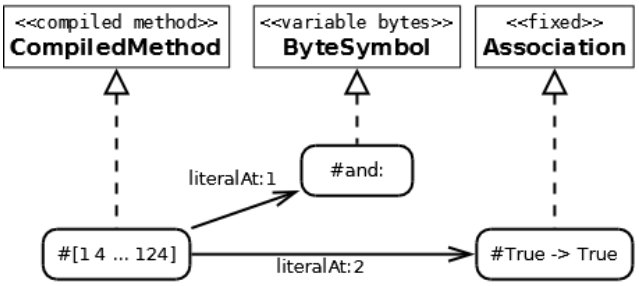

Fig. A.9: Un método es una instancia variable de bytes que, sin embargo, conoce a otros objetos: sus literales.

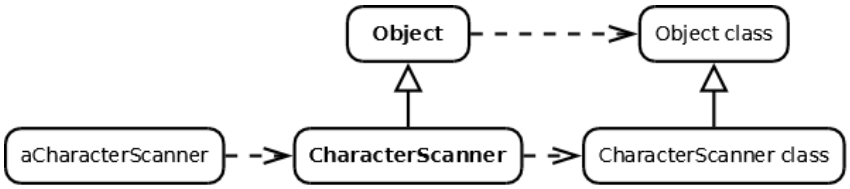

Fig. A.10: Todo objeto es instancia de una clase, la cual es también un objeto. La flecha de líneas punteadas significa "es instancia de", mientras que la otra flecha significa "es subclase de".

2. **literals** (4 bytes cada uno). La máquina virtual los interpreta como punteros a objetos.

3. **bytecodes** (variable). Son normalmente generados por el compilador, e interpretados por la máquina virtual durante la ejecución.

4. **trailer** (variable). Se usa para codificar información sobre el código fuente del método. Normalmente indica en qué archivo y en qué posición del archivo está el código fuente.

A.2.3 Metanivel

Clases y Metaclases

En *Smalltalk*, todo objeto es instancia de una clase, la cual es también un objeto. Ilustramos estas afirmaciones en la Figura A.10, con la clase CharacterScanner, subclase de Object, e instancia de su metaclase.

En la Figura A.11 mostramos la gran complejidad que tiene una clase en *Pharo*. Se trata de CharacterScanner junto a sus colaboradores. Esto no quita que sea un objeto capaz de ser serializado y materializado como cualquier otro. Pensemos en el caso hipotético en el que queremos persistir con detalle a esta clase y a su metaclase, pero sin persistir en detalle a los demás objetos compartidos del ambiente.

El mecanismo actualmente disponible en *Pharo* para realizar la serialización y materialización es el de fileOut y fileIn, comentado en el capítulo anterior. El primero, escribe en un archivo la definición de clase, tanto de CharacterScanner como de su metaclase, además del código fuente de sus métodos. El segundo, lee el archivo,

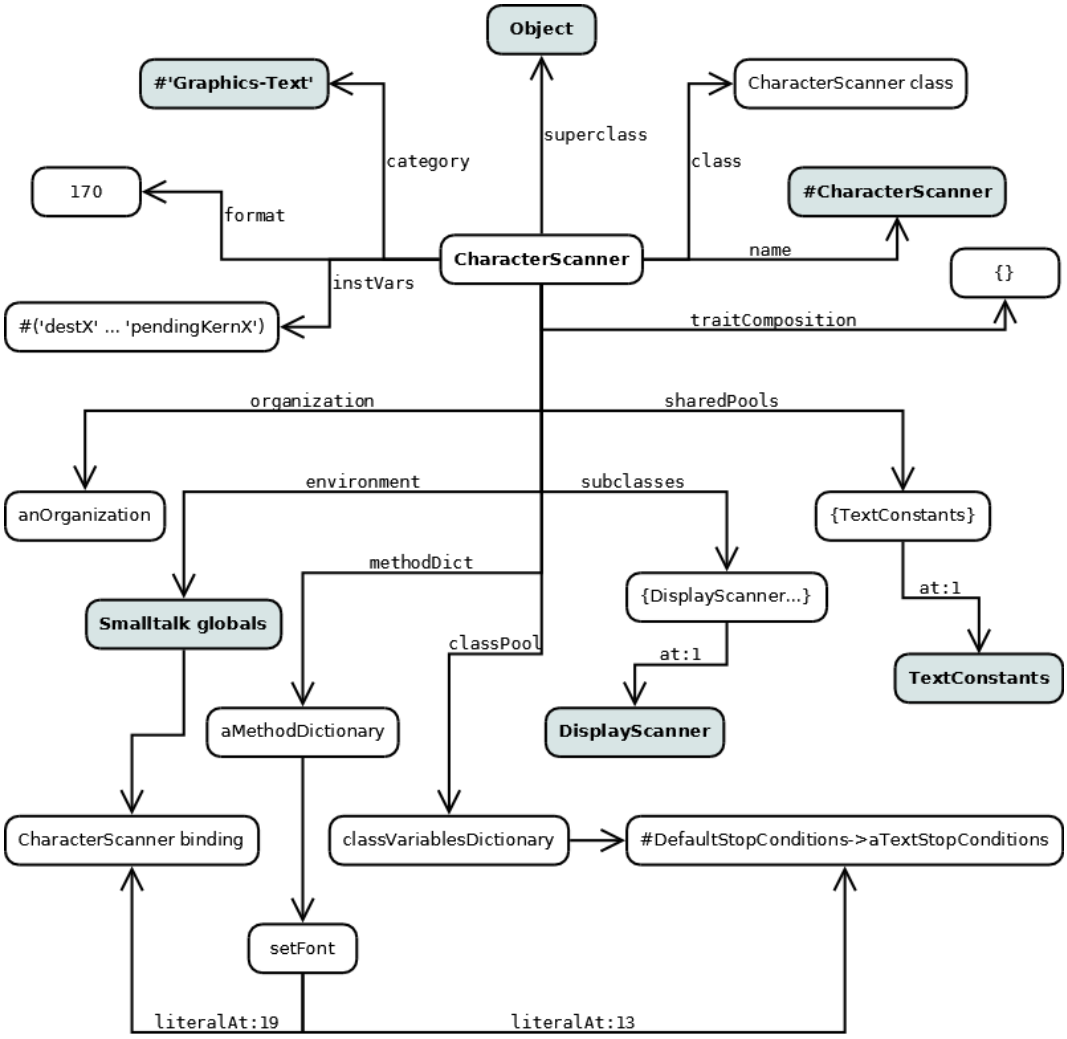

Fig. A.11: La clase CharacterScanner, con sus colaboradores. Para serializarla, una posibilidad es considerar a los objetos resaltados como compartidos.

ejecutando las definiciones de clase, y compilando el código fuente de los métodos. Una instancia de ClassBuilder es la responsable de validar la definición de clase, de construir la clase, de instalarla en el *diccionario de sistema*, y de enviar notificaciones pertinentes acerca de la operación. Luego, el proceso de compilación se ocupa de interpretar el código fuente, compilarlo a *bytecodes*, encontrar los literales correspondientes, construir las instancias de CompiledMethod, e instalarlas en la clase recientemente instalada.

