# Application-Level Virtual Memory for Object-Oriented Systems

Mariano Martinez Peck

**N° d'ordre : 40886**

# THESE

présentée en vue
d'obtenir le grade de

# DOCTEUR

en

Spécialité : informatique

par

**Mariano MARTINEZ PECK**

**DOCTORAT DELIVRE CONJOINTEMENT
PAR MINES DOUAI ET L'UNIVERSITE DE LILLE 1**

Titre de la thèse :
Application-Level Virtual Memory for Object-Oriented Systems

Soutenue le 29/10/2012 à 10h devant le jury d'examen :

| | |
|---|---|
| **Président** | *Jean-Bernard STEFANI (Directeur de recherche – INRIA Grenoble-Rhône-Alpes)* |
| **Directeur de thèse** | *Stéphane DUCASSE (Directeur de recherche – INRIA Lille)* |
| **Rapporteur** | *Robert HIRSCHFELD (Professeur – Hasso-Plattner-Institut, Universität Potsdam, Allemagne)* |
| **Rapporteur** | *Christophe DONY (Professeur – Université Montpellier 2)* |
| **Examinateur** | *Roel WUYTS (Professeur – IMEC & Katholieke Universiteit Leuven, Belgique)* |
| **co-Encadrant** | *Noury BOURAQADI (Maître-Assistant – Mines de Douai)* |
| **co-Encadrant** | *Marcus DENKER (Chargé de recherche – INRIA Lille)* |
| **co-Encadrant** | *Luc FABRESSE (Maître-Assistant – Mines de Douai)* |

Laboratoire(s) d'accueil : Dépt. IA, Mines Douai + RMoD INRIA Lille Nord de France

Ecole Doctorale SPI 072 (Lille I, Lille III, Artois, ULCO, UVHC, Centrale Lille)

# Contents

# Acknowledgments

First of all, I would like to express my complete gratitude to my supervisors Stéphane Ducasse, Noury Bouraqadi, Marcus Denker and Luc Fabresse for giving me the opportunity to be part of the RMoD and URIA teams. You have introduced me to the research and academia world and positively influenced my decision of doing a PhD. Thank you for all your support, encouragement and advice throughout the years. This PhD was an unbelievable experience that I am glad to have had.

Special thanks go the present and former RMOD and URIA team members, most notably: Guillermo Polito, Nick Papoylias, Jannik Laval, Jean Baptiste Arnaud, Martin Dias, Camillo Bruni, Benjamin Van Ryseghem, Esteban Lorenzano, Igor Stasenko and Verónica Uquillas-Gómez. It was a pleasure working with all of you. I also want to particularly acknowledge Matthieu Faure for being an excellent friend and making my life in France so much easier and enjoyable.

I would like to thank Jean-Bernard Stefani, Christophe Dony, Roel Wuyts and Robert Hirschfeld for accepting to be part of my PhD committee. I really appreciate you taking the time to do so and I am honored to count with your presence.

This thesis would have not have been possible without Pharo programming language. I am much obliged to its community, its developers and especially to Marcus Denker, Stéphane Ducasse, Esteban Lorenzano and Igor Stasenko for their generous efforts maintaining and improving Pharo. Throughout this experience I have met lots of interesting people with whom I have shared several nice moments and discussions (even several argentinian asados!). Thanks to all of you for sharing your knowledge and for being always eager to help out with my questions and concerns.

I am deeply grateful to my family and family in-law who have unconditionally supported me in this experience. I am forever indebted to my parents for everything they have provided me in my life. I would also like to especially mention my father in-law Norberto for his support and advice throughout my PhD.

Above all, I want to thank my wife Flor for being part of my life and for sharing with me many unforgettable moments. Flor has selflessly supported and encouraged me through these years while also being a caring and awesome person. This thesis is dedicated to her.

*To Flor,*

# Abstract

During the execution of object-oriented applications, several millions of objects are created, used and then collected if they are not referenced. Problems appear when objects are unused but cannot be garbage-collected because they are still referenced from other objects. This is an issue because those objects waste primary memory and applications use more primary memory than what they actually need. We claim that relying on operating systems (OS) virtual memory is not always enough since it is completely transparent to applications. The OS cannot take into account the domain and structure of applications. At the same time, applications have no easy way to control nor influence memory management.

In this dissertation, we present Marea, an efficient application-level virtual memory for object-oriented programming languages. Its main goal is to offer the programmer a novel solution to handle application-level memory. Developers can instruct our system to release primary memory by swapping out *unused yet referenced objects* to secondary memory.

Marea is designed to: 1) save as much memory as possible *i.e.,* the memory used by its infrastructure is minimal compared to the amount of memory released by swapping out unused objects, 2) minimize the runtime overhead *i.e.,* the swapping process is fast enough to avoid slowing down primary computations of applications, and 3) allow the programmer to control or influence the objects to swap.

Besides describing the model and the algorithms behind Marea, we also present our implementation in the Pharo programming language. Our approach has been qualitatively and quantitatively validated. Our experiments and benchmarks on real-world applications show that Marea can reduce the memory footprint between 25% and 40%.

**Keywords:** Virtual memory; object swapping; object faulting; unused objects; serialization; proxies

# Résumé

Lors de l'exécution des applications à base d'objets, plusieurs millions d'objets peuvent être créés, utilisés et enfin détruits s'ils ne sont plus référencés. Néanmoins, des dysfonctionnements peuvent apparaître, quand des objets qui ne sont plus utilisés ne peuvent être détruits car ils sont référencés. De tels objets gaspillent la mémoire principale et les applications utilisent donc davantage de mémoire que ce qui est effectivement requis. Nous affirmons que l'utilisation du gestionnaire de mémoire virtuel du système d'exploitation ne convient pas toujours, car ce dernier est totalement isolé des applications. Le système d'exploitation ne peut pas prendre en compte ni le domaine ni la structure des applications. De plus, les applications n'ont aucun moyen de contrôler ou influencer la gestion de la mémoire virtuelle.

Dans cette thèse, nous présentons Marea, un gestionnaire de mémoire virtuelle piloté par les applications à base d'objets. Il constitue une solution originale qui permet aux développeurs de gérer la mémoire virtuelle au niveau applicatif. Les développeurs d'une application peuvent ordonner à notre système de libérer la mémoire principale en transférant les *objets inutilisés, mais encore référencés* vers une mémoire secondaire (telle qu'un disque dur).

En plus de la description du modèle et des algorithmes sous-jacents à Marea, nous présentons notre implémentation dans le langage Pharo. Notre approche a été validée à la fois qualitativement et quantitativement. Ainsi, nous avons réalisés des expérimentations et des mesures sur des applications grandeur-nature pour montrer que Marea peut réduire l'empreinte mémoire de 25% et jusqu'à 40%.

**Mots clés:** Mémoire virtuelle; sérialisation; proxies; objets inutilisés; programmation orientée objet

CHAPTER 1

# Introduction

Contents



## At a Glance

This chapter introduces the domain and the context of our research. We explain the problems regarding memory usage of programs written in dynamic object-oriented programming languages. In this context, we place our approach and the solutions offered. We finish this chapter with a summary of the contributions.

## 1.1 Context: The Memory Used by Running Object-oriented Programs

During its execution, a program is loaded into primary memory. Considering object-oriented (OO) programs, the allocated space is occupied by objects. The amount of space may vary during the program execution depending on the creation and destruction of objects.

Since the primary memory is a limited resource, there are several mechanisms and strategies which take place at different levels to optimize its usage and limit the waste of space. For example, at the operating system (OS) level, the virtual memory mechanism splits primary memory into "pages" which can be stored in secondary memory [Denning 1970]. These pages are then reloaded back in primary memory almost transparently depending on what is needed by the program. Another example is the automatic garbage collector (GC) at the language level that frees memory allocated by objects that are transitively unreachable in a system and therefore will never be needed in the future [McCarthy 1960].

Saving and optimizing the primary memory occupied by a running OO program is the context of the work presented in this dissertation.

## 1.2 The Problem of Using Unnecessary Memory

We define as *unused* objects those that a program has not used for a while. Some objects are only used in certain situations or conditions with some not used for a long period of time. The extreme case is when objects are used only once (leaks). In any case, although these objects are *unused*, they are still reachable from other objects.

Unused yet reachable objects cannot be garbage collected. This is an issue because those objects waste primary memory [Kaehler 1986]. Wasting primary memory means less applications running on the same hardware or slowdowns because of operating system virtual memory swapping.

Having unused objects in a software can sometimes be a symptom of an even more serious problem: memory leaks [Bond 2008]. In presence of memory leaks, applications use much more resources than they actually need. They may eventually exhaust the available memory and lead to system crashes.

Apart from the mentioned unused objects, there is yet another category of unused memory. Typical object-oriented applications run on top of a runtime environment such as the J2SE, the .NET framework or a Smalltalk system. Not all applications use all parts of their runtime. Consequently, applications usually occupy more memory than they actually need. Section 2.3.3 presents an experiment to measure unused objects in three real applications. These benchmarks report 80% of *unused* objects. The situation is even worse in small systems. For example, a hello world application in Java SE 1.6 occupies 25MB of RAM.

```
import java.io.IOException;
public class Hello{
  public static void main(String [] args) throws IOException {
    System.out.println("Hello World");
    System.in.read();
  }
}
```

To conclude, unused yet referenced objects and heavy runtimes, usually make object-oriented applications to use more memory than needed.

## 1.3 Shortcomings of Existing Approaches

Operating systems have been supporting virtual memory since a long time [Denning 1970, Stamos 1982, Carr 1981, Williams 1987, Krueger 1993]. Virtual memory is transparent in the sense that it automatically swaps out unused memory organized in pages governed by some strategies such as the least-recently-used (LRU) [Carr 1981, Chu 1972, Denning 1980,Levy 1982]. As virtual memory is transparent, it does not know the application's memory structure, nor does an application have any way to influence the virtual memory manager. The OS only knows about memory pages or similar structures used by virtual memory implementations. Therefore, the OS cannot guess *which* objects are the most appropriate ones to swap, *when* it is a convenient moment to swap, among other policies. As applications become larger and more complex, and the cost of bad memory management grows, it is necessary for the applications to have more control over the virtual memory abstraction [Engler 1995a, Engler 1995b].

An approach to decrease the occupied yet unused memory by application's code is to reduce the runtime they run on. These approaches consist of *removing code* and building small runtimes. For example, J2ME is a stripped-down version of Java for embedded devices. Such platform uses less memory than the standard Java because it does include less functionalities than standard Java, i.e., J2ME contains a strict subset of the Java-class libraries. J2ME degrades the Java environment and APIs right from the specification. Moreover, some J2ME APIs are not compatible with standard Java, breaking the rule "compile once, run everywhere". In this case, decisions behind this reduction are taken by the developers of J2ME. From application developers' perspective, J2ME is a monolith that cannot be adapted.

Another alternative is to let developers decide what each application needs and create a specific and customized runtime for it. For example, VisualWorks Smalltalk provides the Runtime Packager[1] which makes smaller runtimes by explicitly removing classes and packages. However, with these strategies, developers need to know *with absolute certainty* what is required by their applications. At development stage, it is difficult, time-consuming and sometimes impossible to figure out what an application needs for all possible execution paths even with static analyzers. Most static analyzers do not take into account reflective features which are often used to support application evolution and dynamic code loading

[1] Explained in VisualWorks Application Developer's Guide

[Bodden 2011].

There are implementations of virtual memory integrated into object-oriented languages [Kaehler 1986]. This is a step forward since this kind of virtual memory does not depend on the OS. However, such systems are developed on the virtual machine side. In contrast, a language-level implementation decoupled from the virtual machine and the OS has the following advantages:

- It is easier to develop, manage, maintain, debug, extend, install, distribute and understand.
- We can take advantage of all the infrastructure and tools that the language provides. For example, we can use tools like browsers, inspectors, debuggers, versioning system, etc.
- It could potentially work with different virtual machines if the language provides several ones. Besides, language-level implementation ease the port of the tool to other dialects of the same language or even to other languages.
- The user can hook into the algorithms and adapt them for its own needs.

Depending on how the system is implemented, it could also allow the developer to influence the virtual memory decisions. However, they all have the problem of the *swapping unit*, *i.e.,* the portion of memory that is swapped out and in. The goal of a virtual memory manager is to release primary memory by temporary moving data to secondary memory. Regardless the implementation, the system must be carefully designed so that it saves as much memory as possible and minimizes the overhead.

The swapping unit can be an individual object or a group of them. Two facts make the swapping unit a critical design decision. On the one hand, to have good performance, we want to go to secondary memory as less as possible *i.e.,* we want to minimize object faults. When an object is brought into primary memory we would like that all the future needed objects were also in memory, *i.e.,* we want to avoid object faults for all other objects it accesses. Furthermore, when an object is moved to secondary memory, the system should manage the situation when there are objects in primary memory referencing it. On the other hand, the more objects the swapping unit contains, the more likely we would be swapping in unnecessary objets.

In traditional OS' virtual memory, the swapping unit is a page. The OS cannot guess which objects are the most appropriate ones to swap nor how to accommodate them into pages to improve the performance. Virtual memories integrated into object-oriented programming languages do not solve this issue either since in most cases the swapping unit is an individual object.

**Research questions:**

1. *How to build a virtual memory manager that is efficient (reduce the primary memory footprint and minimize the runtime overhead) and application-aware (possibility to adapt it to a specific application)?*

2. *Besides efficiency and application-aware, what are the rest of the properties that a virtual memory manager needs to fulfill?*

## 1.4 Our Solution in a Nutshell

*In this thesis, we argue that an efficient application-level virtual memory for object-oriented programming languages: i) should be implemented at the language level to allow the developer to control when and what to swap according the an application and ii) that the swapping unit should be objects graphs.*

We propose Marea, a virtual memory manager whose main goal is to offer the programmer a solution to handle application-level memory [Martinez Peck 2012a]. Developers can instruct our system to release primary memory by swapping out *unused objects* to secondary memory.

Marea is designed to: 1) save as much memory as possible *i.e.,* the memory used by its infrastructure is minimal compared to the amount of memory released by swapping out unused objects, 2) minimize the runtime overhead *i.e.,* the swapping process is fast enough to avoid slowing down primary computations of applications, and 3) allow the programmer to control or influence the objects to swap.

The input for Marea are *user-defined* graphs, *i.e.,* graphs that the user wants to swap out to secondary memory. Swapped graphs are swapped in as soon as one of their elements is needed. The graphs to swap can have *any* shape and contain *almost any* type of object. This includes classes, methods and closures which are all first-class objects in Marea's implementation language [Black 2009]. This means that our solution works with both of the scenarios mentioned in Section 1.2 *i.e.,* code (runtime) and application-specific objects.

When Marea swaps a graph, it correctly handles all the references from outside and inside the graph. When one of the swapped objects is needed, its associated graph is *automatically* brought back into primary memory. To achieve this, Marea replaces original objects with proxies [Gamma 1993]. Whenever a proxy intercepts a message, it loads back the swapped graph from secondary memory. Since we are changing the living object graph at runtime, this process is completely transparent for the developer and the application. Any interaction with a swapped graph has the same results as if it was never swapped.

Considering object graphs as the swapping unit is a key aspect of our design because:

- We only need a few proxies per graph, hence we increase the memory we are able to release. As we see later in Section 4.4 we only need proxies for the root of the graph and for the objects which are also referenced from outside the graph.

- We decrease the number of unnecessary objects swapped in. If an object is needed and it is swapped in, it is likely it will need its related objects in the future. When considering a page as the swapping unit, for example, there is usually no relation between the objects that are placed into the same page. In the contrary, with graphs, the objects are connected. One drawback with object graphs is that if they are too large, we may be swapping in more unnecessary objects than if grouping them in

pages or groups. This is the reason why as part of the future work proposed in Section 10.3 we discuss about *partial loading*.

- We decrease the number of object faults. When an object is swapped in, its related objects are also swapped in, avoiding more object faults.

The term "Open Implementation" is about having an API to change implementation aspects [Kiczales 1994, Kiczales 1996]. The prime example is a MOP: an API to change a language implementation. From that point of view, Marea's API is an API to give access to one feature (virtual memory) of the program's run-time. At the same time, since Marea and its subsystems (the serializer, the proxy toolbox and the object graph storage) are written in the language side, they can be easily adapted and changed. However the goal of Marea is not to design a full open-implementation because it would require to expose more information and design hooks which may hamper performance.

## 1.5 Contributions

The main contributions of this dissertation are:

1. A precise description of problems, challenges, algorithms and design aspects while building an application-level virtual memory manager for object-oriented systems [Martinez Peck 2011b].

2. A specific virtual machine that can trace object usage. In addition, we provide an API so that the user can query the system to know which objects have been used during a particular scenario (cf. Section 2.3.1).

3. The design of Marea [Martinez Peck 2012a], our efficient solution implemented in the Pharo programming language [Black 2009]. We show that Marea can reduce the primary memory occupied by applications up to 40%. Our implementation also demonstrates that we can build such a tool without modifying the virtual machine (VM) yet with a clean object-oriented design.

4. Ghost, a uniform and lightweight general-purpose proxy implementation [Martinez Peck 2011a, Martinez Peck 2012b]. Ghost provides low memory consuming proxies for regular objects as well as for classes and methods. Moreover, Ghost supports *controlled stratification*: developers decide which message should be understood by the proxy and which should be forwarded to a handler.

5. Fuel, a general-purpose object serializer [Dias 2011, Dias 2012] based on these principles: (1) speed, through a compact binary format and a pickling algorithm which invests time in 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.

6. Visualizations to analyze which parts of the systems are used during a certain scenario [Martinez Peck 2010b]. We can visualize which part of the system is used and which is not, apart from measuring the number of objects and their size in memory.

## 1.6 Structure of the Dissertation

This dissertation is organized as follows:

**Chapter 2** deeply analyzes the problem of managing unused memory in object-oriented languages and provides a list of requirements a novel application-level virtual memory for OO languages should comply with.

**Chapter 3** describes existing approaches that tried to solve these problems and evaluates them against the listed requirements in the previous chapter.

**Chapter 4** presents Marea, our overall solution to the problem of unused memory in object-oriented programming languages. It shows an overview of the complete approach explained in the next chapters.

**Chapter 5** validates our overall approach by benchmarking Marea with real applications and shows the results in terms of speed and memory consumption. We performed several experiments to measure how much memory could be gained using Marea.

**Chapter 6** introduces Fuel, our fast general-purpose framework to serialize and deserialize object graphs using a pickle format which clusters similar objects.

**Chapter 7** presents Ghost, our uniform and lightweight general-purpose proxy toolbox. Ghost provides low memory consuming proxies for regular objects as well as for objects that play an important role in the runtime infrastructure such as classes and methods.

**Chapter 8** discusses the implementation details and key design aspects of Marea. It also lists possible problems and requirements that would be faced when implementing Marea in other programming language.

**Chapter 9** proposes some visualizations that take into account objects and memory usage. It also shows how these visualizations can help Marea users have better results.

**Chapter 10** concludes the dissertation and ends with an outlook on the future work opened by our approach.

**Appendix A** gives instructions on how to install and use the Marea system.

CHAPTER 2

# Unused Memory in OO Programs: The Problem

Contents



## At a Glance

In Section 1.2 we briefly presented the problem this dissertation tackles. In this chapter, we present an experiment that demonstrates such problem and makes it clearer. We also list the requirements a novel application-level virtual memory for OO languages should comply with.


**Keywords:** Unused memory, virtual memory, glossary, requirements, object-oriented programming.

## 2.1 Introduction

This chapter first explains the terminology used in the following of the dissertation. Then it presents an analysis regarding memory usage that clarifies the problem and motivates the goal of this dissertation.

### Structure of the Chapter

In the next section we define the terms used along this chapter and the dissertation as well. Section 2.3 performs an analysis that measures used and unused objects in real applications and demonstrates the problem. Section 2.4 lists the requirements for a novel application-level virtual memory manager. Finally, Section 2.5 summarizes the chapter.

## 2.2 Glossary

In an object-oriented system, related objects reference each other and form a graph. Objects are nodes of such a graph, while references are the edges of it. Unused objects remain in memory (are not garbage collected) if they are connected to other objects. So, in an object graph, some objects are used and some others are unused.

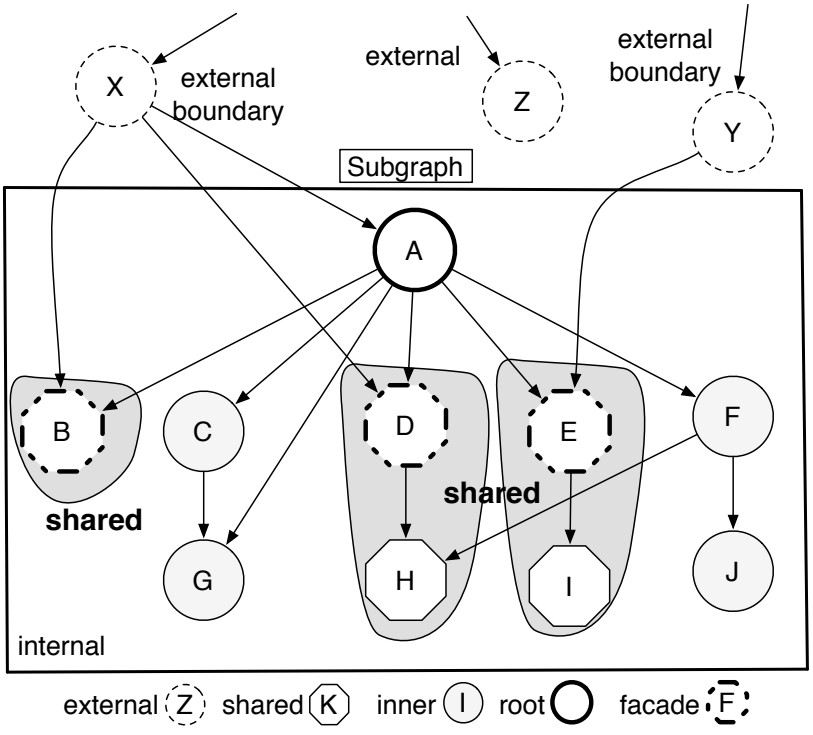


Figure 2.1: Object classification based on graph structure.

When dealing with an application-level virtual memory for object-oriented languages, we actually end up handling objects that are part of graphs. We need to deal with object

graphs when analyzing the system to identify unused objects and also when swapping out and in. Therefore, Figure 2.1 shows an example of an object graph (surrounded by a rectangle). Through this example we define a glossary of terms used in this dissertation to avoid confusion:

**External objects** are those outside the graph. Example: X, Y and Z.

**A root object** is the starting node that allows retrieving all the other nodes of the subgraph by following the edges (references). Example: A.

**Internal objects** are the *root object* and all the objects accessed through it. Example: A, B, C, D, E, F, G, H, I and J.

**External boundary objects** are *external objects* that refer to *internal objects*. Example: X and Y.

**Shared objects** are *internal objects* that are accessed, not only through the root of graph, but also from outside the graph. Example: B, D, E, H and I.

**Internal boundary objects** are *internal objects* which refer to *shared objects*. Example: A and F.

**Facade objects** are *shared objects* which are directly referenced from *external objects*. Example B, D and E.

**Inner objects** are *internal objects* that are only accessible from the *root objects*. Example: C, G, F and J.

## 2.3 Analyzing Memory Usage

To support automatic memory management, most object oriented runtimes rely on garbage collectors (GC) [McCarthy 1960]. The idea behind garbage collection is the automatic destruction of unreferenced objects. The GC collects objects that are not being referenced anymore.

As mentioned above, unused objects are reachable through the object graph and hence they are not garbage collected. To measure the amount of memory wasted with such unused objects, we analyzed several real world applications. We report in this section their memory usage and the number of used vs. unused objects.

### 2.3.1 Detecting Used and Unused Objects

We report here an experiment where we analyzed several real-world applications and measured how many objects and how much memory are actually really used. For that, we implemented a virtual machine able to trace object usage.

The analysis to detect used and unused objects consists of these steps:

**Start analysis.** Mark all objects as unused and then enable the tracing.

**Analysis phase.** At this step we run several scenarios covering as much functionalities of the application as possible. During this execution, each object that is accessed by the virtual machine *e.g.,* when receiving a message, is marked as *used*.

**Stop analysis.** We disable the tracing. From this moment, when an object receives a message, it is not marked anymore.

Once these steps are done, objects are not marked or unmarked anymore so that we can get all the needed information for gathering statistics.

A used object is one that received a message or that has been directly used by the virtual machine during the specific period of time of the analysis phase (between the start and the stop of the analysis).

The information obtained from this type of executions is basically heuristic. Thus, we cannot be sure that a given object is not used at all. It just means it was not used during the analysis phase. Moreover, a certain object that was considered unused, may be needed later on.

This proposed approach could be more conclusive if used in conjunction with a coverage analysis tool together with unit and functional tests. For example, if we have an application that we want to deploy in a minimal runtime, we can run a coverage analysis or even run all the applications tests, in order to detect all possible future used objects.

To implement this analysis, the first challenge is where to store the mark of each object. The first and naive possibility is to simply add an instance variable to Object (the root class of the hierarchy chain) and store there a Boolean object. Nevertheless, this is not possible in Pharo. In addition, the requirement of an extra reference for each object is highly memory consuming.

For our experiment we modified the Pharo VM so that we can use an unused bit of the object header to mark objects as used. In this case, we do not use extra memory and it works efficiently. We also modified the code of the VM that implements the *message send* to turn on the bit when an object receives a message.

Finally, we added all the necessary primitives to the VM to mark all objects, unmark them and get statistics about the memory usage. Furthermore, we implemented custom features on the language side to call the primitives and to take statistics about packages, classes and methods usage.

The modified VM code and the language side code are open-source licensed under MIT license and available here: http://ss3.gemstone.com/ss/ObjectUsageTracer.html.

### 2.3.2 Applications for Analysis

The following is the list of descriptions of the applications we used to perform our analysis.

**DBXTalk CMS Website.** It is the web application of the DBXTalk project[1]. This website is developed with Pier[2], a CMS based on the Seaside Web framework [Ducasse 2010].

[1] The website is currently located in http://dbxtalk.smallworks.com.ar/

[2] www.piercms.com

**Moose Suite.** Moose[3] is a platform for software and data analysis [Nierstrasz 2005]. With such tool, users handle different large models that represent the packages and data that are being analyzed. Moose provides lots of ways to visualize and analyze models.

**Dr. Geo.** It is an interactive geometry software which allows one to create geometric sketches [Fernandes 2007]. Apart from running in GNU/Linux, Windows and Mac OS X, Dr. Geo runs on the XO laptop and (at the time of this writing) it is also being ported to tablets. Because of that, the Pharo runtime used by Dr. Geo is already reduced. It is based on a PharoCore using the production configuration. Besides, developers did compact it even further by removing part of the code included in PharoCore. Since the runtime is only 11.2 MB it is an interesting challenge.

**Pharo Infrastructure.** Apart from benchmarking the memory consumption of different applications built on top of Pharo, we measure the Pharo infrastructure (the IDE and the runtime) itself without anything extra loaded on it. The idea is to compare the infrastructure versus applications and analyze if we can compact Pharo's runtime even more.

### 2.3.3 Unused Objects: Experiments

We have modified the Pharo VM to be able to trace objects usage and we also have several real applications. Now we run different experiments with the mentioned applications. For each experiment we start the analysis, we use the application for a while, we stop the analysis and finally collect the results. For each application we got the following information: the percentage of used and unused objects and the percentage of memory that used and unused objects represent.

| App. | % Used objects | % Unused objects | % Used objects memory | % Unused objects memory |
|---|---|---|---|---|
| DBXTalk | 19% | 81% | 29% | 71% |
| Moose | 18% | 82% | 27% | 73% |
| DrGeo | 23% | 77% | 46% | 54% |
| Pharo | 10% | 90% | 37% | 63% |

Table 2.1: Measuring used and unused objects.

The results are described by Table 2.1. The table shows that, after having navigated all the pages of the DBXTalk website and doing our best to cover all its functionalities, only 19% of the objects were used representing 29% of the runtime memory size.

For Moose, we got that 18% of the objects were used by our experiment. This makes sense because Moose is a really large suite of tools which provides lots of visualizations and integration with different programming languages. In our case, we just imported Smalltalk projects and we run all the visualizations and tools.

[3] http://www.moosetechnology.org/

Dr. Geo example demonstrates that its runtime is already reduced and that is the reason why its percentage of used objects is bigger than the other examples.

In the Pharo infrastructure example, only 10% of the objects were used representing 37% of the runtime memory size. Why the 10% of the objects represents 37% of the memory? This is because the runtime includes all the bitmaps to render the environment. There are a few bitmaps (312 in our example) but each of them is large in memory consumption. Those 312 bitmaps occupy 4.8 MB which represent almost half of the consumed memory.

**Conclusion** To clearly explain the problem that motivates this dissertation we needed to analyze the memory used by applications. Our results show that the analyzed applications only use between 27% and 37% of the memory they occupy. This is the problem we address in this dissertation.

## 2.4 Requirements for Application-Level Virtual Memory Manager

We argue that a novel application-level virtual memory for OO languages should comply with the following requirements:

- **Efficient object-based swapping unit.** Since we are targeting application-level object-oriented virtual memory, the swapping unit has to be at the object level instead of pages of bytes. It can be one object or several together. However, it has to provide an *efficient granularity* to generate the less object faulting as possible and when swapping in to load the minimum of unnecessary objects.

- **Uniformity.** Depending on the language where the system is implemented in, objects can reify the program structure, run-time and other reflective facilities. For example, in Smalltalk, classes, methods, closures, processes, semaphores, among others, are all first class objects. If this is the case, the tool has to be able to improve the memory usage of both, code and application data, *i.e.,* it has to be uniform in the sense that it should be able to swap any kind of object.

- **Reversibility.** Do not remove but instead swap objects. Unused objects should be placed on secondary memory but not removed. Afterwards, if they happen to be needed, they must be swapped in.

- **Automatic swapping in.** As expected from a virtual memory, if an object that was swapped out, is now needed, the system must *automatically* swap it in.

- **Automatic swapping out.** Although the programmer's involvement is a good complement for current virtual memory management, providing also automated mechanism to swap out is appealing *e.g.,* for the case when nothing is specified by the developer.

- **Transparency.** From the point of view of an application's behavior, the system must be transparent in the sense that the application will get the same results whether it is using virtual memory or not. In addition, it has to provide the same API for the programming language. From the application development point of view, the application code should not be polluted with code related to swapping.

- **Controllability at the application level.** The solution should allow application programmers to influence what, when and how to swap, as motivated in Section 1.3. This opens up many new possibilities as we can take domain knowledge into account for the decision of what to swap. The idea is that, since we are at the object level, much more sophisticated strategies can be developed than the simple LRU replacement available with OS virtual memory [Carr 1981, Chu 1972].

- **Portability.** As much as possible, the solution should provide the virtual memory manager as a tool that is completely implemented on the language side without requirements on the virtual machine nor the OS. The reasons were explained in Section 1.3.

## 2.5 Summary

In this chapter we presented some experiments that demonstrate the problem this dissertation tackles. Our results showed that the analyzed applications only use between 27% and 37% of the primary memory they occupy. In addition, we listed the requirements that a novel application-level virtual memory for OO languages should comply with.

Now that we made our problem clear, the next section presents the related work and compares it with the requirements defined in this chapter.

CHAPTER 3

# Approaches to Unused and Application-Level Virtual Memory

## Contents



## At a Glance


This chapter introduces the work related to our research. We detail four domains: (i) virtual memory and garbage collected languages; (ii) reduced language runtimes; (iii) object faulting, orthogonal persistence and object databases; (iv) memory leaks; and (v) general-purpose object graph swappers. We evaluate each work with the list of requirements defined in the previous chapter and we show that these approaches do not satisfy what we define as a application-level object-oriented virtual memory.



**Keywords:** Virtual memory, object faulting, persistence, memory leaks, object graph swapper.

## 3.1 Introduction

In Section 2.4 we introduced the requirements that a novel virtual memory manager for dynamic languages must fulfill. In this chapter we evaluate each related work considering these requirements and we show that there is a lack of approaches to meet all those requirements. Not all the approaches are virtual memory managers nor do they try to solve exactly the problem this dissertation tackles. Therefore, we do not consider our evaluations as a measure of how good or bad each approach is, but instead how much they satisfy the requirements we are interested in.

#### Structure of the Chapter

In the next section we introduce related work to virtual memory and garbage collection. In Section 3.3 we present the approach of customized runtimes object-oriented languages. Section 3.4 shows related work to object faulting, orthogonal persistence and object databases. Memory leaks, a particular problem of unused memory is analyzed in Section 3.5. Section 3.6 details related object graph swappers. Finally, Section 3.7 summarizes the chapter.

## 3.2 Virtual Memory and Garbage Collected Languages

While offering numerous advantages regarding software engineering, garbage collectors do not interact well with OS' virtual memory. The reason is that a full GC traverses all the objects and that can lead to pages reloads causing the so called thrashing of memory pages [Hertz 2005].

### 3.2.1 Operating System Virtual Memory

Since operating systems (OS) provide virtual memory, why should an application program such as a virtual machine for programming languages be responsible for moving objects from primary to secondary memory? Why that task cannot be left to the OS and its efficient management of virtual memory? The reasons are:

- Garbage Collection: Memory not used can, in theory, be swapped by the operating system to disk. However, the objects on disk also need to be garbage collected. As GC's working set involves all objects (including those which are on disk), this can trigger memory thrashing (traffic between primary and secondary memory) that degrades performance by orders of magnitude [Yang 2006, Hertz 2005]. In Section 3.2.4 we discuss some possible solutions to this problem.

- Persistence: Memory swapped by the OS is, by definition, transparent to the language. In image-based systems such as Smalltalk, we can persist the current object memory state in a disk snapshot. When memory is swapped by the OS, saving the system means swapping in all data and writing out the complete heap. As the virtual

machine is not aware of the memory management of the OS, it cannot save its memory in a state where some parts are swapped out while others are not. Even the use of memory mapped files does not help: the GC traversal will force the whole file to be loaded into primary memory.

- The OS only knows about memory pages or similar structures used by virtual memory implementations. Therefore, the OS cannot guess which objects are the most appropriate ones to swap. The application itself is the one who knows the details of objects usage [Engler 1995a, Engler 1995b].

Table 3.1 shows the summary of the evaluation of OS' virtual memory regarding the mentioned requirements.

| | **OS Virtual Memory** |
|---|---|
| Efficient Object-Based Swapping Unit | ○ |
| Uniformity | ● |
| Reversibility | ● |
| Automatic Swapping In | ● |
| Automatic Swapping Out | ● |
| Transparency | ● |
| Controllability at the Application Level | ○ |
| Portability | ○ |

Table 3.1: OS' Virtual Memory evaluation.

### As a Conclusion

*OS' virtual memory is not enough and the developer has no way to influence it. In addition, the granularity is based on pages or similar data structures.*

### 3.2.2 Application-Specific Operating System Virtual Memory

Traditional operating systems are designed as general-purpose systems intended to perform reasonably well on average over all applications. As the number of applications having non-standard memory access patterns increases, the OS' virtual memory policy is degraded. As a result, applications programmers may need to implement their own manager. Such shortcomings have led some to argue for *user-level* virtual memory management.

Operating systems such as Mach [Young 1987], V++ [Cheriton 1988], and Apertos [Yokote 1992, Yokote 1993] are designed to allow users to provide their own virtual memory management module. All these systems, however, require users wishing to exploit these features to build or re-build a significant part of the operating system; this is a complex task beyond the ability of most users.

Krueger et al. [Krueger 1993] developed a technique that allow users to experiment with various page replacement policies and to get immediate feedback from the user interface. Upon startup, a particular application can register its own "page fault handler". When

the OS encounters a page fault, it calls back to the user-defined page handler and invokes a specific routine. Therefore, from the language side, developers can subclass and implement some methods to decide for example, how to handle the page fault *i.e.,* which page to remove from primary memory and move it to secondary one. Users can develop different strategies for their application *e.g.,* they can implement MRU (most recently used) if they prefer it instead of LRU (last recently used). Their conclusion is that application-specific virtual memory policies can increase application's performance.

Exokernel is an operating system architecture for application-level resource management [Engler 1995a] in which traditional operating system abstractions, such as virtual memory and interprocess communication, are implemented at the application level. The same authors also developed AVM (application-level virtual memory) on top of the exokernel. The advantage of this approach is that the user can adapt the policy to choose which pages to swap out while also being able to change some of the virtual memory abstractions *e.g.,* the page-table structure.

Table 3.2 presents an evaluation of the requirements with the application-specific OS virtual memory managers we are aware of.

| | Mach, V++, Apertos | Krueger et al. | Exokernel |
|---|---|---|---|
| Efficient Object-Based Swapping Unit | ○ | ○ | ○ |
| Uniformity | ● | ● | ● |
| Reversibility | ● | ● | ● |
| Automatic Swapping In | ● | ● | ● |
| Automatic Swapping Out | ● | ● | ● |
| Transparency | ● | ● | ● |
| Controllability at the Application Level | ◐ | ● | ● |
| Portability | ○ | ○ | ○ |

Table 3.2: Application-Specific OS Virtual Memory evaluation.

## As a Conclusion

*Being able to change on the language side the policy used by the OS to decide what pages to move between memories, among others abstractions, is definitively a step forward to fulfill the requirement of "Controllability at the Application Level". However, this solution is still low level and depends on a particular OS.*

### 3.2.3 LOOM: Large Object-Oriented Memory

In the eighties, LOOM [Kaehler 1986] (Large Object-Oriented Memory) was a mechanism for Smalltalk 80 which aimed to have a large virtual space that can grow without limits and without impact on Smalltalk performance.

LOOM defined a swapping mechanism between primary and secondary memory. Primary memory was on RAM and it uses 16 bits for the object pointers. The second memory was hard disk and it uses 32 bits pointers. This decision of having different pointer sizes for each memory was a key design decision in LOOM. Most of the problems, solutions and ideas of LOOM were related with this fact.

When objects from primary memory (addresses with 16 bits) pointed to objects in secondary memory (addresses with 32 bits), LOOM made the former to refer a stub object that just stored the long address. Stub objects were marked with a bit that worked as a flag to know whether an object was a stub or not. When the system attempts to access an object *e.g.,* for a message sending, it first checks if it is a stub. A stub is replaced by the original object fetched from secondary memory before performing computations. This technique is known as object faulting [Hosking 1990].

To select which objects to swap, LOOM used a simple LRU (last recently used) [Carr 1981, Chu 1972, Denning 1980, Levy 1982] technique based on a bit used for tracing objects usage.

LOOM was implemented in a context where the secondary memory was much slower than primary memory. This made the overall implementation much more complex. Nowadays, secondary memory is getting faster and faster with random access showing similar results as the RAM memory[1].

