HAL
open science

Visualizing Objects and Memory Usage

Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, Luc Fabresse

To cite this version:

Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, Luc Fabresse. Visualizing Objects and Memory Usage. Smalltalks'2010, Nov 2010, Buenos Ares, Argentina. inria-00531510

HAL Id: inria-00531510

https://inria.hal.science/inria-00531510v1

Submitted on 2 Nov 2010

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

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

Visualizing Objects and Memory Usage

Mariano Martinez Peck[1,2,*], Noury Bouraqadi[2], Marcus Denker[1], Stéphane Ducasse[1], Luc Fabresse[2]

Abstract

Most of the current garbage collector implementations work by reachability. This means they only take care of the objects that nobody else points to. As a consequence, there are objects which are not really used but are not garbage collected because they are still referenced. Such unused but reachable objects create *memory leaks*. This is a problem because applications use much more memory than what is actually needed. In addition, they may get slower and crash. It is important to understand which parts of the system are instantiated but also which are *used* or *unused*. There is a plethora of work on runtime information or class instantiation visualizations but none of them show whether instances are actually used. Such information is important to identify memory leaks.

In this paper, we present some visualizations that show used/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 extend Distribution Maps to represent the way classes are used or not, since we distinguish between a class that just has instances from one that has *used* instances. To identify unused objects, we modified the Pharo Virtual Machine.

Keywords: Memory leaks, program visualization, runtime information, Smalltalk, Object-oriented programming

1. Introduction

In object-oriented programming languages like Smalltalk – and Java to a certain extent–, everything is an object. Objects are allocated and they occupy a certain amount of memory. An application is a graph of interacting objects. During program execution, a large graph of live objects is built, changed and reconfigured. In this graph,

☆This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

*Corresponding author

Email addresses: marianopeck@gmail.com (Mariano Martinez Peck), noury.bouraqadi (Noury Bouraqadi), marcus.denker@inria.fr (Marcus Denker), stephane.ducasse@inria.fr (Stéphane Ducasse), luc.fabresse@mines-douai.fr (Luc Fabresse)

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1.

[2]Université Lille Nord de France, Ecole des Mines de Douai.

objects point to other objects. To support automatic memory management, most object oriented runtimes are based on garbage collectors (GC) [Jon96]. The idea behind garbage collection is the automatic destruction of unreferenced objects. The GC collects objects that are not being referenced anymore. This means that garbage collectors work by reachability.

During a typical run of an application, several millions of objects are created, used and then collected when not referenced. Now, the problem appears when there are objects which are not used but are not garbage-collected because they are still reachable (*i.e.,* are referenced by other objects). Some are used just once, only in certain situations or conditions or they are not used since a long period of time but, in all cases, they are kept in memory. These objects can create memory leaks [BM08].

This is a problem because, in presence of memory leaks, applications use much more memory than what is actually needed. Unused objects lead to slower systems and can even be the cause of severe system crashes. There is a definitive need to identify unused objects. In class-based object-oriented languages, information about class usage is needed.

There is a plethora of work on runtime information [DPKV94, SDBP98, JSB97, Rei03] or class instantiation visualizations [DPHKV93, DLB04] but none of them show whether instances are actually used. This information is important to identify unused objects and, in particular, memory leaks.

In this paper, we present some visualizations that show used/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 extend Distribution Maps to represent the way classes are used or not, since we distinguish between a class that just has instances from one that has *used* instances.

Since Smalltalk 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.

To identify unused objects, we modified the Pharo [BDN+09] Virtual Machine. Distribution Maps is part of the suite Moose, a platform for software and data analysis [NDG05]. Moose is also implemented on top of Pharo Smalltalk and, therefore, we use our modified Virtual Machine to run it.

The contribution of this paper is to provide a different perspective on the problem of used objects and, in particular, to support the perception of potential memory leaks. Distribution Maps' technique provides a quick look at the distribution of properties within a system and it reveals complex information in an intuitive way. The use of Distribution Map fills the gap between the raw results obtained by automated algorithms and the following analysis carried out by a human expert.