Sin embargo, nuestra intención es distinta a la de ese mecanismo: queremos ser capaces de materializar una clase (si se dan las condiciones necesarias), sin hacer uso del ClassBuilder, ni del compilador, sino hacerlo reproduciendo minuciosamente la clase, recreando el grafo colaboradores. Las condiciones necesarias aludidas se refieren al ambiente en donde se pretende materializar, que debe contar con los objetos compartidos que precisa el grafo serializado, y en particular las clases no deben haber cambiado su formato ni su lista de variables, ya que estos hechos podrían invalidar los *métodos compilados* originales.

Traits

Un **trait** es una colección de métodos que puede ser incluida en el comportamiento de una clase sin la necesidad de herencia. Dicho de otra forma, posibilitan a las clases compartir métodos de manera transversal a sus jerarquías. En la Figura A.12 observamos a TaddTest, un *trait* que contiene *tests* para probar variantes de la funcionalidad de agregar elementos a una colección.

Una clase puede usar no solamente un *trait*, sino una composición de ellos. En la Figura A.13 podemos apreciar el ejemplo de BagTest, que en su definición especifica una expresión de composición de *traits*, que se construye mediante ciertas operaciones (+ y -, y @, aunque no aparece en este ejemplo). Dicha composición se representa con una estructura de objetos que sigue, el patrón de diseño *Composite*.

De todas maneras, la dificultad para serializar una composición de *traits* está centrada en la complejidad de los *traits*, semejante a la de serializar clases. El mecanismo de fileIn y fileOut con *traits* funciona de forma muy similar que con clases. La misma diferencia de intención tenemos en este caso: queremos ser capaces de materializar un *trait* sin hacer uso del ClassBuilder ni del compilador, queremos hacerlo recreando minuciosamente el grafo colaboradores.

Pila de ejecución

Un **contexto de ejecución** (MethodContext) representa cierto estado de ejecución, en dos situaciones: en la activación de un método (CompiledMethod) tras el envío de un mensaje, o en la activación de un bloque (BlockClosure), en la evaluación del mismo. Es un objeto de tipo *variable pointers*, aunque con particularidades importantes. Primero, describamos lo que sus variables representan:

- receiver, es el objeto que encarna a la *pseudo-variable*[23] self durante la ejecución.
- method, que contiene los *bytecodes* en ejecución, según el caso es el método activado, o el método que contiene al bloque activado.

[23] Las *pseudo-variables* en *Pharo* son self, super, thisContext, nil, true, y false.

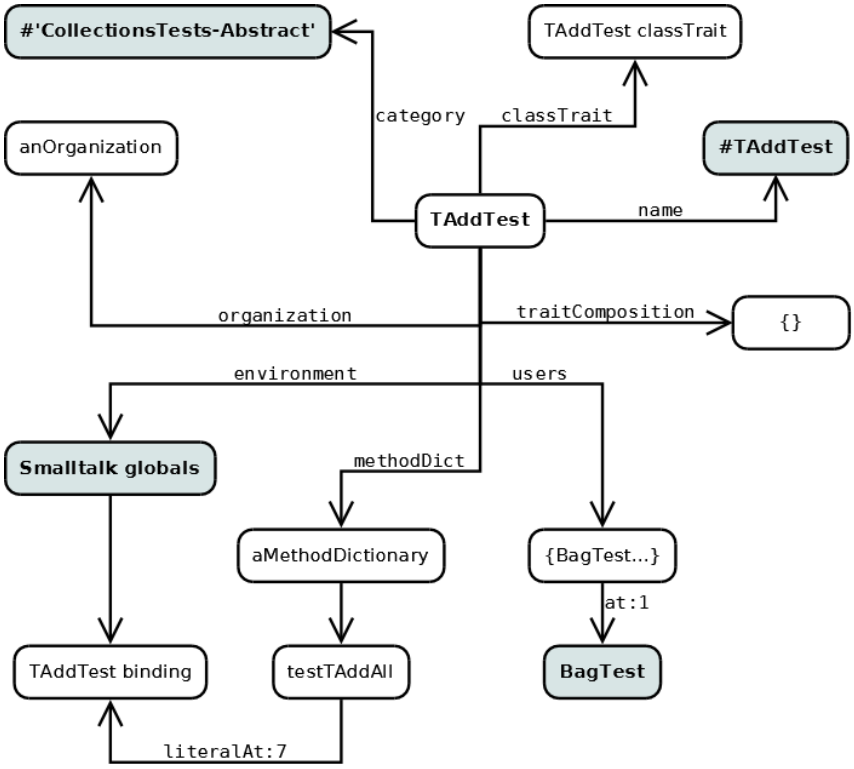

Fig. A.12: El trait TAddTest, con sus colaboradores. Para serializarlo, una posibilidad es considerar a los objetos resaltados como compartidos.

```
CollectionRootTest subclass: #BagTest
 uses: TAddTest + TIncludesWithIdentityCheckTest + TCloneTest + TCopyTest +
  TSetArithmetic + TConvertTest + TAsStringCommaAndDelimiterTest +
  TRemoveForMultiplenessTest + TPrintTest + TConvertAsSortedTest +
  TConvertAsSetForMultiplinessTest + TConcatenationTest +
  TStructuralEqualityTest + TCreationWithTest - #testOfSize +
  TOccurrencesForMultiplinessTest
 instanceVariableNames: '...'
 classVariableNames: ''
 poolDictionaries: ''
 category: 'CollectionsTests-Unordered'
```

Fig. A.13: Ejemplo de definición de clase con expresión de composición de traits.

- pc, el índice en el método del próximo *bytecode* a ejecutar.
- stackp, la posición del tope de la pila de ejecución.
- closureOrNil, que según el caso, puede ser el bloque activado, o nil.
- Como *variables indexadas* están los argumentos del contexto, y las variables temporales.

El tamaño con el que se crean las instancias de MethodContext está restringido a dos valores: 16 ó 56 variables indexadas. Dicho número corresponde al frameSize del método, que depende de la cantidad de argumentos del contexto y de variables temporales necesarias, y está codificado en el encabezado del método. La máquina virtual oculta al ambiente el verdadero tamaño de las instancias de MethodContext, que varía a lo largo de la ejecución, de acuerdo al stackp.