The solution was good but too complex due to the existing restrictions (mostly hardware) at the time. Most of the problems faced do not exist anymore with today's technologies — mainly because of newer and better garbage collector techniques. For example, LOOM had to do complex management for special objects that were created too frequently like MethodContext but, with a generation scavenging [Ungar 1984], this problem is solved by the garbage collector.

An interesting problem faced by LOOM was the fact that since objects could be on secondary memory, certain Smalltalk primitives needed to be modified, such as hash, become, someInstance and nextInstance. It also found that certain critical objects should could not be swapped, such as instances of Process and Semaphore and also all the 50 objects known by the Virtual Machine like nil, true, false and Smalltalk.

Jim Stamos did an extensive simulation comparing LOOM with a corresponding paging system [Stamos 1982, Stamos 1984]. His results showed that if objects that are used together can also be grouped together in secondary memory, then both paging and LOOM perform well. On the other hand, when objects are not grouped and the size of primary memory is small and close to the working set size, both paging and LOOM tend to thrash. One of his conclusion is key for our thesis: the success of grouping schemes is more important than the differences between the schemes.

[1] "Solid-state drives" (SDD) or flash disks have no mechanical delays, no seeking and they have low access time and latency.

Table 3.3 evaluates LOOM in reference to our requirements.

| | LOOM |
|---|---|
| Efficient Object-Based Swapping Unit | ◐ |
| Uniformity | ● |
| Reversibility | ● |
| Automatic Swapping In | ● |
| Automatic Swapping Out | ● |
| Transparency | ● |
| Controllability at the Application Level | ○ |
| Portability | ○ |

Table 3.3: LOOM evaluation.

### As a Conclusion

*LOOM considered objects as the swapping unit rather than pages or similar structures. However, the main downfall was that its swapping unit was too small as it was only one object. In addition, LOOM was implemented at the virtual machine level and it also had to take care about memory management of primary memory and creation and destruction of objects. The objects to swap were detected by LOOM and the user could not influence that.*

## 3.2.4 Solving the OS Thrashing Problem

Section 3.2.1 explained the problem of thrashing in presence of garbage collection. The following are possible solutions:

**Bookmarking Garbage Collector.** Bookmarking collection is a garbage collection algorithm that virtually avoids thrashing [Hertz 2005]. This GC stores information ("bookmarks") of outgoing pointers from pages that have been swapped out to disk. Instead of visiting swapped out pages, the GC uses bookmarks to assist garbage collection. With the cooperation of the virtual memory manager, these bookmarks allow the GC to perform a full-heap, compacting garbage collection without paging.

**CRAMM.** CRAMM (Cooperative Robust Automatic Memory Management) [Yang 2006] is another solution to the thrashing problem. CRAMM consists of two parts: the virtual memory system and the heap sizing model. The former gathers information about the process being executed and the latter dynamically chooses the optimal heap size which allows the system to maintain high performance.

### As a Conclusion

*Since these approaches are not directly related to our concerns, our requirements are not relevant to be evaluated. A virtual memory manager is not*

*intended to solve the OS' thrashing problem. However, since it reduces the amount of objects loaded in primary memory, it also may reduce the amount of pages and thus, the OS' thrashing. Therefore, a close cooperation between GC and virtual memory is required to reduce thrashing.*

## 3.3 Reducing the Memory Occupied by Code

The memory which is occupied but unused can be split in two parts. One part is used by the code of the application and linked libraries. The other part stores data generated as a result of execution. Reducing both parts is of interest.

There are different approaches to reduce the memory occupied by code.

### 3.3.1 Reduced and Specialized Runtimes

Solutions belonging to this family decrease the memory usage by *removing code* and building small runtimes. For example, J2ME is a stripped-down version of Java for embedded devices. Apart from providing functionalities which are specific to embedded devices, such platform uses less memory than the standard Java. This reduced memory footprint is because J2ME does include less functionalities than standard Java, *i.e.,* J2ME contains a strict subset of the Java-class libraries. However, decisions behind this reduction are taken by the developers of J2ME. From developers perspective, J2ME is a monolith that cannot be adapted.

J2ME degrades the Java environment and APIs right from the specification. Moreover, some J2ME APIs are not compatible with standard Java, breaking the rule "compile once, run everywhere". For instance, if an application needs to directly connect to a relational database, it is not possible to do it in the same way it is done in standard Java because J2ME does not provide the JDBC API.

The evaluation of reduced and specialized runtimes regarding our requirements is summarized in Table 3.4.

| | **Reduced and Specialized Runtimes** |
|---|---|
| Efficient Object-Based Swapping Unit | — |
| Uniformity | ○ |
| Reversibility | ○ |
| Automatic Swapping In | — |
| Automatic Swapping Out | — |
| Transparency | ○ |
| Controllability at the Application Level | ○ |
| Portability | ● |

Table 3.4: Reduced and Specialized Runtimes evaluation.

### As a Conclusion

*These approaches may degrade the language API and even provide a non-standard version of the language. Moreover, the decisions of what to reduce are taken by runtime's developers and by application developers. Finally, while the previous approaches try to offer a smaller memory footprint for application code, there is another problem: application data. The traditional way to handle data is to save it to files or databases and load them when necessary. Programmers have to handle this explicitly and provide support for storing and loading objects while keeping track of their references.*

## 3.3.2 Custom and Specific Runtimes

Contrary to the previous alternative where runtime's developers decide what to include, another alternative is to let developers decide what each application needs and create a specific and customized runtime for it.

JITS - Java In The Small - is a tool that allows the developer to customize the runtime to avoid the need of embedding unused packages or features [Courbot 2005, Courbot 2010]. The main problems JITS tries to solve is that the JRE is too large to fit in embedded devices and the mentioned drawbacks of solutions like J2ME or KVM.

The idea of JITS is to develop using standard Java and then during deployment it creates a tailored JRE according to its runtime usage. In addition, there is a process which is before the deploy and done off-board (on the development machine which is different from the target device) which is called Romization. This process creates a memory image containing all the objects (they are already initialized). This image is quite similar to the Smalltalk concept of image. Finally this ROM image can be burnt into a physical memory. Since only used packages are added, the developer does not loose features as the specialization depends on the application usage.

To save more memory, JITS modifies all the Java virtual machine and all its internal representation (tables) to use as less memory as possible. For example, reflection needs certain information at runtime. However, some applications do not need it and hence, we can save memory by discarding them. To detect which parts are used or not by the application, JITS does a call graph analysis for the current thread to mark all possible paths. All fields, methods and classes that were not referenced in this call graph can be removed.

Similar solutions are implemented in other programming languages. For example, VisualWorks Smalltalk provides a runtime packager[2], which makes smaller runtimes by explicitly removing classes and packages.

Still, with this strategy, developers need to know *with absolute certainty* what is required by their applications. At development stage, it is difficult, time-consuming and sometimes impossible to figure out what an application needs for all possible execution paths even with static analyzers. Most static analyzers do not take into account reflective

[2]Explained in VisualWorks Application Developer's Guide

features which are often used to support application evolution and dynamic code loading [Bodden 2011].

Table 3.4 evaluates custom and specific runtimes with the requirements.

| | **Custom and Specific Runtimes** |
|---|---|
| Efficient Object-Based Swapping Unit | — |
| Uniformity | ○ |
| Reversibility | ○ |
| Automatic Swapping In | — |
| Automatic Swapping Out | — |
| Transparency | ○ |
| Controllability at the Application Level | ● |
| Portability | ● |

Table 3.5: Custom and Specific Runtimes evaluation.

### As a Conclusion

*Custom and specific runtimes have almost the same drawbacks as specialized runtimes. The main difference is that at least the application developers take the decision of what to reduce. Nevertheless, with this strategy, developers need to know* with absolute certainty *what is required by their applications. At development stage, it is difficult, time-consuming and sometimes impossible to figure out what an application needs for all possible execution paths even with static analyzers.*

## 3.4 Object Faulting, Orthogonal Persistence and Object Databases

### 3.4.1 Object Faulting

Distributed object managers have been using object faulting for a while [Decouchant 1986]. The goal of such system is to share objects over a network. To achieve that, they use proxy objects [Gamma 1993] that intercept messages and then forward the information to an object manager. The object manager takes care of object migration, objects storage and reclamation, among other functionalities.

Persistent programming languages combine the features of database systems and programming languages. To achieve this the language must provide a mechanism for the detection and handling of references to and from primary and secondary memory.

In these systems, access to persistent objects is detected and managed by the object faulting mechanism, which triggers an automatic retrieval of objects from secondary memory to primary memory.

Hosking et al. [Hosking 1993] analyzed and compared a number of implementations of object faulting. They studied the mechanism by which references to non-resident objects

are detected, and the way in which the object faults themselves are handled.

They also explored an orthogonal design choice: how many objects should be made resident per object fault? Naturally, when there is an object fault, at least that object must be swapped in. Moreover, swapping in that object may imply swapping in other objects also. On the one hand, this can decrease the number of object faults. On the other hand, it can also result in more objects loaded in memory than it is actually needed.

Hosking et al.'s results are conclusive in establishing that software object fault detection mechanisms can provide performance surpassing the performance of comparable hardware-assisted schemes like page-faulting. Object swizzling means converting references between in-memory and on-disk addresses [Moss 1992, Wilson 1991, Kemper 1995]. They also showed that it pays to be eager in object swizzling by swizzling related objects in advance of the application's need for them.

### 3.4.2 Orthogonal Persistence and Object Databases

Orthogonal persistence uses object faulting, pointer swizzling, and read barriers to support transparent storage of objects on disk. However, even if orthogonally persistent systems [Marquez 2000, Hosking 1999, Atkinson 1995, Atkinson 1996, Hosking 1993, Moss 1990] and object databases (ODBMS) [Butterworth 1991] may look similar to a virtual memory manager, they have several differences. The most important one is that their goal is to automatically *persist* a graph of objects into a non-volatile memory. They do not *swap* graphs and, therefore, they do not take into account objects from outside the graph referencing objects inside. Furthermore, they provide features such as transactions, security, fault recovery, queries or query execution.

Another difference with object databases is that, with them, objects live permanently in secondary memory and are temporally loaded into primary memory when needed. When not needed anymore, they are deleted. In traditional virtual memory schemes, objects live in primary memory and they are swapped out when not needed or when the users decide. Moreover, in databases, the user has to always explicitly save data while virtual memory implementations commonly provide automatic graph swapping.

Orthogonal persistence and object databases are evaluated with our requirements and Table 3.6 shows the results.

| | **Orthogonal Persistence and ODBMS** |
|---|---|
| Efficient Object-Based Swapping Unit | ◐ |
| Uniformity | ○ |
| Reversibility | ● |
| Automatic Swapping In | ◐ |
| Automatic Swapping Out | ○ |
| Transparency | ◐ |
| Controllability at the Application Level | ● |
| Portability | ● |

Table 3.6: Orthogonal Persistence and Object Databases evaluation.

### As a Conclusion

*Although a virtual memory manager shares some functionalities with object databases and orthogonal persistence, they both solve different problems. First, object databases and orthogonal persistence only manage data. Second, most databases need user iteration to decide when to persist or when to query. Some ODBMS provide semi-automatic persistence and queries, but still the user needs to clearly mark when a transaction starts and finishes.*

## 3.5 Melt: Tolerating Memory Leaks

Melt [Bond 2008] implements a tolerance approach that safely eliminates performance degradations and crashes due to leaks of dead but reachable objects. The strategy is to have sufficient disk space to hold leaking objects.

Melt assumes that stale objects (objects the program has not used for a while) are likely leaks and moves them to disk. If they are needed later on, Melt brings them back into primary memory.

Similar to LOOM, Melt uses 32 bis for in-memory addresses and 64 bits for the stale space. It uses swizzling [Moss 1992, Wilson 1991] to convert references between 32-bit in-memory and 64-bit on-disk addresses. When an object is moved to secondary memory, a *mapping stub object* is created in primary memory that stores the long address. In-memory objects that refer to stale objects refer to these mapping stub objects. References from stale objects to in-use objects are problematic because the GC may move in-objects around. To solve this problem Melt creates *scion* objects in primary memory that hold a reference to the original in-memory object. These *scion* objects are not moved by the GC. Stale objects then refer these "scion" objects rather than referring directly the in-memory objects.

Melt is implemented in a Java JVM using a copying generational collector. To identify stale objects, Melt modifies (1) the compiler to add read barriers (that causes the object faulting) to the application and (2) the garbage collector to mark heap references and objects stale.

The results of Melt's evaluation are presented in Table 3.7.

| | **Melt** |
|---|---|
| Efficient Object-Based Swapping Unit | ◐ |
| Uniformity | ○ |
| Reversibility | ● |
| Automatic Swapping In | ● |
| Automatic Swapping Out | ● |
| Transparency | ● |
| Controllability at the Application Level | — |
| Portability | ○ |

Table 3.7: Melt evaluation.

### As a Conclusion

> *Because of the nature of memory leaks, Melt assumes that most stale objects are only referenced from other stale objects. If this were not true, then the memory used internal structures of Melt can be even more than the memory we are able to release. Moreover, Melt had good results because its goal is to tolerate leaks* i.e., *memory that never be used again. Performance can be a problem if objects get swapped back in.*
> *In those scenarios where the working set is not all in primary memory* e.g., *a virtual memory manager, it is common to have objects in primary memory referencing stale objects. Moreover, it is common that a swapped graph is loaded back. Because of this, Melt cannot be used as an efficient virtual memory manager.*

## 3.6 General-Purpose Object Graph Swappers

An object graph swapper is a tool that swaps object graphs between primary and secondary memory. Such tools can be used to swap out unused objects and hence release primary memory. With an efficient object graph swapper we can build a virtual memory as presented in the next chapter.

### 3.6.1 ImageSegment Object Swapping Principles

ImageSegment is an object graph swapper and serializer for Squeak Smalltalk developed by D. Ingalls [Ingalls 1997]. We have performed a detailed analysis and an exhaustive experimentation with ImageSegment [Martinez Peck 2010a]. The following is the results of such investigation.

In the ImageSegment's object swapping implementation, there is a list of user defined root objects that are the base of the graph to swap. The graph is then stored in a "segment" which is represented by an instance of the class ImageSegment and contains three sets of objects:

1. *root objects*: these objects are provided by the user and should be the starting point of object graph,
2. *inner objects*, and
3. *shared objects*.

Once the graph is stored in the segment, it can be swapped to disk and the original objects are removed from the Smalltalk system.

In Section 2.2 we call *shared objects* to those objects of a graph that are accessed, not only through the root, but also from outside the graph. If the swapping unit are object graphs as it happens with ImageSegment, detecting and correctly handling shared objects of a graph is a challenging task. This is because there are objects outside the graph with references to objects inside the graph we want to swap.

This is important because it is common to have shared objects inside graphs. In addition, since the system may allow the programmer to freely select any object graph to swap, the probability to get shared objects increases. The problem is that there is no easy or incremental way of detecting *shared objects* because, in most programming languages, objects do not have back-pointers to the objects that refer to them. Hence, a costly full memory traversal is often required *e.g.,* ImageSegment uses garbage collector facilities for such task. In addition, with ImageSegment only the *inner objects* are swapped.

### 3.6.2 Swapping Out and In

The mechanism to swap out works the following way: first, the user provides a *list* of roots. The segment is created and the inner and shared objects are computed. Then, the segment can be swapped out and the root objects are replaced by proxies. The *inner and root objects* of the graph are then written into a file. Once the roots are replaced by proxies, there are no more references from outside the graph to the objects that were written into the file and, therefore, the garbage collector deletes them. As a consequence of this, an amount of memory is released.

To install back the segment from file, there are two different ways:

- Sending any message to one of the proxy objects: remember that roots were replaced by proxies. So, all the objects that were pointing to roots are now pointing to proxies. Whenever a proxy receives a message it will load back the object graph in memory.
- Sending a provided message to the ImageSegment instance (the segment).

### 3.6.3 Evaluation of ImageSegment

**Objects graphs as the swapping unit.** ImageSegment considers object graphs as the swapping unit and we believe this is a step forward (Section 4.2 explains the reasons) in the definition of an object-based virtual memory manager.

**Smart usage of GC facilities.** Doing a full memory traverse to detect shared objects is slow. In fact, Section 4.4 shows our efficient solution to this problem. Nevertheless, ImageSegment is smart by directly using GC facilities to perform this task rather than implementing it from scratch.

**Shared objects are not swapped.** *Shared objects* are not swapped. Moreover, there is an array which remains in primary memory that refers to shared objects. The real problem is that it is difficult to control which objects in the system are referencing objects inside the subgraph. For that reason, most of the times there are several *shared objects* in the graph. The result is that the more *shared objects* there are, the less memory that can be released.

**No support for graph intersections** Object graphs are complex and it is inevitable that a programmer may end up trying to swap out a graph which contains proxies introduced by the swapping of another graph. We call this situation *graph swapping intersection* and it is

an issue that ImageSegment does not support. Therefore, *swapping out* a graph may lead to *swapping in* another unnecessary graph. Even worse, graphs can be swapped in with an inconsistent shape.

**Cannot swap classes and methods.** Since in Smalltalk classes and methods are normal objects and they play an important role in the runtime infrastructure, it is interesting to be able to swap them. This is not solved by ImageSegment. One problem is that there are several objects outside the graph that may be pointing to a class. For example, depending on the Smalltalk implementation, a class can be referenced from its metaclass, from *SmalltalkDictionary*, from its subclasses or superclasses or from its instances[3]. This makes swapping classes very difficult to achieve and the same problem happens when trying to swap methods.

**Implementation.** An important remark is that the steps of marking all objects in the image, traversing the object graph, and calculating shared objects are all implemented on the virtual machine side. That means that most of the algorithms and complexity of ImageSegment is on the VM side. They are implemented as primitives and the main problem with this is that one does not have control over it, while also having the disadvantages listed in Section 1.3.

**Performance.** ImageSegment receives a user defined graph and it needs to distinguish between *shared objects* and *inner objects* [Martinez Peck 2010a]. To do that, it has to do a full memory traversal using the garbage collector infrastructure. In addition, to replace root objects with proxies ImageSegment uses the Smalltalk become: primitive which requires another full memory traversal. Such traversals negatively affects ImageSegment's performance. On the other hand, ImageSegment improves its performance because it is implemented at the virtual machine level.

Table 3.8 shows the evaluation of ImageSegment regarding the listed requirements.

| | **ImageSegment** |
|---|---|
| Efficient Object-Based Swapping Unit | ● |
| Uniformity | ○ |
| Reversibility | ● |
| Automatic Swapping In | ● |
| Automatic Swapping Out | ○ |
| Transparency | ◐ |
| Controllability at the Application Level | ● |
| Portability | ○ |

Table 3.8: ImageSegment evaluation.

[3]This is an implicit reference because objects do not have an instance variable with its class but a pointer to it in the object header.

### As a Conclusion

> *The fact that shared objects are not swapped can be a serious usability concerns. This is a key observation relevant for any serious use of ImageSegment. Its algorithm requires a full memory traversal and its implementation is done at the virtual machine level. Finally, it is not able to swap important objects such as classes or methods.*

## 3.7 Summary

In this chapter we show that the mentioned approaches use object faulting, pointer swizzling, read barriers or proxies and swapping between primary and secondary memory. We also demonstrate that the problems of managing unused memory are many and varied:

- The approach may degrade or limit the runtime where applications run on.
- The developer cannot influence nor has control over the virtual memory scheme.
- The swapping unit is normally objects individually or pages.
- The implementation of the virtual memory is at the virtual machine level or even at OS level.

Table 3.9 shows a summary of the requirements evaluation on related work. The conclusion is that there is no solution that satisfies all the defined requirements.

The next chapter introduces our approach which is fully implemented in an object-oriented programming language and is decoupled from the GC and from the OS' virtual memory. Its key point is that the swapping units are object graphs and the developer can influence on what and when to swap.

| | OS Virtual Memory | Mach, V++, Apertos | Krueger et al. | Exokernel | LOOM | Reduced and Specialized Runtimes | Custom and Specific Runtimes | Orthogonal Persistence and ODBMS | Melt | ImageSegment |
|---|---|---|---|---|---|---|---|---|---|---|
| Efficient Object-Based Swapping Unit | ○ | ○ | ○ | ○ | ◐ | — | — | ◐ | ◐ | ● |
| Uniformity | ● | ● | ● | ● | ◐ | ○ | ○ | ○ | ◐ | ○ |
| Reversibility | ● | ● | ● | ● | ● | ○ | ○ | ● | ● | ● |
| Automatic Swapping In | ● | ● | ● | ● | ● | — | — | ◐ | ● | ● |
| Automatic Swapping Out | ● | ● | ● | ● | ● | — | — | ○ | ● | ○ |
| Transparency | ● | ● | ● | ● | ● | ○ | ○ | ◐ | ● | ◐ |
| Controllability at the Application Level | ○ | ◐ | ● | ● | ○ | ○ | ● | ● | — | ● |
| Portability | ○ | ○ | ○ | ○ | ○ | ● | ● | ● | ○ | ○ |

Table 3.9: Summary of the requirements evaluation on related work.

CHAPTER 4

# Marea, A Model for Application-Level Virtual Memory

## Contents



## At a Glance

This chapter introduces an overview of the approach defended in this dissertation. This approach is a composition of four subsystems. We explain each part and how together they represent our solution.


**Keywords:** Object swapping, object faulting, unused objects, virtual memory, serialization, proxy.

## 4.1 Introduction

Chapter 3 shows the lack of approaches that fulfill our described requirements: *efficient object-based swapping unit, uniformity, reversibility, automatic swapping in, automatic swapping out, transparency, controllability at the application level and portability*. In this chapter, we present Marea, our solution to the problem of unused memory in object-oriented programming languages.

### Structure of the Chapter

In the next section we provide an overview of Marea, our virtual memory manager and its subsystems. Section 4.3 describes the problem of shared objects in graphs. In Section 4.4 we explain Marea's object swapper algorithms. How Marea solves the problem of graph intersections is detailed in Section 4.5. Finally, Section 4.6 summarizes the chapter.

## 4.2 Marea Overview

Our solution to provide an application-level virtual memory for object-oriented languages is to temporarily move object graphs to secondary memory releasing part of the primary memory [Martinez Peck 2012a]. In particular, our solution addresses the problem of shared objects: objects that are swapped but also referenced from outside the swapped graph.

The input for Marea are *user-defined* graphs, *i.e.,* graphs that the user wants to swap out to secondary memory. Swapped graphs are swapped in as soon as one of their elements is needed. The graphs to swap can have *any* shape and contain *almost any* type of object[1]. This includes classes, methods and closures which are all first-class objects in Marea's implementation language: Pharo [Black 2009].

When Marea swaps a graph, it correctly handles all the references from outside and inside the graph. When one of the swapped objects is needed, its graph is *automatically* brought back into primary memory. To achieve this, Marea replaces original objects with proxies [Gamma 1993]. Whenever a proxy intercepts a message, it loads back the swapped graph from secondary memory. This process is completely transparent for the developer and the application. Any interaction with a swapped graph has the same results as if it was never swapped.

Considering object graphs as the swapping unit is a key aspect of our design because:

- We only need a few proxies per graph, hence we increase the memory we are able to release. As we see later in Section 4.4 we only need proxies for the root of the graph and for the objects which are also referenced from outside the graph.

- On the one hand, we decrease the number of unnecessary objects swapped in. If an object is needed and it is swapped in, it is likely it will need its related objects in the future. When considering a page as the swapping unit, for example, there is

[1] In the current implementation of Marea there are only a few objects that cannot be swapped as explained in Section 8.3.1.

usually no relation between the objects that are placed into the same page. In the contrary, with graphs, the objects are connected. On the other hand, if object graphs are too large, we may be swapping in more unnecessary objects than if grouping them in pages or groups. This is the reason why as part of the future work proposed in Section 10.3 we discuss about *partial loading*.

- We decrease the number of object faults. When an object is swapped in, its related objects are also swapped in, avoiding more object faults.

One of our main challenges is how to efficiently handle shared objects. Section 4.3 explains why this is a challenge and Section 4.4 shows how Marea solves it.

### 4.2.1 Marea Subsystems

Figure 4.1 shows that Marea is built on top of four main subsystems: (1) object graph swapper, (2) an advanced proxy toolbox, (3) serializer and (4) object storage. We describe them in the following.

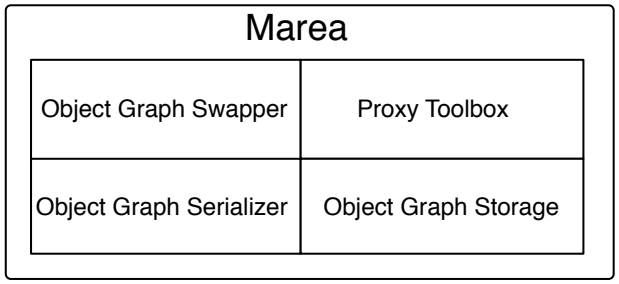


Figure 4.1: Marea subsystems.

**Object Graph Swapper.** The most important subsystem is the object graph swapper (OGS). Its task is to efficiently swap graphs between primary and secondary memory. As we explain later in Section 4.3, detecting and correctly handling the *shared objects* of a graph is a challenging task that the OGS addresses. The OGS uses a serializer and a proxy toolbox to save, load and replace objects.

**Proxy Toolbox.** The OGS replaces some objects of the graph[2] being swapped with proxies [Gamma 1993]. Since proxified objects are referenced from other objects in the system, the OGS needs to update all objects in the system which have a reference to the replaced object so that they point to its proxy. We refer to this functionally as *object replacement*.

Marea uses Ghost [Martinez Peck 2011a, Martinez Peck 2012b], a proxy library able to handle any kind of object (plain objects, classes, methods and other language runtime objects) in a transparent way. We explain this library in detail in Chapter 7.

[2]Section 4.4 explains which objects are actually replaced by proxies.

**Object Serializer.** When an OGS needs to swap out a graph, the first step is to serialize it. Marea is decoupled from the order in which the serializer writes or reads objects from the stream of serialized objects, *i.e.,* Marea can use serializers with different traversal techniques.

An important Marea's requirement is to have access to a fast serializer. When a swapped out object is needed, it is essential to be able to load it back as fast as possible because it is being needed *at that exact moment*.

Furthermore, the serializer must be able to correctly serialize and materialize any kind of objects such as classes, methods or closures in accordance with the implementation language. It should also be flexible enough to be customized to meet the special needs of Marea (explained in Section 8.2.3). To fulfill these requirements, Marea uses Fuel [Dias 2011, Dias 2012]. We describe Fuel in Chapter 6.

**Object Graph Storage.** Its main responsibility is to store and load serialized graphs (each serialized graph is an array of bytes). The graph storage responsibilities are reified in a separate class following the strategy design pattern [Gamma 1993] and hence Marea delegates to such strategy. This allows Marea to easily support different backends and the user can choose which one to use or even create its own. Current backends are the local file system, Riak[3] and MongoDB[4] NoSQL databases.

### 4.2.2 Evaluating Requirements on Marea

In Section 2.4 we list the requirements that a novel virtual memory manager for dynamic languages should satisfy and in Chapter 3 we have evaluated those requirements on each related work. This chapter evaluates those requirements on Marea.

| | **Marea** |
|---|---|
| Efficient Object-Based Swapping Unit | ● |
| Uniformity | ● |
| Reversibility | ● |
| Automatic Swapping In | ● |
| Automatic Swapping Out | ○ |
| Transparency | ● |
| Controllability at the Application Level | ● |
| Portability | ● |

Table 4.1: Marea evaluation.

Table 4.1 provides the summary of the evaluation. The following are the reasons of such results:

- **Efficient Object-Based Swapping Unit.** First, Marea's swapping unit consist of objects. Second, it does not swap objects individually but graphs of objects. This generates few object faults and when swapping in, it loads few unnecessary objects.

[3] http://wiki.basho.com/Riak.html
[4] http://www.mongodb.org/

- **Uniformity.** Marea can handle all kinds of objects whether they represent code, runtime entities, application data, etc.
- **Reversibility.** Marea does not remove objects but instead it swaps them.
- **Automatic Swapping In.** As soon as a swapped object happens to be needed, it is automatically swapped in.
- **Automatic swapping out.** This is not currently available in Marea. Section 10.3 explains that this is the natural future work of this dissertation.
- **Transparency.** From the point of view of an application, Marea is transparent in the sense that the application will get the same results whether it is using Marea or not. From the application's development point of view, the application code is not polluted with code related to swapping. Instead, Marea's code is totally decoupled from the application. However, there is the triggering of the swapping out if the user wants to do it when an application-level event occurs.
- **Controllability at the Application Level.** It allows application programmers to influence or decide what, when and how to swap. This opens up many new possibilities as we can take domain knowledge into account for the decision of what to swap.
- **Portability.** Our solution is decoupled from both, the OS and the virtual machine.

## 4.3 The Main Challenge: Dealing Efficiently with Shared Objects

This section presents the main challenge that an OGS should address: the case of *shared objects*. To explain the problem, we use the example of an object graph surrounded by a rectangle in Figure 4.2. We discuss the issues of detecting objects shared with other graphs, as well as alternatives for handling them. In Section 4.4, we describe in detail how Marea deals with them.

### 4.3.1 The Case of Shared Objects

Detecting and correctly handling shared objects of a graph is a challenging task that an OGS should address.

In the example of Figure 4.2, object `D` has to be considered because it references object `C` which is part of the graph to be swapped out.

Whether *shared objects* should be swapped or not, is an important decision that the solution should address. In any case, it is necessary to correctly deal with them. This is important because it is common to have shared objects inside graphs. Furthermore, since Marea allows the programmer to freely select any object graph to swap, the probability to get shared objects increases.

The following illustrates the problems introduced by shared objects. Figure 4.3 shows that if we simply replace all objects of the graph with proxies, we still need to know that

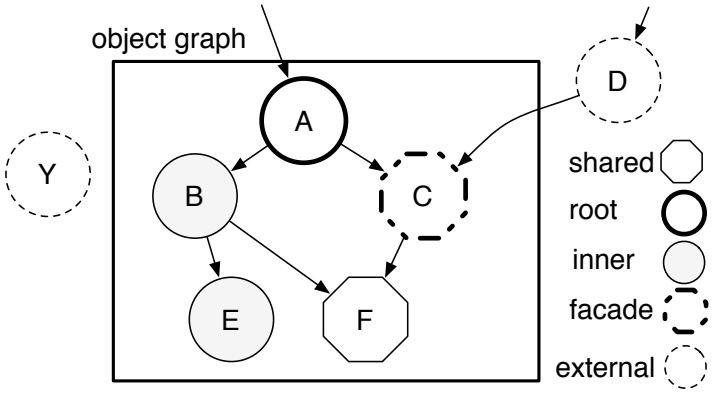


Figure 4.2: Object classification based on graph structure.

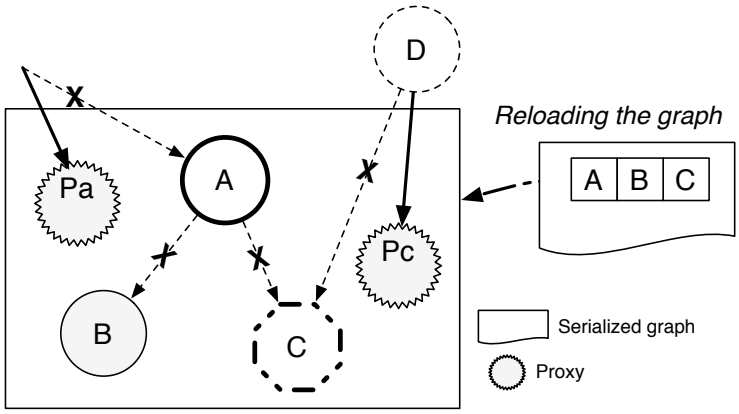


Figure 4.3: The need to manage shared objects.

object C is shared. Otherwise, after having reloaded the graph, D will continue referencing Pc. The correct behavior is to recreate the same graph as it was before the swapping. Therefore, when we reload the graph, D should be updated to refer the materialized C.

The problem is that there is no easy or incremental way of detecting *shared objects* because objects do not have back-pointers to the objects that refer to them. Hence, a costly full memory traversal is often required as explained in Section 3.6. By using the garbage collection infrastructure, ImageSegment identifies which objets of the graph are *inner* and which ones are *shared*. The drawback is the overhead and time spent to do the traversal of the whole memory which is exactly what we want to avoid.

By using weak collections that hold references to proxies and letting the garbage collector do its job, Marea provides a fast and novel approach that avoids a full memory scan as described in Section 4.4.

### 4.3.2 Handling Shared Objects

There are two approaches to deal with shared objects: (1) detect and swap them; (2) detect but do not swap them. Not swapping shared objects can be a limitation because it is common to have objects from outside the graph referencing objects inside it. For example, Section 5.4 shows that, during our experiments, a typical object graph for a class contained 17% of shared objects. Since the goal of Marea is to release as much memory as possible, it implements the first option, *i.e.,* it also swaps shared objects as we present in Section 4.4.

## 4.4 Marea's Object Swapper Algorithms

Marea enables programmers to freely specify which graphs have to be swapped out. As a consequence, it has to support graphs of any shape.

Marea provides an efficient approach to detect and correctly handle shared objects while avoiding a full memory scan. Figure 4.4 shows a small object graph that we use as an example.

Marea creates a proxy for *every* object of the graph (whether it is a root, an inner or a shared object). Then each of them is replaced by its associated proxy (previous object pointers are now pointing to the associated proxy). As a result, proxies for inner objects are not referenced from any other object. Indeed, inner objects were *only* referenced from inside the graph and all objects were replaced by proxies. Hence, as soon as the GC runs, it will garbage collect all inner objects and all proxies leaving only those proxies for *facade objects* and the root of the graph.

During the swap in, the graph is materialized and all proxies associated with the graph are replaced by the appropriate materialized objects. The subsequent sections explain in detail the swapping out and in operations.

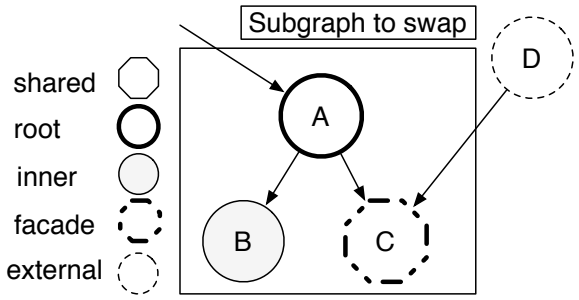


Figure 4.4: A graph to be swapped.

### 4.4.1 Swapping Out

The swapping out phase is triggered explicitly, *i.e.,* the programmer has to instruct the OGS to swap out a graph. In object-oriented systems, runtime memory is composed of an object graph in which objects refer to other objects via instance variables. Objects are nodes of such a graph, while references (*e.g.,* instance variables) are the edges of it. Hence, an object

plus its instances variables can always be considered as a graph. A user may provide an object with instance variables that refer to other objects which also refer to other objects and so on in several levels. For Marea, the boundaries of a graph is its transitive closure. So the way to refer a graph in Marea is simply by referring an object. That object can even be a collection that refers to several objects.

Here is the code to swap out the example of Figure 4.4:

```
| a b c d |
b := 'b'.
c := 'c'.
a := ClassWith2InstVars new.
a instVar1: b.
a instVar2: c.
d := ClassWith1InstVars new.
d instVar: c.
MARObjectGraphSwapper new swapOutGraph: a.
```

Marea's strategy to *swap out* object graphs decomposes into the following steps. As an example, we will swap out the graph of Figure 4.4.

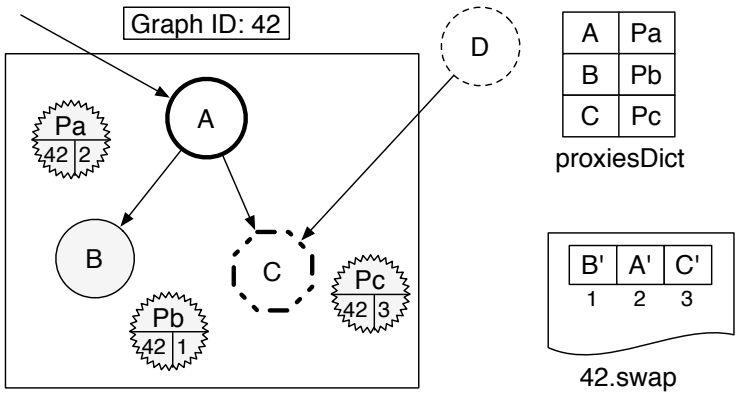


Figure 4.5: Swap out: the object graph is serialized and proxies are created.

1. *Initialization:* Marea assigns an automatically generated and unique ID (a number) to each graph. In this example, it assigns the number 42.

2. *Serialize the object graph:* with the default configuration, Marea serializes the graph into a single file located in a secondary memory (*e.g.,* on hard disk). The filename is the graph ID (42.swap in our example).

3. *Create proxies:* Marea creates a proxy for each object of the graph. We introduce proxiesDict, an identity dictionary that maps objects to proxies. In the example, it maps A, B, and C to their respective proxies, namely Pa, Pb, and Pc. The result is depicted on Figure 4.5[5]. We label the objects serialized into the serialization stream

[5]For sake of clarity, we do not show the object references from structures like proxiesDict that are external to the graph.

as "prime". For example, the object A is the original one and A' represents its serialized version. The stream with contents B' A' C' represents the serialized graph A B C. Each proxy knows the position in the stream of the proxified object apart from its graph ID. Storing the position allows Marea's logic to be decoupled from the order followed by the serializer to write and read objects from the stream. This information stored in the proxy is then used during the swap in process which is explained in Section 4.4.2.

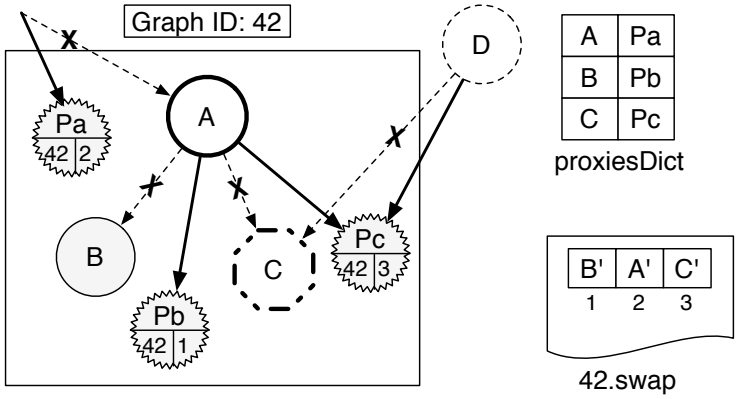


Figure 4.6: Swap out: objects are replaced by proxies.

