Context Visualization of Object Factories

Alison Fernandez Blanco[1], Juan Pablo Sandoval Alcocer[2,3], Alexandre Bergel[4]

[1] Semantics S.R.L., Cochabamba Bolivia
[2] Universidad Catolica Boliviana "San Pablo", Cochabamba, Bolivia
[3]Universidad Mayor de San Simón, Bolivia
[4]Pleiad Lab, DCC, University of Chile

***Abstract*—Profiling the memory consumption of a software execution is usually carried out by characterizing calling-context trees. However, the plurality nature of this data-structure make it difficult to adequately and efficiently exploit in practice. As a consequence, most of anomalies in memory footprints are addressed either manually or in an ad-hoc way.**

We propose an interactive visualization of the execution context related to object productions. Our visualization augments the traditional calling-context tree with visual cues to characterize object factory contexts. We performed a qualitative study involving eight software engineers conducting a software execution memory assessment. As a result, we found that participants perceive our visualization as beneficial to characterize a memory consumption and to reduce the overall memory footprint.

I. Introduction

Modern software execution platforms are designed to efficiently cope with a massive number of objects creations and destructions. However, ensuring that the memory consumed by an application execution is not excessive largely remain a manual activity. It is largely known that debugging memory issues is a tedious and error-prone activity [1].

Most of modern programming environments come with a memory profiling tool. Reporting about the memory consumption is typically produced based on introspecting the memory. Characterizing and identifying the cause of an excessive memory consumption is a problem involving several metrics over several dimensions [2], [3]. Unfortunately, most of popular memory and execution profilers offer disconnected memory report representations, leading to a difficult exploitation.

This paper presents and evaluates a visualization of the method execution context. During the application execution, we monitor the memory blueprint of each method call. After the execution, our visualization summarizes the execution. Our visualization conveys information using a calling-context tree, a natural representation of memory consumption. Each context is augmented with visual cues that indicates the cost of it in the overall memory consumption.

We evaluated our visualization with eight professional software engineers. Each engineer was asked to perform a characterization of a familiar software using our visualization. We found that our visualization is perceived as a significant improvement over ad-hoc and log-based manual technique. Our visualization is perceive as intuitive, easy to understand, and easy to navigate through.

This paper is organized as following: Section II motivates the problem we are addressing. Section III details our visualization. Section IV presents a qualitative study we carried out. Section V describes the related work. Section VI concludes and outlines our future work.

II. Memory Usage & Objects Factories

Object factory. In an object-oriented programming language, size of a heap primarily reflects the occurrence and the physical size of objects contained in the heap. Traditional code execution and memory profilers keep track of the consumed memory by characterizing the execution and the heap obtained after the execution.

Consider the piece of Java code[1] given in the upper part of Figure 1. This contrived code snippet creates an instance of the class `Canvas` filled with 30 instances of `Circle` and 15 instances of `Box`. Each use of the `new` operator represents an object factory, which could, potentially, be the source of many objects. This code contains three object factories: one in method `main(String[])` and two in `create(String)`. This section details how this example appears in the YourKit[2] and JProfiler[3] code profilers. Note that we will use this running example through this paper.

Calling context tree. A common way to represent dynamic information is to use a calling-context tree (CCT). Such a tree gives a representation of each snapshot of the calling-context for all method invocations that occur during an application execution. Each node of the tree groups a number of method invocations performed within the context represented by the node. The calling context tree (CCT) compactly represents all calling contexts in the execution, combining nodes that have the same calling context.

Profiler often use a calling context tree to represent runtime information. In the case of memory profiling, the tree is augmented with metrics associated to each method context. For example, consider the execution of our running example under JProfile (Figure 2). The figure uses a tree text widget to visually render the calling context tree. The tree summarizes

[1]Our profiler and visualization run for the Pharo programming language. We give here Java code and not Pharo's to ease the reading.

[2]`https://www.yourkit.com/`

[3]`https://www.ej-technologies.com/products/jprofiler/overview.html`

```
class Shape { int x, y; }
class Box extends Shape { double h, w; }
class Circle extends Shape { double r; }
class ShapeFactory {
    Shape create(String form){ // D
        if(form.equals("Circle"))
            return new Circle();
        else{
            if(form.equals("Box"))
                return new Box();
            else
                return null;
        }
    }
}
```

```
public class Canvas{
    List<Shape> shapes = new ArrayList<>();
    ShapeFactory factory = new ShapeFactory(); // B

    void add(Shape s){ shapes.add(s); }

    void createCircles(int number){  // C
      for(int i = 0; i < number; i++)
        add(factory.create("Circle"));
    }

    void createBoxes(int number){ // E
      for(int i = 0; i < number; i++)
        add(factory.create("Box"));
    }

    public static void main(String[] argv){ // A
        Canvas canvas = new Canvas();
        canvas.createCircles(30);
        canvas.createBoxes(15);
    }
}
```