The remainder of the paper is structured as follows: Section 2 presents the problem of detecting used and unused objects. Section 3 explains the basis of Distribution Map. In Section 4 we show different Distribution Maps for used and unused objects at different levels like packages, classes and methods. Finally, related work is presented is described in in Section 5, before concluding in Section 6.

2. Detecting used and unused objects

We first need to clarify what we understand by *used* object. A used object is one that receives a message during a specific period of time. Our analysis consists of these steps:

Start analysis. Mark all objects as unused.

Analysis phase. At this step, we enable the tracing and run all the scenarios. We run the code that we want to analyze. During this execution, each object that receives 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.

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.

Our solution modifies the Pharo VM so that we can use an empty 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 modify 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 in the language side to call the primitives and to take statistics about packages, classes and methods usage.

3. Distribution Maps in a Nutshell

Distribution Map visualizes parts of a system and shows how properties of the part elements spread over the initial parts. For example, Distribution Map was used to show how semantics information or authors spread over classes and packages [DGK06]. In our case, a property is for example, if a given package or class has instances, if they are use or not *used* instances, etc.

A Distribution Map is a map showing related information in a spatial space: distance quantifications such as far away or near each other have a meaningful interpretation. Related parts and properties are placed near each other, while unrelated parts or properties are placed far apart. Distribution Maps' technique provides a quick overview of the distribution of properties within a system, and it presents complex information in an intuitive way. The Distribution Map is meant to fill the gap between the raw results obtained by automated algorithms and the following analysis carried out by a human expert.

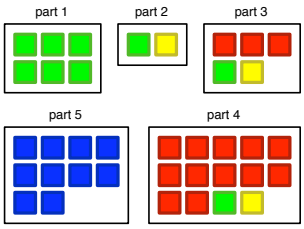

Figure 1: A Distribution Map showing five packages and four properties: Red, Blue, Green and Yellow.

Figure 1 illustrates an example of a Distribution Map with five parts containing 6, 2, 5, 10 and 14 elements respectively and with four properties: Red, Blue, Green and Yellow. 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 1, about the properties we say that Blue is *well-encapsulated*, that Yellow is *cross-cutting* and that Green is like an *octopus* because it has a body and tentacles spread over the system. We can also say that part 1 and part 4 are *self-contained*.

3.1. Basis of a Distribution Map

For creating a Distribution Map you need at least, three elements:

1. A list of containers, 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 you can have multiple properties and not just one. For example, you can have 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.

4. Used Object Maps

The idea of software visualization is helping us to analyze software, to understand the results of an analysis, to answer questions, etc. In our context , it is interesting to know which classes are used for the minimal code execution. This is important for

building minimal systems and for knowing which are the classes that should be part of that base system. Which classes does the system use more? Which classes have several instances and most of them are used? Which classes or packages are not used at all? Which classes have instances but are not generally used?

All these questions are needed when trying to build a base and modular system. With Distribution Maps we can answer them through meaningful visualizations.

4.1. Simple example

The most common and basic question is *"Which classes were used while executing X?"* The second most common questions is *"What happens when X is just one single message sent?"* This means *"Which are the classes that have used instances as a consequence of sending a simple message to a single object?"*.