4. *Replace original objects with proxies:* all references to an original object are updated and changed to point to the appropriate proxy as shown in Figure 4.6. We replace each key (original object) of proxiesDict with its associated value (its proxy). This means that Marea needs the programming language to be able to swap the references between two objects[6]. In the example, A now points to Pb and Pc, and D points to Pc.

5. *Update GraphTable*: the OGS maintains a global table called GraphTable. This is a dictionary in which a key is a graph ID and a value is a collection holding *weak* references to the proxies associated with the graph. We need the GraphTable because during the swap in of a graph we need to retrieve all its proxies and replace them with the appropriate materialized objects. In our example, this step consists of adding graph 42 (*i.e.,* the graph which ID is 42) into GraphTable as shown in Figure 4.7.

6. *Cleaning:* we can now discard the temporary proxiesDict. Once this step is done, none of the internal objects are strongly referenced anymore, *i.e.,* there are no strong references to objects A, B or C. Consequently, when the next GC runs, all these objects are removed. In addition, all weak references to those proxies for inner objects are replaced by *nil* (cleared) in the GraphTable. Figure 4.8 shows the final result after a GC execution. It only remains in memory the proxy for the *root* (Pa in this example) and the proxies for the *facade objects* (only Pc in this example).

[6]In Marea's implementation this is solved by using a reflective Smalltalk operation called "become". This is explained with more details in Section 8.2.2 and Section 8.3.1.

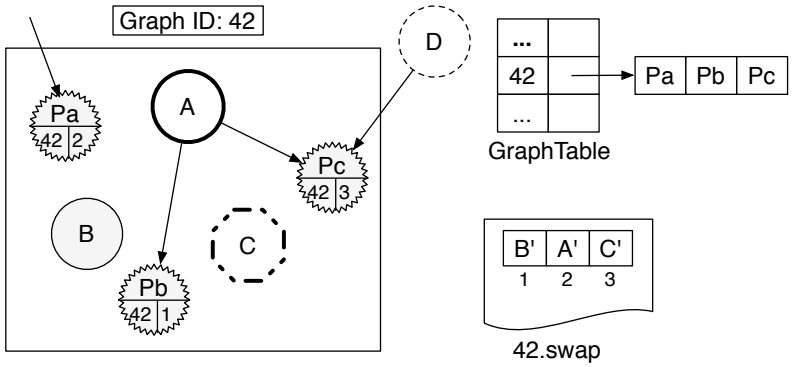


Figure 4.7: Swap out: Updating GraphTable and final result.

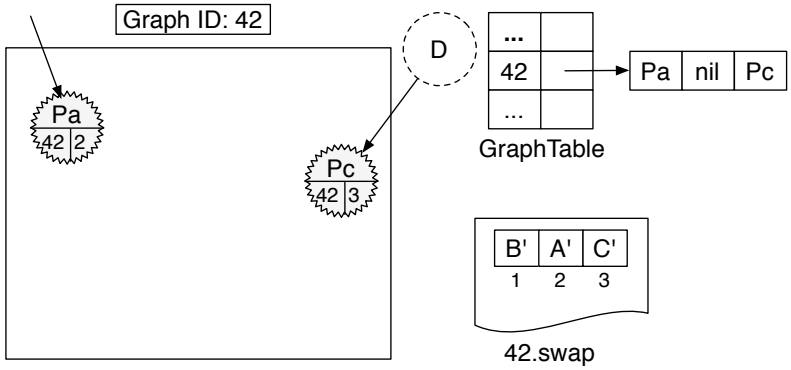


Figure 4.8: Swap out: final result after a GC execution.

### 4.4.2 Swapping In

The swap in of a graph is triggered when one of its proxies is accessed, for example, via a message send or an instance variable access. This situation, *i.e.,* when the system needs an object which was swapped out, causes what is known as "object faulting" [Hosking 1990].

When a proxy intercepts an action it has certain information about it. For example, if the action is a message send, the proxy knows which message was sent and its arguments. The proxy passes all the interception's information to a handler. The handler first makes the OGS to swap in the graph associated with the proxy. Then, the handler forwards the original message to the object that was initially replaced by the proxy. Section 8.2.1 gives more details about Marea's classes and their responsibilities.

We continue our example from Figure 4.8. We assume that the proxy Pa receives a message. The resulting *swap in* decomposes into the following steps:

1. *Materialize the object graph:* this is done by first getting the file named after the proxy's graph ID. Once we have the stream, we materialize the object graph (with all the objects references) into primary memory.

2. *Associate proxies with materialized objects:* during the swapping out, we have stored in each proxy the graph ID and the position in the stream of the corresponding proxi-

fied object. With this information, we can identify the materialized object associated with a given proxy. Figure 4.8 shows that Pa knows that it belongs to the graph of ID 42 and that it represents the object at the position 2 in the stream of this graph. That way we know that we will replace Pa with A'.

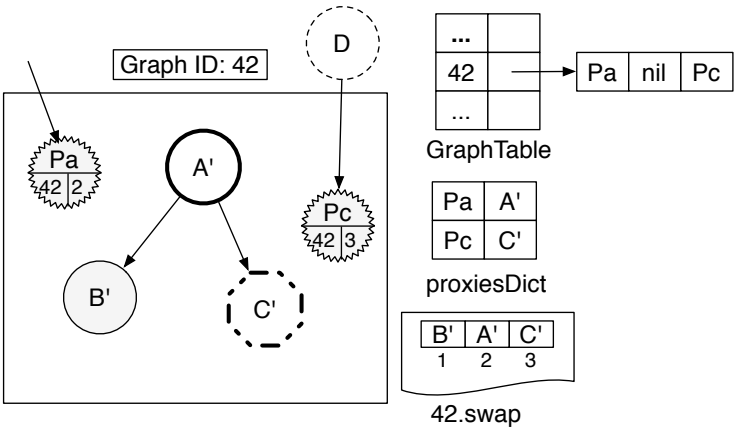


Figure 4.9: Swap in: object graph is materialized and proxies are associated

Similar to what we do when swapping out, we build a temporary proxiesDict (see Figure 4.9) in which the keys are proxies and the values are the associated materialized objects. GraphTable let us retrieve the list of proxies given a graph ID. This list includes only *alive* proxies *i.e.,* proxies that have survived GC because of being referenced. This means that the list of proxies may contain nil entries which have to be ignored. In our example, this is the case of Pb which has been removed. Since no external object references B', it can be materialized without any object replacement.

3. *Replace proxies with original objects:* all references to each proxy are updated so that they now point to the corresponding materialized object. The proxies we need to replace are those that are stored as keys in the proxiesDict and their associated materialized objects are the values of the dictionary. Once the replacement is finished, we discard proxiesDict. Finally, all proxies corresponding to the graph are removed by the GC. Figure 4.10 illustrates the result of this step.

4. *Cleaning:* the current graph is removed from the GraphTable and the file for the serialized graph is deleted[7]. Figure 4.11 presents the final state after a GC run. The result of the swapping in is a graph equal to the one that was swapped out, as Figure 4.4 shows. Notice that even if the materialized objects are called A', B' and C', they are equal to A, B and C.

**Pre- and post-actions.** Some objects may have to execute specific actions before being swapped out or after being swapped in. Another problem while swapping object graphs is

[7]Marea's current implementation offers an API to trigger an automatic compaction of the GraphTable. Such compaction can also be done automatically after each GC.

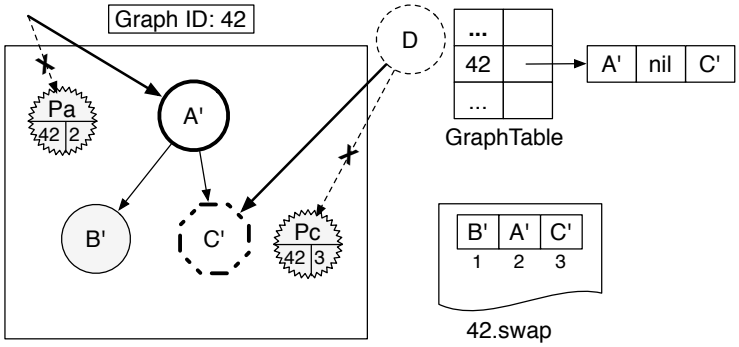


Figure 4.10: Swap in: proxies are replaced by materialized objects.

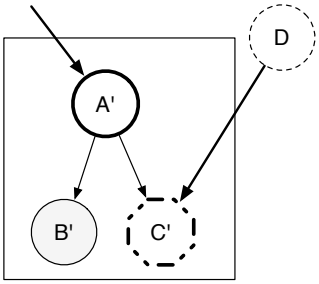


Figure 4.11: Swap in: final result.

to detect implicit information that is necessary to correctly load it back later [Ungar 1995]. For example, when swapping out Smalltalk objects like true, false, nil, Transcript or Processor, we do not want to recreate them when swapping in. Instead, we have to refer to the already existing ones. Two situations occur:

- Pre- and post-actions for serialization and materialization[8] . Most serializers provide a hook to execute a particular action once an object has been serialized or materialized. Some core objects and classes always require this hook for the correct serialization or materialization and this is automatically performed by the serializer. For example, hashed collections need to be rehashed after being materialized because they may refer to objects that have changed their hash. This hook may also be used by developers to define custom actions.

- Marea provides similar hooks so that objects can execute arbitrary actions before or after being swapped out or in. These hooks are used by Marea itself to handle standard objects that need this hook or by programmers to define actions for application-specific objects and classes.

[8] We use the concept of "materialization" to refer to the "deserialization".

## 4.5 Handling Graph Swapping Intersection

Object graphs are complex and it is inevitable that a programmer may end up trying to swap out a graph which contains proxies introduced by the swapping of another graph. We call this situation *graph swapping intersection* and it is an issue that should be addressed. Otherwise, *swapping out* a graph may lead to *swapping in* another unnecessary graph. Even worse, graphs can be swapped in with an inconsistent shape. In this section, we explain the problems related to graph swapping intersection and how Marea addresses them.

### 4.5.1 Problem of Shared Proxies

Following with the example of Figure 4.4, imagine that we swap out graph 42. We end up in the situation of Figure 4.8. Suppose that now the user wants to swap out a graph which includes the object D *e.g.,* graph 43. This raises the question of swapping proxies since the graph of D shown in Figure 4.8 includes a proxy resulting from the previous swapping out of graph 42. Indeed Pc is a proxy of a shared object. We call such a proxy a *shared proxy*.

Swapping graphs with proxies is an issue since it may lead to the loss of shared proxies which in turn results into corrupted graphs. In our example, imagine that we apply the regular swapping out mechanism for the graph of root D and, therefore, we swap out the proxy Pc as if it were a regular object. This will produce two proxies: Pd which replaces D and Ppc which replaces Pc. Once D is garbage collected, the only remaining references to Ppc are weak since they are from the GraphTable. This leads to Ppc being collected too. If now graph 42 is swapped back in, the result will be the correct materialization of A', B' and C'. The problem appears when swapping in graph 43. We will get D' referencing a proxy Pc' that has no relation at all with the already materialized object C' from graph 42. And if now Pc' receives a message, there will be an error because its graph (ID 42) has already been swapped in.

### 4.5.2 Solution part 1: Swapping Out with Shared Proxies

During the swapping out phase, Marea only creates proxies to replace plain objects (*i.e.,* objects that are not proxies). Proxies found in a graph are kept unchanged, and they are inserted into the SharedProxiesTable. This table is a dictionary where a key is a proxy ID and a value is a *strong* reference to the proxy. A proxy ID is unique in the system and it is easily computed from the graph ID and the position in the stream[9] *e.g.,* the proxy ID of Pc is 98346. Depending of the implementation, proxies can store the graph ID and the position and then compute the proxy ID when asked, or store the proxy ID and compute the graph ID and the position when asked. In either case, a proxy is always able to answer a graph ID, a position and a proxy ID.

Following our example, Figure 4.12 shows the result of swapping out, first, the graph 42 and, then, the graph 43 with the handling of shared proxies. When swapping out graph 43, Pc is added to both, GraphTable and SharedProxiesTable.

[9]The proxy ID is the concatenation at the bit level of both numbers, the graph ID and the position.

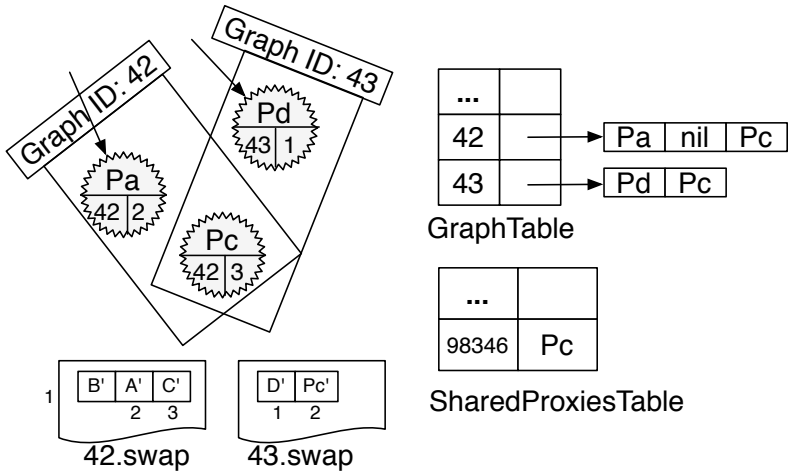


Figure 4.12: Swapping out two graphs with shared proxies. First, graph 42 is swap out and then, graph 43.

### 4.5.3 Solution part 2: Swapping In with Shared Proxies

To handle swapping in graphs with shared proxies, Marea uses the SharedProxiesTable that is filled during the swap out phase. The swap in algorithm performs as described in Section 4.4.2 except for proxies found in the materialized graph. Those proxies are shared ones. Each materialized shared proxy is replaced by the appropriate object from the SharedProxiesTable which is looked up based on the proxy's ID.

To illustrate the algorithm, consider again our example with two graphs of Figure 4.4. After swapping graph 42 (starting with object A) and graph 43 (starting with object D) in this order, we obtain a structure depicted by Figure 4.12. Now, two swapping in scenarios may occur: graph 42 is swapped before 43 or vice-versa. We will consider those scenarios and show that both graphs are correctly rebuilt.

In the first scenario, graph 42 is swapped in first. None of the materialized objects (A', B', and C') is a proxy. We simply replace proxies Pa and Pc found at entry 42 of the GraphTable with the right objects, namely: A' and C'. The replacement, which is system-wide, affects the SharedProxiesTable since C' replaces Pc. When graph 43 is then swapped in, Marea first replaces the proxy Pd found in entry 43 of the GraphTable by the materialized object D'. Entry 43 of the GraphTable also includes C' which is ignored because it is not a proxy. Then, Marea detects a shared proxy issue since there is a proxy (Pc') in the materialized graph. Next, we use the ID of Pc' to get C' from the SharedProxiesTable. Finally, Pc' is replaced by C'. Thus, both graphs are reconstructed correctly and sharing is preserved.

In the second scenario, graph 43 is swapped in first. Marea first replaces the proxy Pd by the materialized object D'. Entry 43 of the GraphTable also includes Pc which is ignored because its graph ID is 42 and not 43. Marea detects a shared proxy issue since there is Pc' in the materialized graph. We use the ID of Pc' to get Pc from the SharedProxiesTable. Next, we replace Pc' by Pc. When graph 42 is then swapped in, Marea replaces proxies

Pa and Pc found at entry 42 of the GraphTable with the right objects, namely: A' and C'. The replacement makes D' reference C'. Again, both graphs are reconstructed correctly and sharing is preserved.

### 4.5.4 Cleaning SharedProxiesTable

Since GraphTable contains weak references, Marea can listen when the GC clears those weak references and then do an automatic cleanup and compaction of the table. However, SharedProxiesTable holds strong references to its values (they can be proxies or normal objects). To clear this table, Marea needs to analyze GraphTable. If a proxy of SharedProxiesTable is not referenced from any entry in GraphTable, it means that all its related graphs were swapped in so it can be removed.

We could make Marea trigger this cleaning when swapping in a graph or to hook into the GC and trigger it automatically after each GC. However, this adds an unnecessary overhead for a small gain in memory. Because of this, the cleaning of SharedProxiesTable is done by Marea sporadically, that is, after having swapped in a customizable number of graphs. Nevertheless, the API allows application programmers to trigger such cleaning when desired.

## 4.6 Summary

This chapter presented Marea, our solution to provide application-level virtual memory. We described the overall approach and the four subsystems of Marea: (1) object graph swapper, (2) an advanced proxy toolbox, (3) serializer and (4) object storage. We detailed each step of the swapping algorithms. We also explained the problem of shared objects and graphs intersections and how Marea efficiently solves it.

The next chapter validates our overall approach by benchmarking Marea with real applications and shows the results in terms of speed and memory consumption. We performed several experiments to measure how much memory could be gained using Marea.

CHAPTER 5

# Marea's Validation

Contents



## At a Glance


This chapter validates our overall approach by benchmarking Marea with real applications and shows the results in terms of speed and memory consumption. We performed several experiments to measure how much memory could be gained using Marea.



**Keywords:** Validation, benchmarks, case studies, API.

## 5.1 Introduction

In Chapter 4 we explained that, to be able to implement a virtual memory fully on the language side and decoupled from the virtual machine, we need to build four main subsystems: (1) object graph swapper, (2) an advanced proxy toolbox, (3) serializer and (4) object storage. We describe them in the following.

In the next chapters we demonstrate that we can develop such subsystems at the language side and with no VM support. We have developed efficient algorithms to swap out/in object graphs that correctly deal with the problem of shared objets while also supporting graphs intersections. We developed a fast general-purpose serializer capable of serializing any type of object and flexible enough to be customized by a virtual memory manager. We also developed a proxy toolbox that is able to "proxify" any type of object and can intercept all kinds of messages. We implemented different object storage engine such a file-based one and one based on a NoSQL database.

As we show along this dissertation, Marea and all the related tools and subsystems were implemented on the language side, with an object-oriented design and with no virtual machine change. In this chapter, we present an overall validation of Marea. Next chapters deeply explain and validate the individual subsystems like the Fuel serializer and the Ghost proxies library.

### Structure of the Chapter

In the next section we give an introduction about how to use Marea. Section 5.3 describes the setup of our experiments and the used applications. Our experiments and benchmarks with real applications is described in Section 5.4. Section 5.5 measures the amount of unused objects in applications and the speed of Marea. Finally, Section 5.6 concludes the chapter.

## 5.2 How to Use Marea?

This section introduces the code needed to put Marea into action. The API of Marea is very straightforward. For example, the following line swaps out a regular graph:

```
MARObjectGraphExporter new  swapOutGraph: rootObject
```

Since the swapping in is automatic, a developer does not have to use it explicitly. Marea also allows the developer to decide which classes to swap. To accomplish that, Marea's API provides some helper methods such as swapOutClass:, swapOutClassAndSubclasses:, swapOutClassWithInstances: and swapOutClassWithSubclassesWithInstances:. The following is an example to swap out the maximum number of classes possible.

```
1 MARMareaExamples>>swapOutClassesForMaxPossiblePackages
2   | allClasses classesNotToSwap |
3   allClasses := self classesOfAllPackages.
4   coreClasses := MARObjectGraphExporter classesFromCorePackages.
5   classesToSwap := allClasses copyWithoutAll: coreClasses.
6   allClasses do: [:each |
7       MARObjectGraphExporter new swapOutClassWithSubclassesWithInstances: each].
```

In this example, each class is considered as a root of an object graph that is swapped out. The graph includes the method dictionary, compiled methods, the class organizer, and the rest of the transitive closure. In addition, it also includes the instances, the subclasses as well as the sub-instances.

In line 3, we obtain all the classes in the system. In line 4, we get the classes that should not be swapped out. Since certain classes are central to the execution of the Pharo system and Marea, we do not to swap them out. MARObjectGraphExporter maintains a list of packages and classes that cannot be swapped out. In line 5, we compute the list of classes that can be swapped and in lines 6-7 we swap them out.

The rest of the API involves messages to clean or reinitialize all Marea internal data structures and to configure which object storage to use, among others.

## 5.3 Experimental Setup

In this section we present all the configuration and description of our experiments. In addition, we provide information about the applications used to perform the experiments.

The following is the setup of our experiments.

1. We took the PharoCore 1.3 runtime as available from the Pharo project site[1]. PharoCore is a small environment with the minimal toolsets and libraries (no external packages or developer tools like refactoring engine or code assist). In addition, we used a PharoCore configuration dedicated for production usage that aggressively cleans caches, fonts, help information and other meta-data and also removes code *e.g.,* all unit tests resulting in a 6.9 MB runtime (excluding the VM size).

2. We loaded some real applications.

3. We wrote some scripts to swap out as many objects as possible considering classes or packages as the roots of the graph, while also to swap regular object graphs. The used scripts were similar to the ones shown by the example of Section 5.2.

4. We run the application over several real case scenarios *i.e.,* we used it and we run actions on it. While doing so, needed graphs were swapped in.

5. We measured the memory used and the swapping speed.

Then, we performed the same experiments but on a PharoCore *without* object swapping and analyzed the gained memory.

[1] http://www.pharo-project.org

**Environment Configuration.** Our experiments were run with PharoCore 1.3 build number 13327 and Cog Virtual Machine version 'CoInterpreter VMMaker-oscog-EstebanLorenzano.139'. The operating system was Mac OS 10.6.7 running in a 2.4 GHz Intel Core i5 processor with 4 GB of primary memory DDR3 1067 MHz. The swapping was done over files in the local filesystem creating one file per swapped graph.

**Applications.** The applications we use for the experiments were the same applications analyzed in Section 2.3.2 *i.e.,* DBXTalk CMS website, Moose suite, Dr. Geo, and Pharo infrastructure.

## 5.4 Benchmarks and Case Studies

In this section, we benchmark Marea with real applications and show the results in terms of speed and memory consumption. We performed several experiments to measure how much memory could be gained using Marea. Our experiments on real-world applications show that Marea reduced the memory footprint between 25% and 40%.

### 5.4.1 Swapping Out Code

Marea can swap different user-provided object graphs. However, a common scenario is when the user wants to swap out "unused code" to make its application's runtime smaller and less memory consuming.

**Results.**

| Application (root of the graph) | Size (MB) before swapping | Objects before swapping | Size (MB) after swapping | Objects after swapping | Size (MB) after experiments | Objects after experiments | Average objects per graph | Average % of shared objects per graph | Gain in size |
|---|---|---|---|---|---|---|---|---|---|
| DBXTalk (class) | 22.7 | 469882 | 10.6 | 254696 | 13.8 | 317107 | 170 | 17% | 40% |
| DBXTalk (package) | 22.7 | 469882 | 16.8 | 327594 | 19.7 | 386907 | 1340 | 24% | 13% |
| Moose (class) | 83.9 | 2908474 | 13.2 | 408549 | 63.7 | 2349701 | 670 | 15% | 25% |
| DrGeo2 (class) | 11.2 | 316212 | 6.9 | 174221 | 7.7 | 193086 | 232 | 17% | 31% |
| Pharo (class) | 11.7 | 242811 | 6.9 | 171460 | 7.2 | 153835 | 205 | 16% | 38% |

Table 5.1: Primary memory footprints showing the benefit of using Marea.

The resulting runtimes after the swapping were working correctly and all examples behaved normally. Table 5.1 shows that, after having navigated and used the applications, the amount of released memory was between 25% and 40% of the original memory footprint.

To explain the columns of Table 5.1, we consider the first line. "DBXTalk (class)" means that we are benchmarking the application DBXTalk and that we consider classes as roots. The table then presents the amount of memory used (in MB) and the total amount of objects before and after performing the swapping. After, it shows the same information but after having run the experiment (using the application). In this case, the values are higher than "after swapping" because, during the experiment, there were graphs swapped in. At the same time, these values are smaller than "before swapping" meaning that we are actually releasing primary memory. The last columns give information about the size of the graph and the amount of shared objects. Finally, the table presents the percentage of memory gain between the original scenario (before swapping) and the last scenario (after swapping and experiments).

- With DBXTalk, when considering classes as graphs, the runtime was 40% smaller than the original one which was already compacted and cleaned for production.
- In the Moose case, the memory was reduced up to 25%. The average number of objects per graph (670) is bigger than the previous example. This is because Moose handles models which are rather large. The graph of a model can include from few hundred thousands of objects to a couple of millions of objects. In this example, we have run *all* visualizations and *all* tools provided by Moose, which is not always needed by all users. Using less of them would cause less swapping in and result into more memory released. Therefore, 25% is the minimum gain to expect in the worst case scenario.
- In the case of Dr. Geo we reduced 31% of the runtime, which is significant knowing that the developers already compacted Dr. Geo as much as they could.
- Regarding Pharo's infrastructure we were able to gain 38% of memory. The actions performed on Pharo were to start it, browse some classes, create a new class and add some methods to it. This analysis shows that Pharo's environment can still be reduced.

**Classes vs. Packages as Roots.** Part of our experiments was to compare the impact of considering packages as the roots of the graph rather than classes. We report here only the experiments with DBXTalk since the results with others applications were similar. As we can see in the first two rows of Table 5.1, the gain when using classes as swap unit is much bigger than with packages. The main reason is that considering a package as root involves larger object graphs. The average number of objects per graph is 1340 with packages while it is 170 with classes. Then as soon as one of these objects is needed, the whole graph is swapped in. One conclusion from this experiment is that deciding which graphs are chosen to swap is important and directly impacts on the results. Another conclusion is that classes are a good default candidate.

**Analyzing Shared Objects.** If we only consider experiments with classes as roots, shared objects represent between 15% and 17% of each graph. This means that the compaction of GraphTable is worthy since 83% to 85% is full of nils. In the example of

DBXTalk, the compaction results in a reduction of memory footprint by 1MB which is already subtracted from the 13.8MB mentioned in Table 5.1.

**Understanding Memory Savings.** Table 5.2 describes with more details the actual objects swapped out. We distinguished between classes, methods and plain instances. The analysis shows that the results in average are approximately the same.

| | Graphs | Total objects | Plain objects | Classes | Methods |
|---|---|---|---|---|---|
| Swap out | 3565 | 596339 | 543650 | 3627 | 49062 |
| Swap in | 402 | 109330 | 100317 | 464 | 8549 |
| % saved | 88.7% | 81.6% | 81.5% | 87.2% | 82.5% |

Table 5.2: Understanding memory savings of the DBXTalk example.

The "% saved" item shows the difference between the original runtime of DBXTalk and the final one (after swapping out and having used the application). The explanation lays on the fact that a lot of classes (and their instances) are not used in this web application. This table also shows that the percentage saved is quite similar (between 80% and 90%) for all the measured items, *i.e.,* the number of graphs, the total objects, plain object, classes and methods. It makes sense since the number of methods per class is linear and the average smoothes the outliers. What this analysis shows in that the execution of this web application uses only a limited set of the available classes. The rest of the applications showed similar results.

**Conclusions.** One conclusion we got from these benchmarks is that, even if the runtimes were small, none of the applications use 100% of them. Furthermore, different applications need different parts (classes and libraries) of the environment.

Our experiments show that Marea significantly decreases primary memory consumption. To gain the described 25% to 40%, all we needed was a few lines of code that simply swapped out as much as possible. No analysis was required. When swapped out code was actually needed by applications, it was just swapped in allowing applications to run smoothly. Nevertheless, as we saw with the experiment of considering packages as roots, the graphs chosen to swap directly impacts on the results. Hence, with certain knowledge in the domain, the developer may be able to choose graphs which may lead to better results.

### 5.4.2 Swapping Out Data

Marea can swap any type of object graph, not necessary those related to code. Measuring the efficiency of data swapping is difficult to assess without building applications like GIMP[2] that require a lot of data. We performed an experiment showing that Marea supports such scenario by using it to swap graphs of plain objects.

[2]Gimp is an image retouching and editing tool. http://www.gimp.org/.

**Moose Example.** Depending on what is being analyzed, Moose models can be quite large *e.g.,* 2 million objects. Because of this, the final user is limited regarding the amount of models loaded at the same time. If several models are opened, Moose uses a very large amount of memory. However, at a given point in time, one analyzes only a subset of models.

Our solution is to use Marea to automatically swap out unused models and then automatically swap them in if needed by the user. In our experiments, we created three different models for different projects: Networking, Morphic and Marea itself. Each model (instance of MooseModel) was considered as root when swapping. Table 5.3 presents the results.

| Moose model | Objects | Swapping out time (ms) | Swapping in time (ms) |
|---|---|---|---|
| Morphic | 2102672 | 170202 | 5360 |
| Network | 539127 | 22751 | 1476 |
| Marea | 323138 | 12127 | 835 |

Table 5.3: Swapping different large Moose models.

**Results.** By using Marea, we were able to automatically manage different models while only leaving in primary memory those models we wanted to analyze at a particular moment in time. An average Moose model between 300.000 and 600.000 objects took between 12 and 22 seconds to swap out and approximately 1 second to swap in. Considering the size of the experimented graphs we conclude that Marea can be used by applications to automatically swap data. The swapping out of large graphs is expensive regarding execution time. On the Moose example, those 12 to 22 seconds to swap out a large model have to be compared with alternatives such as reconstructing the whole model each time or just writing it on disk and re-reading it. Our conclusion is that Marea's overhead is not significant compared to alternatives while it brings a lot of benefits to the application developers.

## 5.5 Measuring Unused Objects and Speed

Apart from benchmarking the amount of memory released, we also measure other properties like the amount of unused objects and the speed of the swapping.

### 5.5.1 Measuring Unused Objects

In Section 2.3.3 we gathered statistics about the memory consumption and objects usage [Martinez Peck 2010b] based on a Pharo VM that we modified to support the identification of unused objects. For each experiment, we start the analysis, we use the system for a while, we stop the analysis and finally collect the results. For each application we got the following information: the percentage of used and unused objects and the percentage of memory that used and unused objects represent. We copy again the results here to ease the comprehension:

| App. | % Used objects | % Unused objects | % Used objects memory | % Unused objects memory |
|---|---|---|---|---|
| DBXTalk | 19% | 81% | 29% | 71% |
| Moose | 18% | 82% | 27% | 73% |
| DrGeo | 23% | 77% | 46% | 54% |
| Pharo | 10% | 90% | 37% | 63% |

Table 5.4: Measuring used and unused objects.

As shown in Section 5.4.1, the graph choice directly impacts on the results. This raises the question of the best results possible if one swaps out the right graphs. The right graphs are those with as much *unused objects* as possible.

**Conclusion.** Using Marea we were able to decrease the amount of memory used. However, the gained memory does not match the expected percentage of unused object previously analyzed. For example, in DBXTalk, we could save 40% of the memory even though we previously measured in our experiments that the 71% was unused. One reason is the granularity of the swapping unit. Depending on the chosen graphs, there may be several unused objects that are swapped in. Another reason is the fact that we did a "blind swapping", *i.e.,* we swapped as much as we could taking classes as roots.

The conclusion from this experiment is that there is still a room of improvement and that with certain knowledge in the domain the memory released can be even bigger.

**Improving the results even more.** Apart from using our object usage tracer VM to analyze whether a virtual memory like Marea is worth or not, we can also use it to improve its performance. Maybe choosing classes as roots is not the best option in all scenarios. Maybe there is a class that has a lot of unused instances but only a few that are used. If we swap out the class together with all its instances, the whole graph is swapped in as soon as one instance is used.

The challenge is how can we help the developer to choose good candidates? Our tool allows the user to execute his application and then query the system to know which objects have been used or unused. Object that have been unused during the execution of the application are good candidates to swap out. In addition, we provide interesting visualizations that help the developer analyze which parts (packages, classes and methods) of the system are being used and which ones are not. These visualizations are presented in Chapter 9.

The result is that these tools can help Marea's user choose good candidates to swap out and therefore increase the performance.

### 5.5.2 Speed Analysis

So far, we have only benchmarked memory consumption. In this section, we also measure the speed to swap out and swap in objects.

| Objects per graph | Swapping out time (ms) | Swapping out time (ms) / object | Swapping in time (ms) | Swapping in time (ms) / object |
|---|---|---|---|---|
| 51 | 40 | 0.7 | 37 | 0.7 |
| 236 | 47 | 0.2 | 44 | 0.2 |
| 777 | 39 | 0.05 | 41 | 0.05 |
| 5758 | 130 | 0.02 | 50 | 0.008 |
| 21753 | 256 | 0.01 | 122 | 0.005 |

Table 5.5: Measuring swapping time.

In Section 5.4.1, we saw that when considering classes as roots, the average number of objects per graph was between 170 and 670. Based on measurements provided by Table 5.5, we can conclude that swapping out an average class takes approximately 40 milliseconds, which is negligible most of the cases. Another characteristic is that the graph size does not significantly impact on the swapping time. Section 8.3.1 explains that the object replacement used to replace objects with proxies is slow in Pharo. It takes, in average, 60% of the swapping time and it does not change much with the size of the graph. We can observe this with the columns that show the swapping time per object.

This benchmark also demonstrates that the swapping in is faster than the swapping out. This is mostly because of the serializer's performance whose materialization (deserialization) is faster than the serialization.

From the user of the application, the swapping in is unnoticeable with the graphs we used. Therefore, the experiences show that we can swap out advantageously classes and reload them on use without significantly performance penalties for the user.

## 5.6 Conclusion

In this chapter we presented different experiments and benchmarks while using Marea on real applications. We measured the amount of memory released and also the speed. We showed that the memory footprint of different applications can be reduced from 25% to 40%, with Marea virtual memory, demonstrating its usefulness.

We also stated that there is still a room of improvement since the unused objects memory for the tested applications was from 63% to 73%. With certain knowledge in the domain and hence by choosing better candidates to swap, the released memory can be even bigger.

After having introduced Marea and presented an overall validation, we continue with explanations of Marea's subsystems. The next chapter introduces Fuel, our fast general-purpose framework to serialize and materialize object graphs using a pickle format which clusters similar objects.

CHAPTER 6

# Fuel: A Fast and Universal Serializer

## Contents



## At a Glance


This chapter introduces Fuel, our general-purpose object serializer based on these principles: (1) speed, through a compact binary format and a pickling algorithm which invests time in serialization for obtaining the best performance on materialization; (2) good object-oriented design, without special help from the virtual machine; (3) serialize any object (closures, contexts, classes, traits, methods, as well as regular objects), thus have a full-featured language-specific format.



**Keywords:** Serialization, marshalling, object graphs, pickle format.

## 6.1 Introduction

In object-oriented programming, since objects point to other objects, the runtime memory is an object graph. This graph of objects lives while the system is running and dies when the system is shutdown. However, sometimes it is necessary, as it happens in Marea, to backup a graph of objects into a non volatile memory so that it can be loaded back when necessary [Martinez Peck 2010a] [Kaehler 1986, Bond 2008] or to export it so that the objects can be loaded in a different system. The same happens when doing migrations or when communicating with different systems. Besides this, databases normally need to serialize objects to write them to disk [Butterworth 1991].

There are a lot of other possible uses for a serializer. For example, in case of remote objects, *e.g.,* remote method invocation and distributed systems [Bennett 1987, Decouchant 1986, Wiebe 1986], objects need to be serialized and passed around the network. A Version Control System that deals with code represented as first-class objects needs to serialize and materialize those objects: Parcels [Miranda 2005] is a typical example. Today's web applications need to store state in the HTTP sessions and move information between the client and the server.

Approaches and tools to export object graphs must scale to large object graphs as well as be efficient. However, most of the existing solutions do not solve this last issue properly. This is usually because there is a trade-off between speed and other quality attributes such as readability/independence from the encoding. For example, exporting to XML [1] or JSON [2] is more readable than exporting to a binary format since it can opened it and edited with any text editor. However, a good binary format is faster than a text based serializer when reading and writing. Some serializers like *pickle* [3] in Python or *Google Protocol Buffers* [4] let the user choose between text and binary representation.

To avoid confusion, we define terms used in this chapter. *Serializing* is the process of converting the whole object graph into a sequence of bytes. We consider the words *pickling* and *marshalling* as synonyms. *Materializing* is the inverse process of serializing, *i.e.,* regenerate the object graph from a sequence of byes. We consider the words *deserialize*, *unmarshalling* and *unpickling* as synonyms. We understand the same for *object serialization*, *object graph serialization* and *object subgraph serialization*. An object can be seen as a subgraph because of its pointers to other objects. At the same time, everything is a subgraph if we consider the whole memory as a large graph.

### 6.1.1 Motivation and Desired Properties

From our point of view, the following five main points shape the space of serializers for class-based object-oriented programming languages. We think all of them are important, yet not necessary in all scenarios. For example, one user can consider that speed is a

[1] SIXX - Smalltalk Instance eXchange in XML- http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.

[2] JSON -JavaScript Object Notation- http://www.json.org.

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

[4] Google Protocol Buffers: http://code.google.com/apis/protocolbuffers/docs/overview.html.

critical requirement while portability not. Another user can require portability but not an outstanding performance.

1. Serializer *speed* is an important aspect since it enables more extreme scenarios such as saving objects to disk and loading them only on demand at the exact moment of their execution [Kaehler 1986, Bond 2008].

2. Serializer *portability and customization*. Since many approaches are often too slow, Breg and Polychronopoulos advocate that object serialization should be done at the virtual machine level [Breg 2001]. However, this implies non portability of the virtual machine and difficult maintenance. In addition, moving behavior to the VM level usually means that the serializer is not easy to customize or extend.

3. Support for *class evolution*. For example, the class of a saved object can be changed after the object is saved. At writing time, the serializer should store all the necessary information related to class shape to deal with these changes. At loading time, objects must be updated in case it is required. Many object serializers are limited regarding this aspect. For example, the Java Serializer [5] does not support the modification of an object's hierarchy nor the removing of the implementation of the Serializable interface.

4. *Storing and loading policies*. Ungar [Ungar 1995] claims that the most important and complicated problem is not to detect the subgraph to export, but to detect the implicit information of the subgraph that is necessary to correctly load back the exported subgraph in another system. Examples of such information are (1) whether to export an actual value or a counterfactual initial value or (2) whether to create a new object in the new system or to refer to an existing one. In addition, it may be necessary that certain objects run some specific code once they are loaded in a new system.

5. *Completeness*. Serializers are often limited to certain kind of objects they save. For example, most of the Smalltalk serializers do not support serialization of objects like BlockClosure, MethodContext, CompiledMethod, etc. Now, in dynamic programming languages *e.g.,* Smalltalk, methods and classes are first class objects, *i.e.,* user's code is represented by objects. Similarly, the execution stack and closures are objects. The natural question is if we can use serializers as a code management system underlying mechanism. VisualWorks Smalltalk introduced a pickle format to save and load code called Parcels [Miranda 2005]. However, such infrastructure is more suitable for managing code than a general-purpose object graph serializer.