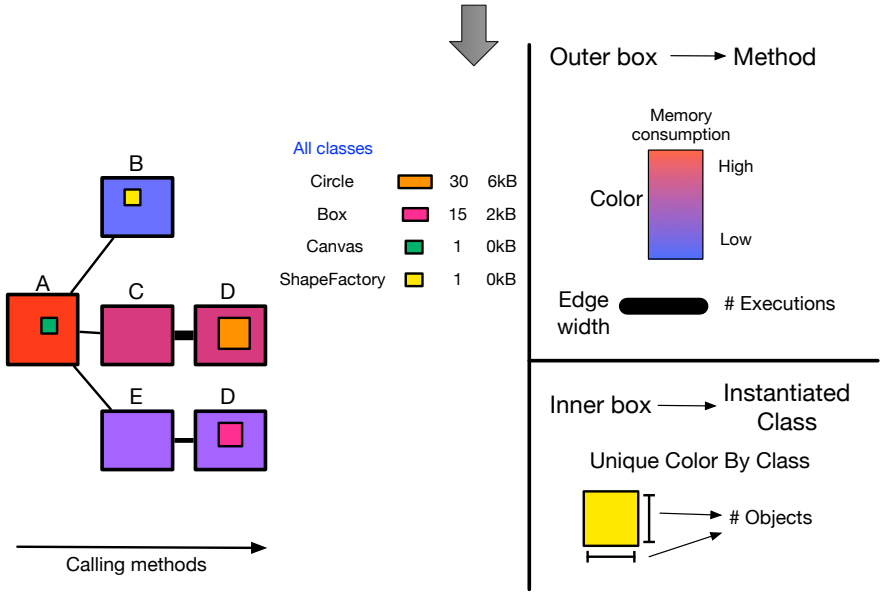

Fig. 1: Running example and detail of our visualization

the execution of the code. The method `Canvas.main` consumes 100% of the memory consumed by the whole execution. This method executes the `Canvas.<init>` constructor, which accounts for 90.8% of the whole memory consumption. The number of object allocations and the actual size in memory are also provided for each context.

Note that the tree does not indicate which classes are actually instantiated. On our example, we can easily deduce it since the produced calling-contact tree is tiny. Identifying the classes that are instantiated is challenging for the execution of any non-trivial piece of code. Note that this situation is not particular to JProfiler. VisualVM[4], NetBeansProfiler[5], and the YourKit profilers provide a similar view and suffer from of the same limitations.

Allocation table. Some profiler complement the range of supported analysis by offering an occurrence table indicating the number of instances for each type and class. Typical information includes: the number of instances created for each type, and the physical size of these instances. Using this table, it is easy to determine the weight of each classes in the overall memory footprint. For example, Figure 3 gives the allocation table given by JProfiler for our running example.

[4] `https://visualvm.github.io/`
[5] `https://profiler.netbeans.org/`

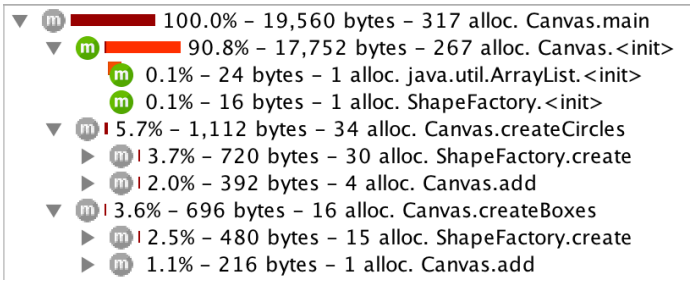

Fig. 2: Calling context tree of memory consumption with JProfiler

Name	Instance Count	Size
char[ ]	315	51,904 byt...
java.lang.String	126	3,024 bytes
byte[ ]	92	22,344 byt...
java.lang.Class	82	26,240 byt...
java.lang.StringBuilder	53	1,272 bytes
Circle	30	720 bytes
java.lang.Object	30	480 bytes
java.io.File	23	736 bytes
java.util.concurrent.Concurr...	23	736 bytes
java.net.URL	21	1,344 bytes
java.util.HashMap$Node	20	640 bytes
java.lang.StringBuffer	18	432 bytes
Box	15	480 bytes

Fig. 3: List of classes instantiated with JProfiler

The figure indicates that the largest amount of memory is consumed by 315 instances of `char[]`. Since the table does not consider the CCT, we have no indication on where in the whole execution these instances are created. Since our example does not create strings, we deduce that the Java runtime or even JProfiler is likely to be the cause of these instances. Information of the `Circle` and `Box` classes are mixed within information related to execution not related to our example. We see that this table is of little help to identifying where and when during a program execution instances are created, which is crucial to detect memory bottlenecks and find memory optimization opportunities.

Limitations. This section illustrates two shortcomings present in popular memory profiling tools. A number of limitations are deduced from these shortcomings:

- *Indirect instantiation.* YourKit and JProfiler annotates the calling context tree with object allocation that are directly or indirectly performed within each context. However, these is no information about which classes are associated to these object allocations. As such, a simple question like *Which classes were instantiated in an given method?* cannot be directly answered.
- *Object distribution.* YourKit and JProfiler provide a summary of the overall consumption using the allocation table. However, mapping the distribution of the reported objects over the calling-context tree is left to be manually done. As a consequence, even a simple question like *Which methods are instantiating the most objects?* cannot be simply answered.
- *Pattern discovery.* It is known that textual listing are suboptimal in identifying relevant values or patterns [4]. A simple question like *Are two or more classes that have similar number of instances?* cannot be answered without a tedious manual work.

The next section presents a visualization we have designed that addresses these limitations.

III. Context Visualization of Object Factories

We propose a new visualization to identify and characterize contexts responsible of creating objects. Our interactive visualization is composed of two main components: (i) a calling-context tree in which object factories are carefully represented and (ii) an interactive class list to narrow the search and highlights particular nodes in the tree.

A. Calling Context Tree (CCT) visualization

We use a polymetric view [5] to visualize the calling context tree obtained from an application execution. Each node of the tree represents a method invoked in a given context that creates at least one object (directly or indirectly). Edges represent the callee/caller relation between method contexts during the execution. The lower part of Figure 1 gives our blueprint for the running example, which is detailed below.