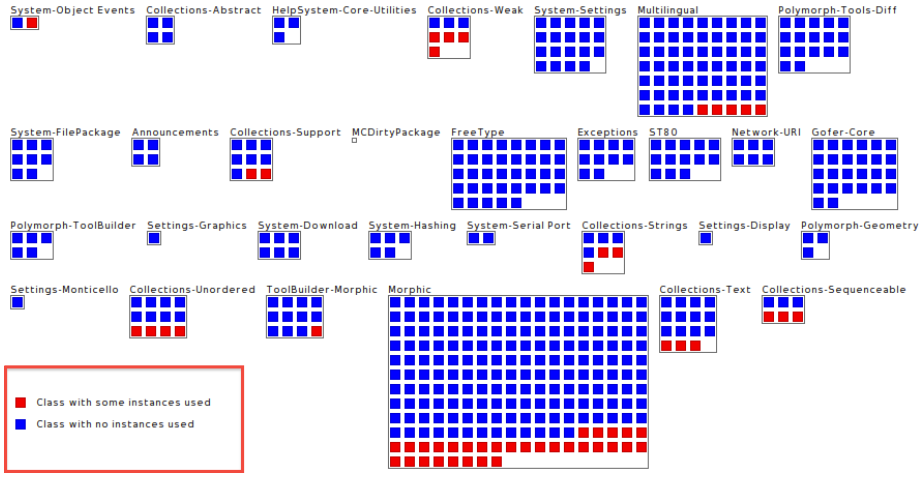

Figure 2: Some classes whose some instances have been used during the execution of a simple message sending

Figure 2 shows an example of a Distribution Map that answers this question. This visualization requires some explanations:

- Containers are visualized as big squares and, in this example, they represent packages (Note that Figure 2 only shows some packages, not all of them).
- Elements are represented with small colored squares inside a container. In Figure 2, each element is a class and it belongs to a container.
- The properties are displayed as the colors of the elements. For example, in Figure 2, the color blue means that the class does not have instances that were used. On the other hand, red classes are those which did have instances that were used.
- When the mouse is over an element, its name is shown. For instance, in our example, elements are classes. Hence, a tooltip shows its name.

To be clear and easy to understand, Figure 2 only shows some packages of Pharo. On the contrary, in Figure 3, you can see all packages of a PharoCore image.