En la Figura A.14 mostramos un fragmento de un instante de la pila de ejecución, al evaluar con *"do it"* en un *Workspace* a la expresión:

```
#(3 4 5) select: [:n| n > 4]
```

El modo de funcionamiento de dicho comando es compilar provisoriamente a la expresión en la clase UndefinedObject con el selector DoIt, y removerlo al terminar la ejecución. Para la serialización podríamos considerar compartidos tanto al objeto nil, como al método Array>>select:, pero no así al método UndefinedObject>>DoIt, puesto que su fugacidad nos hace suponer que el mismo no estará presente al momento de materialización. En consecuencia, al serializar una pila de ejecución, debe incluirse en el grafo a este método con total detalle. Por otra parte, si se codifica al método Array>>select: como objeto compartido, se corre el riesgo de que al momento de materializar, el mismo haya cambiado, y que los *bytecodes* apuntados por pc sean obsoletos. Esto podría ocasionar problemas serios en el ambiente, por lo tanto deberían tomarse medidas como incluir un código de verificación para los *bytecodes*.

A.3 Desafíos del Dominio

Habiendo realizado en los capítulos anteriores un estudio de escenarios y mecanismos de serialización existentes y luego un análisis detallado de la tecnología de objetos elegida y las problemáticas concretas que presenta para la serialización, creemos válido formular en este capítulo una descripción más genérica de las problemáticas del dominio de la serialización de objetos.

A.3.1 Identidad

Una de las principales problemáticas de este dominio es materializar los objetos con la identidad correcta. Hemos explicado casos bien conocidos de objetos compartidos en *Pharo*. En los siguientes párrafos los repasaremos brevemente y agregaremos otras formas habituales de compartir objetos. Excepto los objetos true, false y nil, que son accesibles por medio de *pseudo-variables*, encontraremos que en general los objetos compartidos se obtienen a partir del *diccionario del sistema* (Smalltalk).

Las *variables globales* se pueden obtener enviando a Smalltalk el mensaje associationAt: con el nombre de la variable como parámetro. Otra posibilidad es utilizar el

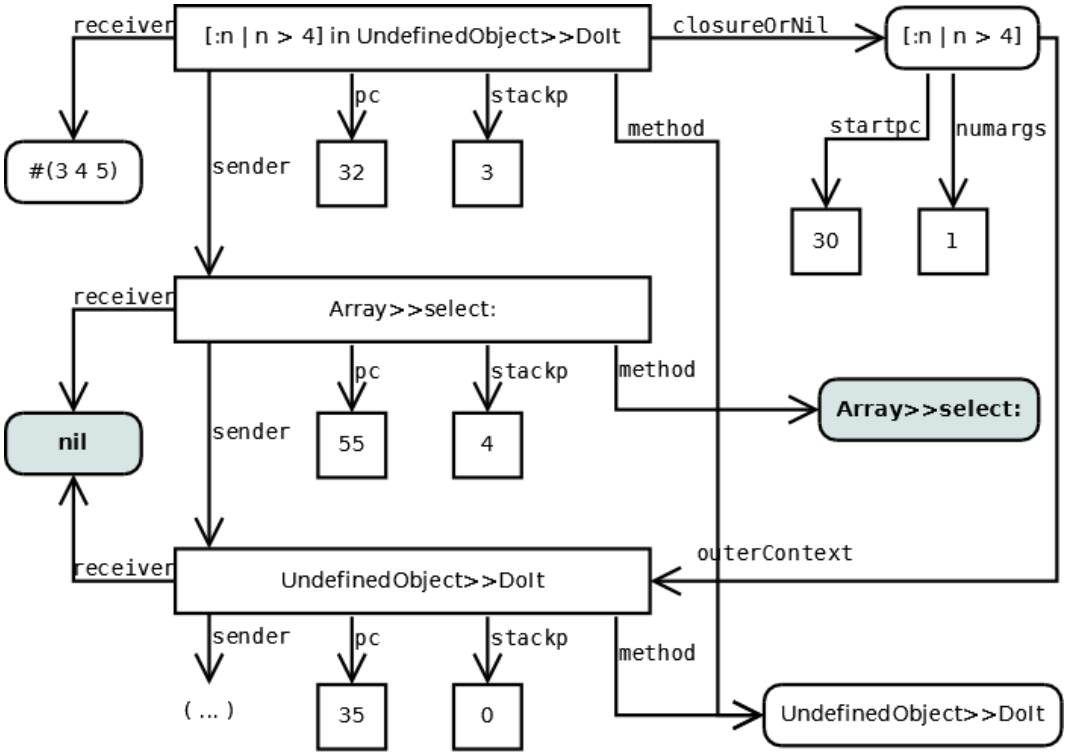

Fig. A.14: Fragmento de un instante de la pila de ejecución, al evaluar con "do it" en un Workspace a la expresión `#(3 4 5) select: [:n| n > 4]`. A los objetos resaltados, o sea a `nil` y al método `Array>>select:`, podríamos considerarlos compartidos, pero no así al método `UndefinedObject>>DoIt`, ya que fue compilado por el comando "do it" en esa clase temporariamente, y será removido al terminar la ejecución.

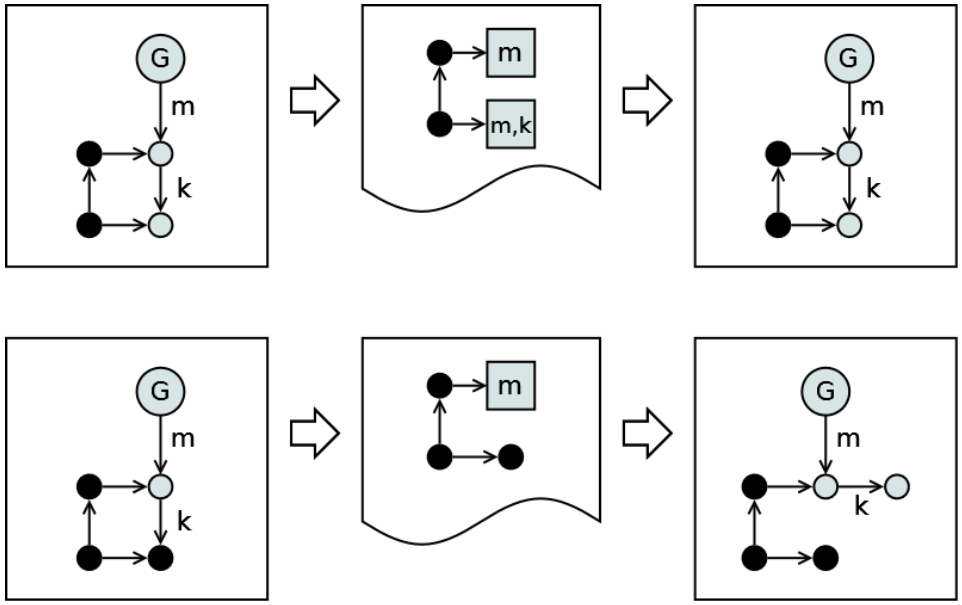

Fig. A.15: Dos casos de serialización y materialización de un grafo con objetos compartidos en el ambiente. Los objetos en negro reproducen detalladamente su estado; los claros se consideran compartidos y se obtienen a partir de G, que simboliza a algún objeto bien conocido del ambiente. Las letras k y m simbolizan maneras de obtener a los objetos compartidos.

mensaje associationOrUndeclaredAt:, que la define con un valor por defecto en caso de no estar previamente definida.