Tree Layout. Nodes of the tree are located using an horizontal tree layout. Roots, which typically includes the `main` method, are located on the left hand side and leafs on the right hand side. This organization favors reading the visualization from left to right, as most non-semitic natural languages do (*e.g.,* latin and anglo-saxon). A caller method is placed on the left of the called methods, and an edge indicates the execution control flow, from left to right.

Calling contexts are ordered from up to bottom to reflect the order in which methods are called. A branch located at the top of the visualization was the first one to be executed while the bottom-most branch was the last one before the execution complete.

Nodes. Each node represents a node of the calling context tree (CCT). A node in a CCT represents a method and the context in which the method is invoked. A method called by several other methods may therefore appear several times in the tree (*e.g.,* Method `D` in Figure 1).

A node color indicates the amount of memory directly and indirectly consumed by this node. Color fades from red to blue, where red means that the node is the most memory-consuming while a blue node indicates that little memory is consumed. For example, in the lowest branch in Figure 1, method `createBoxes` (marked `E` in the lower part) calls the method `create` (`D`). Both `E` and `D` have the same color since they consume the exact same amount of memory. Method `E` consumes more than `B`, the constructor of `Canvas`.

A node may contains inner boxes. Each box represents a group of objects created in the encapsulating node context. Each group of objects are instances of the same class. In Figure 1 we have four groups of created objects. Method `A` creates a `Canvas`, `B` creates a `ShapeFactory`, `D` in the middle

branch creates 30 `Circles`, and `D` in the lower branch creates 15 `Boxes`. The exact number of instances is indicated in the list located on the right of the CCT.

Each class as a unique color to favor identifying patterns in method contexts based on the instantiated classes. For example, in Figure 1, we immediately see that the four object factories creates instances from different classes. Size of the inner box linearly indicates the number of objects created in this method context.

Figure 4 illustrates our visualization on a large and representative code execution. The mouse cursor is above a context of the method `renderCountries`, defined in the class `RTMapBuilder`. This context has one inner box and calls two other methods, each having an inner box of the same size. We can therefore deduce that these three contexts creates the same number of instances, for three different classes since the color of the inner boxes are distinct. As described below, the popup details the exact number of created class instances.

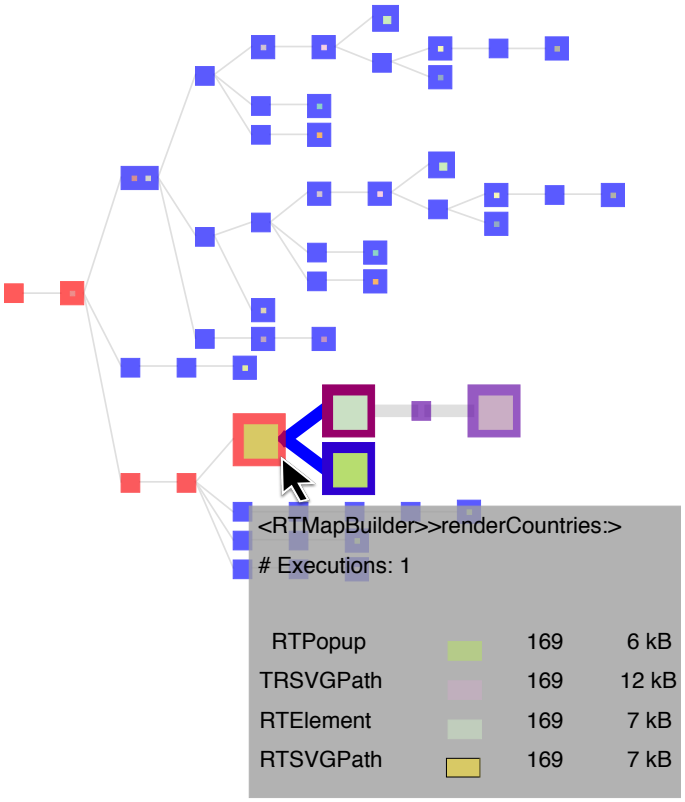

Fig. 4: Call context tree interaction when the mouse is over the method `renderCountries:` of the class `RTMapBuilder`

Edges. Call flow between method context nodes is indicated using edges: a context on the left calls another connected context on the right. The width of an edge is proportional to the number of times that the method is called by the calling method context. A context represents the path from a particular node to the root. A method may be called from different contexts.

For example, the method `D` is called in two different contexts, in Figure 1. We also see the that the method `D` is called by `C` more times than when called by `E`, since the edge between `C` and `D` is thick. The exactly number of execution is given by a popup.

Interactions. The method context nodes (*i.e.,* outer boxes) support a number of interactions, actionable by moving the mouse cursor:

- *Highlighting.* When passing over a node its outgoing edges and connected nodes are highlighted. Figure 4 illustrates this situation.
- *Node popup.* A popup window appears when the mouse is above a method context node. The popup indicates the method name, the number of times its has been invoked by the calling method context. Furthermore, the popup summarizes direct and indirect objects creation by the pointed method. This summary is presented as a list, in which:
  - *Instantiated classes.* The first column contains the name of the classes that are directly or indirectly instantiated by the method context. In Figure 4, we see that the context that corresponds to the method `renderCountries:` instantiate four classes.
  - *Black border.* A class that is directly instantiated by the method context is marked with a black border. We see that the class `RTSVGPath` is directly instantiated by the pointed method. It means that the source code of `renderCountries:` contains the expression `new RTSVGPath().`
  - *Number of instances.* The third column gives the number of objects directly and indirectly created. We see that each of the four classes is instantiated 169 times.
  - *Memory usage.* The fourth column indicates the memory footprint in Kilobytes (kB).
- *Inner node popup.* For inner nodes, the popup indicates the number of objects produced of the inner node corresponding class.
- *Source code.* When clicking an outer node the source code corresponding to that method is opened (not shown in the figure).