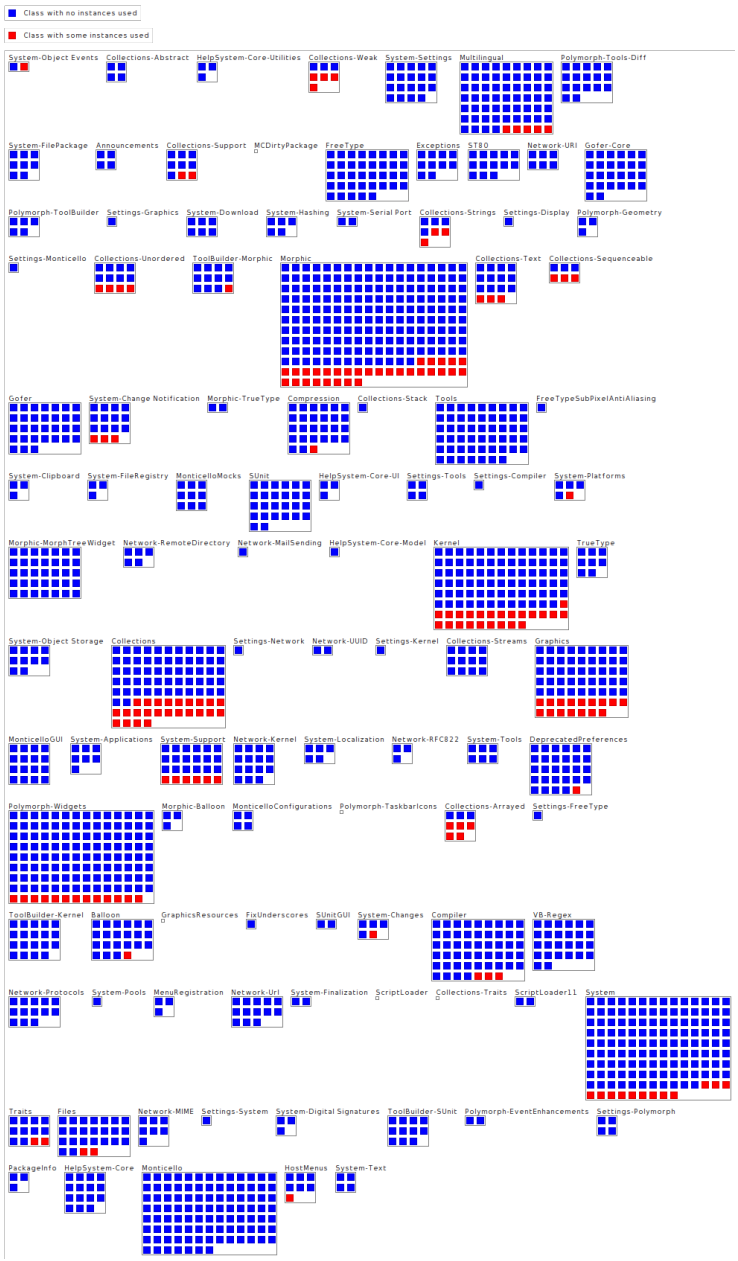

Figure 3: All classes with used instances during the execution of a simple message send.

Notice that there are still extra features of a Distribution Map:

- You can easily detect large packages which can indicate that refactoring is needed (probably split the package into smaller ones).
- You can compare different Distribution Maps of different scenarios to detect which packages and classes are used. Suppose you now want to run a Seaside web application, you can start your application, do the analysis and then compare it with the previous example.
- It is easy to see the average of usage of a certain package. Once again, this feature helps you to detect possible refactors.

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 really used?"* Figure 4 shows a part of a Distribution Map where each container is a class (from the 'Kernel' package) and each element is a method. Again, the analysis of this example is just a single message sent to an object.

From now onwards, we will show only part of the Distribution Maps since they are large. For example, in Figure 4, there are much more classes in the "Kernel" package but we just show a few.

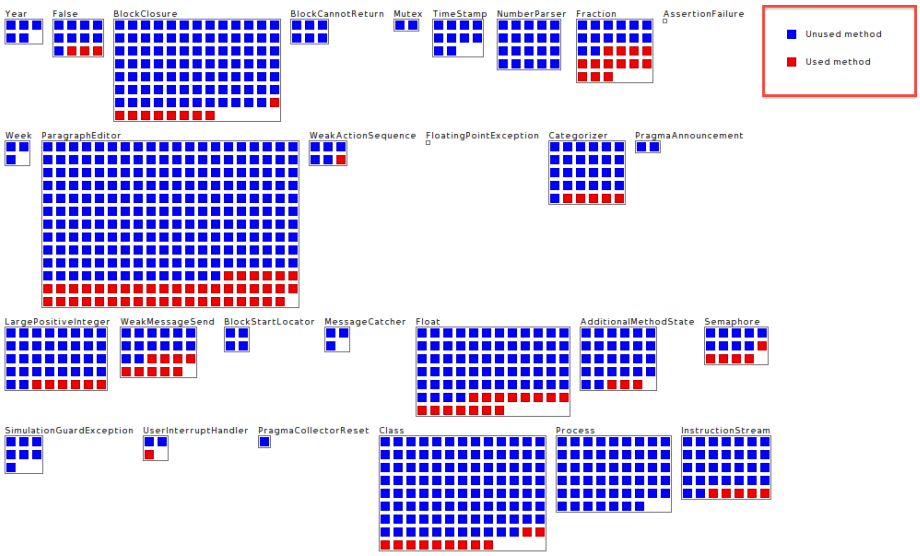

Figure 4: Used methods during the execution of a simple message send

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. Figure 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".