Las *clases* (que son variables globales) se obtienen enviando a Smalltalk el mensaje at: con el nombre de clase como parámetro. En general, podemos esperar que al momento de materialización ya existan en el ambiente.

En el patrón de diseño *Singleton*, la instancia única de una clase se obtiene a través de un mensaje que la misma clase brinda. La instancia podría no existir antes del envío del mensaje si se crea de manera *lazy*, como es habitual. En general, podemos esperar que dicha clase sea global y, por lo tanto, también puede obtenerse la instancia indirectamente a partir de Smalltalk.

En el patrón *Flyweight*, un conjunto de instancias son compartidas por medio de un objeto que actúa como *factory*, que normalmente es una clase.

En la Figura A.15 esquematizamos el patrón que siguen los casos descriptos, con dos casos de serialización y materialización de un grafo con objetos compartidos en el ambiente. Los objetos en negro reproducen detalladamente su estado; los claros se consideran compartidos y se obtienen a partir de G, que simboliza a algún objeto bien conocido del ambiente. Las letras k y m simbolizan maneras de obtener los objetos compartidos a partir de otros. En el ejemplo de la parte superior, el grafo contiene dos objetos considerados compartidos en la serialización, y por lo tanto los mostramos codificados como [m] y como [m,k], simbolizando que esa información podrá ser luego decodificada, para obtener los objetos compartidos correspondientes en el ambiente de materialización.

En la parte inferior mostramos un eventual problema que aparece cuando, al serializar al objeto [m,k], éste no es considerado compartido y, por lo tanto, es recreado

en una nueva instancia en lugar de encarnar en la otra ya existente en el ambiente. Dependiendo del escenario de uso, esto podría ser grave. Una posible solución general es considerar compartida a toda la clausura transitiva de objetos apuntados a partir de Smalltalk. Esa es la estrategia que utiliza ImageSegment, que por razones de eficiencia la implementa en la *máquina virtual*, re-usando parte del mecanismo de garbage collection. Otra solución que puede ser suficiente es la de brindar al usuario la posibilidad de configurar el mecanismo, adaptándolo a su escenario.

Instanciando el esquema en un caso concreto, podemos pensar en la clase Color, que implementa el patrón *Flyweight* para los colores más habituales. Por ejemplo, el color blanco se obtiene mediante el envío del mensaje white a Color. Sin embargo, no es conceptualmente equivocado crear una instancia no-idéntica que represente al color blanco, mediante el mensaje r:g:b:. La primer expresión es verdadera y la segunda falsa:

```
(Color r: 1 g: 1 b: 1) = Color white
(Color r: 1 g: 1 b: 1) == Color white
```

Supongamos que el grafo a serializar incluye a la clase Color, y al objeto Color white. Podemos esperar que, a menos que se indique lo contrario, la clase Color sea considerada objeto compartido (en el esquema, el objeto [m]). Por otro lado, podríamos esperar que el Color white (el objeto [m,k]), sea recreado por duplicado, a menos que se configure especialmente al serializador para que lo materialice enviando el mensaje white a Color.

A.3.2 Flexibilidad

A veces existen múltiples formas de capturar un mismo objeto. Hemos hablado de los objetos compartidos, que en realidad pueden considerarse compartidos en la mayoría de los casos, pero que en otros casos podría ser más útil capturar y reproducir su estado minuciosamente. Un caso es el de una clase, que puede considerarse un objeto compartido y codificarse sólo con su nombre, que servirá como identificación para obtenerlo en el ambiente de materialización. Otro caso es el de las asociaciones compartidas que, más precisamente, son instancias de la clase Association que pertenecen a diccionarios compartidos. Este ejemplo sirve para hacer notar que el condicionante para decidir cómo serializar un objeto, no siempre es homogéneo para todas las instancias de una clase y, por lo tanto, para algunos casos es deseable contar con mecanismos centrados en objetos y no en clases, para definir los criterios utilizados.

No es única, tampoco, la forma de capturar el estado de un objeto. Para explicarlo, tomemos el caso de OrderedCollection. En *Smalltalk* en general está establecido que esa clase representa secuencias de objetos, conservando el orden en que fueron agregados, y su capacidad crece por demanda, sin poner límite a la cantidad de objetos que puede albergar. Además, se pueden suponer ciertas características particulares de complejidad algorítmica en sus operaciones básicas diferentes, por ejemplo, a las de LinkedList, que está implementada como una *lista encadenada simple*.

A pesar del acuerdo en relación a qué representa una OrderedCollection, no hay una única implementación establecida para esta clase, ni siquiera una representación única. En la Figura A.16 mostramos dos distintas, la de *Pharo* y la de *VisualWorks*[24],

[24] Tomado de la versión 7.7.1, que data del año 2010.

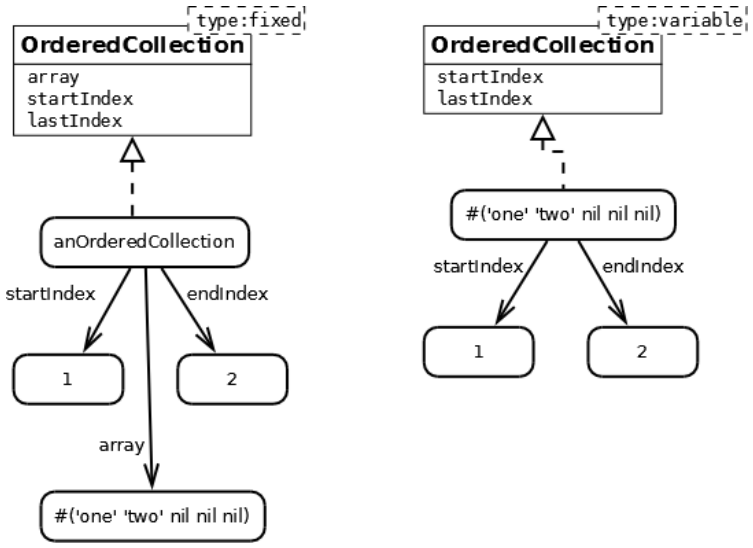

Fig. A.16: La lista ordenada con los strings 'one' y 'two', representada en dos implementaciones distintas de OrderedCollection, de *Pharo* y de *VisualWorks*.

las cuales no sólo difieren en las variables de nombre, sino también en el tipo de clase: en la primera, la instancia es *fixed pointers*, y sus elementos están en una instancia de Array; en la segunda, la instancia es *variable pointers* y conoce directamente a sus elementos, a través de variables indexadas.