### 6.1.2 Overview of Fuel

In Marea, we are interested in all these properties, but most notably in the serializer speed and the completeness. We did not find any reliable solution and therefore we have developed our own serializer. This chapter presents Fuel, our fast open-source general-purpose

[5] Java Serializer API: http://java.sun.com/developer/technicalArticles/Programming/serialization/.

framework to serialize and deserialize object graphs using a pickle format which clusters similar objects.

Traditional serializers encode the objects of the graph while traversing it. The stream is a sequence of bytes where they store each object plus an identifier of its type. The deserialization then starts to read objects from the stream. For each object it reads, it needs to read its type as well as determine and interpret how to materialize that encoded object. In other words, the materialization is done *recursively*.

In the contrary, in Fuel there is a first traversal of the graph (we call this phase "analysis") where each object is associated with an specific type which is called "cluster" in Fuel. Fuel first writes only the objects (without their references to other objects) and then the references from each object. During materialization, Fuel first materializes the instances. Since all the objects of a cluster have the same type, Fuel reads that information in the stream only once. The materialization can be done in a bulk way which means that we can just iterate and instantiate the objects. Finally, Fuel iterates and sets the references for each of the materialized object. Fuel's materialization is done *iteratively*.

We show in detailed benchmarks that we have the best performance in most of the scenarios: For example, with a large binary tree as sample, Fuel is four times faster writing and seven times faster loading than its competitor SmartRefStream. If a slow stream is used *e.g.,* a file stream, Fuel is sixteen times faster for writing thanks to its internal buffering. We have implemented and validated this approach in Pharo [Black 2009] and Fuel has already been ported to other Smalltalk implementations. Fuel is also used to serialize classes in Newspeak. The pickle format presented in this chapter is similar in spirit to the one of Parcels [Miranda 2005]. However, Fuel is not focused on code loading and is highly customizable to cope with different objects.

In addition, this chapter demonstrates the speed improvements made in comparison to traditional approaches. We demonstrate that we can build a fast serializer without specific VM support, with a clean object-oriented design and providing the most possible required features for a serializer.

### Structure of the Chapter

We start by exposing some elements to evaluate a serializer in Section 6.2. In Section 6.3, we present our solution and an example of a simple serialization which illustrates the pickling format. We apply the evaluation criteria to Fuel in Section 6.4. Section 6.5 gives an overview of Fuel's design. A large amount of benchmarks are provided in Section 6.6. We present Fuel real life usages in Section 6.7. Finally, we discuss related work in Section 6.8 just before concluding in Section 6.9.

## 6.2 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.

### 6.2.1 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 [6], Google Protocol Buffers, Oracle Coherence*Web [7], Hessian [8], have low-level language-independent formats oriented to exchange structured data between many languages. In contrast, SmartRefStream in Pharo and Pickle 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

[6] Action Message Format - AMF 3: http://download.macromedia.com/pub/labs/amf/amf3_spec_121207.pdf.

[7] Oracle Coherence: http://coherence.oracle.com.

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

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.

**Customizability.** 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.

### 6.2.2 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 environment 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 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 6.1. 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.

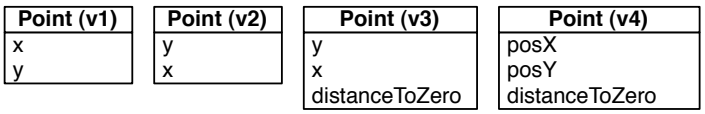


Figure 6.1: Several kinds of class shape changing.

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 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 without materializing 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.

## 6.3 Fuel's Foundation

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

### 6.3.1 Pickle Format

Riggs defines a pickle format as:

> *"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 ensure 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."* *(* [Riggs 1996]*)*

Fuel's 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 the type information from the stream only once. The materialization can be done in a bulk way (it can just iterate and instantiate the objects). Finally, Fuel iterates over the materialized objects and sets their instance variables with the appropriate references.

Even if the main goal of Fuel is materialization speed, the benchmarks of Section 6.6 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 aFileStream |
anOrigin := Point x: 10 y: 20.
aCorner := Point x: 30 y: 40.
aRectangle := Rectangle origin: anOrigin corner: aCorner.
aFileStream := FileStream newFileNamed: 'example'.
(FLSerializer on: aFileStream) serialize: aRectangle.
```

Figure 6.2 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.

### 6.3.2 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. That means that when serializing the Transcript what we actually serialize it is just the global name and then during materialization we get the global with such name from the system (for more details see Section 6.3.3). Another example is that 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 cluster knows how to encode and decode the objects it groups. Therefore, Fuel delegates to them for the encoding and decoding of objects. To know how to associate a particular object to a cluster, Fuel has an *analysis phase* as explained in Section 6.3.3. Here are some examples of clusters: PositiveSmallIntegerCluster groups positive instances of SmallInteger; NegativeSmallIntegerCluster groups

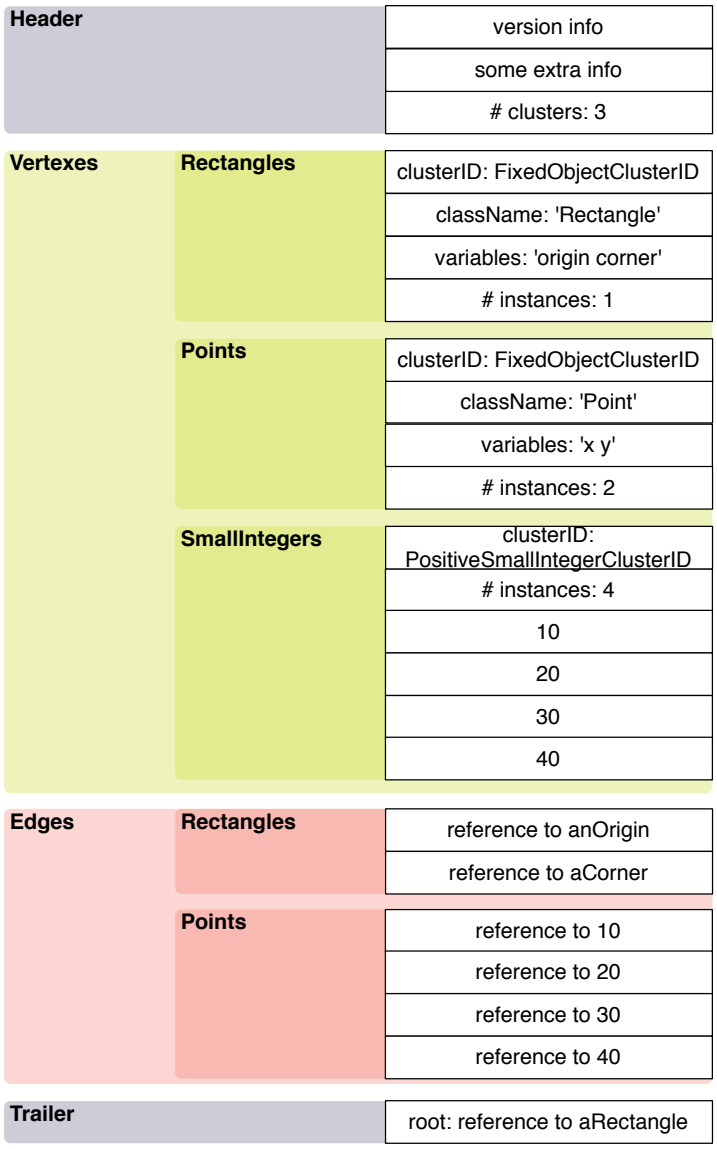


Figure 6.2: A graph example encoded with the pickle format.

negative instances of SmallInteger; FloatCluster groups Float instances. FixedObjectCluster is the cluster for instances of 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 6.2, 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, PositiveSmallInte-

gerCluster or NegativeSmallIntegerCluster do not store such information because it is implicit in the cluster implementation.

In Figure 6.2, 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 specific clusters.

### 6.3.3 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 traverse 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.

### 6.3.4 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.

### 6.3.5 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 one of the most important characteristics of Fuel's pickle format and the main reason why materializing is much faster in Fuel than in other approaches. 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.

### 6.3.6 Breadth-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 do a breadth-first traversal since there are two phases for writing instances and references as previously explained. 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%.

## 6.4 Fuel's Features

In this section, we analyze Fuel in accordance with the concerns and features defined in Section 6.2.

### 6.4.1 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 6.6.

**Completeness.** To our knowledge, and considering all our tests and benchmarks, 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. Fuel provides hooks to solve the mentioned problems. For example, there is a hook so that a message is send 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 right now is the ability to materialize in a dialect or language a stream which was serialized in another language. It is not our immediate plan to communicate with another language. Nevertheless, we believe that with certain amount of work, Fuel's design *e.g.,* the fact of having reified the clusters, may allow us to adapt the serializer to output XML files instead than our binary pickle format.

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.

**Customizability.** 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 alternative to do it: 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.

### 6.4.2 Addressing Challenges with Fuel

In this section, we explain how Fuel implements some of the features previously commented. The mentioned challenges *partial loading* and *custom reading* are not included here because Fuel does not support them at the moment of this writing.

**Cyclic object graphs and duplicates.** Fuel uses an identity set to store each visited object of the graph. When visiting each object, Fuel checks if that object was already visited *i.e.,* if it is already present in the collection. Therefore, it only processes objects only once, allowing us to support 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* [9], 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 a hook method the user can specify 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 at the very beginning with a fixed string prefix *e.g.,*'FUEL', and then we write the version number of the serializer *e.g.,*'18'. Then, when materializing, the signature and the version has to match with the current materializer. Otherwise, we signal an appropriate error. At the moment, we do not support backward compatibility.

[9] Fuel allows the user to specify that a certain class must be considered as internal *i.e.,* that it has to be serialized in full detail.

### 6.4.3 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 claim 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 on 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 6.7, 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. This is deeply explained in Section 6.5.2.

- It does not need any special support from the VM.

- It is covered by tests and benchmarks.

## 6.5 Fuel Design and Packages

Fuel is open-source and developed under the MIT license[10]. The website of the project with its documentation is in: http://rmod.lille.inria.fr/web/pier/software/Fuel. The source code is available in the SqueakSource3 server: http://ss3.gemstone.com/ss/Fuel.html.

### 6.5.1 Fuel Design

Figure 6.3 shows a simplified UML class diagram of Fuel's design.

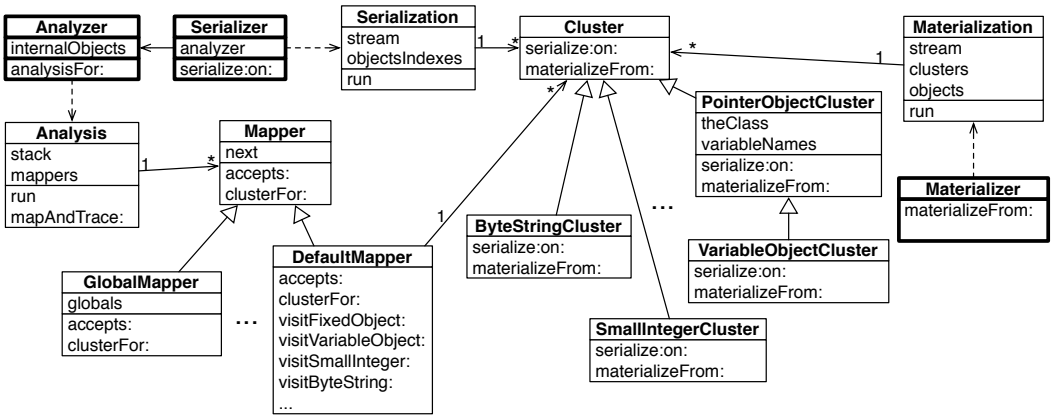


Figure 6.3: Fuel's design.

- 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: By implementing the *Visitor* design pattern, the classes in this hierarchy make possible the complete customization of how the graph is traversed (visited) and serialized. At the same time, they implement the *Chain of Responsibility* design pattern for determining what cluster corresponds to an object [Alpert 1998]. An Analyzer creates a chain of mappers each time we serialize.

- The hierarchy of Cluster has 35 subclasses, where 10 of them are optional optimizations.

[10] http://www.opensource.org/licenses/mit-license.php

**Code statistics.** 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.

### 6.5.2 Fuel Packages and Libraries

Contrary to other serializers, Fuel is not focused only in one usage like code loading. Fuel is a general-purpose serializer and it is highly customizable to cope with different objects and scenarios. Fuel is the infrastructure on top of which we can then build other tools. One important aspect to achieve that is to have good modularity and that is why we have several packages in Fuel.

Figure 6.4 shows some of the available Fuel packages. The packages Fuel, FuelMetalevel and FuelPackageLoader are in grey because they are the most important ones. The arrows represent dependencies between packages *e.g.,* FuelBenchmarks needs Fuel.

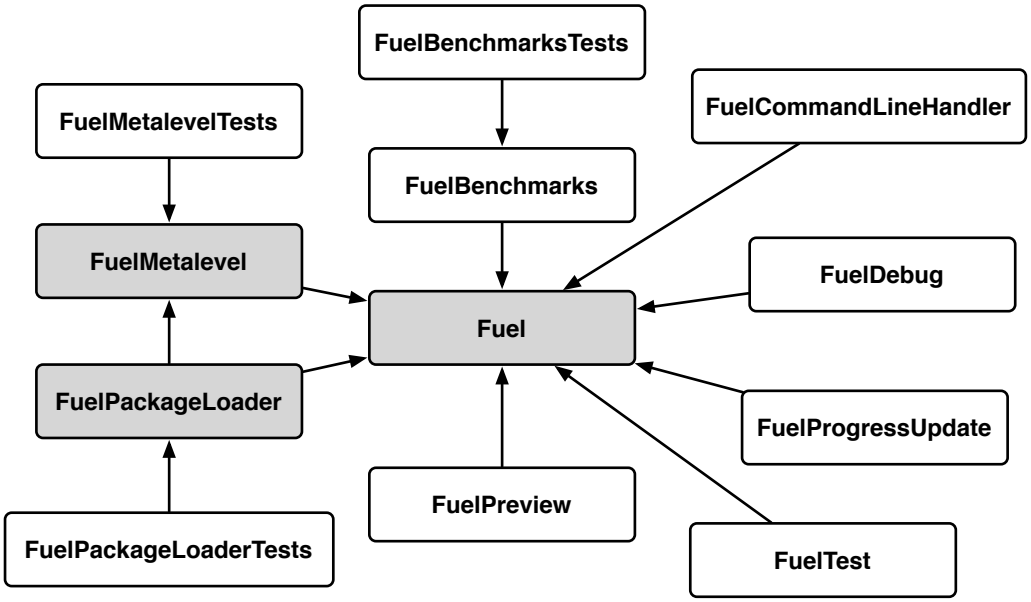


Figure 6.4: Fuel's available packages.

**Fuel.** This is the *core* of Fuel. It works as a general-purpose serializer and all it does is serialize and materialize. If the graph the user wants to serialize contains classes, they will all be considered as "global". That means that during serialization we just store its global name and during materialization we read the global name and we search it in the system. Therefore, classes have to be present in the environment we are materializing.

**FuelMetalevel.** This package adds the knowledge of how to correctly serialize and materialize classes, metaclasses, traits, method dictionaries, compiled methods and closures, *i.e.,* all the entities related to code and runtime infrastructure. It only knows how to serialize and correctly materialize. However, FuelMetalevel *only* do that: serialize and materialize. Nothing else. It does not initialize classes, it does not notify the system about the materialized classes, etc.

**FuelPackageLoader.** Right now the way to export and import packages with Pharo is by exporting the source code and the compile it during the import. FuelPackageLoader is a prototype to see whether Fuel is able to export and import packages of code in a binary way and avoid having to compile from sources during the import. This packages uses FuelMetalevel and its responsibility is to take into account the concept of package of code and the correct integration of them into the system. For example, it initializes classes, sends notifications, etc.

**FuelBenchmarks.** Fuel contains a large suite of benchmarks that analyzes the speed of the serialization and materialization and the resulted size of the stream. It can be used to benchmark itself by comparing the results with previous versions or after certain change. For important changes we always run these benchmarks to see if we have not significantly decreased in performance. In addition, we have created adaptors so that we can benchmark Fuel agains other serializers as Section 6.6 shows.

**FuelCommandLineHandler.** This is a tool to be able to materialize Fuel files directly from the command line. This is used to build Pharo images from kernel images, as described in Section 6.7

**FuelDebug.** FuelDebug helps us debugging Fuel in both cases, serialization and materialization. The output of the tool let us know, for example, which objects were serialized or which are the paths to a certain object. This is useful to detect bugs in Fuel but it can also help the user discover unwanted objects that are being serialized because they are reachable from the input graph.

**FuelPreview.** It is a package to visualize the objects graphs being serialized. This is very useful to understand the transitive closure it is being serialized. FuelPreview can also visualize the results obtained from FuelDebug. Figure 6.4 shows two simple examples in which objects are represented as circles and references as arrows. On the left, there is a normal graph that we want to serialize and the red circle is the root. On the right, we take the same graph but we only display the paths to those objects (with yellow color) which are integers less than 2. The API provides a way to visualize paths to objects that fulfill certain condition specified by the user. Finally, notice that this package depends on Roassal[11] a visualization engine.

[11] http://www.objectprofile.com/pier/Products/Roassal

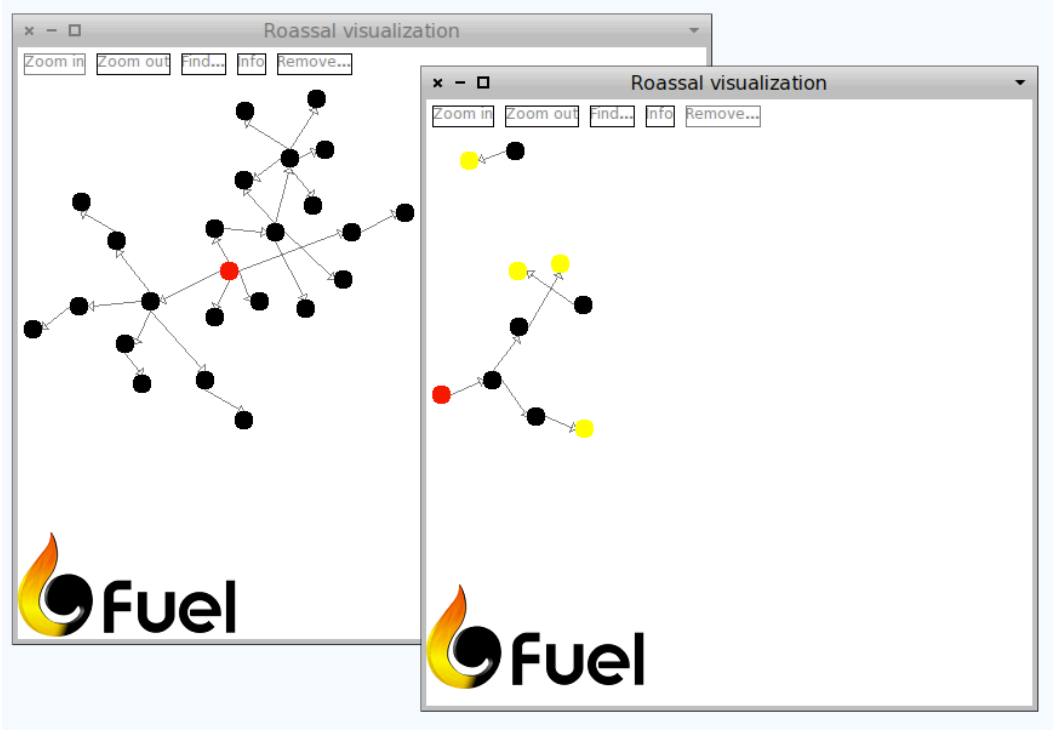


Figure 6.5: A visualization example of a graph.

## 6.6 Benchmarks

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 (comma separated values) files which ease the immediate build of charts.

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 6.8.

### 6.6.1 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 because different users may consider one more important than the other one. For example, if we are building a control version system, materialization time is more important since a version is loaded much more frequently than produced.

**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.

### 6.6.2 Benchmarks serializing primitive and large objects

**Primitive objects serialization.** Figure 6.6 shows the results of primitive objects serialization and materialization using memory-based streams. The horizontal axis corresponds to the duration (time). Hence, the shorter the bar is, the better the serializer/materializer is.

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.

The conclusions are:

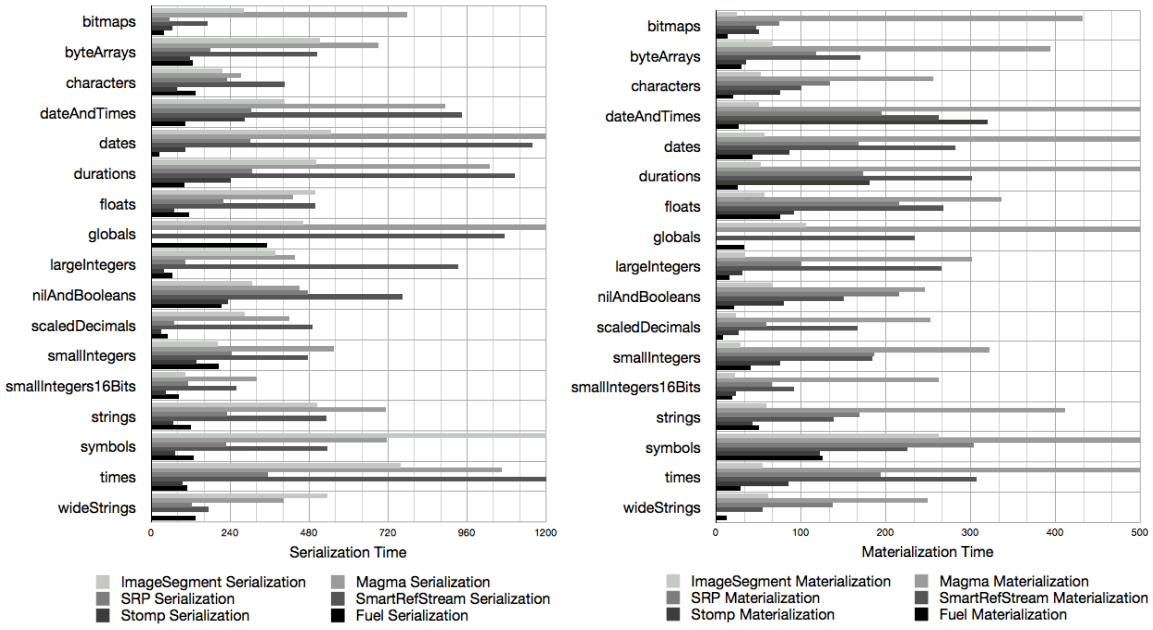


Figure 6.6: Time (in ms) for primitive objects serialization and materialization (the smaller the better).

- 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 implemented in the virtual machine.
- ImageSegment is really slow with Symbol instances. We explain the reason later in Section 6.6.3.
- 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 1.3 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 6.7 shows the results of large objects serialization and materialization. The conclusions are:

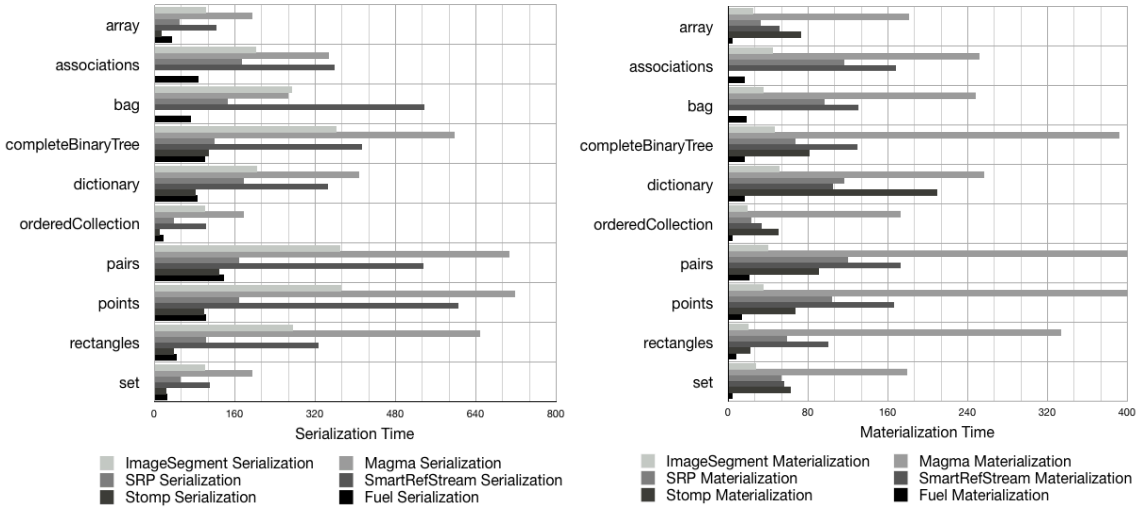


Figure 6.7: Time (in ms) for large objects serialization and materialization (the smaller the better).

- 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 dif-

ferent 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 build 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.

### 6.6.3 ImageSegment results explained

ImageSegment seems to be really fast in certain scenarios but too slow in others. However, it deserves some explanations of how ImageSegment works [Martinez Peck 2010a]. 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 6.6 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 6.8 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

- 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.

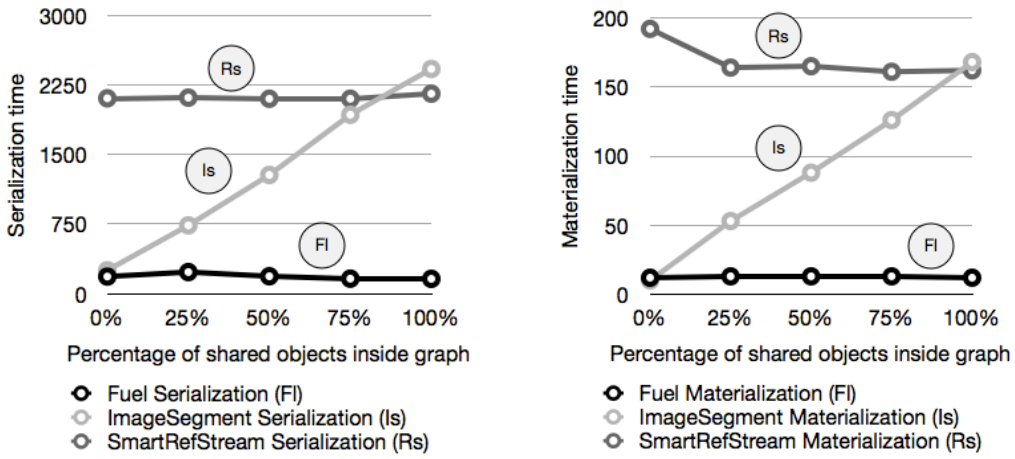


Figure 6.8: ImageSegment serialization and materialization in presence of shared objects.

- For serialization, when all objects are shared, 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.

### 6.6.4 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 6.9 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.** 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 for serialization is similar than StOMP. Fuel performs then well with small graphs as well as large ones.

### 6.6.5 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

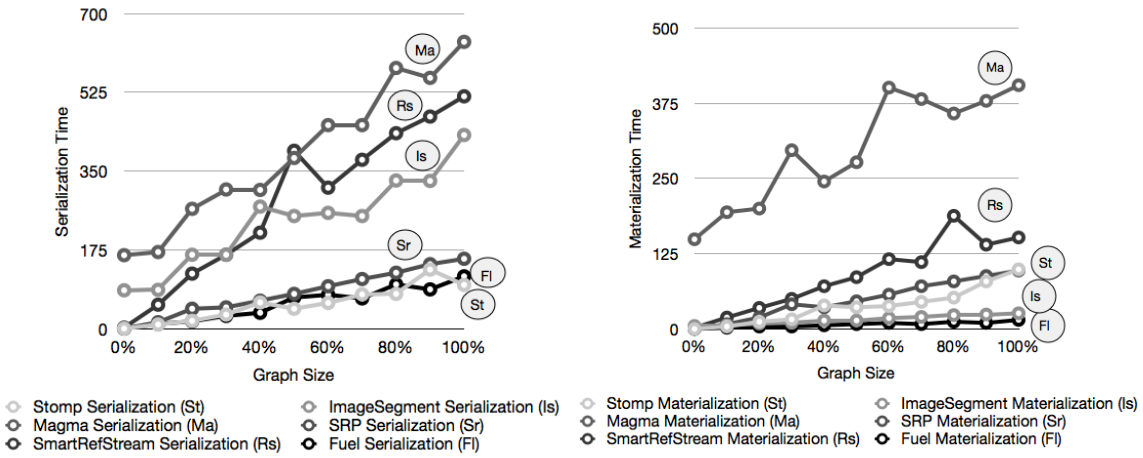


Figure 6.9: Serialization and materialization of different graph sizes.

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 on the language side but not in VM inside itself, VM primitives or plugins.

Cog is around 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 on 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 on 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 6.10 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. Note, however, that ImageSegment relies on SmartRefStream (implemented on the language side) to serialize the shared objects. In this experiment we used graphs with none or a few shared objects. Otherwise, the difference in ImageSegment might be bigger between uses of the two virtual machines.

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

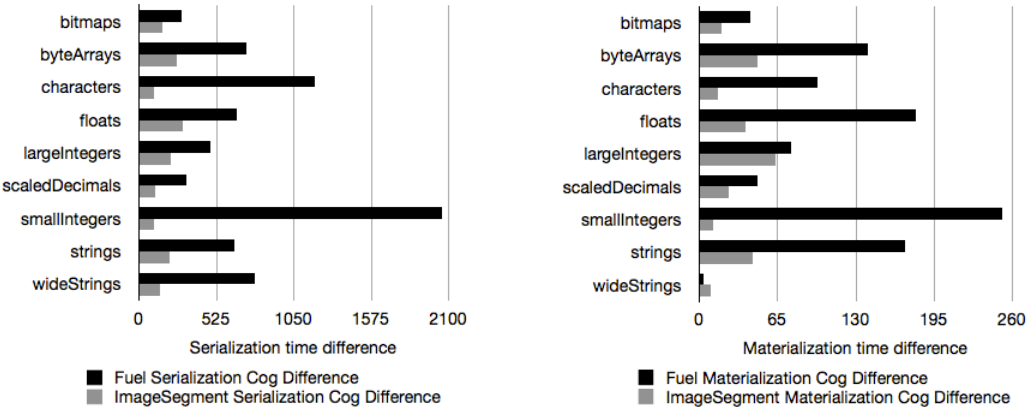


Figure 6.10: Serialization and materialization differences when using CogVM

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 chapter 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.

## 6.7 Real Cases Using Fuel

Apart from Marea, Fuel is used in different 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 [Nierstrasz 2005]. It uses large data models that can be exported and imported from files. The models produces 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[12] which has been integrated into Moose Suite 4.4.

[12] http://www.moosetechnology.org/tools/fuel

**Pier CMS persistency.** Pier is a content management system that is light, flexible and free [Renggli 2007]. It is implemented on top of the Seaside web framework [Ducasse 2010]. In Pier all the documents and applications are represented as objects. A simple persistency strategy has been implemented using Fuel[13]. This tool persists Pier CMS kernels (large object graphs), *i.e.,* Pier CMS websites. This way the user can backup its system and load it back later on if desired.

**SandstoneDB with Fuel backend.** SandstoneDB[14] 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.

**SimplePersistency with Fuel backend.** One can use Fuel as the serializer/materializer for the Simple Image Based Persistence scheme[15]. The implementation is in this repository http://www.squeaksource.com/SimplePersistence.html.

**Fuel in BioSmalltalk.** BioSmalltalk uses Fuel to serialize big DNA or proteins[16]. Moreover, BioSmalltalk has a special requirement on Fuel: to customize the serialization strategy to save primary memory which is important when dealing with these kind of objects. This means changing the serializer on-the-fly when a particular object is found in a graph of objects. Specifically, if a DNA or protein sequence with a particular threshold is found, we zip it. Then, during materialization, BioSmalltalk uncompress the needed strings.

**PharoKernel generation.** PharoKernel is a Pharo distribution that contains only the real kernel packages[17]. The image is only about 2.5 MB and works correctly. Pharo uses FuelPackageLoader to export *all* Pharo packages (all but kernel ones) and then to import them in a PharoKernel, giving us back the original Pharo image but generated from the kernel.

**Newspeak port.** Newspeak[18] 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 [Bracha 2010b] 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.

[13] http://ss3.gemstone.com/ss/pierfuel.html
[14] http://onsmalltalk.com/sandstonedb-simple-activerecord-style-persistence-in-squeak
[15] http://onsmalltalk.com/simple-image-based-persistence-in-squeak/
[16] http://biosmalltalk.blogspot.fr/2012/07/custom-serialization-of-big-dna-or.html
[17] https://ci.lille.inria.fr/pharo/view/Pharo-Kernel%202.0/
[18] http://newspeaklanguage.org/

## 6.8 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 [Riggs 1996]. 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 or JSON. 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 [Miranda 2005] 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 [19] (Smalltalk Objects on MessagePack[20]) and the mature SRP (State Replication Protocol) [21] 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 or Oracle Coherence*Web HTTP session management.

[19] StOMP - Smalltalk Objects on MessagePack: http://www.squeaksource.com/STOMP.html.
[20] http://msgpack.org
[21] State Replication Protocol Framework: http://sourceforge.net/projects/srp/.

## 6.9 Conclusion

In this chapter, 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.

To continue explaining Marea's subsystems, next chapters presents Ghost, our uniform and lightweight general-purpose proxy toolbox. Ghost provides low memory consuming proxies for regular objects as well as for objects that play an important role in the runtime infrastructure such as classes and methods.

CHAPTER 7

# Ghost: Uniform and Flexible Proxies

## Contents



## At a Glance


In this chapter, we present Ghost: a uniform and lightweight general-purpose proxy implementation for the Pharo programming language. Ghost provides low memory consuming proxies for regular objects as well as for classes and methods.



**Keywords:** Message passing control, proxy, interception.

## 7.1 Introduction

A proxy object is a surrogate or placeholder that controls access to another target object. A large number of scenarios and applications have embraced and used the Proxy Design Pattern [Gamma 1993, Eugster 2006]. Proxy objects are a widely used solution for different scenarios such as remote method invocation [Shapiro 1986, Santos 2002], distributed systems [Bennett 1987, McCullough 1987], future objects [Pratikakis 2004], behavioral reflection [Ducasse 1999, Kiczales 1991, Welch 1999], aspect-oriented programming [Kiczales 1997], wrappers [Brant 1998], object databases [Lipton 1999], inter-languages communications and bindings, access control and read-only execution [Arnaud 2010], lazy or parallel evaluation, middlewares like CORBA [Wang 2001, Koster 2000, Hassoun 2005], encapsulators [Pascoe 1986], security [Van Cutsem 2010], memory management and object swapping [Martinez Peck 2011b, Martinez Peck 2011c], among others.

Most proxy implementations support proxies for instances of common classes only. Some of them, *e.g.,* Java Dynamic Proxies [Eugster 2006], even need that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages.

Creating uniform proxies for not only regular objects, but also for objects with an important role in the runtime infrastructure such as classes or methods has not been considered. In existing work, it is impossible for a proxy to take the place of a class and a method and still be able to intercept messages and perform operations such as logging, security, remote class interaction, etc.

Marea replaces the original (probably unused) objects with proxies. When one of the proxies intercepts a message, the original objects are brought back into primary memory. The original objects can be instances of common classes but they can also be methods, classes, method context themselves, etc. Therefore, the proxy implementation for Marea must deal with all kind of objects including classes and methods.

Another property of proxy implementations is memory footprint. As any other object, proxies occupy memory. In Marea's scenario, the number of proxies and their memory footprint is crucial because, at some point, the amount of released memory depends on that.

Traditional implementations in dynamic languages such as Smalltalk are based on error handling [Pascoe 1986]. This results in non stratified proxies meaning that not all messages can be trapped leading to severe limits. In Marea, not being able to intercept messages is a problem because those messages will be directly executed by the proxy instead of being intercepted. This can lead to different execution paths in the code, errors or even a VM crash.

Traditionally, proxies not only intercept messages, but they also decide what to do with the interceptions. We argue that these are two different responsibilities that should be separated. Proxies should only intercept, which is a generic operation that can be reused in different contexts. Processing interceptions is application-dependent. It should be the responsibility of another object that we call *handler*.

In this chapter, we present Ghost: a uniform and lightweight general-purpose proxy implementation for Pharo programming language [Black 2009]. Ghost provides low memory consuming proxies for regular objects, classes and methods. It is possible to create a proxy

that takes the place of a class or a method and that intercepts messages without breaking the system. When a proxy takes the place of a class, it intercepts both the messages received by the class and the lookup of methods for messages received by instances. Similarly, when a proxy takes the place of a method, then the method execution is intercepted too. Last, Ghost supports *controlled stratification*: developers decide which message should be understood by the proxy and which should be intercepted and transmitted for processing to the handler.

#### Structure of the Chapter

Section 7.2 defines and unifies the vocabulary and roles used throughout the chapter and presents the list of criteria used to compare different proxy implementations. Section 7.3 describes the typical proxy implementation and it presents the problem by evaluating it against the previously defined criteria. An introduction to Pharo reflective model and its provided hooks is explained by Section 7.4. Section 7.5 introduces and discusses Ghost proxies and shows how the framework works. Section 7.6 explains how Ghost is able to proxify methods and classes. Certain messages and operations that need special care when using proxies is analyzed in Section 7.7. Section 7.8 provides an evaluation of Ghost based on the defined criteria. Section 7.9 evaluates whether virtual machine support for proxies is worth it or not. Real case studies of Ghost are presented in Section 7.10. Finally, in Section 7.11, related work is presented before concluding in Section 7.12.

## 7.2 Proxy Evaluation Criteria

### 7.2.1 Vocabulary and Roles

For sake of clarity, we define here the vocabulary used throughout this chapter and we highlight the roles that objects are playing when using proxies (see Figure 7.1).

*Target.* It is the original object that we want to *proxify*.

*Client.* It is an object which uses or holds a reference to a target object.

*Interceptor.* It is an object whose responsibility is to *intercept* messages that are sent to it. It may intercept some messages or all of them.

*Handler.* The handler is responsible of *handling* messages caught by the interceptor. By *handling* we refer to whatever the user of the framework wants to do with the interceptions, *e.g.,* logging, forwarding the messages to the target, control access, etc.

One implementation can use the same object for taking the roles of interceptor and handler, *i.e.,* the proxy plays both roles. In another solution, such roles can be achieved by different objects. With this approach, the proxy usually takes the role of interceptor.

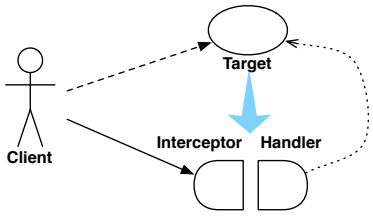


Figure 7.1: Roles in Proxy.

### 7.2.2 Proxies Implementation Criteria

From the implementation point of view, there are criteria that have to be taken into account to compare and characterize a particular solution [Ducasse 1999, Van Cutsem 2010]:

**Stratification.** Most solutions to implement proxies are based on dedicated messages such as doesNotUnderstand:. The problem with approaches that reserve a set of messages for the proxy implementation is that there is a clash between the API of the proxified object and the proxy implementation.

To address such problem, some solutions proposed stratification [Van Cutsem 2010]. Stratification means that there is a clear separation between the proxy support and application functionalities. In a fully stratified proxy, all messages received by a proxy should be intercepted and transmitted for processing to a handler. The proxy API should not pollute the application's namespace. Besides, having this stratification is important to achieve security and to fully support transparency of proxified objects for the end-programmers [Bracha 2004].

Stratification highlights two responsibilities in a proxy toolbox: (1) trapping or intercepting messages (interceptor role) and (2) managing interceptions (handler role), *i.e.,* performing actions once messages are intercepted. In a stratified proxy framework, the first responsibility is covered by a proxy itself and the second one by a handler.

**Interception levels.** There are the following possibilities:

- Intercept *all* messages, even those not defined in the object API *e.g.,* inherited from superclasses.
- Intercept all messages excluding a list of messages defined by the user.
- Intercept all messages excluding some messages imposed the proxy toolbox *e.g.,* inherited methods if we are using a solution based on error handling such as using the doesNotUnderstand: message.

With the last option, the developer has no control over messages that are not intercepted and hence performed by the proxy itself. This can be a problem because it is impossible to distinguish messages sent to the proxy from the ones that should be trapped. For example, when a proxy is asked its class, it must answer not its own class but the class of the target object. Otherwise, this can cause errors difficult to manage.

**Object replacement.** Replacement is making client objects refer to the proxy instead of the target. Two cases exist:

1. Often, the target is an existing object with other objects referencing it. The target needs to be *replaced* by a proxy, *i.e.,* all objects in the system which have a reference on the target should be updated so that they point to the proxy instead. For instance, for a virtual memory management, we need to swap out unused objects and to replace them with proxies. We refer to this functionally as *object replacement*.

2. In the other case, the proxy is just created and it does not replace another already existing object. For example, when doing a query with a database driver, it can let proxies to perform lazy loading on some parts of the graphs. As soon as a proxy receives a message, the database driver loads the rest of the graph. Another example is remote method invocation where targets are located in a different memory space from the clients' one. This means that, in the client memory space, we have proxies that can forward messages and interact with the real objects in the other memory space.

**Uniformity.** We refer to the ability of creating a proxy for any type of object (regular object, method, class, block, process, etc) and replacing the object with it. Most proxy implementations support proxies only for regular objects *i.e.,* proxies cannot replace a class, a method, a process, etc., without breaking the system. Certain particular objects like nil, true and false cannot be proxified either.

This is an important criterion since there are scenarios where being able to create proxies for any runtime entity is mandatory. As described in Section 7.10 an example is the mentioned virtual memory which replaces all type of unused objects with proxies

**Transparency.** A proxy is fully transparent if clients are unaffected whether they refer to a proxy or the target. No matter what message the client sends to the proxy, it should answer the same as if it were the target object.

One of the typical problems related to transparency is the identity issue when the proxy and the target are located in the same memory space. Given that different objects have different identities, a proxy's identity is different from the target's identity. The expression proxy == target will answer false revealing the existence of the proxy. This can be temporary hidden if there is object replacement between the target object and the proxy. When we replace all references to the target with references to the proxy, clients will only see the proxy. However, this "illusion" will be broken as soon as the target provides its own reference (self) as an answer to a message.

Another common problem is asking a proxy the class or type since, most of the times, the proxy answers its own type or class instead of the one of the target. The same happens if there is special syntax or operators in the language such as "+", "/", "=", ">", etc. To have the most transparent proxy possible, these situations should be handled in a way which allows the proxy to behave like the target.

**Efficiency.** The proxy toolbox must be efficient from the performance and memory usage points of view. In addition, we can distinguish between installation performance and runtime performance. For example, for installation, it is commonly evaluated if a proxy installation involves extra overhead like recompiling.

Depending on the usage, the memory footprint of the proxies can be fundamental. The space analysis should consider not only the size in memory of the proxies, but also how many objects are needed per target: it can be either only one proxy instance or a proxy instance and a handler instance.

**Ease of debugging.** It is difficult to test and debug in the presence of proxies because debuggers or test frameworks usually send messages to the objects present in the current stack. These messages include, for example, printing an object, accessing its instance variables, etc. When the proxy receives any of these messages it may intercept such message, making debugging more complicated.

**Implementation complexity.** This criterion measures how difficult it is to implement a solution. Given a fixed set of functionalities, a simpler implementation is better.

**Constraints.** The toolbox may require, for example, that the target implements certain interface or inherits from a specific class. In addition, it is important that the user of the proxy toolbox can easily extent or adapt it to his own needs.

**Portability.** A proxy implementation can depend on specific entry points of the virtual machine or on certain features provided by the language.

## 7.3 Common Proxy Implementation

Although there are different proxy implementations and solutions, there is one that is the most common among dynamic programming languages. This implementation is based on error raising and the resulting error handling [Ducasse 1999, Black 2009]. We briefly describe it and show that it fails to fulfill important requirements.

### 7.3.1 Typical Proxy Implementation

In dynamic languages, the type of the object receiving a message is resolved at runtime. When an unknown message is sent to an object, an error exception is thrown. The basic idea is to create objects that raise errors for all the possible messages (or a subset) and customize the error handling process.

In Pharo, for instance, the virtual machine sends the message doesNotUnderstand: to the object that receives a message that does not match any method. To avoid infinite recursion, all objects must understand the message doesNotUnderstand:. That is the reason why such method is implemented in the class Object, the root of the hierarchy chain. The default

implementation throws a MessageNotUnderstood exception. Similar mechanisms exist in dynamic languages like Ruby, Python, Objective-C and Perl.

Since doesNotUnderstand: is a normal method, it can be overwritten in subclasses. Hence, if we can have a minimal object and we override the doesNotUnderstand: method to do something special (like forwarding messages to a target object), then we have a possible proxy implementation. This technique has been used for a long time [McCullough 1987, Pascoe 1986] and it is the most common proxy implementation. Most dynamic languages provide a mechanism for handling messages that are not understood as shown in Section 7.11.

**Obtaining a minimal object.** A minimal object is one which understands none or only a few methods [Ducasse 1999]. In some programming languages, the root class of the hierarchy chain (usually called Object) already contains several methods. In Pharo, Object inherits from a superclass called ProtoObject which inherits from nil. ProtoObject understands a few messages[1]: the minimal messages that are needed by the system. Here is a simple Proxy implementation.