B. Interactive class list

In addition to the calling-context tree, our visualization summarizes the number of instances for each class involved in the execution (Figure 5). This summary is given as an interactive list of class names, located on the right of the tree. The list is decreasingly ordered with respect to the number of instances created for each class. Content of the list is similar to the node popup, previously described: each row contains the class name, an horizontal rectangle indicating the color and whose width indicates the number of instances, and the memory consumption expressed in kB.

When first opened, the visualization shows the complete calling-context tree and all the classes are listed, as illustrated

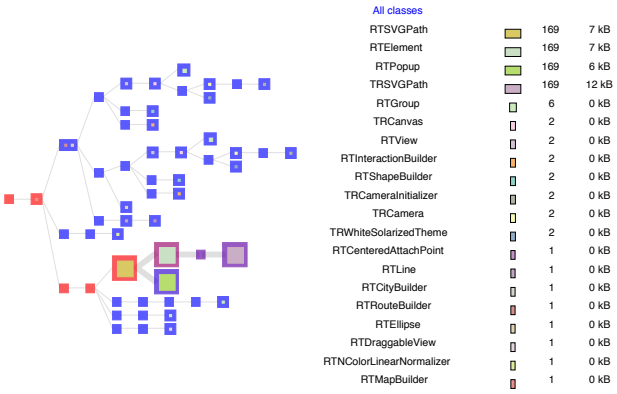

Fig. 5: Call context tree and the interactive class list

in Figure 5. To mitigate the necessity to make a visualization scalable, the interactive class list allows for filtering and highlighting, two operations detailed below.

Class Exclusion. A practitioner may exclude a class from the tree visualization by right clicking on the name in the class list. Excluding a class results in removing all corresponding inner boxes in the tree. Note that a branch that is left without inner boxes is also removed from the tree.

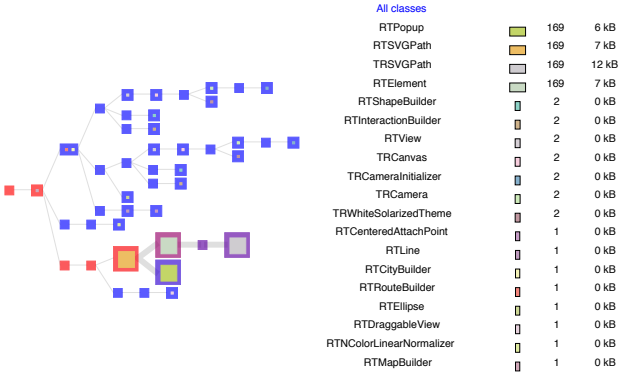

Fig. 6: Calling context tree without objects of `RTGroup`

Figure 5 indicates that the class `RTGroup` has 6 instances. Figure 6 shows the result of excluding this class from the tree. By removing this class from the visualization, several branches of the tree are empty and therefore are removed.

Method context highlighting. Method context may be highlighted for a given class. Locating the mouse above a class name shade all the method contexts that do not directly or indirectly instantiate that class. Figure 7 illustrates the highlighting of all method context that are related to the instantiation of the class `RTGroup`. The four branches in the calling context tree, highlighted by shading all other nodes, end with an instantiation of `RTGroup`.

By clicking on a class name, the highlight is fixed to let further exploration by moving the above the method context. Clicking back on the class name unfix the highlight.

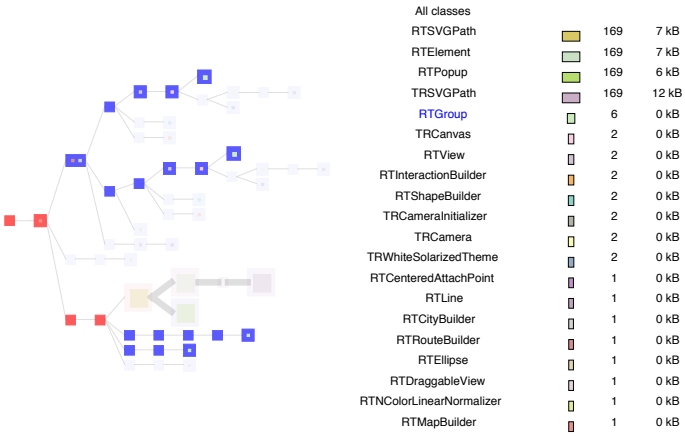

Fig. 7: Paths of execution that produce at least one object of the class `RTGroup`

C. Example

Figure 8 and Figure 9 represent the evolution of the execution of a large application. Figure 8 describes the original execution of a sunburst application [6]. During the execution, the application executes 141 methods (not represented on the figure), for which only 17 methods contain an object factory. The tree presented in Figure 8 contains 87 method contexts, representing the different contexts in which the 17 methods and their callers are executed.

In our setting, the application computes a sunburst for an arbitrary set of 382 input data. The interactive class list in Figure 8 indicates that 382 instances of the class `RTElement` and `TRArcShape` are created. We can therefore deduce that the number of created instances is likely to be related to the number of provided inputs. The class `RTGroup` is instantiated 766, thus it is likely that each Sunburst input element creates two instances of `RTGroup`. In total, the creation of the sunburst produces over 1,500 objects consuming more than 70kB of memory. In presence of a modern executing platform, such figures may appear as insignificant. However, the sunburst is integrated within the Pharo programming environment to visualize source code while a programmer is programming. It is therefore crucial to have the visualization snappy with no latency. It is therefore essential to avoid any unnecessary memory consumption.