Para serializar una instancia de cualquiera de estas listas, bien podría interesarnos hacerlo con total detalle para materializarla con precisión, como instancia de la misma clase original, y manteniendo el tamaño original de los objetos *variable pointers*. Pero también podría interesarnos reproducirla menos literalmente y capturar lo que está representando más conceptualmente, es decir, serializar sus elementos en el orden adecuado, para luego, a la hora de materializar, utilizar la implementación de lista ordenada que esté disponible en el ambiente, abstrayéndose de los detalles implementativos y brindando mayor compatibilidad.

Los párrafos precedentes sirven para mostrar por qué un mecanismo de serialización de propósitos generales debería ser capaz de adaptarse a distintos escenarios, varios de los cuales pueden coexistir en un mismo ambiente. Resaltamos este hecho porque hemos observado varios serializadores existentes que solo brindan la posibilidad de hacer adaptaciones estáticas en el ambiente, pero están limitados a que cada tipo de objeto sea capturado de una única manera. Además, a menudo la solución que dan para la adaptación está orientada a clases y no a objetos individuales, utilizando el patrón de diseño *template methods*.

A.3.3 Grafos Cíclicos

Resulta natural que los grafos de objetos a serializar puedan tener ciclos, y por lo tanto es necesario detectarlos durante la iteración. Cumplir con este requerimiento manteniendo la eficiencia de tiempo y espacio no es trivial. Una solución sencilla es mantener un conjunto de los objetos ya iterados, con el cual verificar si cada objeto del grafo que aparece al recorrerlo ya fue previamente procesado o no. Pero pueden aprovecharse algunas particularidades para optimizar este mecanismo, tales como

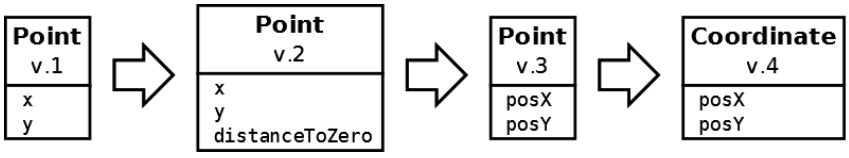

Fig. A.17: Cambios en el metanivel

que ningún String o SmallInteger puede generar un ciclo.

A.3.4 Cambios en el Metanivel

En un ambiente con clasificación, como lo es *Smalltalk*, el estado de cada objeto está regido por su clase, que describe aspectos tales como la cantidad de colaboradores y con qué nombre se los manipula. Dado que esta meta-descripción puede mutar en el tiempo, nos interesa poder materializar un objeto que fue serializado con una meta-descripción distinta a la actual. En la Figura A.17 mostramos un caso de evolución de la clase Point.

Supongamos que serializamos un punto cuando la versión 1 era vigente. Luego la clase muta a su versión 2, agregando una variable de instancia. Este caso podría resolverse fácilmente poniendo nil como valor inicial, ya que se acostumbra utilizar dicho valor para significar "no inicializado".

Ahora supongamos que serializamos en versión 2 y materializamos en la versión 3. Encontramos que se realizaron dos tipos de cambio. Por un lado, hubo una eliminación de variable, y por otro, un renombre de variables. El primer caso es sencillo de resolver automáticamente, ya que basta con ignorar al colaborador que había sido capturado durante la serialización. El segundo caso requeriría información adicional aportada por el usuario o el ambiente, que indique que x cambió a posX, e y cambió a posY.

Por último, en la cuarta versión hay un cambio en el nombre mismo de la clase. En este caso también sería necesario usar información adicional para materializar correctamente la instancia serializada.

A.3.5 Especializaciones

Es deseable darle al desarrollador la libertad de modificar la forma genérica en que se serializan y se materializan los objetos, como hemos mostrado en las secciones Identidad y Flexibilidad. Para muchos casos, debería ser suficiente con definir estas especializaciones a nivel de clase, para aplicarlas por igual a todas sus instancias, aunque en otros casos es interesante poder aplicarlas a nivel de objeto, de manera transversal a las clases.

A continuación enumeraremos varios tipos de especializaciones posibles que, según hemos observado, los serializadores suelen permitir. A grandes rasgos se dividen en: qué información se extrae del objeto, cómo se codifica y decodifica esa información en forma de secuencia, y cómo se usa esa información para reproducir al objeto en el ambiente de materialización.

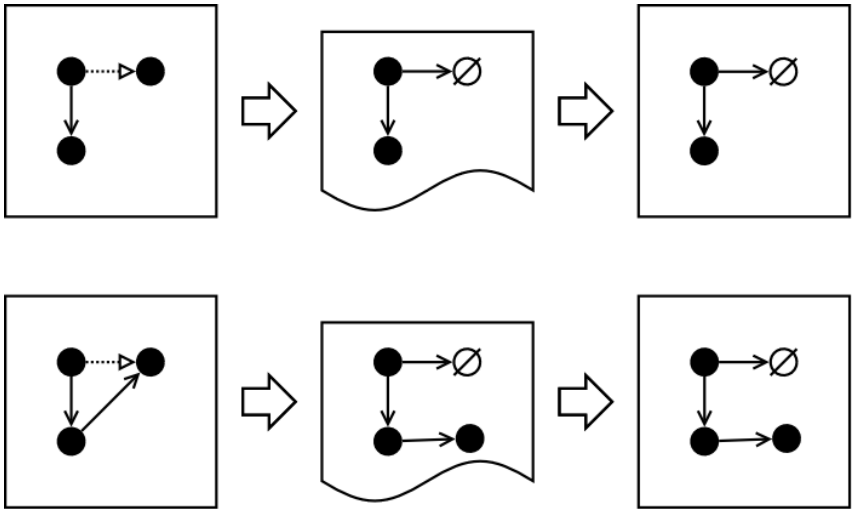

Fig. A.18: Ignorando variables de un objeto. En la parte superior mostramos un caso simple donde la colaboración ignorada se indica con la flecha punteada. En la parte inferior, una variante del ejemplo anterior, con un efecto posiblemente indeseable, donde se pierde consistencia en el grafo. El objeto tachado simboliza a nil.

Ignorar colaboradores. Al serializar, muchas veces resulta cómodo o necesario excluir ciertas variables del estado de un objeto. Puede verse como una operación de podado del grafo, evitando serializar ramas innecesarias. Al materializar, lo habitual es recrear dichas variables con un valor inicial nulo (nil), para inicializarlo por demanda, de manera *lazy*.