```
ProtoObject subclass: #Proxy
    instanceVariableNames: 'targetObject'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Proxies'
```

```
Proxy >> doesNotUnderstand: aMessage
    |result|
    ...    "Some application specific code"
    result := aMessage sendTo: targetObject.
    ...    "Other application specific code"
     ^result
```

Figure 7.2: Naive proxy implementation based in minimal object and handling not understood methods in Pharo.

**Handling not understood methods.** Common behaviors of proxies include logging before and after the method, forwarding the message to a target object, validating some access control, etc. If needed, it is valid to issue a super send to access the default doesNotUnderstand: behavior.

To be able to forward a message to an object, the virtual machine usually reifies the message. In Pharo, the argument of the doesNotUnderstand: message is an instance of the class Message. It specifies the method selector, the list of arguments and the lookup class (in normal messages it is the class of the receiver and, for super sends, it is the superclass of the class where the method issuing the super send is implemented). To forward a message to another object, the class Message provides the method sendTo: anotherObject.

This solution is independent of Pharo. For example, the Pharo's doesNotUnderstand: and sendTo: are in Ruby method_missing and send, in Python __getattr__ and getattr, in Perl autoload, in Objective-C forwardInvocation:. In Section 7.11, we explain some of these examples with more detail.

[1] ProtoObject has 25 methods in PharoCore 1.4.

### 7.3.2 Evaluation

We now evaluate the common proxy implementation based on the criteria we described in Section 7.2.2.

**Stratification.** This solution is unstratified:

- The method doesNotUnderstand: cannot be trapped like a regular message. Moreover, when such message is sent to a proxy, there is no efficient way to know whether it was because of the regular error handling procedure or because of a proxy trap that needs to be handled. In other words, the doesNotUnderstand: occupies the same namespace as application-level methods [Van Cutsem 2010].
- There is no separation between proxies and handlers.

**Interception levels.** It cannot intercept all messages but *only* those that are not understood. As explained, this generates method name collisions.

**Object replacement.** It is usually unsupported by most programming languages. Nevertheless, Smalltalk implementations do support it using pointer swapping operations such as the become: primitive. However, with such solution, target references may leak when the target remains in the same memory: the target might provide its own reference as a result of a message. This way the client gets a reference to the target so it can by-pass the proxy.

**Uniformity.** There is a severe limit to this implementation since it is not uniform: proxies can only be applied to regular objects. Classes, methods and other core objects cannot be proxified.

**Transparency.** This solution is not transparent. Proxies do understand some methods (those from its superclass) generating method name collisions. For instance, if we evaluate Proxy new pointersTo[2] it answers the references to the proxy instead of intercepting the message and forwarding it to a target. The same happens with the identity comparison or when asking the class.

In Pharo, it is possible not only to subclass from ProtoObject but also from nil in which case the subclass do not inherit any method. This solves some of the problems, such as the one of method name collisions, but the solution is still not stratified and makes debugging more complicated.

**Efficiency.** From the speed point of view, this solution is reasonably fast (it is based on two lookups: one for the original message and one for the doesNotUnderstand: message) and it has low overhead. In contrast to other technologies, there is no need to recompile the application and the system libraries or to modify their bytecode or to do other changes

[2] pointersTo is a method implemented in ProtoObject.

such as in Java where it is necessary to modify the environment variable CLASSPATH or the class loader. Regarding the memory usage, there is no optimization.

**Ease of debugging.** The debugger sends messages to the proxy which are not understood and, therefore, intercepted. To be able to debug in presence of proxies, one has to implement all these methods directly in the proxy. The drawback is that the action of enabling or disabling the debugging facilities means adding or removing methods from the proxy. Instead of implementing the methods in the proxy, we could also have a trait (if the language provides traits or any other composable unit of behavior) with such methods. However, we still need to add the trait to the proxy class when debugging and remove it when we are not.

**Implementation complexity.** This solution is easy to implement: it just requires to create a subclass of the minimal object and implement the doesNotUnderstand: method.

**Constraints.** This solution is flexible since target objects do not need to implement any interface or method, nor to inherit from specific classes. The user can easily extent or change the purpose of the proxy adapting it to his own needs by just reimplementing the doesNotUnderstand:.

**Portability.** This approach needs just a few requirements that have to be provided by the language and the VM. Moreover, almost all available dynamic languages support these needs by default: a message like doesNotUnderstand:, a minimal object and the possibility to forward a message to another object. Therefore, it is easy to implement this approach in different dynamic languages.

## 7.4 Pharo Support for Proxies

Before presenting Ghost, we first explain the basis of the Pharo reflective model and the provided hooks that our solution uses. We show that Pharo provides, out of the box, all the support we need for Ghost's implementation *i.e.,* object replacement, interception of methods' execution and interception of all messages.

### 7.4.1 Pharo Reflective Model and VM Overview

Being a Smalltalk dialect, Pharo inherits the simple and elegant reflective model of Smalltalk-80. There are two important rules [Black 2009]: 1) *Everything is an object*; 2) *Every object is instance of a class*. Since classes are objects and every object is an instance of a class, it follows that classes must also be instances of classes. A class whose instances are classes is called a metaclass. Figure 7.3 shows a *simplified* reflective model of Smalltalk.

Figure 7.3 shows that a class is defined by a superclass, a method dictionary, an instance format, subclasses, name and a couple of others. The important point here is that the first

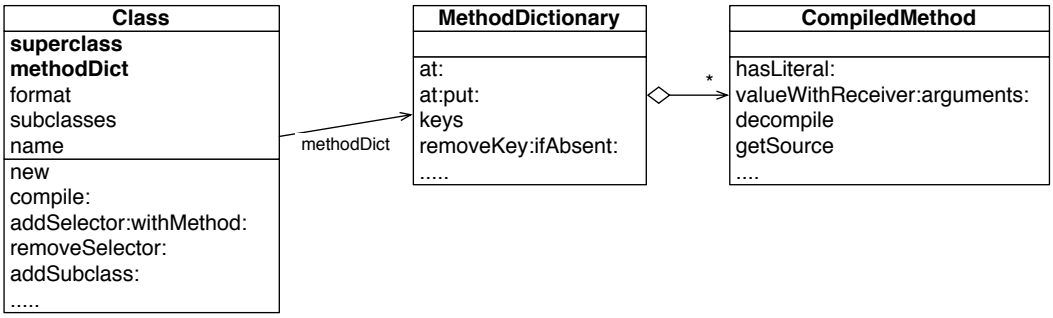


Figure 7.3: The basic Smalltalk reflective model. Bold instance variables are imposed by virtual machine logic.

two are imposed by the virtual machine[3]. The method dictionary is a hash table where keys are the methods names (called selectors in Smalltalk) and the values are the compiled methods which are instances of CompiledMethod.

### 7.4.2 Hooks and Features Provided by Pharo

The following is a list of the Pharo reflective facilities and hooks that Ghost uses for implementing proxies.

**Class with no method dictionary.** When an object receives a message and the VM does the method lookup, if the method dictionary of the receiver class (or of any other class in the hierarchy chain) is nil, the VM sends the message cannotInterpret: aMessage to the receiver. Contrary to normal messages, the lookup for the method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil. Otherwise, there would be an infinite loop. This hook is powerful for proxies because it let us intercept *all* messages that are sent to an object.

Figure 7.4 depicts the following situation: we get one object called myInstance, instance of the class MyClass whose method dictionary is nil. This class has a superclass called MyClassSuperclass. Figure 7.4 shows how the mechanism works when sending the message printString to myInstance. The message cannotInterpret: is sent to the receiver (myInstance) but starting the lookup in MyClassSuperclass.

**Objects as methods.** This facility allows us to intercept method executions. We can put an object that is not an instance of CompiledMethod in a method dictionary. Here is an example:

```
MyClass methodDict at: #printString put: Proxy new.
MyClass new printString.
```

[3]The VM actually needs three instances variables, the third being the format. But, the format is accessed only by a few operations *e.g.,* instance creation. Since the proxy intercepts all messages including creational ones, the VM will never need to access the format while using a proxy.

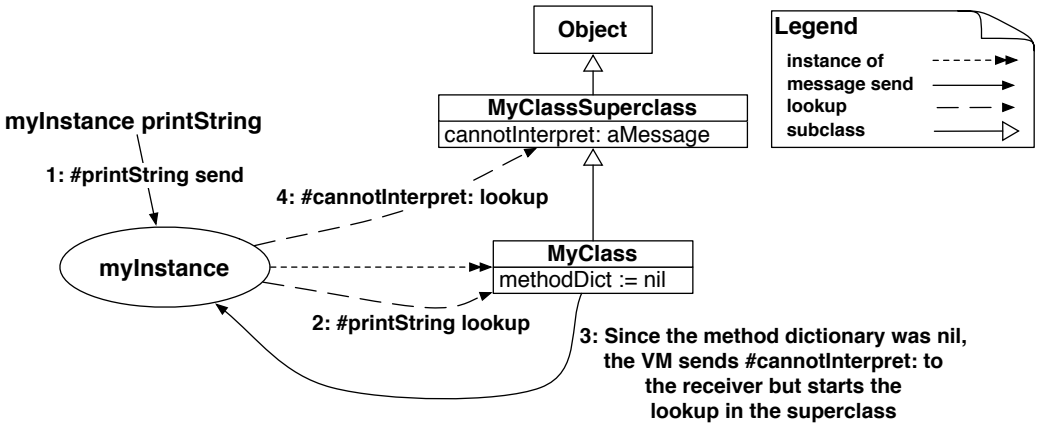


Figure 7.4: Message handling when a method dictionary is nil.

When the printString message is sent, the VM does the method lookup and finds an entry for #printString in the method dictionary. Since the object associated with the printString selector is not a compiled method, the VM sends a special message run: aSelector with: arguments in: aReceiver to that object, *i.e.,* the one that replaces the method in the method dictionary.

The VM does not impose any shape to objects acting as methods such as having certain amount of instance variables or certain format. The only requirement is to implement the method run:with:in:.

**Object replacement.** The primitive Object » become: anotherObject is provided by the VM and it atomically swaps the references of the receiver and the argument. All variables in the entire system that used to point to the receiver now point to the argument and vice-versa. In addition, there is also becomeForward: anotherObject which updates all variables in the entire system that used to point to the receiver so that they point to the argument, *i.e.,* it is only one way.

This feature enables us to replace a target object with a proxy so that all variables that are pointing to the target object are updated to point to the proxy.

## 7.5 Ghost's Design and Implementation

Ghost is open-source and developed under the MIT license[4]. The website of the project with its documentation is in: http://rmod.lille.inria.fr/web/pier/software/Marea/GhostProxies. The source code is available in the SqueakSource3 server: http://ss3.gemstone.com/ss/Ghost.html.

### 7.5.1 Overview Through an Example

Ghost distinguishes between *interceptors* and *handlers*. Proxies solely play the role of interceptors while handlers define the treatment of the trapped message. Data related to a

[4]http://www.opensource.org/licenses/mit-license.php

trapped message is reified as an object we call *interception*. Figure 7.5 shows Ghost's basic design which is explained in this section. The most important features of Ghost are: (1) to be able to intercept all messages but also to exclude a user-defined list, (2) to be uniform (to be able to proxify any objects, even sensitive ones like classes or method), and (3) to be stratified (*i.e.,* clear separation between proxies and handlers) in a *controlled manner*.

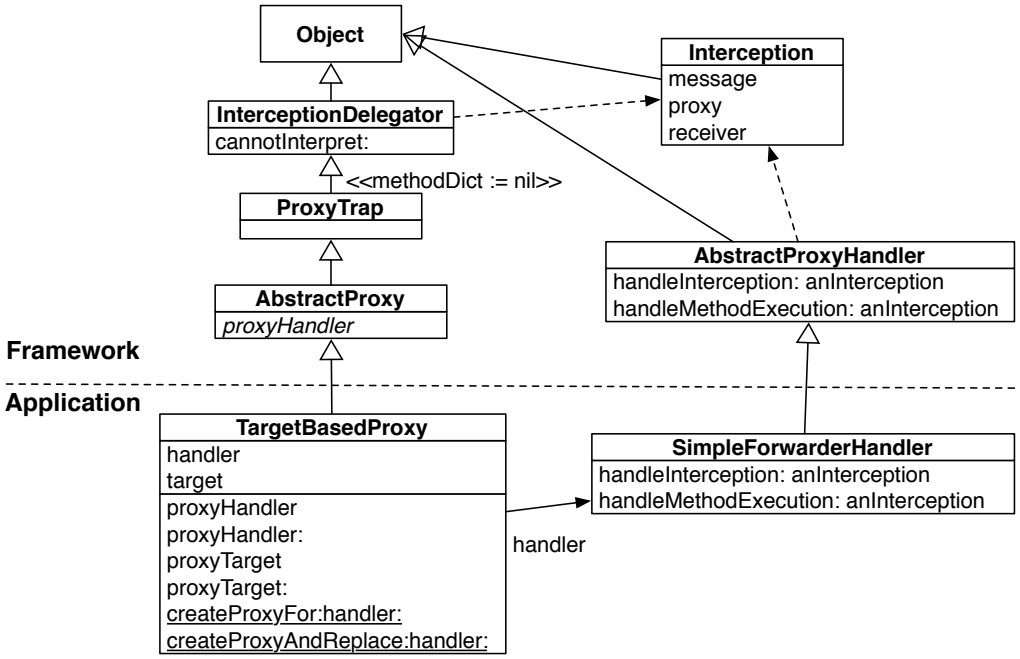


Figure 7.5: Part of the Ghost framework and an example of proxies for regular objects.

Ghost's implementation uses the following reflective facilities: classes with no method dictionary, objects as methods and object replacement. The basic kernel is based on the hierarchies of AbstractProxy (whose role is to intercept messages) and AbstractProxyHandler (whose role is to handle intercepted messages) together with the communication from the former to the latter through Interception instances.

The handlers' responsibility is to manage message interceptions trapped by proxies. What the handler does with the interception, depends on what the user wants. To illustrate the implementation, we use a SimpleForwarderHandler which just forwards the intercepted message to a target object. In this example, each proxy instance uses a particular handler instance which is accessed by the proxy via an instance variable. Another user of the framework may want to use the same handler instance for all proxies. Consequently, different proxies can use the same or different handlers. How proxies are mapped to handlers depends on the user needs and it is controlled by the method proxyHandler as explained later.

The information passed from a proxy to a handler is reified as an instance of the class Interception. It includes the message which reifies the selector and its arguments, the proxy and the receiver (as we see later sometimes the receiver is not the proxy but a different

object).

In real-world scenarios (see Section 7.10), a proxy often needs to hold some specific information, for example, a target object, an address in secondary memory, a filename, an identifier or any important information. Thus, the proxy should provide at least an accessor to allow the handler to retrieve this information. The need for these application-specific messages understood by the proxy leads us to *controlled stratification*. The developer *controls* and decides the set (usually small) of messages understood by the proxy. By carefully choosing selectors for these messages (usually we use a specific prefix), one avoids collisions with applications messages and enhances proxy's transparency.

A typical reason for controlled stratification is saving memory by sharing a unique handler among all proxies. In the context of the simple forwarder example, we need to map each proxy to a target object that will eventually perform the messages trapped by the proxy. If one goes for full stratification, the proxy will intercept all messages and send them to the handler. But, the handler should hold a reference to the target. Then, for every target object we would have two placeholders: a proxy and a handler. If we use the same object for both responsibilities, then there is no clear division between proxies and handlers. Therefore, in this example, to have a smaller memory footprint, we introduced an instance variable in class TargetBasedProxy that stores the target object. A singleton handler, shared among all proxies, asks each proxy for its target before forwarding the intercepted message. To let the handler access the target object of a given proxy, TargetBasedProxy class implements the method proxyTarget.

### 7.5.2 Proxies for Regular Objects

This section shows Ghost's implementation for regular objects. Subclasses of AbstractProxy (such as TargetBasedProxy) provide proxies for regular objects, *i.e.,* objects that do not need any special management. Their responsibility is to intercept messages.

**Proxy creation.** The following code shows how to create a proxy for a point (3,4). Since the handler is a simple forwarder, the messages are forwarded to the proxy's target.

```
testSimpleForwarder
    | proxy |
    proxy := TargetBasedProxy createProxyFor: (Point x: 3 y: 4) handler: SimpleForwarderHandler new.
    self assert: proxy x equals: 3.
    self assert: proxy y equals: 4.
```

The class method createProxyFor:handler: creates a new instance of TargetBasedProxy and sets the handler (the user specifies which handler to use just by passing it as a parameter) and the target object.

**Message Trapping in Action.** ProxyTrap is a special class whose method dictionary is nilled out once created. When we send a message to an instance of TargetBasedProxy if the message is not implemented in that class, the method lookup continues in the hierarchy until ProxyTrap, whose method dictionary is nil. For all those messages (the ones not implemented in TargetBasedProxy and AbstractProxy), the VM will eventually send the message

cannotInterpret: aMessage. Note that there are a few messages that are not executed but inlined by the compiler and the virtual machine (See Section 7.7). From now onwards, we consider that when we use cannotInterpret:, we intercept *all* messages except a specific list that we do not want to intercept. This is to distinguish it from doesNotUnderstand: where one can only intercept the messages not understood.

Coming back to the cannotInterpret:, remember that such message is sent to the receiver (in this case the aProxy instance) but the method lookup starts in the superclass of the class whose method dictionary is nil which, in this case, is InterceptionDelegator (see Figure 7.5). Because of this, InterceptionDelegator implements the mentioned method:

```
InterceptionDelegator >> cannotInterpret: aMessage
    | interception |
    interception := Interception for: aMessage proxy: self.
    ^ self proxyHandler handleInterception: interception.
```

An Interception instance is created and passed to the handler. In this case, for the interception we only need the proxy and the message. The receiver is unused here. InterceptionDelegator sends proxyHandler to get the handler. Therefore, proxyHandler is an abstract method which must be implemented by concrete proxy classes, *e.g.,* TargetBasedProxy and it must answer the handler to use.

Handler classes are user-defined and the example of the simple forwarder handler logs and forwards the received message to a target object as shown below.

```
SimpleForwarderHandler >> handleInterception: anInterception
    | answer |
    self log: 'Message ', anInterception message , ' intercepted'.
    answer := anInterception message sendTo: anInterception proxy proxyTarget.
    self log: 'The message was forwarded to target'.
    ^ answer
```

Direct subclasses from AbstractProxy *e.g.,* TargetBasedProxy are only used for regular objects. We see in the following sections how Ghost handles objects that do require special management such as methods (see Section 7.6.1) or classes (see Section 7.6.2).

### 7.5.3 Extending and Adapting Proxies and Handlers

To adapt the framework, users have to create their own subclass of AbstractProxyHandler and implement the method handleInterception:. They also need to subclass AbstractProxy and define which handler to use by implementing the method proxyHandler. It is up to the developer to store the handler in the proxy or to share a singleton handler instance among all proxies, or any other option. Other customizations are possible depending on the application's needs:

- Which state to store in the proxy. For example, rather than a simple target object, proxies for remote objects may require an IP, a port and an ID identifying the remote object. A database or object graph swapper may need to store a secondary memory address or ID.

- Which messages are implemented in the proxy and directly answered instead of being intercepted. The most common usage is implementing methods for accessing instance variables so that the handler can invoke them while managing an interception. Next section presents different examples.

### 7.5.4 Intercepting Everything or Not?

One would imagine that the best proxy solution is one that intercepts *all* messages. However, this is not what the user of a proxy library needs most of the times. Usually, developers need to send messages to a proxy and get an answer instead of being intercepted. Here are a few examples:

- Storing proxies in hashed collections means that proxies need to answer their hash.
- With remote objects, it is likely that the system will need to ask a proxy its target in the remote system or information about it *e.g.,* URI or ID.
- Serializing proxies to a file or network means that the serializer will ask its class and its instance variables to be serialized as well.
- Debugging, inspecting and printing proxies only makes sense if a proxy answers its own information rather than intercepting the message.

The question "Intercepting Everything or Not?" is really a difficult one. On the one hand, to use a proxy as a placeholder, it is useful that it understands some basic messages such as identityHash, inspect, class, etc. On the other hand, it is a problem since those messages are not intercepted anymore.

Not only the user of the proxy framework usually needs to send messages to a proxy, but also the proxy toolbox itself. For example, imagine the framework wants to validate and be sure that a proxy should not be proxified again. It should check whether the object to proxify is already a proxy or not. Therefore, the proxy should answer to a message like isProxy.

To support these requirements, Ghost provides a flexible design so that proxies can understand and answer specific messages. The way to achieve this is simply by implementing methods in proxy classes. All methods implemented below ProxyTrap in the hierarchy are not intercepted. With our solution, we have the best scenario: *the user* controls stratification and decides *what* to exclude in the proxies interception and intercept *all* the rest. With solutions like doesNotUnderstand:, one can also implement methods in proxy classes to avoid being intercepted but proxies are forced by the system to understand (and hence do *not* intercept) even more messages like those methods that every object understands (e.g. identityHash, initialize, isNil, etc.). Such messages are not defined by the user but by the system.

### 7.5.5 Messages to be Answered by the Handler

Apart from the possibility of adding methods to the proxy and avoiding interception of messages, Ghost supports special messages to which the *handler* must answer itself in-

stead of managing them as regular interceptions. A handler keeps a dictionary that maps selectors of messages intercepted by the proxy to selectors of messages to be performed by the handler. This user-defined list of selectors is used with different objectives such as debugging purposes, *i.e.,* those messages that are sent by the debugger to the proxy are answered by the handler and they are not managed as a regular interception. This eases the debugging in presence of proxies. The handler's dictionary of special messages for debugging can be defined as following:

```
SimpleForwarderHandler >> debuggingMessagesToHandle
    | dict |
    dict := Dictionary new.
    dict at: #basicInspect put:#handleBasicInspect:.
    dict at: #inspect put:#handleInspect:.
    dict at: #inspectorClass put:#handleInspectorClass:.
    dict at: #printStringLimitedTo: put: #handlePrintStringLimitedTo:.
    dict at: #printString put: #handlePrintString:.
    ^ dict
```

The dictionary keys are selectors of messages received by the proxy and the values are selectors of messages that the handler must send to itself. For example, if the proxy receives the message printString, then the handler sends itself the message handlePrintString: and answers that. All the messages to be sent to the handler (*i.e.,* the dictionary values) take as parameter an instance of Interception which contains the message, the proxy and the receiver. Therefore, such messages have access to all the required information.

These special messages are pluggable *i.e.,* they are easily enabled and disabled. Moreover, they are not coupled with debugging so they can be used every time a user wants certain messages to be implemented and answered directly by the handler rather than performing the default action for an interception. As we explain in the next section, this feature is used, *e.g.,* to intercept method's execution.

This feature is similar to the ability of define methods in the proxy so that they are understood instead of intercepted. Nevertheless, there are some differences which help the user to decide which of the two ways to use in each situation:

- The mechanism of the handler is pluggable, while defining methods in a proxy is not.

- Some methods like those accessing instance variables of the proxy (such as the target object) *have* to be in the proxy. Another example is primitive methods. For example, if we want the proxy to understand proxyInstVarAt: so that it can be used for serialization purposes, we have to define such method in the proxy because it calls a primitive. It is impossible to define that method in the handler without changing the primitive.

- Handlers can be shared among several proxy instances and even different types of proxies. Therefore, we cannot put specific behavior to a shared handler that applies only to a specific type of proxy.

## 7.6 Proxies for Classes and Methods

Ghost supports proxies for regular objects as well as for classes, methods and any other class that requires special management such as block closures or processes. In this section, we show how to correctly proxify methods and classes.

### 7.6.1 Proxies for Methods

In some dynamic languages, methods are first-class objects. This means that it is necessary to handle two typical and different scenarios when we want to proxify methods: (1) handling message sending to a proxified method object and (2) handling execution (from the VM) of a proxified method.

- *Sending a message to a method object.* In Pharo, for example, when a developer searches for senders of a certain method, the system has to check in the literals of the compiled method if it is sending such message. To do this, the system searches all the literals of the compiled methods of all classes. This means it sends messages (sendsSelector: in this case) to the objects that are in the method dictionary. When creating a proxy for a method, we need to intercept such messages.
- *Method execution.* This is when the compiled method is executed by the virtual machine. Suppose we want to create a proxy for the method username of User class. We need to intercept the method execution, for example, when doing User new username. Note that this scenario *only* exists if there is object replacement, *i.e.,* when the original method is replaced by a proxy.

**Proxy creation.** To clarify, imagine the following test:

```
testSimpleProxyForMethods
    | mProxy kurt method |
    kurt := User named: 'Kurt'.
    method := User compiledMethodAt: #username.
    mProxy := TargetBasedProxy createProxyAndReplace: method handler: SimpleForwarderHandler new.
    self assert: mProxy getSource equals: 'username ^ name'.
    self assert: kurt username equals: 'Kurt'.
```

The test creates an instance of User class and a proxy for method username. Using the method createProxyAndReplace:handler:, we create the proxy and we *replace* the original object (the method username in this case) with it. By replacing an object, we mean that all the pointers to the existing method then point to the proxy. Finally, we test both types of messages: sending a message to the proxy (in this case mProxy getSource) and executing a proxified method (kurt username in our example).

**Handling both cases.** Ghost solves both scenarios. In the first one, *i.e.,* mProxy getSource, Ghost has nothing special to do. It is just a message sent to a proxy and it behaves exactly the same way we have explained so far. In the second one, illustrated by kurt username, a proxified method is *executed*. When the VM looks for the method username, it

notices that, in the method dictionary, there is not a CompiledMethod instance but instead an instance of another class. Consequently, it sends the message run:with:in to such object. Since such object is a proxy in this case, the message run:with:in: is intercepted and delegated to the handler just like any other message.

As already explained, the handler can have a list of messages that require special management rather than performing the default action. With that feature, we map run:with:in to handleMethodExecution:, meaning that if the handler receives an interception with the selector run:with:in it sends to itself handleMethodExecution: and answers that. Subclasses of AbstractProxyHandler that want to handle interceptions of methods' execution must implement handleMethodExecution: to fit their needs, for example:

```
SimpleForwarderHandler >> handleMethodExecution: anInterception
    | targetMethod receiver arguments |
    targetMethod := anInterception proxy proxyTarget.
     "Remember the message was run: aSelector with: arguments in: aReceiver"
    receiver := anInterception message arguments third.
    arguments := anInterception message arguments second.
    ^ targetMethod valueWithReceiver: receiver arguments: arguments
```

That method just gets the required data from the interception and executes the method with the correct receiver and arguments by using the method valueWithReceiver:arguments:.

Notice that the Pharo VM does not impose any shape to methods. Therefore, as we showed in the previous example, we can use the same proxy class (TargetBasedProxy) that we use for regular objects.

**Alternatives.** Another approach to manage interceptions of methods' execution is to implement run:with:in: in the proxy itself. In such situation, we can get the data from the parameters, create an Interception instance and pass it to the handler. However, we believe proxies should understand as less as possible from the proxy toolbox machinery and leave such responsibilities for their handlers.

### 7.6.2 Proxies for Classes

Pharo represents classes as first-class objects and they play an important role in the runtime infrastructure. It is because of this that Ghost has to take them into account. Developers often need to be able to replace an existing class with a proxy. Instances hold a reference to their class and the VM uses this reference for the method lookup. Therefore, object replacement must not only update the references from other objects, but also the class references from instances.

AbstractClassProxy provides the basis for class proxies (See Figure 7.6). AbstractClassProxy is necessary because the VM imposes specific constraints on the memory layout of objects representing classes. The VM expects a class object to have the two instance variables superclass and methodDict in this specific order starting at index 1. We do not want to define TargetBasedClassProxy as a subclass of TargetBasedProxy because the two instance variables target and handler would get index 1 and 2, not respecting the imposed order. However, not being able to subclass is not a real problem in this case because there

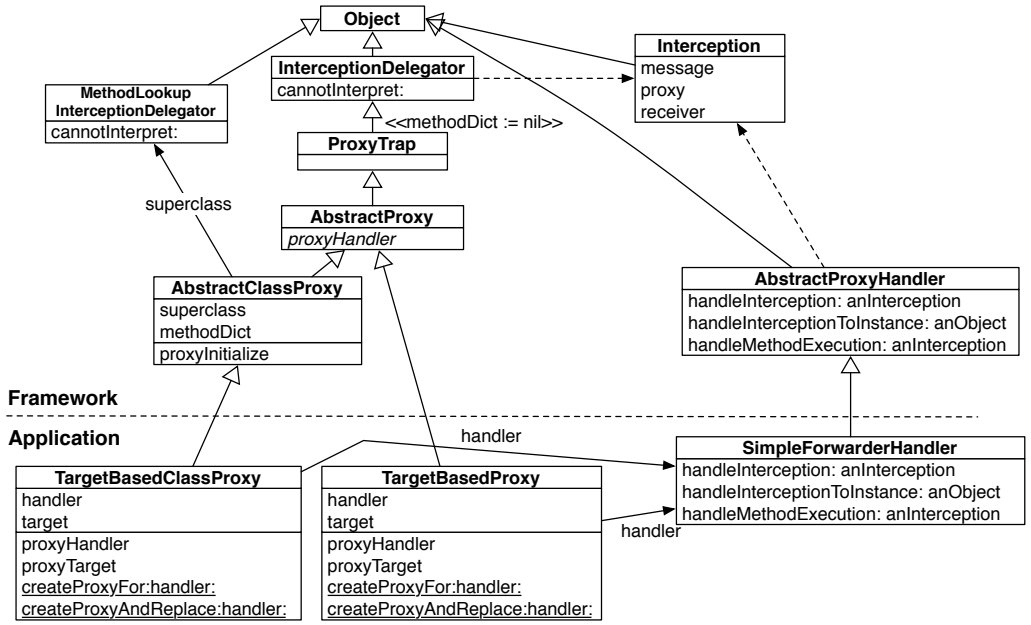


Figure 7.6: Part of the Ghost framework and an example of proxies for classes.

are only a few methods in common so we are not duplicating much code because of that.

**Requirements.** AbstractClassProxy has to be able to intercept the following kinds of messages:

- Messages that are sent directly to the class as a regular object.
- Messages that are sent to an instance of the proxified class, *i.e.,* an object whose class reference is pointing to the proxy (which happens as a consequence of the object replacement between the class and the proxy). This kind of message is only necessary when an object replacement takes place.

**Proxy creation.** To explain class proxies, consider the following test:

```
testSimpleProxyForClasses
    | cProxy kurt |
    kurt := User named: 'Kurt'.
    cProxy := TargetBasedClassProxy createProxyAndReplace: User handler: SimpleForwarderHandler new.
    self assert: User name equals: #User.
    self assert: kurt username equals: 'Kurt'.
```

This test creates an instance of User and then, with the message createProxyAndReplace:handler:, it creates a proxy that replaces the User class. Finally, it tests that we can intercept both kind of situations: messages sent to the proxy (in this case User name) and messages sent to instances of the proxified class (kurt username in this case).

**Handling two cases.** The first message, User name, has nothing special and it is handled like any other message. The VM will end up sending cannotInterpret: to the receiver and starting the method lookup in the class which method dictionary was nil, *i.e.,* InterceptionDelegator. The second message is more complicated and needs certain explanation.

The method createProxyAndReplace:handler: is equal to the one of AbstractProxy but something different happens. After creating a new instance, the method createProxyAndReplace:handler: sends the message proxyInitialize to it. In TargetBasedClassProxy we do not only set a handler and a target (as in the case of TargetBasedProxy), but also the minimal information required by the VM so that an *instance* of TargetBasedClassProxy can act as a *class*. Thus, we set its method dictionary to nil and its superclass to MethodLookupInterceptionDelegator.

Coming back to the example, when we evaluate kurt username, the class reference of kurt is pointing to the created TargetBasedClassProxy instance (as a result of object replacement). This proxy object acts as a class and it has its method dictionary instance variable to nil. Hence, the VM sends the message cannotInterpret: to the receiver (kurt in this case) but starting the method lookup in the superclass of the class with nilled method dictionary which is MethodLookupInterceptionDelegator. A simplified definition (later in this section we see the real implementation) of the cannotInterpret: of class MethodLookupInterceptionDelegator is the following:

```
MethodLookupInterceptionDelegator >> cannotInterpret: aMessage
    | proxy |
    proxy := aMessage lookupClass.
    interception := Interception for: aMessage proxy: proxy receiver: self.
    ^ proxy proxyHandler handleInterceptionToInstance: interception.
```

It is important to notice the difference in this method in comparison with the implementation of InterceptorDelegator. In both situations, User name and kurt username, we always need to get the proxy to perform the desired action.

User name case. The method cannotInterpret: is called on InterceptorDelegator and the receiver, *i.e.,* what self is pointing to, is the proxy itself.

kurt username case. The method cannotInterpret: is called on MethodLookupInterceptionDelegator and self points to kurt and not to the proxy. The proxy is the looked up class, *i.e.,* the receiver's class, which we can get from the Message instance. Then we send the message handleInterceptionToInstance: to the handler. We use that message instead of handleInterception: because the user may need to perform different actions. What the implementation of handleInterceptionToInstance: does in SimpleForwarderHandler is to execute the desired method with the receiver without sending a message to it[5] avoiding another interception and an infinitive loop.

To conclude, with this implementation, we can successfully create proxies for classes,

[5]Pharo provides the primitive method receiver:withArguments:executeMethod: which directly evaluates a compiled method on a receiver with a specific list of arguments without actually sending a message to the receiver.

*i.e.,* to be able to intercept the two mentioned kind of messages and replace classes by proxies.

**Discussion.** We could have reused cannotInterpret: implementation of InterceptionDelegator instead of using MethodLookupInterceptionDelegator and set it also in the method proxyInitialize of AbstractClassProxy. That way, InterceptionDelegator is taking care of both types of messages. The solution has to check which kind of message it is and, depending on that, perform a specific action. We think the solution with MethodLookupInterceptionDelegator is much cleaner.

Ghost's implementation uses AbstractProxyClass not only because it is cleaner from the design point of view, but also because of the memory footprint. Technically, we can *also* use AbstractProxyClass for regular objects and methods. However, this implies that, for every target to proxify, the size of the proxy is unnecessary bigger in memory footprint because of the additional instance variables needed by AbstractProxyClass.

**Problem with subclasses of proxified classes.** When we proxify a class but not its instances and one of the instances receives a message, Ghost intercepts the method lookup and finally uses the cannotInterpret: method from MethodLookupInterceptionDelegator. In that method, the proxy can be obtained using aMessage lookupClass, because the class of the receiver object is a proxy. However, this is not always possible. If we proxify a class but not its subclasses and a subclass' instance receives a message which does not match any method, the lookup eventually reaches the proxified class. Ghost intercepts the method lookup and executes the cannotinterpret: method from MethodLookupInterceptionDelegator. At this stage, we need to *find* the trapping class, i.e., the first class in the hierarchy with a nilled method dictionary. In this scenario, message lookupClass does not return a proxy but an actual class: a subclass of the proxified class. To solve this problem, Ghost does the following implementation:

```
MethodLookupInterceptionDelegator >> cannotInterpret: aMessage
    | proxyOrClass proxy |
    proxyOrClass := aMessage lookupClass.
    proxy := proxyOrClass ghostFindClassWithNilMethodDictInHierarchy.
    interception := Interception for: aMessage proxy: proxy receiver: self.
    ^ proxy proxyHandler handleInterceptionToInstance: interception.
```

The method ghostFindClassWithNilMethodDictInHierarchy checks if the current class has a nilled method dictionary and, if it does not, it recurs to superclasses. This method is also implemented in AbstractClassProxy just answering self.

## 7.7 Special Messages and Operations

Being unable to intercept messages is a problem because it means they will be directly executed by the proxy instead. This can lead to different execution paths in the code, errors or even make the VM to crash.

One common problem when trying to intercept all messages is the existing optimizations for certain methods. In Pharo, as well as in other languages, there are two kinds of optimizations that affect proxies: (1) inlined methods and (2) special bytecodes.

### 7.7.1 Inlined Methods

These are optimizations done by the compiler. For example, messages like ifTrue:, ifNil:, and:, to:do:, etc. are detected by the compiler and are not encoded with the regular bytecode of message send. Instead, such methods are directly encoded using different bytecodes such as jumps. As a result, these methods are never executed and cannot be intercepted by proxies. The second kind of optimization is between the compiler and the virtual machine.

Ideally, we would like to handle inlined messages the same way than regular ones. The easiest yet naive way is to disable the inlining. However, disabling all optimizations produces two important problems. First, the system gets significantly slower. Second, when optimizations are disabled, those methods are executed and there can be unexpected and random problems which are difficult to find. For instance, in Pharo, everything related to managing processes, threads, semaphore, etc., is implemented in Pharo itself without proper abstractions. The processes' scheduler can only switch processes between message sends. This means that there are some parts in the classes like Process, ProcessorScheduler, Semaphore, etc., that have to be atomic, *i.e.,* they cannot be interrupted and switched to another process. If we disable the optimizations, such code is not atomic anymore. Other examples are the methods used to enumerate objects or to get the list of objects pointing to another one. While iterating objects, each send to whileTrue: (or any other of the inlined methods) will create more objects like MethodContext generating an infinitive loop.

The messages that are inlined in Pharo 1.4 are stored in the class variable MacroSelectors of the class MessageNode and they are:

1. ifTrue:, ifFalse:, ifTrue:ifFalse:, ifFalse:ifTrue:, and:, or:, implemented in True and False.

2. ifNil:, ifNotNil:, ifNil:ifNotNil: and ifNotNil:ifNil:, implemented in both classes: ProtoObject and UndefinedObject.

3. to:do: and to:by:do:, implemented in Number.

4. whileFalse:, whileTrue:, whileFalse, whileTrue and repeat, implemented in BlockClosure.

Theses messages involve special objects that can be split into two categories. In the first category, we have true, false, nil, and numbers which are related to the messages of items 1 to 3. These objects are so low-level that they cannot be replaced by proxies. Indeed, sending the become: message to one of them result into VM hang or crash. To our knowledge there is no use case where these objects should be proxified. Hence, inlining their messages is not an issue.

Block closures form the second category of objects which are involved in inlined messages. Actually, the inlining of the messages whileFalse:, whileTrue:, whileFalse, whileTrue and repeat is performed only when the receiver is a closure. In situations where the receiver

is the result of a message or a variable like in the code below, the message is not inlined. The following code illustrates these two cases:

```
|welcomeBlock|
[Transcript cr; show: 'Hello'] repeat.  "Message inlined"
welcomeBlock := [Transcript cr; show: 'Welcome'].
welcomeBlock repeat.  "No inlining"
```

Since a proxy is not recognized as closure by the compiler, the message is not inlined. Therefore, messages sent to a proxy for a block closure are intercepted.

### 7.7.2 Special Associated Bytecodes and VM optimization

The second type of optimization correspond to a special list of selectors[6] that the compiler does not encode with the regular bytecode of message send. Instead, these selectors are associated with special bytecodes that the VM directly interprets. For these selectors, there are three groups:

- Methods that may be executed depending on the receiver or argument type. For example, the execution 1 + 2 never sends the message + to 1 because both are 32 bit integers but 1 + 'aString' will do. Analyzing the implementation of the bytecode associated with each of those selectors shows us that all of them check the type of the receiver and arguments: the method is only executed when there is a type mismatch. For example, all arithmetic operations and bit manipulation expect small integers or floats, boolean operations expect booleans, size expects strings or arrays, etc. Whenever the receiver or arguments do not satisfy the conditions, the bytecode follows with the normal method execution, *i.e.*,, the message is sent. Since proxies never satisfy the conditions, then the messages are sent by the VM and trapped like normal messages.
- Methods that are always sent such as new, next, nextPut:, do:, etc. Here the only optimization done by the VM is just a quick and internal set of the selector to execute and the argument count. These methods are not a problem for proxies since they are always executed.
- Methods which are never executed but directly answered by the VM internal execution. In Pharo 1.4, there is only one single method of this type[7]: ==. It answers whether the receiver and the argument are the same object.

The conclusion is that only == is not intercepted by proxies. Nevertheless, even if it were possible, object identity should never be handled as a regular message as demonstrated by Marc Miller [8]. In addition, == is not a problem when using proxies with object replacement. For example, given the following code:

[6] In Pharo 1.4, we can get the list of those selectors by executing Smalltalk specialSelectors. Those are: +, -, <, >, <=, >=, =, ~=, *, /, \\, @, bitShift:, //, bitAnd:, bitOr:, at:, at:put:, size, next, nextPut:, atEnd, ==, class, blockCopy:, value, value:, do:, new, new:, x and y

[7] In earlier versions of Pharo class was also of this type, but the special bytecode associated it was removed in Pharo 1.4.

[8] http://erights.org/elib/equality/grant-matcher/index.html

```
(anObject == anotherObject)
    ifTrue: [ self doSomething]
    ifFalse: [self doSomethingDifferent]
```

Imagine that anObject is replaced by a proxy, *i.e.,* all objects in the system which were referring to the target (anObject), will now refer to the proxy. Since all references have been updated, == continues to answer correctly. For instance, if anotherObject was the same object as anObject, == answers true since both are referencing the proxy now. If they were not the same object, == answers false. Hence, identity is not a problem when there is object replacement.

## 7.8 Evaluation

In this section, we evaluate Ghost using the criteria defined in Section 7.2.

**Stratification.** This solution is stratified. On the one hand, there is a clear separation between proxies and handlers. On the other hand, interception facilities are separated from application functionality. Indeed, the application can even send the cannotInterpret: message to the proxy and it will be intercepted like any other message. Thus, the proxy API does not pollute the application's namespace. Moreover, stratification is controlled: users can still select which messages they do not want to be intercepted.

**Interception levels.** It can intercept all messages while also providing a way to exclude user defined messages.

**Object replacement.** Such feature is important since it allows one to seamlessly substitute an object with a proxy. This is provided by Ghost thanks to the become: primitive of Pharo.

**Uniformity.** This implementation is uniform since proxies can be used for regular objects as well as for classes and methods. Moreover, they all provide the same API and can be used polymorphically. Nevertheless, there is still non-uniformity regarding some other special classes and objects. Most of them are those that are present in what is called the *special objects array*[9] which contains the list of special objects that are known by the VM. Examples are the objects nil, true, false, etc. It is not possible to do a correct object replacement of those objects with proxies. The same happens with immediate objects, *i.e.,* objects that do not have object header and are directly encoded in the memory address such as SmallInteger.

The special objects array contains not only regular objects, but also classes. Those classes are known and used by the VM so it may impose certain shape, format or responsibilities in their instances. For example, one of those classes is Process and it is not possible

[9] Check the method recreateSpecialObjectsArray in Pharo for more details.

to correctly replace a Process instance by a proxy. These limitations occur only when object replacement is desired. Otherwise, there is no problem and proxies can be created for those objects. Since classes and methods play an important role in the runtime infrastructure of Pharo, creating proxies for them useful in several scenarios (as we see in the next section) and that is why Ghost provides special management for them. From our point of view, the mentioned limitations only exist in the presence of unusual needs. Nevertheless, if the user also needs special management for certain objects like Process instances, then he can create a particular proxy that respects the imposed shape.

**Transparency.** Ghost proxies are transparent even with the special messages inlined by the compiler and the VM. The only exception is object identity (message ==).

**Efficiency.** From the CPU point of view, this solution is fast and it has low overhead.
Ghost provides an efficient memory usage with the following optimizations:

- TargetBasedProxy and TargetBasedClassProxy are defined as compact classes. This means that in a 32 bits system, their instances' object header is only 4 bytes long instead of 8 bytes. For those instances whose body part is more than 255 bytes and whose class is compact, their header is 8 bytes instead of 12. The first word in the header of regular objects contains flags for the garbage collector, the header type, format, hash, etc. The second word is used to store a reference to the class. In compact classes, the reference to the class is encoded in 5 bits in the first word of the header. These 5 bits represent the index of a class in the compact classes array set by the language[10] and accessible from the VM. With these 5 bits, there are 32 possible compact classes. This means that, from the language side, the developer can define up to 32 classes as compact. Declaring the proxy classes as compact, allows proxies to have smaller header and, consequently, smaller memory footprint.

- Proxies only keep the minimal state they need. AbstractProxy defines no structure and its subclasses may introduce instance variables needed by applications.

- In the methods for creating proxies presented so far (createProxyFor:handler: and createProxyAndReplace:handler:), the last parameter is a handler. This is because, in our example, each proxy holds a reference to the handler. However, this is only necessary when the user needs one handler instance per target object which is not often the case. The handler is sometimes stateless and can be shared and referenced through a class variable or a global one. In that scenario, proxyHandler must be implemented to answer a singleton. Therefore, we can avoid the memory cost of a handler instance and its reference from the proxy. If we consider that the handler has no instance variable, then it is 4 bytes saved for the instance variable in the proxy and 8 bytes for the handler instance. That gives a total of 12 bytes saved per proxy in a 32 bits.

[10] See methods SmalltalkImage»compactClassesArray and SmalltalkImage»recreateSpecialObjectsArray.

**Ease of debugging.** Because Ghost provides a way to have messages answered directly by the handler, we can make debugging with proxies very easy. The handler can answer all methods related to debugging, inspecting, etc. In contrast with traditional proxy implementation based on doesNotUnderstand:, this mechanism is pluggable, *i.e.,* it can be enabled or disabled at runtime by just changing a dictionary. There is no need to remove or add methods to the proxy.

**Implementation complexity.** Ghost is easy to implement[11] in Pharo: it consists of 11 classes with an average of 6 methods per class and each method has an average of 4 lines of code. The total amount of lines of code is approximately 300. Ghost is covered by approximately 10 unit tests that cover all use cases.

**Constraints.** The solution is flexible since the objects to proxify can inherit from any class and are free to implement or not all the methods they want. There is no kind of restriction imposed by Ghost. In addition, the user can easily extend or change the purpose of the toolbox adapting it to his own needs by just subclassing a handler and a proxy.

**Portability.** Ghost is not portable to other Smalltalk because it is based on a VM hook (the cannotInterpret: message) present in the Pharo VM. In addition, it also needs object replacement (become: primitive) and objects as methods (run:with:in: primitive). The become: primitive is present in all Smalltalk dialects because it is used by the language itself. The hook method run:with:in: is not available in all dialects but we only need it if we need to intercept method execution. Furthermore, we rely on the primitive method receiver:withArguments:executeMethod: to be able execute a method on a receiver object without actually sending a message to it. We only use this method when we proxify classes and this method is present in some Smalltalk dialects.

Without these reflective facilities, we cannot easily implement all the required features of a good proxy library. In the best scenario, we can do it but with substantial development effort such as modifying the VM or compiler or even creating them from scratch. Pharo provides all those features by default and no changes are required for Ghost.

## 7.9 Discussing VM Support for Proxies

One important question of a proxy implementation is whether virtual machine support is worth it or not. To continue our experiment in Pharo, we analyze the advantages and disadvantages of including proxy support in the Pharo VM. Here is the list of needed modifications:

- The VM has to distinguish a proxy from a regular object. The simplest approach is to make the VM assume that an object is a proxy when it is an instance of a certain class, say TargetBasedProxy. However, this limits the different proxy classes we can have in the system. Then the VM can have a list of proxy classes which can be

[11] The source code is available at: http://ss3.gemstone.com/ss/Ghost.html

defined from the language side. Another approach is to use a bit in the object header to mark objects as proxies. From the language side, there can be a primitive to set the value of such bit. This is more flexible since every object can be marked as proxy.

- The method lookup in the VM has to be modified to check whether the receiver is a proxy or not. If it is not, the method lookup follows the normal path. If it is a proxy, it should delegate the interception to its handler.

These modifications are easy to achieve in a standard Pharo VM. However, from our point of view, it has the following drawbacks:

- It is difficult to have a custom list of messages that should not be intercepted. This is because a proxy is always seen by the VM as a proxy so it always delegates the interception to the handler. One can also change the VM and define a list of messages that should be excluded but, again, this means that the user has to set such list. Furthermore, the method lookup will be slower.
- Intercepting the lookup is easy in a basic VM but, when the VM has a JIT (just in time compiler) and PICs (polymorphic inline caches), the lookup starts to be split in several places: jitted code calling non jitted code, jitted code calling jitted code, found in PIC, not found in PIC, etc.
- Even when there are no proxies, there is an overhead in the method lookup. With Ghost, the cost is only paid for proxies.
- We did not find a problem that we are unable to solve directly with Ghost's implementation. The only case where we think having VM support for proxies can help is if one wants to proxify true, false, nil or numbers. However, such change is insufficient since the problem of the inlined messages such as ifTrue:, ifNil:, etc. still needs to be fixed.

To conclude, after our experiment with Ghost proxies, we argue that moving the proxies support to the VM side is not only not worth it, but also a bad choice. With the current implementation of Ghost, we did not find any blocking problems that could only be solved by using VM supported proxies. For this reason, our analysis concludes that having proxies only handled on the language side is better (more flexible, less overhead, less complexity).

## 7.10 Case Studies

As a matter of showing possible uses of Ghost proxies, we present the following real cases:

### 7.10.1 Marea

The main user of Ghost is Marea. Marea's goal is to use less memory by only leaving in primary memory what is needed and used swapping out the unused objects to secondary memory. 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 a message, it loads back the swapped out object from secondary memory. Since Marea replaces each object with a proxy, the amount of released memory varies significantly depending on the memory footprint of the proxies.

Marea needs that all objects which were pointing to the existing target object are updated so that they point to the proxy, *i.e., object replacement*. All kinds of objects are swapped whether they are normal objects, classes or methods. In Marea, a proxy does not hold a reference to a target object but instead the address (composed by a graph ID and a position) of the target object in secondary memory. Therefore, Marea implements its own proxy classes Proxy and ClassProxy. To know which proxy to instantiate for a particular object of the graph being swapped, Marea delegates to the original objects the creation of the proxy. Object (the root of the class hierarchy in Pharo) implements the method newProxyWith:graphID: answering an instance of Proxy. Then, Class overrides that method and returns an instance of ClassProxy.

When a proxy intercepts a message, it means that the swapped object is needed again. Because of this, for every interception the handler delegates to ObjectGraphImporter. This class reads the swapped object from disk (the proxy has the needed information) loading it into primary memory and, then, it uninstalls the proxy, *i.e.,* the original object is replaced again by the proxy.

```
MareaHandler >> handleInterception: anInterception
    | originalObject |
    originalObject := ObjectGraphImporter swapIn: anInterception proxy.
    ^ anInterception message sendTo: originalObject.
```

Since for Marea it is important that proxies have the minimum memory footprint possible, we take advantage of some features provided by Ghost:

- Proxies are instances of *compact classes* (Proxy and ProxyClass).
- Since the handler is stateless, it is shared among proxies.

### 7.10.2 Method Wrappers

Method wrappers [Brant 1998] control the execution of methods by wrapping them into other objects (*i.e.,* the wrappers) that perform some task on method evaluation. The implementation of method wrappers is straightforward with Ghost. In fact, it has been already shown with the self log: in the example of SimpleForwarderHandler of Section 7.6.1. We can use the regular TargetBasedProxy class with a CompiledMethod as a target and implement the following handler:

```
AbstractMethodWrapper >> handleMethodExecution: anInterception
    | answer |
    self preExecutionFor: anInterception.
    answer := anInterception message sendTo: anInterception proxy proxyTarget.
    self postExecutionFor: anInterception.
    ^ answer
```

Then concrete subclasses implement preExecutionFor: and postExecutionFor:. In addition, if the user needs to wrap all methods of a class and perform the same action for all the interceptions, then a better approach is to directly create a TargetBasedClassProxy rather than creating a TargetBasedProxy for every method.

Apart from intercepting method execution, we can also intercept messages sent to proxies that take the place of methods:

```
AbstractMethodWrapper >> handleInterception: anInterception
    | answer |
    self preMessageSentFor: anInterception.
    answer := anInterception message sendTo: anInterception proxy proxyTarget.
    self postMessageSentFor: anInterception.
    ^ answer
```

In this case, concrete subclasses must implement preMessageSentFor: and postMessageSentFor:.

## 7.11 Related Work

### 7.11.1 Proxies in dynamic languages

**Objective-C.** Objective-C provides a proxy implementation called NSProxy [12]. This solution consists of an abstract class NSProxy that implements the minimum number of methods needed to be a root class. Indeed, this class is not a subclass of NSObject (the root class of the hierarchy chain), but a separate root class (like subclassing from nil in Smalltalk). The intention is to reduce method conflicts between the proxified object and the proxy. Subclasses of NSProxy can be used to implement distributed messaging, future objects or other proxies usage. Typically, a message to a proxy is forwarded to a proxified object which can be an instance variable in a NSProxy subclass.

Since Objective-C is a dynamic language, it needs to provide a mechanism like the Smalltalk doesNotUnderstand: when an object receives a message that cannot understand. When a message is not understood, the Objective-C runtime sends methodSignatureForSelector: to see what kind of argument and return types are present. If a method signature is returned, the runtime creates a NSInvocation object describing the message being sent and then sends forwardInvocation: to the object. If no method signature is found, the runtime sends doesNotRecognizeSelector:.

NSProxy subclasses must override the forwardInvocation: and methodSignatureForSelector: methods to handle messages that they do not implement themselves. By implementing the method forwardInvocation:, a subclass can define how to process the invocation *e.g.,* forwarding it over the network. The method methodSignatureForSelector: is required to provide argument type information for a given message. A subclass' implementation should be able to determine the argument types for the messages it needs to forward and it should be able to build a NSMethodSignature object accordingly. Note that, from this point of view, Objective-C is not so dynamic.

[12] Apple developer library documentation: http://developer.apple.com/library/ios/#documentation/cocoa/reference/foundation/Classes/NSProxy_Class/Reference/Reference.html.

To sum up, the developer needs to subclass NSProxy and implement the forwardInvocation: to handle messages that are not understood by itself. One of the drawbacks of this solution is that the developer does not have control over the methods that are implemented in NSProxy. For example, such class implements the methods isEqual:, hash, class, etc. This is a problem because those messages will be understood by the proxy instead of being intercepted. This solution is similar to the common solution in Smalltalk with doesNotUnderstand:.

**Ruby.** In Ruby, there is a proxy implementation which is called Delegator. It is just a class included in Ruby standard library but which can be easily modified or implemented from scratch. Similar to Objective-C and Smalltalk (and indeed, to most dynamic languages), Ruby provides a mechanism used when an object receives a message that it does not understand. This method is called method_missing(aSelector, *args). Moreover, from Ruby version 1.9, there is a new minimal class called BasicObject which understands a few methods and is similar to ProtoObject in Pharo.

Ruby's proxies are similar to Smalltalk's proxies using doesNotUnderstand: and to Objective-C' NSProxy as they have a minimal object (subclass from BasicObject) and implement method_missing(aSelector, *args) to intercept messages.

**Javascript.** Mozilla's Spidermonkey JavaScript engine has long included a nonstandard way of intercepting method calls based on Smalltalk's doesNotUnderstand:. The equivalent method is named noSuchMethod. Such solution is not stratified and it only intercepts the messages that are not understood.

Van Cutsem et al. implemented what they call "Dynamic Proxies" [Van Cutsem 2010] which are objects that act like normal objects but whose behavior is controlled by another object known as handler. They provide a clear division between proxies and handlers. Similarly to our possibility of creating proxies for methods, they can create proxies for Javascript functions since they are objects too. As well as we do, they have several reasons to avoid intercepting === (object identity in Javascript).

From what we can understand, their solution requires changes in the VM. Contrary to Smalltalk where almost everything is a message send, in Javascript, apart from function calls, there are operators. We understand that the VM needs to be changed so that each of these operators can check whether the receiver is a proxy or not redirecting the invocation to the handler rather than following the normal steps when the answer is positive. They also provide way for the user to specify a list of properties that are answered directly by the proxy instead of being intercepted.

The authors also claim that having only one trap message *e.g.,*doesNotUnderstand: does not scale if we were to introduce additional traps to intercept not only method invocation, but also property access, assignment, lookup, enumeration, etc. Nonetheless, some items of such list do not apply to all programming languages. For example, in Pharo, the only way to access instance variables from outside an object is by sending a message so it is a message send. Enumerations are also just messages. The lookup can be intercepted with Ghost by using proxies for classes. So, from such list, the only item Ghost is unable to intercept is

assignment but it is not even clear why one would want to intercept assignments.

Javascript is a prototype-based language which, besides function calls, it has more language constructors. Smalltalk is an object-oriented programming language and system where most of the language constructions are message sends. Furthermore, their work takes a more reflective standpoint. They want a way to reify every operation such as message send, instance variable access, etc. In Smalltalk, this is not needed for proxies. Nevertheless, if we want to do behavioral reflection and change *e.g.,* instance variable access (to make it persistent, for example), then we would need a way to intercept them even when accessing from within the object.

### 7.11.2 Proxies in static languages

**Java.** Java, being a statically-typed language, supports quite limited proxies called *Dynamic Proxy Classes*. It relies on the Proxy class from the java.lang.reflect package.

> *"Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.*
>
> *Each proxy instance has an associated invocation handler object, which implements the interface InvocationHandler. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the invoke method of the instance's invocation handler, passing the proxy instance, a java.lang.reflect.Method object identifying the method that was invoked, and an array of type Object containing the arguments. The invocation handler processes the encoded method invocation as appropriate and the result that it returns will be returned as the result of the method invocation on the proxy instance."*
>
> (Java Dynamic Proxies. The Java Platform 1.5 API Specification)

The creation of a dynamic proxy class can only be done by providing a list of java interfaces that should be implemented by the generated class. All messages corresponding to the declarations in the provided interfaces will be intercepted by a proxy instance of the generated class and delegated to a handler object.

Java proxies have the following limitations:

- The user *cannot* create a proxy for instances of a class which has not all its methods declared in interfaces. This means that, if the user wants to create a proxy for a domain class, he is forced to create an interface for it. Eugster [Eugster 2006] proposed a solution which provides proxies for classes. There is also a third-party framework based on bytecode manipulation called CGLib [13] which provides proxies for classes.

- *Only* the methods defined in the interface will be intercepted which is a big limitation.

[13] cglib Code Generation Library: http://cglib.sourceforge.net.

- Java interfaces do not support private methods. Since Java proxies require interfaces, private methods cannot be intercepted either. Depending on the proxy usage, this can be a problem.

- Proxies are subclass from Object forcing them to understand several messages *e.g.,* getClass. So, a proxy answers its own class instead of the target's one. Therefore, the proxy is not transparent and it is not fully stratified. Moreover, there are some specific exceptions: when the messages hashCode, equals or toString (declared in Object) are sent to a proxy instance, they are encoded and dispatched to the invocation handler's invoke method, *i.e.,* they are intercepted.

**.Net.** Microsoft's .NET proposes a closely related concept of Java dynamic proxies with nearly the same limitations. There are other third-party libraries like *Castle DynamicProxy* [14] or *LinFu* [15]. DynamicProxy differs from the proxy implementation built into .NET which requires the proxified class to extend MarshalByRefObject. This is a too heavy constraint since instances of classes that do not subclass MashalByRefObject cannot be proxified. In LinFu, every generated proxy dynamically overrides all of its parent's virtual methods. Each of its respective overridden method implementations delegates each method call to the attached interceptor object. However, none of them can intercept non-virtual methods.

### 7.11.3 Comparison

Statically typed languages, such as Java or .NET, support quite limited proxies [Barrett 2003]. Java and .Net suffer from the lack of replacement issue and transparency. Another problem in Java is that one cannot build a proxy with fields storing any specific data. Therefore, one has to put everything in the handler meaning no handler sharing is possible which ends in a bigger memory footprint. Proxies are far more powerful, flexible, transparent and easy to implement in dynamic languages than in static ones.

Dynamic languages just need two features to implement a basic Proxy solution: 1) a mechanism to handle messages that are not understood by the receiver object and 2) a minimal object that understands a few or no messages so that the rest are managed by the mentioned mechanism. Objective-C NSProxy, Ruby Decorator, etc, all work that way. Nevertheless, none of them solve all the problems mentioned in this chapter:

**Uniformity.** All the investigated solutions create proxies for specific objects but none of them are able to create proxies for classes or methods.

**Object replacement.** Some proxy solutions can create a proxy for a particular object X. The user can then use that proxy instead of the original object. The problem is that there may be other objects in the system referencing X. Without object replacement, those references will still be pointing to X instead of pointing to the proxy. Depending on the proxies usage, this can be a limitation.

[14] Castle DynamicProxy Library: http://www.castleproject.org/projects/dynamicproxy.
[15] LinFu Proxies Framework: http://www.codeproject.com/KB/cs/LinFuPart1.aspx.

**Memory footprint.** None of the solutions take special care of the memory usage of proxies. This is a real limitation when proxies are being used to save memory.

## 7.12 Conclusion

In this chapter, we described the need for proxies, their different usages and common problems while trying to implement them. We introduced Ghost: a generic and lightweight proxy implementation on top of Pharo Smalltalk.

Our solution provides uniform proxies not only for regular instances, but also for classes and methods. Ghost optionally supports object replacement. In addition, Ghost proxies can have a small memory footprint. Proxies are powerful, easy to use and extend and its overhead is low.

Ghost's implementation takes advantages of Pharo VM reflective facilities and hooks. Nevertheless, we believe that such specific features, provided by Pharo and its VM, can also be ported to other dynamic programming language.

Now that we have already presented Fuel serializer and Ghost proxies, in the next chapter we discuss the implementation details and key design aspects of Marea. We also lists possible problems and requirements that would be faced when implementing Marea in other programming language.

CHAPTER 8

# Marea's Implementation Details

## Contents



## At a Glance

This chapter provides details of Marea's implementation and design. In addition, it discusses issues we found in Marea but that represent recurrent problems that other implementations will face.


**Keywords:** Object swapping, object faulting, virtual memory, serialization, proxy, Smalltalk.

## 8.1 Introduction

Along this dissertation, we claimed we wanted to build the virtual memory manager at the programming language level. This is because it is much easier to develop, maintain, port, adapt and understand.

In this chapter, we describe Marea's object-oriented design and we demonstrate that Marea is built completely at the language level. It does not modify the virtual machine nor the OS.

#### Structure of the Chapter

In the next section we present the details of Marea's implementation and design. Section 8.3 discusses issues related to general implementations. Finally, Section 8.4 summarizes the chapter.

## 8.2 Marea Implementation

Marea is fully implemented in Pharo, an open-source Smalltalk-inspired programming language [Black 2009]. Marea is open-source and developed under the MIT license[1]. The website of the project with its documentation is at: http://rmod.lille.inria.fr/web/pier/software/Marea. The source code is available in the SqueakSource3 server: http://ss3.gemstone.com/ss/Marea.html. As explained in Section 4.2, we relied on Ghost (the advanced proxy toolbox [Martinez Peck 2011a, Martinez Peck 2012b] described in Chapter 7) and Fuel (the fast serializer [Dias 2011, Dias 2012] described in Chapter 6). In this section, we explain Marea's design and its requirements from the programming language.

### 8.2.1 Marea Design

Marea is built with a clean object-oriented design and *without any change to the virtual machine* making it easy to understand, maintain and extend. Marea has approximately 2000 lines of code split in 35 classes. The average number of methods per class is 9 while the average lines of code per method is 6 conforming to Smalltalk standards [Klimas 1996]. The 80 unit tests that cover all Marea's use cases are implemented in approximately 1900 lines of code which is almost as long as Marea's implementation. Figure 8.1 shows a simplified UML class diagram of Marea's design. Its key classes are:

- ObjectGraphExporter is the entry point for the final user. It provides methods to swap out graph of objects and relies on Fuel for the serialization. The algorithm is the one explained in Section 4.4.

- ObjectGraphImporter implements the algorithm to swap in graphs described in Section 4.4 and its input is a proxy. It relies on Fuel for the materialization.

[1] http://www.opensource.org/licenses/mit-license.php

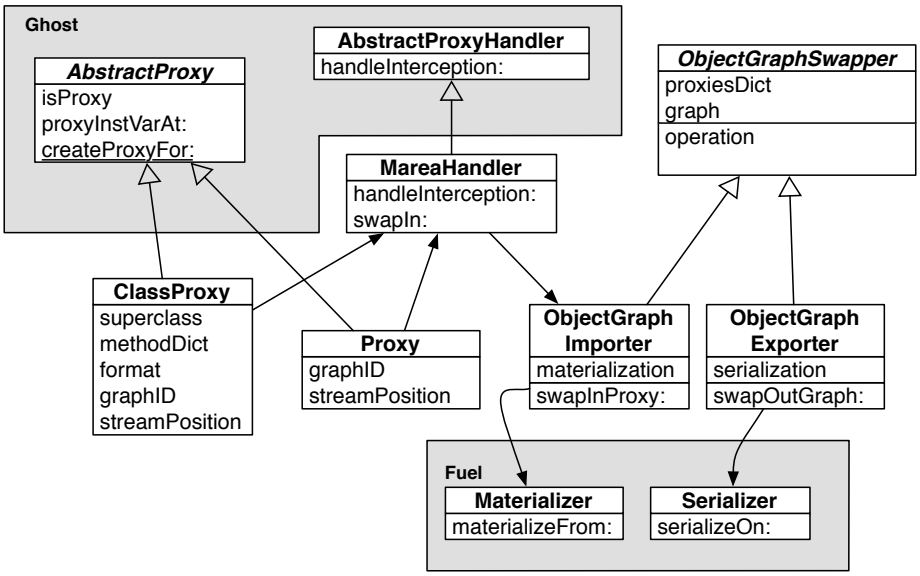


Figure 8.1: Simplified UML class diagram of Marea.

- Proxy and ClassProxy are concrete subclasses of Ghost's AbstractProxy and AbstractClassProxy respectively. Their only purpose is to intercept messages to trigger the swapping in. ClassProxy is needed because we want to proxy classes themselves and the VM expects certain shape for classes. Once a message is intercepted, it is forwarded to its associated handler which is MareaHandler in our case. There are also certain special messages that the proxies implement and answer themselves rather than intercepting *e.g.,* isProxy or proxyInstVarAt:.

- MareaHandler is a concrete subclass of Ghost's AbstractProxyHandler whose main goal is to manage interceptions. An interception is a reification of a message that was sent to a proxy and intercepted. Each interception contains everything needed to swap in the receiver of the intercepted invocation *i.e.,* the proxy which forwarded the interception, the receiver and the arguments. MareaHandler triggers the swap in of the graph associated with the proxy (by delegating to the ObjectGraphImporter and passing the proxy as parameter) that forwarded the interception and then it forwards to the just swapped-in object the message intercepted by the proxy.

### 8.2.2 Requirements from the VM and Language

Although implementing a whole OGS usually requires development on the virtual machine side or even at the OS level, our implementation relies on the default Pharo VM. Nevertheless, Ghost, the proxy library that Marea uses, takes advantage of the following reflective features and hooks provided by the VM and the language.

- *Object replacement:* the primitive become: anotherObject atomically swaps the references of the receiver and the argument. All references in the entire system that

used to point to the receiver now point to the argument and vice versa. There is also becomeForward: anotherObject which is only one way. This feature enables us to replace target objects with proxies and vice versa.

This is the strongest dependency of Marea on the virtual machine. To our knowledge, Smalltalk is the only programming language that provides such a feature at the language side. Nevertheless, we believe that it is not that complex to implement it in different object-oriented languages. The reason is that several of them include a moving garbage collector, and if GC moves objects around, then it needs to update pointers. Such task is the most significant and important part of the become: primitive. Moreover, if the VM is based on an object table instead of on direct pointers, the implementation of the become: is even easier because it just means swapping two pointers.

Since Smalltalk provides become: at the language level and does not keep it hidden in the VM, we were able to implement Marea without any VM changes. In a language with a moving GC (such as the default Java collector) or based on object tables, implementing Marea would require to modify the VM either to provide a become primitive at the language level or to implement the whole Marea in the VM.

- *Objects as methods:* Pharo lets us replace a method in a method dictionary with an object that is not an instance of CompiledMethod. While performing the method lookup, the VM detects that the object in the method dictionary is not a method so it sends the message run: aSelector with: arguments in: aReceiver to that object. Therefore, by handling run:with:in:, Ghost can even intercept method execution.

- *Class with no method dictionary:* the method dictionary is stored as an instance variable of a class, hence it can be changed. When an object receives a message and the VM does the method lookup, if the method dictionary of the receiver class (or of any other class in the hierarchy chain) is nil, then the VM directly sends the message cannotInterpret: aMessage to the receiver. However, the lookup for method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil. This hook allows Ghost to create proxies which intercept almost all possible messages.

### 8.2.3 Marea's Serializer Customization

Marea requires a fast serializer to avoid increasing the overhead in the overall system. Furthermore, the serializer must be able to correctly serialize and materialize any kind of objects such as classes, methods or closures. Finally, it should also be flexible enough to be customized to meet the special needs of Marea. Along this dissertation we have already shown that Fuel solves the first two problems. This section explains how we use Fuel to solve a special need of Marea.

When swapping out, the serializer sends messages to each object of the graph to query its state or hash (a serializer can temporally store each processed object into a hashed

collection). Since a graph may include a proxy, the serializer is likely to trigger the swap in of the graph associated with the proxy.

To avoid this issue, Marea uses the customizability of Fuel to send proxies only a few special messages that they do understand avoiding swapping in. The set of special messages varies according to the kind of proxy. Fuel has been developed to support such customizations. It has the notion of "clusters" which groups objects of the same type and also defines how objects are serialized and materialized. Our customization consists of a special cluster for proxies, meaning that we adapt the serializer to handle proxies in a special manner.

### 8.2.4 Using Low Memory Footprint Proxies

Marea's goal is to reduce the memory footprint of applications. However, Marea itself uses proxies, which also occupy memory. To reduce memory occupied by proxies, we use Ghost.

**Compact Classes.** To decrease memory even more, we take advantage of the Pharo internal object representation: we declare our proxy classes as *compact*. In Pharo, up to 32 classes can be declared as compact classes. In a 32 bits system, compact class instances' object headers are only 4 bytes long instead of the 8 bytes that apply to instances of regular classes. The first word in the header contains flags for the garbage collector, the header type, format, hash, compact class index, etc. The second word is used to store a reference to the class. In compact classes, the reference to the class is encoded in 5 bits in the first word of the header. These 5 bits represent an index in the compact classes' array, internally used by the virtual machine.

**Reducing Instance Variables in Proxies.** Secondly, we encode the proxy instance variables position and graphID in one unique proxyID. The proxyID is a SmallInteger which uses 15 bits for the graphID and 16 bits for the position. Since SmallInteger are immediate objects[2], we do not need an object header for the proxyID. With these optimizations, an instance of Proxy is only 8 bytes (4 for the header and 4 for the proxyID).

By using small integers for proxyIDs, we have a limit of 32767 for the graphID and 65535 for the position. Still, Marea may exceed those limits. Pharo can represent integers with an arbitrary large number of digits. However, instead of using SmallIntegers which are immediate objects, it uses instances of the class LargePositiveInteger which are plain objects and hence occupy more memory. Nevertheless, LargePositiveInteger is a compact class so its instances have a small header.

### 8.2.5 Alternative Object Graph Storage

By default, Marea does the simplest approach when storing graph, *i.e.,* it stores each graph into a separate file in the local file system. This approach may not be the fastest since

[2]In Smalltalk, immediate objects are objects that are encoded in a memory address and do not need an object header. In Pharo, integers are immediate objects.

we are creating and removing files all the time. In addition, there could be lot of small graphs which serialized stream size is smaller than the minimum file size of the file system. Therefore, we end up using much more secondary space than actually needed. Finally, this approach implies that the device where we are running the system has a secondary memory such as a hard disk. This is frequently not the case with mobile devices such as a cellphone or a robot.

Marea OGS delegates the graph storage responsibility to a separate class following the strategy design pattern. There are currently three implementations: simple file based, Riak based and MongoDB based. The user can choose which one to use or even create its own.

One problem that appears when using a database as a backend for storing graphs, is that it adds functionalities that we do not particularly need and that have performance penalties, for example, transactions support, security and validations. In addition, we need to maintain in primary memory all the objects related to the database driver. Some recent NoSQL databases may *not* provide those functionalities avoiding that extra overhead. Moreover, some of these NoSQL databases *e.g.,* CouchDB[3] or Riak[4] expose their interface (API) through the network. That means that we only need a HTTP client library and a small database driver built on top of that.

The way to store data in NoSQL databases is usually following the convention of key/-value, *i.e.,* at a certain key we put certain value. Our solution is to put the graph ID as key and the serialized graph (an array of bytes) in the value. This way it is easy to use the client library to store a BLOB[5] representing our object graph. When swapping in, the proxy has the graph ID so we can get the corresponding array of bytes and materialize it.

This approach can be used for a distributed object graph swapper (OGS). Instead of swapping out object graphs on the same machine, they will be swapped out to a remote server or to the cloud. Note that this adds network communication costs.

## 8.3 Discussion

In this section we start with some specific Marea discussions and then we discuss general issues with an OGS.

### 8.3.1 Infrastructure-Specific Issues

While implementing Marea in Pharo, we encountered some issues that are specific to Pharo but represent recurrent problems that most of the implementations will face. We report them here.

#### 8.3.1.1 Special Objects Are Never Swapped

The swap out algorithm replaces each object of the graph with a proxy. Since the graph is specified by the end programmer, it may contain any type of object *e.g.,* system objects

[3] http://couchdb.apache.org/

[4] http://wiki.basho.com/

[5] BLOB stands for "binary large object", a database type for storing a collection of binary data.

which cannot be replaced without breaking the system. In Pharo, there are three kinds of objects that cannot be replaced: (1) nil, true, false, Smalltalk, Processor, etc[6]. (2) Immediate objects *e.g.,* instances of SmallInteger. (3) Instances of Symbol and Character are also special in Smalltalk because they are created uniquely and shared across the system. In addition, symbols and characters are also used in critical parts of the machinery of method execution.

Marea handles this problem by systematically checking the objects to proxify. Special objects are not proxified and they are handled specifically during materialization. When a graph is rebuilt, Marea inserts references to the special objects that already exist in the system instead of creating duplicates.

#### 8.3.1.2 Proxies and Primitives

Most programming languages, if not all, have methods called primitives implemented in the virtual machine. For example, in Pharo, arithmetic operations, checking whether two references represent the same object or not, file and sockets management, graphics processing, etc., all end up using primitives at some point.

Primitives usually impose a shape in the receiver object or arguments. For example, they expect some objects to be an instance of a certain class or to have a specific format. The problem appears when we replace an object with a proxy which is then passed as an argument to a primitive. In Pharo, most primitives do not crash the VM in that situation but instead they notify their failure which can be managed at language level. By default, a primitive failure is handled by raising a PrimitiveFailed exception.

Marea must be transparent and avoid raising such exceptions. One approach is to not proxify objects which are passed as argument to primitives. Doing an exhaustive analysis of primitives, it is possible to know the classes of the objects passed as arguments. However, this limits the OGS.

We used another solution that captures the PrimitiveFailed exception, gets the original receiver and arguments, swaps in proxies and then re-execute again the same method. This solution is transparent for the user and, if there were proxies, the user will not even be aware.

The detailed steps are:

1. Handle (catch) the PrimitiveFailed.
2. From the exception we can get the signaler method context, from where we can get the sender context which is the context of the method which has the primitive and has failed.
3. From the context that has the primitive method, we can get the receiver and arguments.
4. Check receiver and arguments and swap in graphs in presence of proxies.
5. Re-execute again the original primitive method with the new receiver and arguments (they will have only changed if there were proxies).

[6]These objects are present in the specialObjectsArray and are known and directly used by the VM.

6. Make the original context which has the primitive method to respond the answer from the previous step.

Thanks to Pharo facilities, such solution is approximately 10 lines of code but it relies on the fact that primitives raise exceptions and that we can dynamically access the stack:

```
primitiveFailed
    | primitiveContext method receiver arguments |
    "Just to avoid loops"
    (thisContext sender sender sender method selector = #primitiveFailed)
        ifTrue: [ ^ self primitiveFailed: thisContext sender selector ].
    primitiveContext := thisContext sender.
    method :=  primitiveContext method.
    ((method primitive > 0) or: [method isNamedPrimitive])
        ifFalse: [ ^ self primitiveFailed: thisContext sender selector ].
    receiver := primitiveContext receiver.
    arguments := primitiveContext arguments.
    self unInstallProxiesFrom:  receiver arguments: arguments.
    primitiveContext return: (method valueWithReceiver: receiver arguments: arguments)
```

#### 8.3.1.3 Special Proxies

Some objects can be replaced but only by proxies that respect certain shape. Some object-oriented programming languages like Pharo, represent classes and methods as first-class objects, *i.e.,* they are no more than just instances from other classes known as Metaclass and CompiledMethod respectively. Nevertheless, for example, during method lookup, the VM directly accesses some specific instance variables and imposes a memory layout on the objects representing classes.

Ghost proxies solve this problem by creating special proxies for classes and methods that respect the shape needed by the VM.

#### 8.3.1.4 Object Replacement in Pharo

The object replacement in Pharo is done by using the primitive become:. The problem is that such primitive scans the whole memory to swap all references to the receiver and the argument. Additionally, Pharo provides a "bulk become" that replaces multiple objects at the same time. Marea uses a bulk become to convert all the proxies of a graph. That way, at least we pay the memory traversal only once.

ImageSegment also uses this primitive but in addition it does yet another full memory scan to detect shared objects. Marea algorithms do not require a full memory scan. This problem of the become: primitive is an implementation defect of the current virtual machine. For example, Visual Works Smalltalk does not perform a full memory scan when doing the become:.

While performing the benchmarks described in Section 5.4, we measured the time of the "bulk become" for each graph and we calculated which percentage of the total swapping time it represents. Finally, taking into account this information, we conclude that, in average, this primitive takes about 60% of the total swapping time.

### 8.3.2 Messages That Swap In Lots of Objects

Having classes and methods as first-class objects offers solid reflective capabilities. However, there are large system queries that access *all* classes or methods in the system. When swapping classes and packages, that may cause the swap in of most of the graphs.

#### 8.3.2.1 Examples

During a typical development session using an IDE, when a programmer asks for all the senders of a certain message the system has to check if each compiled method of each class is sending such a message. This means that the system sends messages (sendsSelector: in this case) to the objects that are in the method dictionary causing the swap in if they happen to be proxies. Another example is when a new class is created or removed. The system needs to flush and recreate an internal cache with the class names. To achieve that, it sends the message name to all classes causing the swap in of those which were proxified.

Most of these scenarios happen during application development and not during deployment. As Marea is intended to reduce memory for deployed applications, this is not usually a problem. Still, understanding potential solutions is important. We identified two possible solutions.

#### 8.3.2.2 Explicit Verification

To support such facilities, ImageSegment (an object swapper developed by D. Ingalls for Squeak [Ingalls 1997]) modified a large amount of queries of the base system (Squeak) to explicitly check which objects where in memory and only perform actions on them. The implementation is simple: one method isInMemory defined in Proxy returns false and one method in Object returns true. The two following methods illustrate the use of isInMemory at classes but also at graphical object level.

```
Behavior>>allSubclassesDoGently: aBlock
  "Evaluate the argument for each of the receiver's subclasses."
  self subclassesDoGently:
    [:cl |   cl isInMemory ifTrue: [
         aBlock value: cl.
         cl allSubclassesDoGently: aBlock]]
```

```
Morph>>isFullOnScreen
  "Answer if the receiver is full contained in the owner area."
  owner isInMemory
    ifFalse: [^ true].
  ^ owner clearArea containsRect: self fullBounds
```

In other cases, for certain messages, ImageSegment swaps in the graph, sends the message to the objects that were just swapped in and then it swaps them out again. An example of this is when a class changes its shape and its swapped instances need to be updated.

This type of solution raises the question of the transparency of an OGS.

#### 8.3.2.3 Proxy API Adaption

When the proxy implementation supports it, defining that certain messages are handled by the proxy itself instead of forwarding it to the handler (that will swap in the graph) offers a solution. Often, such possibility is based on the fact that the proxy can play the role of a cache by keeping certain information of the proxified object (as explained later).

Since Ghost provides the mentioned feature, this is the solution chosen by Marea. For example, we use it for the case of class proxies. In Pharo, it is common that the system simply select classes by sending messages such as isBehavior, isClassSide, isInstanceSide, instanceSide or isMeta. Therefore, we defined such methods in ClassProxy to answer appropriate results, *i.e.,* isBehavior and isInstanceSide answers true, isClassSide and isMeta answers false and instanceSide answers self. This way, we avoid swapping in classes. Maybe just after receiving any of those messages, the proxy receives yet another one which causes the swap in. However, sometimes this is not the case. For example, when filtering a collection, we may want to remove all objects which are classes. In this case, even if we send the message isBehavior we are not swapping in all classes just to reject them from a collection.

Apart from classes, Pharo has metaclasses and traits. The metaclass is the class of a class, *i.e.,* the unique instance of a metaclass is a class. A trait is a group of methods that act as a unit of reuse from which classes are composed [Ducasse 2005, Ducasse 2006b, Ducasse 2009]. Therefore, additionally to ClassProxy, Marea provides MetaclassProxy and TraitProxy which also understand some methods to avoid the possible unnecessary swap in. This approach is not limited to them and can also be used for any kind of object. The only requirement is to create a specific proxy class (subclass of AbstractProxy) for that type of object. This allows the developers to adapt Marea to particular use cases and avoid swapping in objects when it is not necessary. Nevertheless, such approach is limited to return simple values and it is complemented by the following one.

#### 8.3.2.4 Using Proxies as Caches

We define that a graph has a good swapping ratio when the amount of objects we swap out is bigger than the amount of proxies that remain in the system (the proxies for facade objects). When the swapped graph offers a good swapping ratio, it may be worth using proxies as caches of certain object properties. This requires more work from the developer but the gain is less swapping in.

When swapping out a class, the swapping ratio makes such scenarios effective. Section 5.4 shows that during our experiments a typical object graph for a class included between 15% and 17% of facade objects. That means that for those graphs we can release between 85% and 83% of its objects. Moreover, for each graph there is only *one* instance of ClassProxy and the rest are normal proxies. Hence, ClassProxy can store some specific data and still obtain a memory-efficient solution. For example, ClassProxyWithName is a class proxy that stores its proxified class name and that has the accessor method. With this, we avoid swapping in all classes when the system flushes the caches of class names.

Another possible improvement that we did not implement because of the lack of time is to create a special proxy for methods that stores the literals of the proxified method. Even

if the gain will not be so significant, the proxy is still smaller than the proxified methods. By storing literals, a method proxy is able to answer messages requiring this data avoiding the problem of swapping in all compiled methods when doing certain operations such as searching for senders of a message.

These alternative proxies have a tradeoff between the memory they occupy and the messages they are able to correctly answer without swapping in their target. It is up to the user of Marea to decide which strategy to use.

## 8.4 Conclusion

In this chapter we look into the details of Marea's implementation and design. We also presented some discussions regarding our implementation and issues that other implementations will face. We demonstrated why we claim that Marea is fully implemented on the language side and without modifications of the virtual machine or the OS.

After having explained Marea's subsystems and its implementation details, the next chapter proposes some visualizations that take into account objects and memory usage. Such visualizations can help the user to improve Marea's performance.

CHAPTER 9

# Visualizing Objects Usage

## Contents



## At a Glance


This chapter presents some visualizations that take into account objects and memory usage in object-oriented applications. For this, we use Distribution Map which is a visualization showing spread and focus of properties across systems. We can visualize which part of the system is used and which is not, apart from measuring the number of objects and their size in memory.



**Keywords:** Software visualization, unused objects, runtime information, memory analysis.

## 9.1 Introduction

In Chapter 2 we showed that the fact of having several unused yet referenced objects makes applications to use more memory than needed. With our modified Pharo VM we traced objects usage. The results we collected indicated that the analyzed applications only use between 27% and 37% of the memory they occupy.

Having the percentage of unused objects is not enough. There are several cases where we want to know *which* objects are actually being used or unused. For example, in the context of Marea, such analysis can improve its performance by helping Marea users choose better candidates to swap. There is a plethora of work on runtime information [De Pauw 1994, Stasko 1998, Jerding 1997, Reiss 2003] or class instantiation visualizations [De Pauw 1993, Ducasse 2004] but none of them show whether instances are actually used. This information is important to identify unused objects.

In this chapter, we present some visualizations that show used and unused objects in object-oriented applications. For this, we use Distribution Map which is a visualization showing spread and focus of properties across systems. We propose a solution that applies Distribution Maps to represent the way classes are used or not and also the amount of instances and the memory they occupy. The Moose platform for software and data analysis [Nierstrasz 2005] supports building Distribution Maps. Since Pharo considers packages, classes and methods as objects, the mechanism to track unused objects is the same and we can analyze any objects including classes, packages and methods.

#### Structure of the Chapter

The remainder of the chapter is structured as follows: The next section describes the basis of Distribution Map. In Section 9.3 we show different Distribution Maps representing used and unused objects at different levels like packages, classes and methods. Section 9.4 explains how these visualizations helps improving Marea's performance. Finally, related work is presented in Section 9.5, before concluding in Section 9.6.

## 9.2 Visualizing Results with Distribution Maps

In this section, we present some visualizations that show used/unused objects in object-oriented applications. We use Distribution Maps to represent classes and we visually distinguish a class that has instances from one that has *used* instances.

Since Moose is also implemented on top of Pharo and it supports Distribution Maps, we used it with our modified Pharo VM able to trace objects usage.

### 9.2.1 Distribution Maps in a Nutshell

Distribution Map enables the user to visualize parts of a system while focusing on some properties. It provides a quick overview of the distribution of these properties within a system, and it presents complex information in an intuitive way. This visualization is meant to fill the gap between the raw results obtained by automated algorithms and the following

analysis carried out by a human expert. For example, Distribution Map was used to show how semantics information or authors spread over classes and packages [Ducasse 2006a].

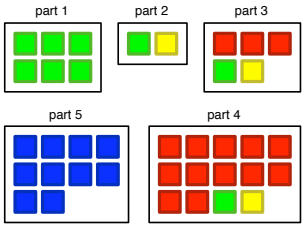


Figure 9.1: A Distribution Map showing five parts and four properties.

Figure 9.1 illustrates an example of a Distribution Map with five parts containing 6, 2, 5, 10 and 14 elements respectively and with four properties. The colors are just a way to represent properties. On the visualization, for each part $p_n$ there is a large rectangle and within that rectangle, for each element $s_i \in p_n$ there is a small square whose color refers to the property $q_m$ attributed to that element.

From the visualization we can characterize both the parts with respect to the contained properties, and the properties with respect to their distribution over the parts. In our example from Figure 9.1, about the properties we say that Blue is *well-encapsulated*, that Yellow is *cross-cutting*. We can also say that part 1 and part 4 are *self-contained*.

Creating a Distribution Map needs at least, three elements:

1. A list of containers (parts), while each has a list of elements.
2. List of elements.
3. List of properties. These are applied to each element.

For example, a list of containers can be a list of packages. Elements are the classes of each package and a property can be whether a class has used instances or not. In another example, a container can be a class, the elements its methods and the property whether they were used or not.

It is also important to take into account that there can be multiple properties. For example, in the context of Marea, we can define the following properties: classes that have instances; classes that have instances where, at least, one instance is really used; classes that do not have instances at all; abstract classes; etc.

## 9.3 Unused Object Maps

The idea of software visualization is helping us to analyze software, to understand the results of an analysis and to answer questions. In our context, it is important to know which objects and classes are unused because they are good candidates to swap out.

We want that our visualizations help Marea users answer, for example, the following questions: Which classes are not used by the application? Among classes with several

instances which ones have most of their instances unused? Which classes or packages are not used at all? Which classes and instances are unused while also occupying significant amount of memory?

All the answers to these questions help Marea's users increase their performance by choosing good swapping candidates. With Distribution Maps we can answer them through meaningful visualizations.

### 9.3.1 Simple example

The most common and basic question is *"Which classes were unused while executing X?"* (where X is an arbitrary sequence of one or more expressions). Since message sending is a typical expression in OO languages, we can then derive the following question: *"Which are the classes that have used instances as a consequence of sending a single message to a single object?"*. As an example we consider sending the single message size to the string 'thisIsATest'. This idea can be generalized to any code in any application. To run the example, we evaluate the following in a workspace of the Pharo environment:

```
UnusedObjectDiscoverer current startDiscovery.
'thisIsATest' size.
UnusedObjectDiscoverer current stopDiscovery.
```

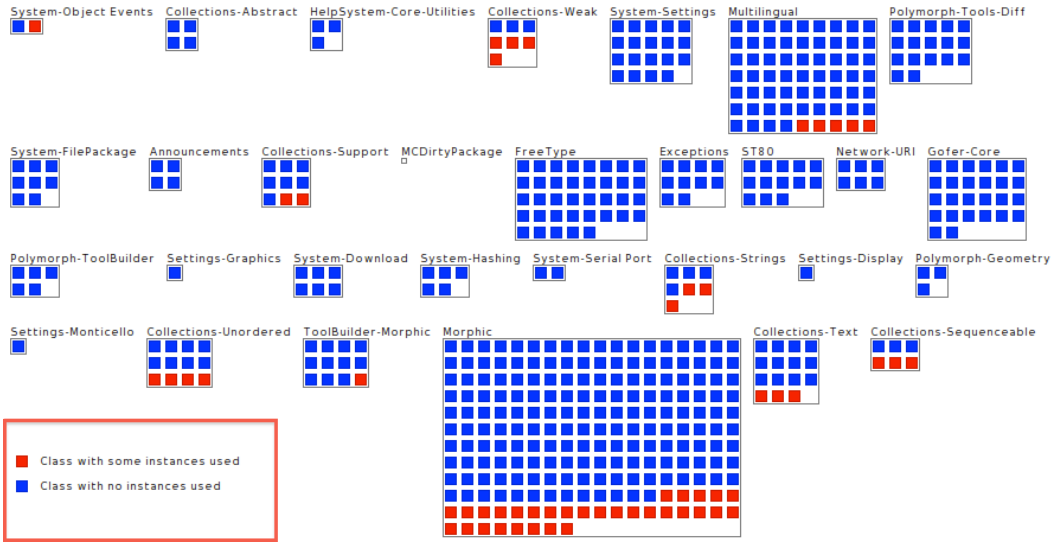


Figure 9.2: Classes which have some instances that have been used during the execution of a single message send.

The result of what has been used during this message send is shown in Figure 9.2 as a Distribution Map of packages and their classes. This visualization deserves some explanations:

- Containers are visualized as big squares and, in this example, they represent packages.

- Elements are represented as small colored squares inside a container. In Figure 9.2, each element is a class which belongs to a package.
- The properties are displayed as the colors of the elements. In this example, all instances of classes in blue are unused, while classes in red have at least one instance used during the execution of the message.
- Moose provides us with a facility to quickly obtain the name of an element in a distribution map. When the mouse is over an element, its name is shown. In our case, a tooltip shows the name of the selected class.

Figure 9.2 only shows some packages of Pharo. On the contrary, Figure 9.3 shows all packages of Pharo[1].

Notice that a Distribution Map is even useful to:

- Easily detect large packages which can indicate that refactoring is needed (probably split the package into smaller ones).
- Compare different Distribution Maps of different scenarios to detect which packages and classes are used. Suppose we want to run a Seaside web application [Ducasse 2010], we can start our application, do the analysis and then compare it with the previous example.
- Visualize the usage of a certain package. Once again, this feature helps us to detect possible refactorings.

Sometimes analyzing at package-class level is not enough and we need to go deeper. For example, we need to know *"Which methods of the class X were actually unused?"* Figure 9.4 shows a subset of a Distribution Map where each container is a class (from the 'Kernel' package) and each element is a method. Similarly to the distribution map for packages shown above, the visualization provided here is the result of the analysis during the evaluation of the message size sent to a string.

From now onward, we will show only part of the Distribution Maps since they are large. For example, there are more classes in the "Kernel" package than what Figure 9.4 shows. Fortunately, the API of DistributionMap is flexible enough to let us filter the containers or elements to narrow the view. For example, we are able to only show packages with more than four elements, or show only elements that satisfy certain conditions.

In addition, analyzing only if instances of a class were used or not is not enough. It is important, for example, to distinguish between concrete and abstract classes[2]. Figure 9.5 shows a Distribution Map with more properties like "Class with used instances", "Class with instances", "Class without instances", "Abstract used class" and "Abstract unused class".

[1]This distribution map is applied on a PharoCore distribution, which is a special runtime for production purposes which does not include developer packages. Standard Pharo has more packages.

[2]In Smalltalk an abstract class is that one that has at least one method implemented as self subclassResponsibility.

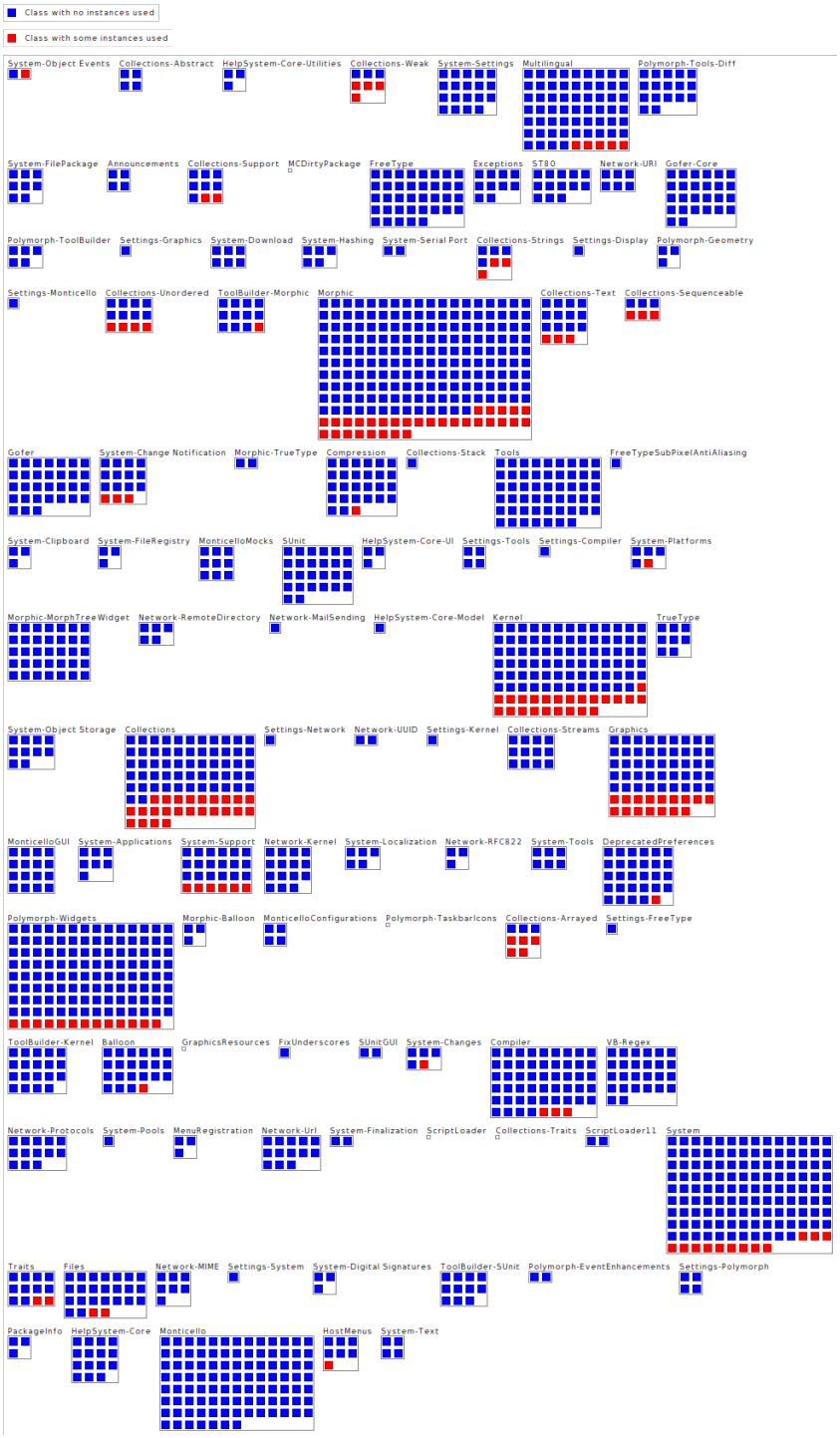


Figure 9.3: All classes with used instances during the execution of a single message send.

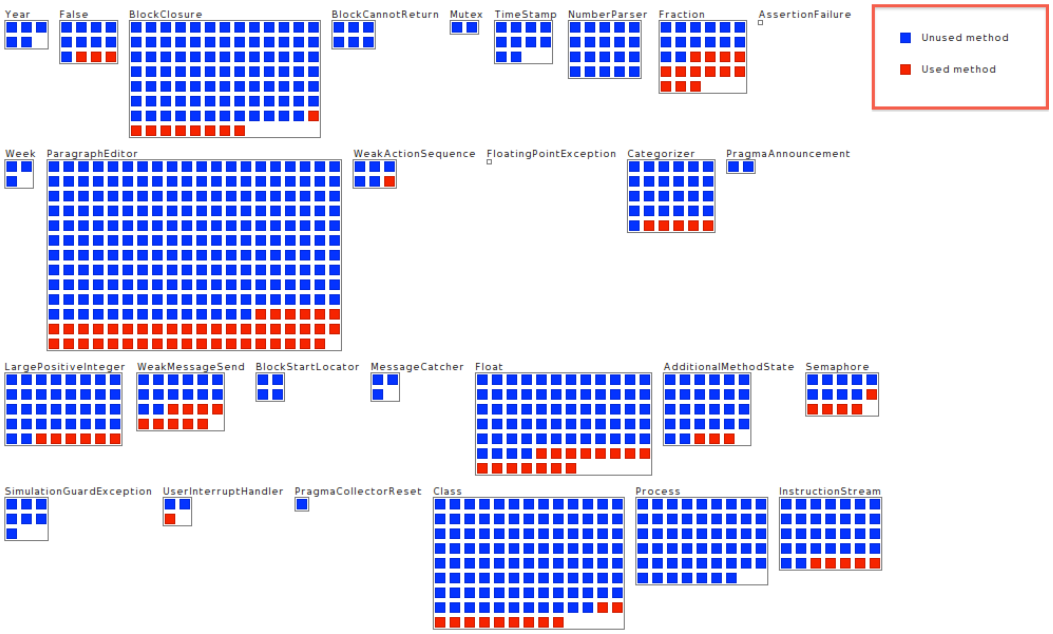


Figure 9.4: Used methods during the execution of a single message send.

### 9.3.2 Amount of Instances and Memory Usage

Probably, one needs to go beyond just knowing if a class has used instances or not. For example, one may need to know how many used instances each class has. It is not the same if a class has forty two used instances or a thousand. Figure 9.6 shows an example.

Notice that in these Distribution Maps the properties legend (at the bottom of the figure) is sorted decrementally (highest to lowest) by the number of occurrences of each property. In our example, the first one is "Class without instances" which is the property that has more occurrences. Figure 9.6 shows that the Blue boxes are the majority. On the other hand, the property "Class with used instances between 1001 and 10000" is the one that has the least amount of occurrences. This makes sense since only few classes have so many used instances.

Figure 9.6 depicts that the classes with the biggest amount of used instances are ByteSymbol, ByteString, Float, CompiledMethod, and Array. Then it follows with Association, MethodDictionary, Metaclass, Point and Rectangle.

Another important property to analyze is the memory occupied by objects. It is not the same having four instances of one MB each, than having a thousand instances of twenty bytes each. Figure 9.7 shows an example with the memory occupied by used instances. The classes that have used instances that occupy most memory are ByteSymbol, ByteString, Bitmap, CompiledMethod, MethodDictionary, and Array.

It is interesting to compare this to the previous example. The class Bitmap does not even appear in the previous example (Figure 9.6 ) as one of the classes with more used instances. However, in Figure 9.7, it appears as one the classes with used instances that occupy most of the memory. What does it mean? It means that there are not too many used instances of

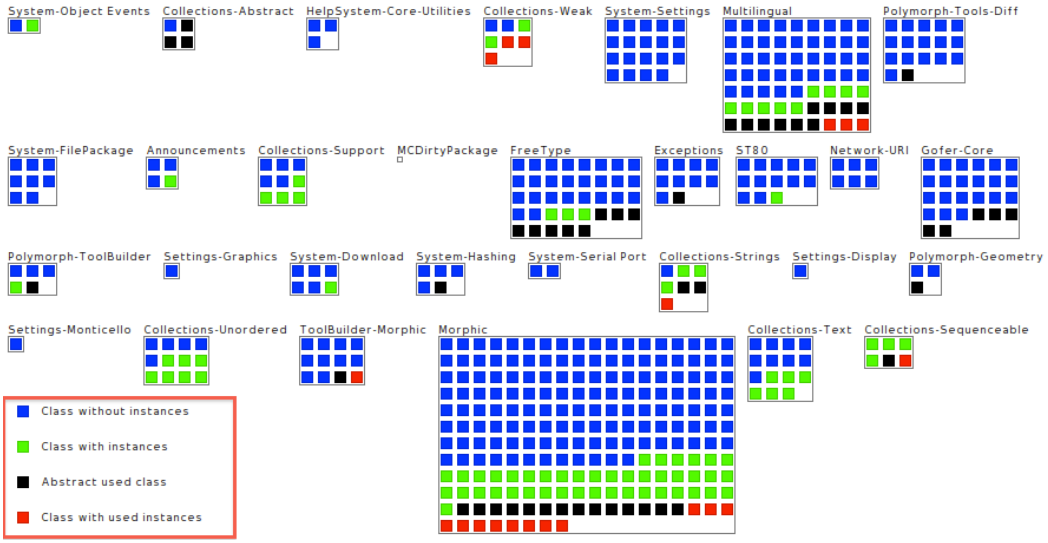


Figure 9.5: A more accurate view on live classes during the execution of a single message send.

Bitmap but each instance occupies much more memory than the average.

But why there are used Bitmap instances? This is because to start, run and stop de analysis, we evaluate such code in a Workspace. And this Workspace is part of the Smalltalk IDE. Therefore, even when the minimal code is executed inside the Smalltalk IDE, Bitmap instances are used.

## 9.4 Improving Marea's Performance

The visualizations presented in this chapter can be used whenever a developer want to analyze which parts of the system is being used in a specific scenario or to know which parts use most of the memory.

In Marea, the challenge is how can we help the developer to choose good candidates? Objects that have been unused during the execution of the application are good candidates to swap out. With our visualizations, the user can run the application and known which classes, methods and packages have been used. In addition, we also present the memory they occupy.

## 9.5 Related Work

There is a plethora of work on runtime information [De Pauw 1994, Stasko 1998, Jerding 1997, Reiss 2003, Lange 1995b, Systä 1999] or class instantiation visualizations [De Pauw 1993, Ducasse 2004] but *none* of them show whether instances are actually used or not. To the best of our knowledge we are the first ones to treat such information and present it to engineers. This information is important to identify unused objects and, in


Figure 9.6: Live classes grouped by their used instances count.

particular, memory leaks [Bond 2008]. In our context, Marea users can use it as a tool to choose good candidates to swap to secondary memory.

Jerding et al. proposed visualization about system execution. The challenge is to display the execution trace of the system and allow users to navigate through it. For this purpose they propose the "mural technique" which acts as a zoom out on a large amount of information [Jerding 1997, Jerding 1996]. They identify message patterns over a large amount of interaction traces.

Win de Pauw et al. present how class communicates and their instance creation behavior [De Pauw 1993]. Ducasse et al. use polymetric views [Lanza 2003] to show the behavior of instances creation [Ducasse 2004].

In the same area of Lange's work [Lange 1995a,Lange 1995b] regarding the interactive understanding of design pattern execution, there are several works such as the one of Systa, Richner, or Greevy [Systä 1999,Richner 1999,Greevy 2005] which mix dynamic with static information.

Robert Walker et al [Walker 1998] present a tools to navigate dynamic information offline at architectural level. This approach complements and extends existing profiling and visualization approaches. However, they do not mention any point on used objects.

## 9.6 Conclusion

We did not find any tool capable of analyzing the objects usage and visualizing its results with meaningful visualizations in the context of object-oriented applications. To achieve this, we proposed a solution that applies Distribution Maps to take into account instance usage and to distinguish between used and unused instances.

Analyzing and visualizing the amount of instances of a class, of used and unused instances, of methods used per class, of classes used per package, etc, are also excellent tools to improve and refactor the system. For example, they indicate that some classes cannot

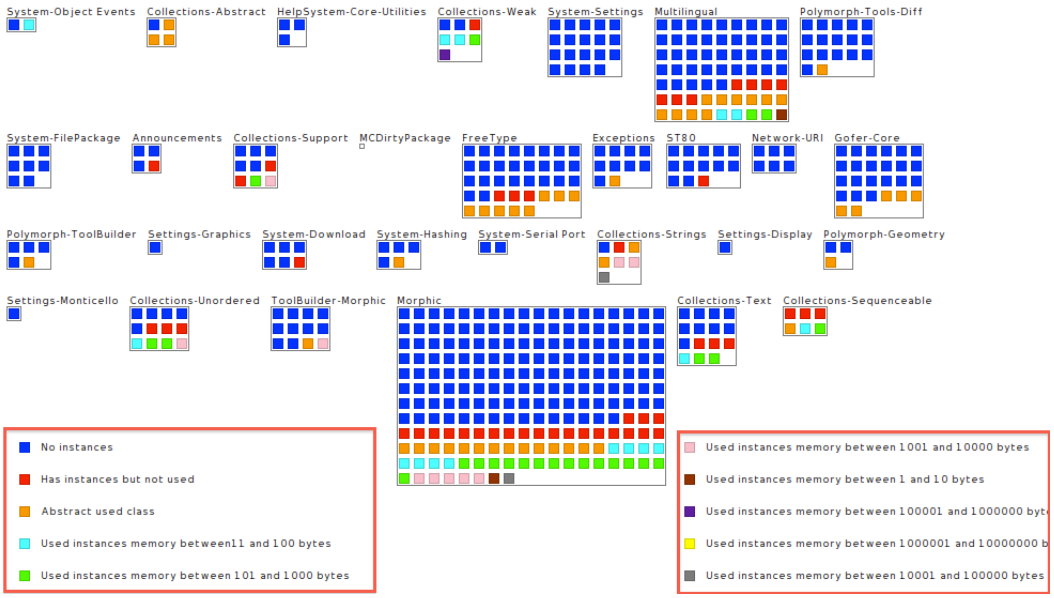


Figure 9.7: Distribution Map with the memory occupied by used instances.

easily be removed from the system, that some classes must be split (create subclasses), that a package must be divided into less coupled smaller ones, etc.

In addition, they help us understand which objects, methods, classes and packages are used in different scenarios. For example, when trying to build a minimal kernel system where we need to know which is the minimal set of objects or code to include in such kernel.

The next chapter concludes the dissertation by summarizing the ideas proposed in Marea. Then we discuss the possible future work and open issues.

CHAPTER 10

# Conclusion

**Contents**



## At a Glance

This chapter provides an overall summary. We briefly review the contributions of this dissertation and discuss possible future work.

## 10.1 Summary

The study conducted in this dissertation starts with the observation that most applications end up using more memory than what is actually needed by the processing they perform. One way of dealing with this problem is by swapping data from primary memory to secondary memory. Traditionally, this is achieved with a virtual memory provided by the host operating system. However, OS' virtual memory manager acts blindly since it only knows about memory pages or similar structures. It has no knowledge of the application nor its domain. Therefore, it cannot guess which data is the most appropriate to swap. Moreover, in the presence of a garbage collector like in most modern object-oriented languages, performance of the virtual memory can significantly degrade. Indeed, the GC scans all application's objects even if they have been swapped out. This results in memory page faulting and can lead to the so called memory trashing where the virtual memory manager dramatically degrades the system's performance by transferring data back and forth between primary and secondary memory.

We have conducted a study of this problem in the context of dynamic object-oriented programming languages. After analyzing memory usage of 4 real scale systems, we arrived to the following requirements for a virtual memory manager. It must: (i) provide an object-based smart swapping unit which generates the less object faulting possible and that, when swapping in, brings the minimum number of unnecessary objects; (ii) be uniform by being able to swap any kind of object; (iii) swap and not remove; (iv) automatically swap in; (v) automatically swapping out; (vi) be transparent by providing the same API for the programming language whether the virtual memory is being used or not; (vii) allow application programmers to influence what, when and how to swap (viii) be decoupled from the OS and the virtual machine.

There are various approaches related to our research. We classified them in different groups: (i) virtual memory and garbage collected languages; (ii) reduced language runtimes; (iii) object faulting, orthogonal persistence and object databases; (iv) memory leaks and (v) general-purpose object graph swappers. We have evaluated each work with the defined list of requirements. Although they exhibit interesting properties, none of them matches our requirements listed above. Therefore, we introduced Marea, an *application-level object-oriented virtual memory*.

We describe how our solution addresses the shortcomings we found in existing approaches while also explaining how Marea fulfills the stablished requirements. Being an application-level virtual memory, Marea allows developers to influence the virtual memory management. Contrary to traditional virtual memory managers where the swapping unit is a page or an individual object, Marea considers objects graphs as the swapping unit. Swapped graphs are swapped in as soon as one of their elements is needed. The graphs to be swapped can have *any* shape and contain *almost any* kind of object. This includes classes, methods and closures which are all first-class objects in Marea's implementation language.

When Marea swaps a graph, it correctly handles all the references from outside and inside the graph. When one of the swapped objects is needed, its graph is *automatically* brought back into primary memory. To achieve this, Marea replaces original objects with

proxies. Whenever a proxy intercepts a message, it loads back the swapped graph from secondary memory. This process is completely transparent for the developer and the application. Any interaction with a swapped graph has the same results as if it was never swapped.

Marea decomposes into four main subsystems: (1) object graph swapper, (2) advanced proxy toolbox, (3) serializer and (4) object storage. In this dissertation, we describe each of these parts from the conceptual level. We also provide a detailed description of their implementations on top of the Pharo programming language. These implementations have been used in substantial experiments that allowed us to empirically validate our proposal. These experiments complement our theoretical validation where we show how Marea matches our requirements for an application-level object-oriented virtual memory.

## 10.2 Contributions

Our findings have implications for the research in this field. As our proposed approach has demonstrated, it allows application programmers to control what, when and how to swap. This opens up many new possibilities as we can take domain knowledge into account for the decision of what to swap. Since we are at the object level, much more sophisticated strategies can be developed than the simple LRU replacement available with OS virtual memory.

Marea relies on Ghost, our new generic and lightweight proxy implementation. Ghost provides uniform proxies not only for regular instances, but also for classes and methods. Ghost optionally supports object replacement. In addition, Ghost proxies have a small memory footprint and their overhead is low. Proxies are powerful, easy to use and extend.

We have also proposed 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. Instead of being done in a recursive way which is what most serializers do, this is done in an iterative way. 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. Fuel can serialize any kind of object. We demonstrate that it is possible to build a fast serializer without specific VM support with a clean object-oriented design while providing the most possible required features for a serializer.

Both, Fuel and Ghost are *general-purpose* frameworks that can be used outside Marea. Their validations show that they have outstanding performance and can be used in real-world applications. For example, Fuel has been used for persistency purposes, bioinformatics, importing and exporting of code packages, kernel systems constructions while also being ported to the Newspeak programming language.

Our implementation of Marea demonstrates that we can bring a fast, object-based virtual memory to the application-level without modifying the VM yet with a clean object-oriented design. It also allows us to validate empirically our solution by experimenting Marea with different real-world applications. We have focused on measuring the efficiency

and usefulness of it. Our benchmarks demonstrate that the memory footprint of different applications can be reduced from 25% to 40%. However, there is still a room of improvement since the unused objects memory for the tested applications was from 63% to 73%. With certain knowledge in the domain and, by choosing better candidates to swap, the released memory can be even bigger. For this reason, we have proposed some visualizations that take into account objects and memory usage and allow Marea users to choose better candidates.

## 10.3 Future Work

There are a few open questions that were not discussed, but should also be explored in future work.

### 10.3.1 Automatic Graphs Detection

A natural follow up to this work is to automate the process of swapping out graphs. The challenge is to *automatically* detect graphs of *unused yet referenced objects* which are good candidates to be swapped out. First and foremost, we need to answer the question: how to identify unused objects? After using an object (*e.g.,* message reception), how long should the system wait before considering it as unused? Besides, different criteria should be considered to select graphs to swap out such as: the graph's size, the percentage of unused objects or the percentage of objects shared with other graphs.

Another interesting direction to explore is the relationship between application-level virtual memory and garbage collection. Could they take advantage one of the other? Can they reuse the same memory traversal? Can they share the information or flags stored in object headers?

### 10.3.2 Improved Algorithms

With the current implementation, when a proxy intercepts a message, Marea swaps in the whole swapped out graph. Meaning that the whole graph is materialized and all proxies are replaced by the just materialized objects. It can happen that the graph we are swapping in is quite large and maybe just the object associated to the proxy is needed. Or maybe the system needs that object and a few others reachable from it. In addition, in Marea the size and structure of the graphs that are chosen to swap out directly impact on performance (speed and memory saved). Therefore, swapping in to primary memory the whole graph may not be the best approach in this particular scenario.

In the future, we plan to investigate what we call *partial swapping*. When a proxy receives a message, instead of swapping in the whole graph, we only swap in the transitive subgraph taking the object associated to the proxy as the root. Another possibility is to do what some database drivers do *i.e.,* swap in the graph at a certain level of depth and replace the rest with proxies. This is known as *lazy loading*. In either case, these improvements would make graph selection less important because they will have less impact on performance. However, it can happen also that soon after partially swapping in, another part of

the original complete graph is needed. Hence, we need to do another swap in for another subgraph. Swapping in several times (one per subgraph) is slower than swapping in the whole graph only once. Taking both scenarios into account we plan to investigate in which cases it is worth swapping in a subgraph instead of the whole graph.

### 10.3.3 Marea for Other Use Cases

In the case of Marea, we want to swap in the graph when a proxy intercepts a message, but there can be other scenarios where something different is desired. For example, imagine a graph that was swapped out because the hardware where the system is running has less memory than what is required. In this scenario, we do not want to swap in. If the graph was migrated to another system running with more hardware (or imagine the cloud), we may be able to forward the messages to remote objects running in the other system. Or imagine a system where a graph is swapped out because of security reasons and it needs that if the graph is intended to be swapped in, the system throws an error.

We want to analyze whether Marea would be useful for those scenarios. The first step in the exploration of these scenarios is to study which specific proxy handlers might be required.

### 10.3.4 Thread Safety

One of the current limitations of Marea is when a graph changes while being swapped out. All our algorithms to swap and the serialization do not work properly when the graph changes in the middle of the operation. This is because Marea is implemented in language side and therefore it is not atomic.

In the future, we plan to investigate the following approaches:

- Run Marea's in a process with high priority. This is just a workaround and does not solve the problem. It just reduces the chances to alter the graph while being swapped.
- We can protect all mutations of objects in the graph with a Mutex/Monitor whose critical section covers the serialization *i.e.,* blocking all other processes that wants to mutate objects in the graph until serialization and swapping is complete. This could be implemented but it may need adapting application code. In either case, another way to address the problem is to adapt the VM to support immutability. The virtual machine of Newspeak [Bracha 2010a] (which is a fork of the current Pharo VM) supports that so we can experiment with it.

### 10.3.5 Transparency

Another interesting topic that we did not yet fully solved is the transparency for the system. In Pharo, for example, each class can answer to the message allInstances. What should that answer when we also have instances on secondary memory? On the one hand, we believe that such message should answer only the objects in memory and then add another message that answers both. On the other hand, this makes the system inconsistent. For example, if a class which is in memory changes (like adding, removing or renaming instance

variables), its instances need to be updated. Since this functionality relies on the message allInstances, we are not updating the instances that are in secondary memory. Of course there are workaround like swapping in all instances, update them and then swap them out or check if they need to be updated when swapping.

Usually, these kind of messages that needs all instances, all subclasses or similar are used during the *development* of an application. Once an application is *deployed*, there is less probability to need this type of messages. Marea is mostly intended for deployed applications rather than for developing.

Nevertheless, we still want to analyze how the system should be designed to cope with the fact that the working set of objects is not all in primary memory.

APPENDIX A

# Getting Started

Contents



This appendix gives instructions on how to install and use the Marea system.

## A.1 Installation

There are two possibilities to install Marea. One is downloading an already prepared environment and the other is to install Marea in a standard Pharo distribution.

### A.1.1 Downloading a one-click image

1. Download the one-click Marea distribution from https://gforge.inria.fr/frs/download.php/31256/Pharo-1.4-Marea.zip.
2. Launch the executable of your platform:
   - Mac: Marea.app
   - Linux: Marea.app/Marea.sh
   - Windows: Marea.app/Marea.exe

### A.1.2 Building a custom image

To install Marea in a Pharo environment:

1. Get a Pharo image from http://www.pharo-project.org/. The tested versions of Pharo with Marea are 1.4 and 2.0.
2. Open a Workspace and evaluate the following code:

```
Gofer it
url: 'http://ss3.gemstone.com/ss/Marea';
package: 'ConfigurationOfMarea';
load.

(Smalltalk at: #ConfigurationOfMarea) project stableVersion load.
```

   This will install Marea and all its dependencies such as Ghost and Fuel.

## A.2 Examples

In this section we present some examples to show how to use Marea.

### A.2.1 Swapping Regular Objects

The API of Marea is very straightforward and minimalistic. For example, the following line swaps out a regular graph:

```
| today yesterday |
today := Date today.
MARObjectGraphExporter new swapOutGraph: today.
yesterday := Date yesterday.
MARObjectGraphExporter new swapOutGraph: yesterday.
```

The default persistency strategy of Marea stores each swapped graph in a separate file located in a directory called *swapSpace*. Such directory is created in the directory where the Pharo image is. For example, if the Pharo image is in */Users/mariano/marea/marea.image* then the directory is */Users/mariano/marea/swapSpace*.

The name of the file created for each graph is its graph ID and the file extension is *.swap*. In our example, today is written into a file named *1.swap*. The next swapped graph yesterday in *2.swap* and so on.

Since the swapping in is automatic, a developer does not have to use it explicitly. As soon as one of the proxies of the graph is used, the graph is swapped in. In our example, if we do:

```
today asSeconds.
```

The proxy intercepts the message asSeconds and the graph is swapped in and answers the value of today as seconds.

The object passed to swapOutGraph: can be any type of object. So for example we can also do:

```
MARObjectGraphExporter new swapOutGraph: Date allInstances.
```

In this case we swap out all instances of Date. We can also do:

```
MARObjectGraphExporter new swapOutGraph: (
    Array
        with: Date allInstances
        with: DateAndTime allInstances
        with: (TextStyle named: 'Bitmap DejaVu Sans')
        with: UITheme currentSettings ).
```

Here we swap out all instances of Date and DateAndTime, the fonts named 'Bitmap DejaVu Sans' and the current settings of the UI theme.

We could continue giving more examples, but we hope it is clear that the method swapOutGraph: provides the swapping mechanism for almost any type of object graph. In the next section we show how to swap code.

### A.2.2 Swapping Classes

Marea also allows the developer to swap classes. To accomplish that, Marea's API provides some helper methods such as swapOutClass:, swapOutClassAndSubclasses:, swapOutClassWithInstances: and swapOutClassWithSubclassesWithInstances:. Examples:

```
"Here we just swap the class DateAndTime"
MARObjectGraphExporter new swapOutClass: DateAndTime.
"Here we swap the class DateAndTime and all its instances"
MARObjectGraphExporter new swapOutClassWithInstances: DateAndTime.
"Here we swap the class DateAndTime and its subclasses (TimeStamp)"
MARObjectGraphExporter new swapOutClassAndSubclasses: DateAndTime.
"Here we swap the class DateAndTime, TimeStamp and all their instances"
MARObjectGraphExporter new swapOutClassWithSubclassesWithInstances: DateAndTime.
```

We can also, for instance, swap out some development tools:

```
classesToSwap := { TestRunnerBrowser. SystemBrowser. FlatMessageListBrowser. HierarchyBrowser.
    ClassListBrowser. ChangedMessageSet. MessageNames. ProtocolBrowser. RecentMessageSet. ChangeList.
    VersionsBrowser. ChangeSorter. ChangeSetBrowser. DualChangeSorter. TimeProfiler. TimeProfileBrowser.
    PointerExplorer. ViewHierarchyExplorer. ObjectExplorerWrapper. ViewHierarchyExplorer.
    FileContentsBrowser. FileList. Inspector. ProcessBrowser. Debugger. PreDebugWindow. TimeProfileBrowser.
    TimeProfiler. Finder. FinderUI. FinderNode. }.

classesToSwap do: [:each | MARObjectGraphExporter new swapOutClassWithSubclassesWithInstances: each].
```

Or we can swap all UI themes:

```
classesToSwap := { BlueUITheme. UIThemeStandardSqueak. UIThemeVistary. UIThemeW2K.
    GLMOrangeUITheme. VistaryThemeIcons. WateryThemeIcons. }.
classesToSwap do: [:each | MARObjectGraphExporter new swapOutClassWithSubclassesWithInstances: each].
```

There is also a helper method to swap out traits (groups of methods that act as a unit of reuse from which classes are composed):

```
MARObjectGraphExporter new swapOutTrait: TAssertable.
```

Finally, we can also swap a whole package, which means swapping all its classes:

```
MARObjectGraphExporter new swapOutPackage: (PackageInfo named: 'AST').
```

### A.2.3 Using Different Storage Engines

By default, Marea has a persistency strategy based on files in the local file system. To install persistency strategies based on No-SQL databases one has explicitly tell the installer to do so:

```
(Smalltalk at: #ConfigurationOfMarea) project stableVersion load: 'MareaRiak'.
```

or:

```
(Smalltalk at: #ConfigurationOfMarea) project stableVersion load: 'MareaMongoDB'.
```

That installs the Marea backend for those databases and also the database driver for them. Once we have install them we can set them as the storage engine strategy by doing:

```
MARObjectGraphSwapper graphPersistencyStrategy: MARMongoDBGraphPersistency.
```

or:

```
MARObjectGraphSwapper graphPersistencyStrategy: MARRiakDBGraphPersistency.
```

We can also change the configuration of those databases. For example, for Riak, the default connection is on *http://localhost:8098/riak*. In MongoDB it is *(Mongo host: 'localhost' port: 27017) open*. To change it:

```
MARRiakDBGraphPersistency databaseConnection: 'http://somedomain:PPPP/riak'.
MARMongoDBGraphPersistency databaseConnection: (Mongo host: 'somedomain' port: PPPP) open.
```

### A.2.4 Rest of the API

**Resetting.** Marea provides a method to reset all its internal configuration and data. This leaves the environment as clean as it was before starting to do anything with Marea. It is useful for testing. The method MARObjectGraphSwapper resetAll resets all the internal tables mentioned in Chapter 4 such as the GraphTable and SharedProxiesTable.

```
MARObjectGraphSwapper resetAll.
```

It also resets the counter to get the graph ID, swaps in all swapped out graphs and removes all stored graphs remaining in the storage engine. The methods used internally by resetAll can also be used directly by the user. Examples:

```
MARObjectGraphSwapper uninstallAllProxies.
MARObjectGraphSwapper resetGlobalTable.
MARObjectGraphSwapper resetSharedProxiesTable.
MARObjectGraphSwapper resetGraphID.
MARObjectGraphSwapper resetGraphStorage.
```

**Cleaning.** Marea could clean the system after every swap in. However, to avoid adding extra overhead, we let the user decide when to clean Marea. By cleaning Marea we mean to compact the tables GraphTable and SharedProxiesTable to release even more memory.

```
MARObjectGraphSwapper clean.
```

**Logging.** The user of Marea can decide whether to log or not and also *where* to log. If log is enable, Marea logs information when a graph is swapped out and when it is swapped in. In the latter it also logs the sequence of messages (stack trace) that triggered the swap in. Hence, the logging adds a significant overhead to Marea's performance. Because of that, we disable the logging by default for general-purpose usage and the user has to enable it when needed:

```
MARLogger createAndSetLogger.
```

That method creates, by default, a file *MareaLog.txt* which is placed in the image directory. In addition, the user can choose to log into the Pharo Transcript instead of files or to directly disable again the log:

```
MARLogger changeToFileLogger.
MARLogger changeToTranscriptLogger.
MARLogger disableLogging.
```

# Bibliography


[Alpert 1998] Sherman R. Alpert, Kyle Brown and Bobby Woolf. The design patterns Smalltalk companion. Addison Wesley, Boston, MA, USA, 1998. 75

[Arnaud 2010] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel and Mathieu Suen. *Read-Only Execution for Dynamic Languages*. In Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS'10), Malaga, Spain, June 2010. 90

[Atkinson 1995] Malcolm Atkinson and Ronald Morrison. *Orthogonally Persistent Object Systems*. VLDB JOURNAL, vol. 4, pages 319–401, 1995. 26

[Atkinson 1996] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis and S. Spence. *An orthogonally persistent Java*. SIGMOD Rec., vol. 25, no. 4, pages 68–75, 1996. 26

[Barrett 2003] Tom Barrett. *Dynamic Proxies in Java and .NET*. Dr. Dobb's Journal of Software Tools, vol. 28, no. 7, pages 18, 20, 22, 24, 26, July 2003. 120

[Bennett 1987] John K. Bennett. *The Design and Implementation of Distributed Smalltalk*. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA '87, pages 318–330, New York, NY, USA, 1987. ACM. 60, 90

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009. 5, 6, 34, 62, 90, 94, 97, 124

[Bodden 2011] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati and Mira Mezini. *Taming Reflection*. In International Conference on Software Engineering, 2011. 4, 25

[Bond 2008] Michael D. Bond and Kathryn S. McKinley. *Tolerating memory leaks*. In Gail E. Harris, editeur, OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, October 19-23, 2008, Nashville, TN, USA, pages 109–126. ACM, 2008. 2, 27, 60, 61, 143

[Bracha 2004] Gilad Bracha and David Ungar. *Mirrors: design principles for meta-level facilities of object-oriented programming languages*. In Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices, pages 331–344, New York, NY, USA, 2004. ACM Press. 92

[Bracha 2010a] Gilad Bracha. *Newspeak Programming Language Draft Specification Version 0.06*, 2010. 149

[Bracha 2010b] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox and Eliot Miranda. *Modules as Objects in Newspeak*. In Theo D'Hondt, editeur, Proceedings of the 24th European conference on Object-oriented programming, ECOOP'10, pages 405–428, Berlin, Heidelberg, 2010. Springer-Verlag. 86

[Brant 1998] John Brant, Brian Foote, Ralph Johnson and Don Roberts. *Wrappers to the Rescue*. In Proceedings European Conference on Object Oriented Programming (ECOOP'98), volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. 90, 116

[Breg 2001] Fabian Breg and Constantine D. Polychronopoulos. *Java virtual machine support for object serialization*. In Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande, JGI '01, pages 173–180, New York, NY, USA, 2001. ACM. 61

[Butterworth 1991] Paul Butterworth, Allen Otis and Jacob Stein. *The GemStone object database management system*. Commun. ACM, vol. 34, no. 10, pages 64–77, 1991. 26, 60

[Carr 1981] Richard W. Carr and John L. Hennessy. *WSCLOCK a simple and effective algorithm for virtual memory management*. In Proceedings of the eighth ACM symposium on Operating systems principles, SOSP '81, pages 87–95, New York, NY, USA, 1981. ACM. 3, 15, 21

[Cheriton 1988] David Cheriton. *The V distributed system*. Commun. ACM, vol. 31, no. 3, pages 314–333, March 1988. 19

[Chu 1972] Wesley W. Chu and Holger Opderbeck. *The page fault frequency replacement algorithm*. In Proceedings of the December 5-7, 1972, fall joint computer conference, part I, AFIPS '72 (Fall, part I), pages 597–609, New York, NY, USA, 1972. ACM. 3, 15, 21

[Courbot 2005] Alexandre Courbot, Re Courbot, Jean-Jacques Vandewalle, Gilles Grimaud and Jean jacques V. *Romization: Early Deployment and Customization of Java Systems for Constrained Devices*. In In Proceedings of Second International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, volume 3956 of *Lecture Notes in Computer Science*, pages 57–76. Springer, 2005. 24

[Courbot 2010] Alexandre Courbot, Gilles Grimaud and Jean-Jacques Vandewalle. *Efficient off-board deployment and customization of virtual machine-based embedded systems*. ACM Transaction on Embedded Computer Systems, vol. 9, pages 21:1–21:53, mar 2010. 24

[De Pauw 1993] Wim De Pauw, Richard Helm, Doug Kimelman and John Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), pages 326–337, October 1993. 136, 142, 143

[De Pauw 1994] Wim De Pauw, Doug Kimelman and John Vlissides. *Modeling Object-Oriented Program Execution*. In M. Tokoro and R. Pareschi, editeurs, Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), volume 821 of *LNCS*, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag. 136, 142

[Decouchant 1986] Dominique Decouchant. *Design of a Distributed Object Manager for the Smalltalk-80 System*. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPLSA '86, pages 444–452, New York, NY, USA, 1986. ACM. 25, 60

[Denning 1970] Peter J. Denning. *Virtual Memory*. ACM Comput. Surv., vol. 2, no. 3, pages 153–189, September 1970. 2, 3

[Denning 1980] Peter Denning. *Working Sets Past and Present*. IEEE Transactions on Software Engineering, vol. SE-6, no. 1, pages 64–84, January 1980. 3, 21

[Ducasse 1999] Stéphane Ducasse. *Evaluating Message Passing Control Techniques in Smalltalk*. Journal of Object-Oriented Programming (JOOP), vol. 12, no. 6, pages 39–44, June 1999. 90, 92, 94, 95

[Ducasse 2004] Stéphane Ducasse, Michele Lanza and Roland Bertuli. *High-Level Polymetric Views of Condensed Run-Time Information*. In Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press. 136, 142, 143

[Ducasse 2005] Stéphane Ducasse, Nathanael Schärli and Roel Wuyts. *Uniform and Safe Metaclass Composition*. Journal of Computer Languages, Systems and Structures, vol. 31, no. 3-4, pages 143–164, December 2005. 132

[Ducasse 2006a] Stéphane Ducasse, Tudor Gîrba and Adrian Kuhn. *Distribution Map*. In Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06), pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society. 137

[Ducasse 2006b] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts and Andrew P. Black. *Traits: A Mechanism for fine-grained Reuse*. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28, no. 2, pages 331–388, March 2006. 132

[Ducasse 2009] Stéphane Ducasse, Marcus Denker and Adrian Lienhard. *Evolving a Reflective Language*. In Proceedings of the International Workshop on Smalltalk Technologies (IWST 2009), pages 82–86, Brest, France, aug 2009. ACM. 132

[Ducasse 2010] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone and Michael Davies. Dynamic web development with seaside. Square Bracket Associates, 2010. 12, 86, 139

[Engler 1995a] Dawson R. Engler, M. Frans Kaashoek and James O'Tool. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. In SOSP, pages 251–266, 1995. 3, 19, 20

[Engler 1995b] D.R. Engler, S.K. Gupta and M.F. Kaashoek. *AVM: Application-Level Virtual Memory*. In Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on, pages 72 –77, may 1995. 3, 19

[Eugster 2006] Patrick Eugster. *Uniform proxies for Java*. In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM. 90, 119

[Fernandes 2007] Hilaire Fernandes, Stéphane Ducasse and Thibault Caron. *Dr Geo II: Adding Interactivity Planes in Interactive Dynamic Geometry*. In Proceedings of 5th International Conference on Creating, Connecting and Collaborating through Computing (C5 2007), pages 153–162. IEEE Computer Society, 2007. 13

[Gamma 1993] Erich Gamma, Richard Helm, John Vlissides and Ralph E. Johnson. *Design Patterns: Abstraction and Reuse of Object-Oriented Design*. In Oscar Nierstrasz, editeur, Proceedings ECOOP '93, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag. 5, 25, 34, 35, 36, 90

[Greevy 2005] Orla Greevy and Stéphane Ducasse. *Correlating Features and Code Using A Compact Two-Sided Trace Analysis Approach*. In Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society. 143

[Hassoun 2005] Youssef Hassoun, Roger Johnson and Steve Counsell. *Applications of dynamic proxies in distributed environments*. Software Practice and Experience, vol. 35, no. 1, pages 75–99, January 2005. 90

[Hertz 2005] Matthew Hertz, Yi Feng and Emery D. Berger. *Garbage collection without paging*. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM. 18, 22

[Hosking 1990] Antony L. Hosking, J. E Moss and Cynthia Bliss. *Design of an Object Faulting Persistent Smalltalk*. Rapport technique, University of Massachusetts, Amherst, MA, USA, 1990. 21, 42

[Hosking 1993] Antony L. Hosking and J. Eliot B. Moss. *Object Fault Handling for Persistent Programming Languages: A Performance Evaluation*. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, pages 288–303, October 1993. 25, 26

[Hosking 1999] Antony L. Hosking and Jiawan Chen. *PM3: An Orthogonal Persistent Systems Programming Language - Design, Implementation, Performance*. In Proceedings of the 25th International Conference on Very Large Data Bases, VLDB '99, pages 587–598, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. 26

[Ingalls 1997] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay. *Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself*. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), pages 318–326. ACM Press, November 1997. 28, 131

[Jerding 1996] Dean F. Jerding and John T. Stasko. *The Information Mural: Increasing Information Bandwidth in Visualizations*. Rapport technique GIT-GVU-96-25, Georgia Institute of Technology, October 1996. 143

[Jerding 1997] Dean J. Jerding, John T. Stasko and Thomas Ball. *Visualizing Interactions in Program Executions*. In Proceedings of International Conference on Software Engineering (ICSE'97), pages 360–370, 1997. 136, 142, 143

[Kaehler 1986] Ted Kaehler. *Virtual Memory on a Narrow Machine for an Object-Oriented Language*. Proceedings OOPSLA '86, ACM SIGPLAN Notices, vol. 21, no. 11, pages 87–106, November 1986. 2, 4, 21, 60, 61

[Kemper 1995] Alfons Kemper and Donald Kossmann. *Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis*. The VLDB Journal, vol. 4, no. 3, pages 519–567, July 1995. 26

[Kiczales 1991] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. The art of the metaobject protocol. MIT Press, 1991. 90

[Kiczales 1994] Gregor Kiczales and Andreas Paepcke. *Open Implementations and Metaobject Protocols*. Expanded tutorial notes, 1994. At http://db.stanford.edu/ paepcke/shared-documents/Tutorial.ps. 6

[Kiczales 1996] Gregor Kiczales. *Beyond the Black Box: Open Implementation*. IEEE Software, January 1996. 6

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming*. In Mehmet Aksit and Satoshi Matsuoka, editeurs, Proceedings ECOOP '97, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. 90

[Klimas 1996] Edward J. Klimas, Suzanne Skublics and David A. Thomas. Smalltalk with style. Prentice-Hall, 1996. 124

[Koster 2000] Rainer Koster and Thorsten Kramp. *Loadable Smart Proxies and Native-Code Shipping for CORBA*. In Claudia Linnhoff-Popien and Heinz-Gerd Hegering, editeurs, Trends in Distributed Systems: Towards a Universal Service Market, Third International IFIP/GI Working Conference, USM 2000, Munich, Germany, September 12-14, 2000, Proceedings, volume 1890 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 2000. 90

[Krueger 1993] Keith Krueger, David Loftesness, Amin Vahdat and Thomas Anderson. *Tools for the Development of Application-Specific Virtual Memory Management*. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, pages 48–64, October 1993. 3, 19

[Lange 1995a] Danny Lange and Yuichi Nakamura. *Interactive Visualization of Design Patterns can help in Framework Understanding*. In Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95), pages 342–357, New York NY, 1995. ACM Press. 143

[Lange 1995b] D.B. Lange and Y. Nakamura. *Program Explorer: A Program Visualizer for C++*. In Proceedings of Usenix Conference on Object-Oriented Technologies, pages 39–54, 1995. 142, 143

[Lanza 2003] Michele Lanza and Stéphane Ducasse. *Polymetric Views—A Lightweight Visual Approach to Reverse Engineering*. Transactions on Software Engineering (TSE), vol. 29, no. 9, pages 782–795, September 2003. 143

[Levy 1982] H. Levy and P. H. Lipman. *Virtual Memory Management in the VAX/VMS Operating System*. IEEE Computer, vol. 16, no. 3, page 35, March 1982. 3, 21

[Lipton 1999] Paul Lipton. *Java Proxies for Database Objects*. http://www.drdobbs.com/windows/184410934, 1999. 90

[Marquez 2000] Alonso Marquez, Stephen M Blackburn, Gavin Mercer and John Zigman. *Implementing Orthogonally Persistent Java*. In In Proceedings of the Workshop on Persistent Object Systems (POS, pages 218–232, 2000. 26

[McCarthy 1960] John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I*. CACM, vol. 3, no. 4, pages 184–195, April 1960. 2, 11

[McCullough 1987] Paul L. McCullough. *Transparent Forwarding: First Steps*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 331–341, December 1987. 90, 95

[Miranda 2005] Eliot Miranda, David Leibs and Roel Wuyts. *Parcels: a Fast and Feature-Rich Binary Deployment Technology*. Journal of Computer Languages, Systems and Structures, vol. 31, no. 3-4, pages 165–182, May 2005. 60, 61, 62, 87

[Moss 1990] J. Eliot B. Moss. *Design of the Mneme persistent object store*. ACM Trans. Inf. Syst., vol. 8, no. 2, pages 103–139, April 1990. 26

[Moss 1992] J. Eliot B. Moss. *Working with Persistent Objects: To Swizzle or Not to Swizzle*. IEEE Transactions on Software Engineering, vol. SE-18, no. 8, pages 657–673, August 1992. 26, 27

[Nierstrasz 2005] Oscar Nierstrasz, Stéphane Ducasse and Tudor Gîrba. *The Story of Moose: an Agile Reengineering Environment*. In Michel Wermelinger and Harald Gall, editeurs, Proceedings of the European Software Engineering Conference (ESEC/FSE'05), pages 1–10, New York NY, 2005. ACM Press. Invited paper. 13, 85, 136

[Pascoe 1986] Geoffrey A. Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80*. In Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, pages 341–346, November 1986. 90, 95

[Pratikakis 2004] Polyvios Pratikakis, Jaime Spacco and Michael Hicks. *Transparent proxies for java futures*. In OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, pages 206–223, New York, NY, USA, 2004. ACM Press. 90

[Reiss 2003] Steven P. Reiss. *Visualizing Java in Action*. In Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization), pages 57–66, 2003. 136, 142

[Renggli 2007] Lukas Renggli. *Pier — The Meta-Described Content Management System*. European Smalltalk User Group Innovation Technology Award, August 2007. Won the 3rd prize. 86

[Richner 1999] Tamar Richner and Stéphane Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. In Hongji Yang and Lee White, editeurs, Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press. 143

[Riggs 1996] Roger Riggs, Jim Waldo, Ann Wollrath and Krishna Bharat. *Pickling State in the Java System*. Computing Systems, vol. 9, no. 4, pages 291–312, 1996. 66, 87

[Santos 2002] Nuno Santos, Paulo Marques and Luis Silva. *A Framework for Smart Proxies and Interceptors in RMI*, 2002. 90

[Shapiro 1986] Marc Shapiro. *Structure and Encapsulation in Distributed Systems: The Proxy Principle*. In Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS), pages 198–205, Washington, DC, 1986. IEEE Computer Society. 90

[Stamos 1982] James William Stamos. *A Large Object-Oriented Virtual Memory: Grouping Strategies, Measurements, and Performance*. Technical Report SCG-82-2, Xerox Palo Alto Research Center, Palo Alto, California, may 1982. 3, 21

[Stamos 1984] James W. Stamos. *Static grouping of small objects to enhance performance of a paged virtual memory*. ACM Trans. Comput. Syst., vol. 2, no. 2, pages 155–180, May 1984. 21

[Stasko 1998] John T. Stasko, John Domingue, Marc H. Brown and Blaine A. Price. Software visualization — programming as a multimedia experience. The MIT Press, 1998. 136, 142

[Systä 1999] Tarja Systä. *On the relationships between static and dynamic models in reverse engineering Java software*. In Working Conference on Reverse Engineering (WCRE99), pages 304–313, October 1999. 142, 143

[Ungar 1984] Dave Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm*. ACM SIGPLAN Notices, vol. 19, no. 5, pages 157–167, 1984. 21

[Ungar 1995] David Ungar. *Annotating Objects for Transport to Other Worlds*. In Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications, OOPSLA '95, pages 73–87, New York, NY, USA, 1995. ACM. 44, 61

[Van Cutsem 2010] Tom Van Cutsem and Mark S. Miller. *Proxies: design principles for robust object-oriented intercession APIs*. In Dynamic Language Symposium, volume 45, pages 59–72. ACM, oct 2010. 90, 92, 96, 118

[Walker 1998] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson and Jeremy Isaak. *Visualizing Dynamic Software System Information through*

*High-Level Models*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98), pages 271–283. ACM, October 1998. 143

[Wang 2001] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt and Ossama Othman. *The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware*. In Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01), pages 103–118, Berkeley, California, 2001. USENIX Association. 90

[Welch 1999] Ian Welch and Robert Stroud. *Dalang - A Reflective Extension for Java*, February 1999. 90

[Wiebe 1986] Douglas Wiebe. *A Distributed Repository for Immutable Persistent Objects*. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPLSA '86, pages 453–465, New York, NY, USA, 1986. ACM. 60

[Williams 1987] Ifor Williams, Mario Wolczko and Trevor Hopkins. *Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy*. In J. Bézivin, J-M. Hullot, P. Cointe and H. Lieberman, editeurs, Proceedings ECOOP '87, volume 276 of *LNCS*, pages 79–88, Paris, France, June 1987. Springer-Verlag. 3

[Wilson 1991] Paul R. Wilson. *Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware*. SIGARCH Comput. Archit. News, vol. 19, no. 4, pages 6–13, July 1991. 26, 27

[Yang 2006] Ting Yang, Emery D. Berger, Scott F. Kaplan, J. Eliot and B. Moss. *Cramm: Virtual memory support for garbage-collected applications*. In In USENIX Symposium on Operating Systems Design and Implementation, pages 103–116, 2006. 18, 22

[Yokote 1992] Yasuhiko Yokote. *The Apertos Reflective Operating System: The Concept and its Implementation*. In Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, pages 414–434, October 1992. 19

[Yokote 1993] Yasuhiko Yokote. *Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach*. In Object Technologies for Advanced Software, First JSSST International Symposium, volume 742 of *Lecture Notes in Computer Science*, pages 145–162. Springer-Verlag, November 1993. 19

[Young 1987] Robert L. Young. *An Object-Oriented Framework for Interactive Data Graphics*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 78–90, December 1987. 19

# List of Publications


[Dias 2011] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse and Gabriela Arévalo. *Clustered Serialization with Fuel*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011), Edinburgh, Scotland, 2011. 6, 36, 124

[Dias 2012] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse and Gabriela Arévalo. *Fuel: A Fast General Purpose Object Graph Serializer*. Software: Practice and Experience, 2012. 6, 36, 124

[Martinez Peck 2010a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Experiments with a Fast Object Swapper*. In Smalltalks 2010, Concepción del Uruguay, Argentina, 2010. 28, 30, 60, 82

[Martinez Peck 2010b] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Visualizing Objects and Memory Usage*. In Smalltalks 2010, Concepción del Uruguay, Argentina, 2010. 7, 55, 6

[Martinez Peck 2011a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Efficient Proxies in Smalltalk*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011), Edinburgh, Scotland, 2011. 6, 35, 124

[Martinez Peck 2011b] 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. 6, 90

[Martinez Peck 2011c] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse and Luc Fabresse. *Object Swapping Challenges: an Evaluation of ImageSegment*. Journal of Computer Languages, Systems and Structures, vol. 38, no. 1, pages 1–15, nov 2011. 90

[Martinez Peck 2012a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Object-Based Virtual Memory Brought To The Application Level (submitted + passed first review round)*. Journal of Object Technology, November 2012. 5, 6, 34

[Martinez Peck 2012b] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse and Marcus Denker. *Ghost: A Uniform and General-Purpose Proxy Implementation (submitted + passed first review round)*. Journal of Science of Computer Programming (SCP), October 2012. 6, 35, 124

## Application-Level Virtual Memory for Object-Oriented Systems

During the execution of object-oriented applications, several millions of objects are created, used and then collected if they are not referenced. Problems appear when objects are unused but cannot be garbage-collected because they are still referenced from other objects. This is an issue because those objects waste primary memory and applications use more primary memory than what they actually need. We claim that relying on operating systems (OS) virtual memory is not always enough since it is completely transparent to applications. The OS cannot take into account the domain and structure of applications. At the same time, applications have no easy way to control nor influence memory management.

In this dissertation, we present Marea, an efficient application-level virtual memory for object-oriented programming languages. Its main goal is to offer the programmer a novel solution to handle application-level memory. Developers can instruct our system to release primary memory by swapping out *unused yet referenced objects* to secondary memory.

Marea is designed to: 1) save as much memory as possible *i.e.,* the memory used by its infrastructure is minimal compared to the amount of memory released by swapping out unused objects, 2) minimize the runtime overhead *i.e.,* the swapping process is fast enough to avoid slowing down primary computations of applications, and 3) allow the programmer to control or influence the objects to swap.

Besides describing the model and the algorithms behind Marea, we also present our implementation in the Pharo programming language. Our approach has been qualitatively and quantitatively validated. Our experiments and benchmarks on real-world applications show that Marea can reduce the memory footprint between 25% and 40%.

**Keywords:**

Virtual memory; object swapping; object faulting; unused objects; serialization; proxies

## Mémoire virtuelle au niveau applicatif pour les systèmes orientés objet

Lors de l'exécution des applications à base d'objets, plusieurs millions d'objets peuvent être créés, utilisés et enfin détruits s'ils ne sont plus référencés. Néanmoins, des dysfonctionnements peuvent apparaître, quand des objets qui ne sont plus utilisés ne peuvent être détruits car ils sont référencés. De tels objets gaspillent la mémoire principale et les applications utilisent donc davantage de mémoire que ce qui est effectivement requis. Nous affirmons que l'utilisation du gestionnaire de mémoire virtuel du système d'exploitation ne convient pas toujours, car ce dernier est totalement isolé des applications. Le système d'exploitation ne peut pas prendre en compte ni le domaine ni la structure des applications. De plus, les applications n'ont aucun moyen de contrôler ou influencer la gestion de la mémoire virtuelle.

Dans cette thèse, nous présentons Marea, un gestionnaire de mémoire virtuelle piloté par les applications à base d'objets. Il constitue une solution originale qui permet aux développeurs de gérer la mémoire virtuelle au niveau applicatif. Les développeurs d'une application peuvent ordonner à notre système de libérer la mémoire principale en transférant les *objets inutilisés, mais encore référencés* vers une mémoire secondaire (telle qu'un disque dur).

En plus de la description du modèle et des algorithmes sous-jacents à Marea, nous présentons notre implémentation dans le langage Pharo. Notre approche a été validée à la fois qualitativement et quantitativement. Ainsi, nous avons réalisés des expérimentations et des mesures sur des applications grandeur-nature pour montrer que Marea peut réduire l'empreinte mémoire de 25% et jusqu'à 40%.

**Mots clés:**

Mémoire virtuelle; sérialisation; proxies; objets inutilisés; programmation orientée objet