The visualization presents the repetition of a visual pattern. Inspecting the source code of the method A indicates the presence of a non-terminal recursion, which is simplified as:

```
RTAbstractTreeBuilder>>createElements: atree using: ablock depth:
        depth
  e := shapeBuilder elementOn: atree.
  children := children collect: [ :child | | e2 |
      self createElements: child using: ablock depth: depth + 1.
      ]
```

The call to `createElements: using: depth:` execute the method marked as B in the visualization, which is `elementOn:`. The source code of this method is:

```
RTShapeBuilder>>elementOn: object
```

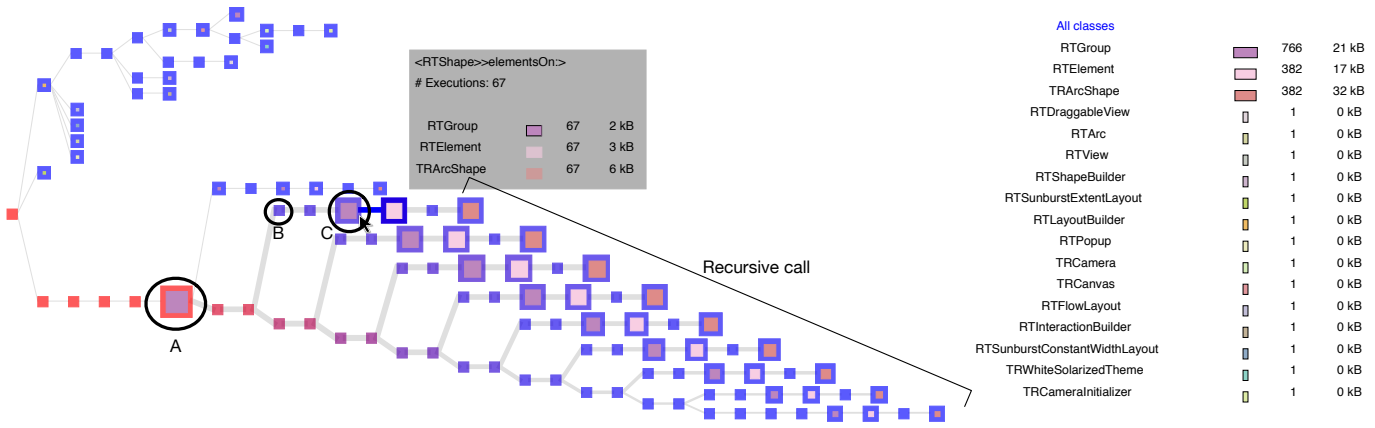

Fig. 8: Visualizing the execution of the Sunburst application

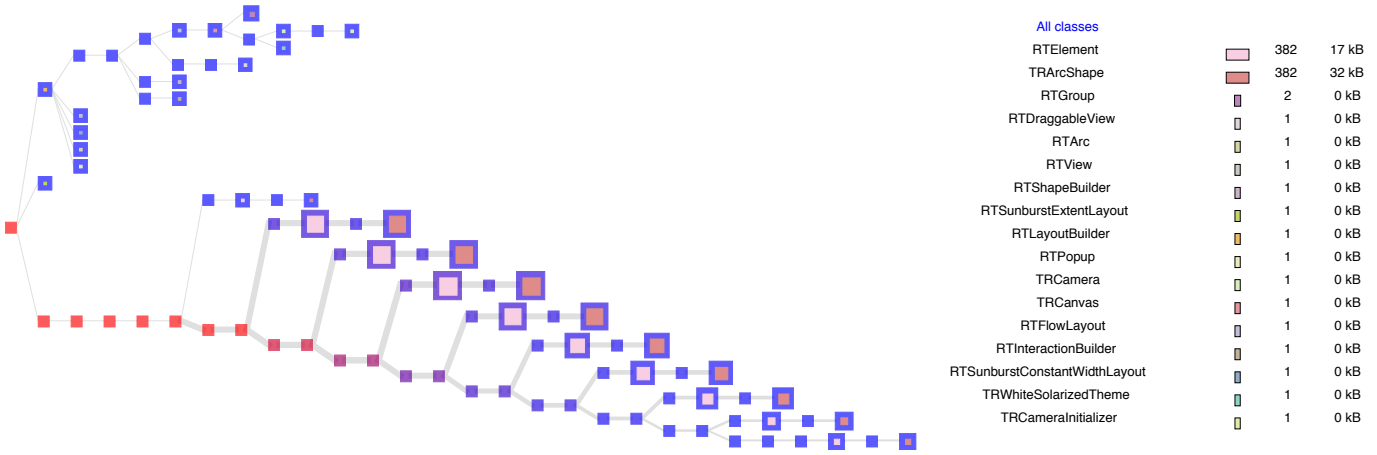

Fig. 9: The memory reduction of Sunburst

```
    ^ (self elementsOn: (RTGroup with: object)) first
```

The method calls indirectly to the method marked as C in the visualization, which creates several instances of `RTGroup`. The source code of this method is:

```
RTShape>>elementsOn: models
  | group |
  group := RTGroup new.
  models do: [ :m | group add: (self elementOn: m) ].
  ^ group
```

This method return an instance of `RTGroup` with elements created based on `models` each time that is called. On this case, we only need the elements created based on `models` calling to the method `RTShape>>elementOn:`. To avoid the creation of several instances of `RTGroup` we modify the method B as follows (simplified version):

```
RTShapeBuilder>>elementOn: object
   answer := shape elementOn: object.
   createdElements add: answer.
```

Avoiding the unnecessary creation of `RTGroup` has a visual impact, as presented in Figure 9. The calling context related to `RTShapeBuilder>>elementOn:` now don't call to `RTShape>>elementsOn:` instead it calls to `RTShape>>elementOn:`, so the recursive call is free from inner boxes related to `RTGroup`.