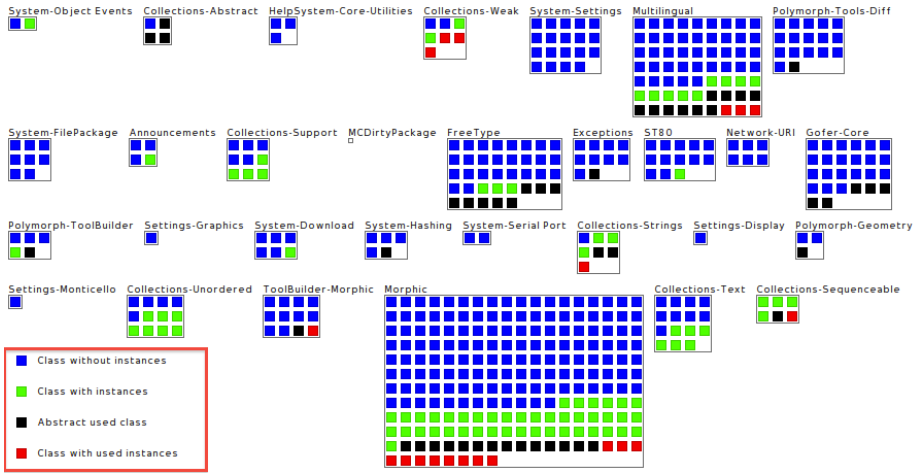

Figure 5: A more accurate view on live classes during the execution of a simple message send

4.2. Amount of instances and memory usage

Probably, you need to go beyond just knowing if a class has used instances or not. For example, you 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 6 shows an example.

Notice that in these Distribution Maps the properties legend (at the top of the visualization) 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. As you can notice in Figure 6, 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.

In Figure 6, you can also notice that the classes with the biggest amount of used instances are ByteSymbol, ByteString, Float, CompiledMethod, ByteSymbol 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 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, ByteSymbol, Array.

It is interesting to compare this to the previous example. The class Bitmap does not even appear in the previous example (Figure 6) as one of the classes with more used instances. However, in Figure 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 Bitmap but each instance occupies much more memory than the average.

Similar to Bitmap, the class WideString has few instances (less than five) but it is one

Figure 6: Live classes grouped by their used instances count.

the the classes that has more memory occupied by used instances.

5. Related Works

There is a plethora of work on runtime information [DPKV94, SDBP98, JSB97, Rei03, LN95b, Sys99] or class instantiation visualizations [DPHKV93, DLB04] but *none* of them show whether instances are actually used or not. To the best of our knowledge we are the first one to treat such information and present it to engineers. This information is important to identify unused objects and, in particular, memory leaks [BM08].

Ducasse *et al.* [DGK06] define Distribution Map, which is a visualization showing spread and focus of properties across systems. Our visualizations show and analyze used/unused objects in object-oriented applications. For this, we use Distribution Map and we define two measurements to represent object usage. We distinguish between a class that just has instances from another one that has *used* instances.

Jerding *et al.* proposed visualization about system execution. The challenge there is to be able to display the execution trace of the system and to navigate it. For this purpose they present mural technique which acts as a zoom out on a large amount of information [JSB97, JS96]. They identify message patterns over large amount of interaction traces.

Win de Pauw *et al.* present how class communicates and their instance creation behavior [DPHKV93]. Bertuli *et al.* use polymetric views [LD03] to show the behavior of instances creation [DLB04].

In the same area than the work of Lange [LN95a, LN95b] on the interactive understanding of design pattern execution, Several works such as the one of Systa, Richner,