En la parte superior de la Figura A.18, mostramos un ejemplo simple donde se ignora una relación de colaboración. En la parte inferior hay una variante del ejemplo anterior, en donde la colaboración es ignorada pero, sin embargo, el colaborador no es excluido del grafo serializado. Allí señalamos un hecho a tener en cuenta: ignorar una colaboración significa quitar un eje del grafo, pero no necesariamente un nodo. Debemos tener presente que podrían generarse inconsistencias en algún caso.

Sustitución en serialización. Durante la serialización, en lugar de capturarse el estado de un objeto, se captura el de otro, que lo reemplaza a los efectos de la serialización. El grafo original permanece intacto en el ambiente.

Visto como una operación de grafo, puede pensarse como reemplazar a un nodo por otro, o si se quiere, en una analogía botánica, como injertar una rama en lugar de otra. En la parte superior de la Figura A.19 observamos un ejemplo sencillo, de un objeto que es sustituido por nil. Nuestra intención es contrastar con el ejemplo en la figura anterior, mostrando que en este caso definitivamente se eliminan nodos del grafo y no está el eventual problema de inconsistencia.

Sustitución en materialización. Dado un objeto que fue codificado durante una serialización, en la materialización ese objeto no pasa a formar parte del grafo reconstruido, sino que se pone en su lugar un reemplazo del mismo.

En la parte inferior de la Figura A.19, se sustituye un objeto por nil al momento de materializar.

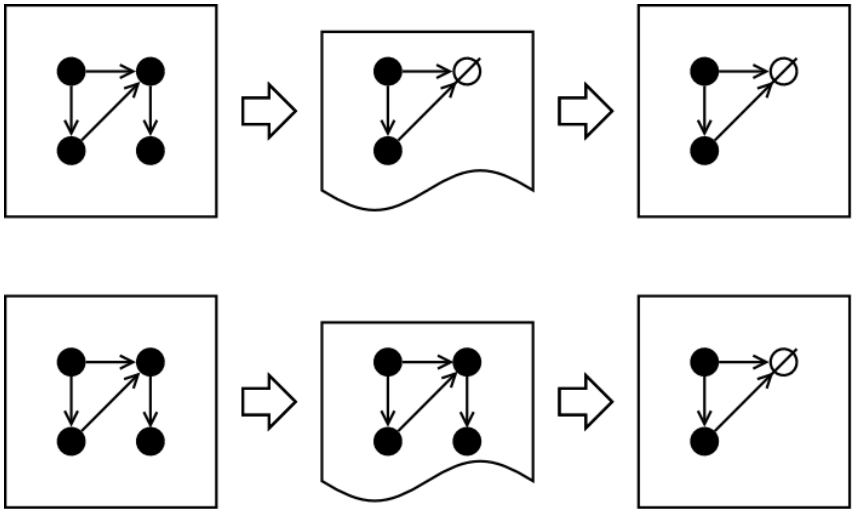

Fig. A.19: Sustituciones. En la parte superior, un objeto del grafo es sustituido por nil en la serialización. En el inferior, es sustituido por nil en la materialización.

Codificación primitiva. Por cuestiones de eficiencia, ya sea en tiempo o espacio, en algunos casos es importante brindar la posibilidad de especializar cómo se codifican ciertos objetos en la secuencia. Una instancia de Time representa un instance del día, como puede ser a las '5:19:45 pm'. Cada instancia tiene dos números enteros como colaboradores internos: seconds, con la cantidad de segundos transcurridos desde medianoche, y nanos, para más precisión. Claramente, conocer la cantidad de segundos es redundante si además conocemos a la cantidad de nanosegundos. También, calculando podemos ver que la cantidad de nanosegundos en un día puede expresarse en 6 bytes sin perder precisión. Estas instancias abundan en una imagen de *Pharo*, por lo tanto optimizar su serialización puede ser ventajoso en algunos casos. Entonces, podríamos especializar la serialización de las instancias de Time para codificar el estado completo en 6 bytes en lugar de hacerlo normalmente, lo que implicaría codificar referencias a dos colaboradores.

A.3.6 Integridad de la representación secuencial

A la hora de materializar un grafo de objetos a partir de una secuencia de bytes hay varias problemáticas a resolver.

Versionado. La secuencia podría haber sido generada por una versión diferente, ya sea anterior o posterior, del serializador. En caso de ser una anterior, es deseable que se brinde al usuario la posibilidad de leerlo (*back-compatibility*).

Daños. Como es bien sabido, una secuencia de bits puede sufrir daños durante su transmisión o persistencia. La corrupción de tan sólo un bit ya podría tener consecuencias. Es deseable no sólo poder detectar inconsistencias, sino también aislar y reparar el daño, y de esa forma recuperar el grafo serializado aunque sea de forma parcial.

Seguridad. La fuente de procedencia de la secuencia puede no ser del todo confiable en algunos casos y, dado que la materialización introduce un grafo de objetos en el ambiente, puede ser útil que el usuario pueda configurar ciertos criterios de seguridad. Por ejemplo, que los arreglos no puedan superar cierto tamaño, que la cantidad de objetos del grafo no supere cierta cota, o que no se introduzcan clases nuevas o comportamiento nuevo al ambiente.

A.3.7 Modalidades Alternativas de Lectura

En algunas situaciones puede ser útil leer una secuencia de bytes con un objetivo distinto al de recrear el grafo serializado, por ejemplo, para realizar algún cálculo sobre el grafo o incluso sobre varias secuencias. Entonces, es deseable hacer una lectura de la secuencia más económica, con menor consumo de procesador y memoria, especialmente cuando los grafos pudieran ser grandes. Damos algunos ejemplos que son interesantes, aunque no son los principales casos de uso de estas herramientas.

Resumen. Realizar una lectura recaudando información sobre el contenido de la secuencia. Por ejemplo, conocer la cantidad de instancias total, o por clase; la memoria requerida para recrear el grafo; instancias de qué clases hay en el grafo; las clases requeridas al ambiente para recrear el grafo.

Consultas. Una variante del punto anterior es brindar la posibilidad de obtener de forma rápida algún dato específico del grafo serializado. Por ejemplo, saber si contiene algún método que envíe o implemente cierto mensaje.

Podado. Puede bastar con reconstruir de manera incompleta el grafo serializado, eliminando ciertas ramas del grafo.

Materialización por demanda. Muy similar al punto anterior, pero no eliminar ramas sino substituirlas por *proxies* que carguen el grafo sólo en la medida que sea necesario.

A.4 Fuel

En esta sección presentamos a *Fuel*, el mecanismo de serialización que hemos desarrollado.

A.4.1 Algoritmo

Normalmente, la serialización de un objeto implica dos acciones principales: iterar al grafo de sus colaboradores internos, y codificarlos en la secuencia de bytes.

El algoritmo de serialización más habitual consiste en iterar al grafo en profundidad (*depth-first*). Cada objeto iterado es codificado en bytes. Los casos base de la recursión son los objetos que se codifican sin necesidad de iterar sus colaboradores internos, tales como las instancias de SmallInteger o de ByteArray. También son casos base los objetos que ya habían sido iterados antes, lo cual evita entrar en ciclo infinitamente.