IV. User Study

This section presents the design and the results of an explorative user study [7]. Our results support the viability of using our visualization as an effective tool (i) to analyze the memory consumption and (ii) to identify optimization opportunities.

A. Participants

Since our visualization is implemented in Pharo and our experiment requires comprehending and modifying Pharo code, our participants must be familiar with the Pharo programming language.

We selected eight participants, all have a solid software engineering experience. From the eight participants, 4 are profesional software engineers in a local Chilean software company, 2 are PhD students in the field of software engineering, and the remaining 2 develop software in an academic context. The experience of our participants span over various

programming languages and programming environments, in addition to Pharo. In particular, Java, Python, and JavaScript are mentioned as known programming languages.

Table I summarizes the profile of each participant. The first column assigns an identifier to each participant. We will use these identifiers along the description of our user study. The second column refers to the number of experience years in software development. The third column gives a brief explanation on how a participant usually addresses an excessive memory consumption in a running application:

- *Logging*: A memory anomaly can be identified by inserting an event generation next to an object production site. For example, printing a message whenever a class constructor is executed.
- *Manual*: The code execution is manually traced until a line of code prone to an excessive memory consumption is reached.
- *No experience*: The participant has no experience in addressing memory issue.
- *Debugger*: The participant used a dedicated memory debugger and profiler to track down the issue.

The fourth column of Table I indicates whether the participant authored the benchmark that will be analyzed in the user study (as described below). The last column gives the name of the benchmark.

Par.	Exp. Prog.	Experience in addressing memory issue	Exp. Bench.	Benchmark
P1	15	Logging	●	Sunburst
P2	6	Manual	●	BundleBuilder
P3	5	Debugger	●	GraphQL
P4	14	No experience	●	Discord
P5	7	Manual	●	HartLib
P6	7	No experience	○	Grapher
P7	3	Manual	○	MapBuilder
P8	5	Manual	○	Mondrian

TABLE I: Information of participants.

B. Benchmarks

A software execution benchmark provides a reference point against which to measure absolute and relative performances. Our study requires some benchmarks on which participants will use our visualization to solve some generic tasks.

Analyzing memory consumption is not a trivial and common activity. A great knowledge is required (i) to understand if a memory consumption footprint indicates an anomaly, and (ii) to identify the cause of that anomaly. As a consequence, we need to carefully chose the benchmarks our participants will have to consider in our study.

We selected for each participant a project that we know the participant is familiar with (either as an author or as a user). The fourth column of Table I indicates whether the participant is the author of the benchmark: ● indicates the participant authored the benchmark, ○ if the participant is a user of the project.

Table II describes the benchmarks considered in our study.

Project	Benchmark
Grapher	Creates ellipses representing 598 classes where the size is proportional to the lines of code and the position corresponds to the number of methods and the number of variables.
MapBuilder	Creates circles on the map representing 2200 earthquakes, positioned given the latitude and longitude, with a size proportional to the magnitude of the earthquake.
SunBurst	Creates a visualization of sunburst of 382 classes, where each class has a color and its position depends on the subclasses.
BundleBuilder	Creates a visualization of 598 classes, where each node is a class and it's connected by bezier lines to his dependent classes.
Mondrian	Creates circles representing 585 classes, each class connected with their superclass, with the color and the size normalized and using a cluster layout.
GraphQL	Given a simple schema and an entry point return the answer to a simple request, all this following the specification of GraphQL.
Discord	Given an user on discord, login to the account and recollect all the last 50 messages of 3 servers where the user is involved.
HartLib	Construct a pdf document given several specifications.

TABLE II: Projects and benchmarks.

C. Tasks and Work Session

A work session involves one participant and one benchmark. The work session is carried out as follows:

1) *General questions* - Four questions are asked to the participant about general experience and experience with dealing with memory issues.
2) *Learning material* - We provide a tutorial-like description of our visualization.
3) *Tasks* - Three or four questions are asked to the participant. It is necessary to use our visualization in order to answer the questions. If the participant authored the benchmark (●), then four questions are asked. In case the author is a user (○) three questions are asked. The questions are divided into three categories: Identifying, Understanding, and Improving. Questions and their rationales are given in Table III.
4) *Experiment feedback* - Open comments and a few open questions are gathered. We gathered comments in an informal and oral fashion to not pressure the participant into giving an answer that we expect.

We observe and monitor the execution of each work session, each participant performed the tasks on a computer and reported their answers in an online form[6]. Running a work session for each of the 8 participants totals 4.18 hours.

D. Results and Discussion

This section presents and summarizes the results we obtained.

- *Identifying object factories.* The eight participants understood the visualization. All the participants were able to characterize the memory consumption of the application for which the benchmark is run. Some participants used the visualization to spot object factories by searching for method context colored in red (we recall that red

[6]`https://goo.gl/forms/kqsvABhADjlqjodn2` (●), `https://goo.gl/forms/tYli8bmkoBpqV0dA3` (○)

Category	Par.	Question	Rationale
Identifying	●○	Can you characterize the memory consumption of the benchmark? Why?	Identify where are the top memory consumption, where are the objects created on the path of execution and what methods were related to each other on the call context tree.
Understanding	●○	Do you understand the visualization?	Understand how is executed the program and the memory that consumes on all the path of execution.
		Do you find it intuitive? Why?	We want to participantes could use the visualization without problems to understand what is representing.
	●	Do you find some anomalies in the way memory is consumed? Why?	Understand if the number of objects created is accord to the schema mental or the idea that they have, and also if they really know how much memory they are using.
Improving	●	Can you modify your application to reduce the memory consumption?	Modify their application to see what features are more used of the memory profiler and how they deal with memory consumption issues using the memory profiler.
	○	If you would reduce the memory consumption of that application, how would you do it?	Try to reduce de memory consumption of the application to see how they deal with memory consumption issues using the profiler.