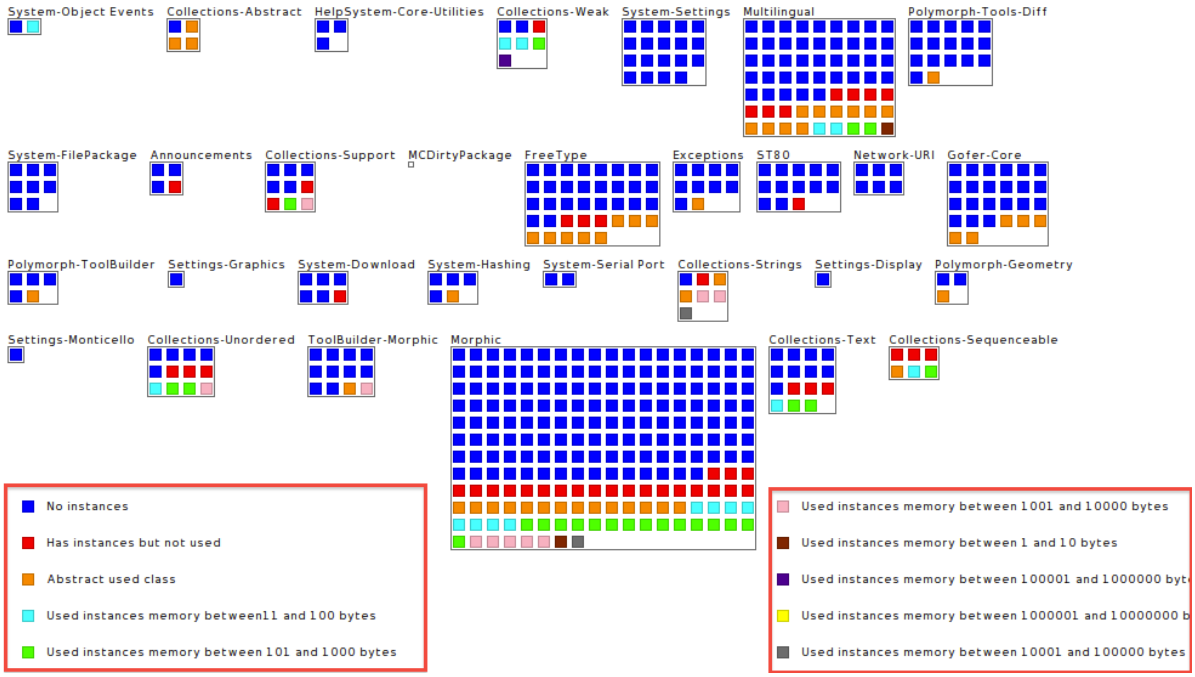

Figure 7: Distribution Map with the memory occupied by used instances.

or Greevy [Sys99, RD99, GD05] mix dynamic information and static one.

Robert Walker et al [WMFB+98] present a tools to navigate dynamic information off-line at architectural level. This approach complements and extends existing profiling and visualization approaches. However they do not mention any point on used objects.

6. Conclusion and Future Work

In this paper, we looked at the problem of visualizing and understanding objects and memory usage in the context of Object Oriented applications. We proposed a solution that extends 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 excellent tools to improve and refactor the system. For example, they indicate that some classes cannot easily be removed from the system, that some classes must be split (create subclasses), that a package must be divided into smaller and less coupled packages, etc.

In addition, they help us understand which objects, methods, classes and packages are used in different scenarios. For example, in case you are developing a web application, you can be interested in knowing which objects, methods, classes or packages

your application is using. The same happens when trying to build a minimal kernel system where you need to know which is the minimal set of objects or code to include in such kernel.

Right now it is difficult to see the differences after running certain code. You have to create a Distribution Map before running your code (for example, the navigation of a Web Application) and another one once you finish. Then, take both Distribution Maps and check the differences. We plan to create a Distribution Map comparator that receives as an input two Distribution Maps and creates a third one with the differences between them.

Acknowledgements.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region (CPER) 2007-2013.

References

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

[BM08] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[DGK06] 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.

[DLB04] 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.

[DPHKV93] 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.

[DPKV94] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings of the European Conference on Object-Oriented Programming*

(ECOOP'94), volume 821 of *LNCS*, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

[GD05] 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.

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[JS96] Dean F. Jerding and John T. Stasko. The information mural: Increasing information bandwidth in visualizations. Technical Report GIT-GVU-96-25, Georgia Institute of Technology, October 1996.

[JSB97] 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.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LN95a] 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.

[LN95b] 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.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[Rei03] Steven P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[SDBP98] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[Sys99] 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.

[WMFB+98] 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.