El algoritmo que implementa *Fuel* realiza la serialización en dos etapas separadas: el análisis y la codificación. Durante el análisis, cada colaborador iterado se asocia a un **cluster**, que le corresponde según su tipo. Al finalizar el análisis, se obtiene una serie de *clusters*, cada uno de los cuales agrupa objetos del grafo de un mismo tipo. En la etapa de codificación, se delega en cada cluster la responsabilidad de escribir en la secuencia a sus objetos asociados. Como mostramos en la sección de Diseño, hay una jerarquía de clases Cluster, las cuales se especializan en codificar y decodificar eficientemente cierto tipo de objetos.

Serializando un rectángulo. Para continuar presentando nuestro algoritmo en manera intuitiva, introducimos el ejemplo de serialización de un rectángulo. Como vemos en el código más abajo, creamos una instancia de Rectangle usando dos instancias de Point que definen su extensión, que pasan a ser sus colaboradores internos. Luego, el mismo pasa como argumento al serializador.

```
| aRectangle anOrigin aCorner |
anOrigin := Point x: 10 y: 20.
aCorner := Point x: 30 y: 40.
aRectangle := Rectangle origin: anOrigin corner: aCorner.
FLSerializer newDefault serialize: aRectangle on: aFileStream.
```

Para que la materialización pueda realizarse de manera iterativa, la etapa de codificación se divide en dos partes: Primero se codifica la información estrictamente necesaria para recrear los vértices del grafo, es decir, para instanciar los objetos; luego, la información para recrear las aristas del grafo, es decir, para restaurar las referencias entre las instancias. La Figura A.20 esquematiza cómo queda serializado el rectángulo. El grafo es codificado en cuatro secciones, dispuestas especialmente para que la materialización requiera sólo una lectura lineal:

header tiene la versión de *Fuel* con el que se serializó, usada para chequear compatibilidad. También tiene la cantidad total de clusters codificados en la secuencia.

vertexes tiene la información necesaria para instanciar al rectángulo, a los puntos, y a los enteros que representan las coordenadas de los puntos.

edges tiene la información para restaurar del referencias del rectángulo y los puntos. Los enteros no tienen referencias a otros objetos.

trailer tiene la referencia al rectángulo, porque es necesario saber cuál de los objetos recreados fue pasado al serializador.

A.4.2 Diseño

La Figura A.21 es un diagrama de clases que muestra el diseño de *Fuel*.

- Serializer, Materializer and Analyzer, tienen el protocolo externo *API* del framework. Son *façades* que brindan a los usuarios lo necesario para serializar y materializar. Además, actúan como *builders*, construyendo en cada ejecución una

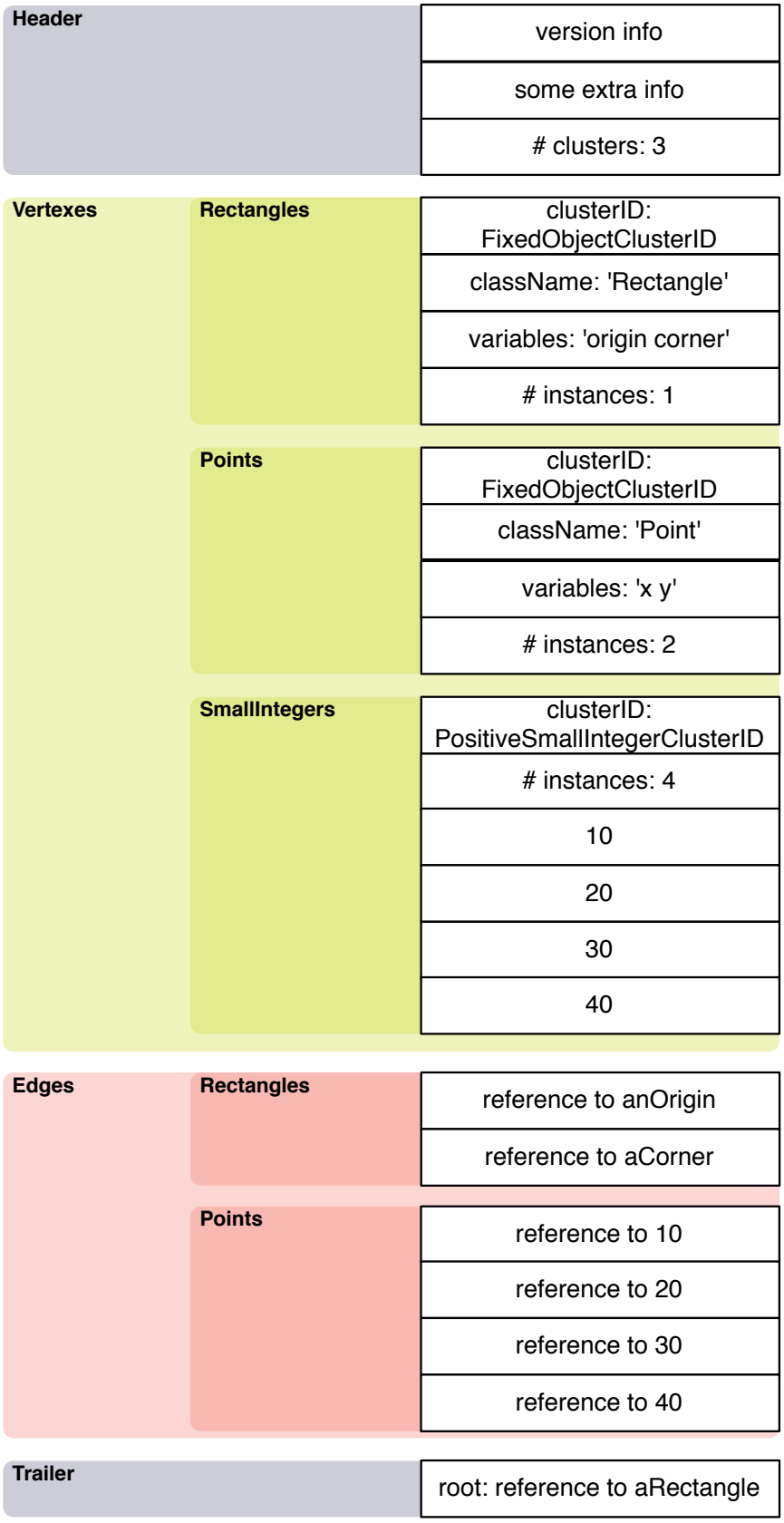

Fig. A.20: Esquema de cómo queda serializado el rectángulo.

instancia de Serialization, Materialization y Analysis, respectivamente, las cuales implementan los algoritmos. A través de *extension methods* agregamos al protocolo estándar funcionalidad extra de manera modular. Por ejemplo, el paquete opcional FuelProgressUpdate agrega el mensaje showProgress a estas clases *façade*, el cual activa una barra de progreso mientras se procesa.