TABLE III: Questions to the participants.

indicates a high memory consumption). As a result, all the participants have successfully identified object factories on their respective benchmarks.

- *Understanding object factories.* All the participants understood the execution of the program and had no difficulties understanding what is a calling-context tree. Actually, participants consider a calling context tree to be a familiar structure. Participants correctly understood the meaning of inner boxes and contrasting the visualization with method source codes did not generate doubts. We found that 80% of the benchmark authors found anomalies in their application because of the creation of unnecessary objects. Participant P5 found an opportunity for introducing a cache.
- *Improving object factories.* All the five benchmark author tried to modify their applications, by proposing several solutions to reduce the memory consumption. Two of the five authors could actually implement the changes they proposed. The remaining three could not implement an improvement, essentially because of the large and deep modification this code improvement implied. Significantly more time was necessary to complete the tasks.
  For the application authors, we asked them a possible solution to reduce the memory consumption. All of them gave solutions according to what they think were unnecessary object creation during the execution. For example, P6 and P8 identified unnecessary creation of `RTGroup` according the benchmark, and P7 introduced a lazy object creation.

Observations. By observing the participants use our visualization, we found a number of relevant elements.

- *Interactive menu.* All the participants use the interactive class list menu to see the total number of created objects.
- *Class highlights.* All the participants highlighted classes in order to identify methods that creates objects of a specific class.
- *Node popups.* All the participants uses the visual popups to obtain the method's name, the number of times it was executed, the classes instantiated, and the number of instances.
- *Source code.* 87.5% of the participants use the interaction of source code at the time of the identify anomalies and to modify the code to reduce memory consumption.
- *Class exclusion.* 25% of the participants use the operation of class exclusion to discard the objects of a given class, because they thought that the total number of objects was the right one.

Experiment feedback. After the experiment we ask our participants about their opinion about the memory profiler. We summarize the answers we collected for two informally asked questions:

- *Do you feel more efficient at identifying memory issue using memory blueprint than without?* All the participants agree that using the memory profiler make them more efficient to identify memory issue than doing it manually. Participants P5 and P8 commented that our profiler is better than other profilers, including JProfiler and YourKit, because they consider that a visual representation is more compact than a large textual description. The visual cues offered by our visualization are also perceived as important in order to understand and find object factories. However, these two participants made some suggestions on how our visualization can be improved.
- *Do you have some suggestion on how to improve the visualization?* We obtained several propositions:
  - *Easy use* P3, P4 and P7 wants an easy way to install the visualization and use it. The participant suggests a better integration within the Pharo programming environment. Having an example of a visualization is also mentioned as relevant.
  - *Object's life* P5 and P6 suggested that the life time of the objects could also be represented. In particular, some of the spotted objects may be garbage collected, which are not represented on the visualization. Knowing when exactly an object dies is perceived to be relevant. These two participants wish to know when

exactly an object is removed from the memory or whether the objects live until the end of the execution.
- *Exclude with condition* P7 thinks that the operation of class exclusion should satisfy a condition. For example, classes with less instances than a particular threshold could be automatically excluded from the visualization.
- *Line of Code* P6 and P8 thinks that it would be relevant to indicate the line of code where the object is created. This feature is perceived as a way to simplify finding object factories.

E. Threats to validity

As most experimental studies, our work is subject to threats to validity. We presents these threats using the classical structure *conclusion*, *internal*, *construct*, and *external*.

Conclusion Validity. A threat to conclusion validity is a factor that may lead our effort to an incorrect conclusion about a relationship in our observations.

Our conclusion is based on the an explorative user study that involves a group of participants. We considered 8 participants. Although this number if relatively low, we mitigated possible bias by choosing participants with a diverse background.

Internal Validity. This threat present influences that could affect the independent variable. Internal validity refers to how well our experiment was designed and conducted (*e.g.,* identifying possible independent variables).

All our participants have a solid background in Pharo, are knowledgable in other programming languages, and have a certain level of knowledge about the benchmark. We use different benchmarks, one per participant. We therefore avoid having independent variables that are related to the programming language or the benchmark.

Construct Validity. This threat presents the influences of social factors or the design of the experiment to generalize the experiment result.

We voluntarily focussed on the Pharo programming language. We ensured that the participants we selected have knowledge about the benchmark and fully understand the problem they have to solve. Output of each participant session was carefully measured using observation while a particular task was being completed.

External Validity. This threat presents the conditions that generalizes the results of our study to other situations (*e.g.,* in industry).

This paper voluntary focus in real benchmarks of the Pharo ecosystem. Besides we cover diverse categories of software projects, the external validity of our user study is limited. However, we believe that our study provide relevant evidence of the viability of our approach in the studied projects.

V. Related Work

Visualizing memory consumption is an active research area. This section highlights the most relevant related works.

Production sites. Infante [8] presents memory blueprint as a visual representation of a graph of calls that indicates object production sites for a given application execution. Similar to our visualization, memory blueprint helps to identify methods that creates objects. Contrary to our visualization, memory blueprint looses the context of the method whence one cannot immediately identify which objects exactly are produced based on the callees. In addition, one cannot see the source code of the method, exclude class from the analysis or see the paths where certain class is instantiated.

Allocation and death of objects. Veroy [9] presents a graph with hive plots [10] the relationship of the objects allocation context to the death context. This work has a different objective to ours. Their hive plots focuses on identifying which classes are hot spots for object death, and not to identify or understand the object factories that may lead to the creation of unnecessary objects.