- La jerarquía de clases de Mapper es una característica importante de neustro diseño: Estas clases hacen posible personalizar completamente cómo el grafo es atravesado y serializado. Implementan el patrón de diseño llamado *Chain of Responsibility* [2] para determinar qué *cluster* corresponde a cada objeto. Una instancia de Analyzer se ocupa de crear la cadena de *mappers* cada vez que se serializa.

- La jerarquía de clases de Cluster tiene 44 subclases, diez de las cuales son optimizaciones opcionales.

- *Fuel* tiene 1861 líneas de código, separado en 70 clases. El número promedio de métodos por clase es 7 y el promedio de líneas por método es 3.8. *Fuel* cuenta con una batería de 192 tests que cubren los principales casos de uso, y su cubrimiento es del 90%. La cantidad de líneas de código de tests es 1830, casi la misma cantidad que el código testeado.

- *Fuel* cuenta con una completa *suite* de *benchmarks*, que nos ha resultado muy útil para medir la performance temporal. Permite obtener muestras con una amplia variedad de objetos, y de tamaños de grafo. Nos ha permitido comparar fácilmente con otros serializadores disponibles en *Pharo*, así como también con otras versiones de *Fuel*. Ha sido esencial para verificar cuánto afecta cada cambio, durante el desarrollo. Los resultados pueden ser exportados en formato separado por comas (*CSV*), que suelen entender las herramientas para la generación de gráficos.

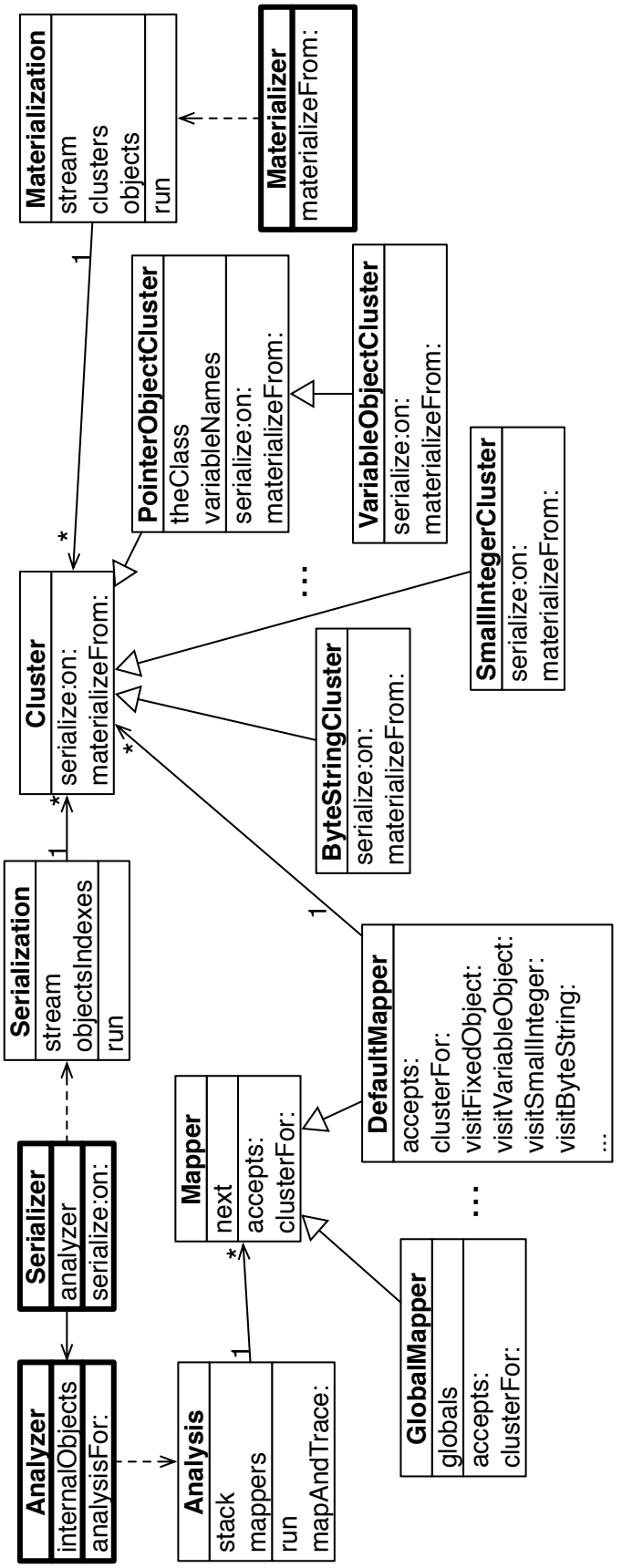

Fig. A.21: Diseño de *Fuel*

BIBLIOGRAFÍA

[1] Action message format - amf 3. http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf.

[2] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[3] John K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, December 1987.

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

[5] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[6] Gilad Bracha, Peter von der Ahé, Vassili Bykov3, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in newspeak. In *ECOOP 2009*, LNCS. Springer, 2009.

[7] Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object serialization. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, 2001.

[8] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[9] Dominique Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 444–452, November 1986.

[10] Michael Franz and Thomas Kistler. Slim binaries. *Commun. ACM*, 40(12):87–94, 1997.

[11] Hessian. http://hessian.caucho.com.

[12] Java serializer api. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[13] Json (javascript object notation). http://www.json.org.

[14] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, November 1986.

[15] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, 2010.

[16] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Problems and challenges when building a manager for unused objects. In *Proceedings of Smalltalks 2011 International Workshop*, Bernal, Buenos Aires, Argentina, 2011.

[17] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. Object swapping challenges: an evaluation of imagesegment. *Journal of Computer Languages, Systems and Structures*, 38(1):1–15, November 2012.

[18] Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[19] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[20] Oracle coherence. http://coherence.oracle.com.

[21] Pickle. http://docs.python.org/library/pickle.html.

[22] Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[23] Lukas Renggli. Pier — the meta-described content management system. European Smalltalk User Group Innovation Technology Award, August 2007. Won the 3rd prize.

[24] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, 1996.

[25] Seaside home page. http://www.seaside.st.

[26] Sixx (smalltalk instance exchange in xml). http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.

[27] State replication protocol framework. http://sourceforge.net/projects/srp/.

[28] Stomp - smalltalk objects on messagepack. http://www.squeaksource.com/STOMP.html.

[29] David Ungar. Annotating objects for transport to other worlds. In *Proceedings OOPSLA '95*, pages 73–87, 1995.

[30] Douglas Wiebe. A distributed repository for immutable persistent objects. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 453–465, November 1986.