GCspy framework. Printezis [11] presents GCspy, a framework for the collection, storage and replay of memory management behavior. Printezis allows one to visualize used memory space like a history graph and offers an user interface view to show detailed information about the space management. Contrary to our work, Printezis focuses on the used and the free spaces. Later Cheadle [12] extends the GCspy framework to visualize memory allocators like `dlmalloc` and how concurrent garbage collectors make use of the heap memory. The visualization presented by the extensions is based on a serial of nested boxes with representative colors for the heap and their respective spaces. Contrary to our work, Cheadle's work focuses on the use of the heap by garbage collectors and `dlmalloc` and not identify the creation of objects.

Visualizing dynamic information. Walker [13] presents an approach to visualize operations of an object-oriented system at the architectural level, their approach abstracts the number of objects involved in the execution and the interaction between objects, without representing the memory consumption. Later Ducasse [14] applied polymetric views to visualize dynamic information as boxes with features associated to dynamic information. Ducasse's work is an example of a visualization for the classes that are the most instantiated, but contrary to our work they do not focus on information about the memory consumption.

Unused objects. Peck [15] presents a visualization of nested boxes to understand the memory usage analyzing the amount of instances of a class, of used and unused instances, and other metrics. The objective of this work is to detect unused instances of a program execution to avoid creating them, and not identify and characterizing object factories.

Object churn. Duseau [16] presents Vasco, a visual approach to explore object churn in framework intensive applications, an object churn consists in the excessive use of temporary objects generating possible memory bloats. Vasco visualizes the object churn on a sunburst representing a CCT. They work has a different purpose than ours therefore.

Memory leaks. There is a number of works to address memory leaks. Jinsight [17] is a tool with a number of visualizations to explore the lifetimes of objects, showing the interactions, deadlocks and the garbage collector activity. Contrary to our visualization, Jinsight represents the call context tree render as text like JProfiler and presents the limitations previously seen.

Heapviz [18] is a tool to visualize and explore the heap graph using a radial layout of the heap snapshots, obtained from a running Java program. Reiss *et al.* [19] present a plane compact interactive graph visualized as a tree map to show the memory behavior of a executed program. Also Reiss [20] presents a tool visualization where each class and package is displayed using a box display visualization. This tool indicates the behavior of the memory, the allocations, and deallocations. Reiss decided to do an improvement called JOVE [21] taking a different visualization to show the threads. All these works attempt to identify memory leaks in their on way, but using visual graphs or tree maps. As a consequence, they loose the context of the method, and do not have filter criteria and cannot view the source code. They use information for deallocations to identify memory leaks.

VI. Conclusions

This paper presented a visualization of object factories. It visualizes the method calling context in which objects are created. Our approach visually maps object creations to the node of a calling-context tree. Several metrics are provided to characterizing the memory consumption. Our visualization is interactive to navigate and reduces the number of nodes in the tree.

We present a user study that involves 8 software engineers who use our approach to identify, understand and improve the memory usage of their own projects or projects that are familiar with. Our results shows that all participants were able to identify and characterize object factories. In addition, some of them were able also to reduce the creation number of objects in their benchmarks, reducing in this way the memory usage.

References

[1] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, J. Murphy, Patterns of memory inefficiency, in: Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 383–407.
URL http://dl.acm.org/citation.cfm?id=2032497.2032523

[2] D. Marinov, R. O'Callahan, Object equality profiling, in: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 313–325. doi:10.1145/949305.949333.
URL http://doi.acm.org/10.1145/949305.949333

[3] A. Infante, A. Bergel, Object equivalence: Revisiting object equality profiling (an experience report), in: Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages, DLS 2017, ACM, New York, NY, USA, 2017, pp. 27–38. doi:10.1145/3133841.3133844.
URL http://doi.acm.org/10.1145/3133841.3133844

[4] E. Tufte, P. Graves-Morris, The visual display of quantitative information.; 1983 (2014).

[5] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.
URL http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf

[6] M. Mamani, A. Infante, A. Bergel, Inti: Tracking performance issue using a compact and effective visualization, in: 2014 33rd International Conference of the Chilean Computer Science Society (SCCC), 2014, pp. 132–134. doi:10.1109/SCCC.2014.28.

[7] G. Ellis, A. Dix, An explorative analysis of user evaluation studies in information visualisation, in: Proceedings of the 2006 AVI workshop on BEyond time and errors: novel evaluation methods for information visualization, ACM, 2006, pp. 1–7.

[8] A. Infante, A. Bergel, Efficiently identifying object production sites, in: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on, IEEE, 2015, pp. 575–579.

[9] R. L. Veroy, N. P. Ricci, S. Z. Guyer, Visualizing the allocation and death of objects, in: Software Visualization (VISSOFT), 2013 First IEEE Working Conference on, IEEE, 2013, pp. 1–4.

[10] M. Krzywinski, I. Birol, S. J. Jones, M. A. Marra, Hive plots?rational approach to visualizing networks, Briefings in bioinformatics 13 (5) (2011) 627–644.

[11] T. Printezis, R. Jones, GCspy: an adaptable heap visualisation framework, Vol. 37, ACM, 2002.

[12] A. M. Cheadle, A. Field, J. Ayres, N. Dunn, R. A. Hayden, J. Nystrom-Persson, Visualising dynamic memory allocators, in: Proceedings of the 5th international symposium on Memory management, ACM, 2006, pp. 115–125.

[13] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. Isaak, Visualizing dynamic software system information through high-level models, in: ACM SIGPLAN Notices, Vol. 33, ACM, 1998, pp. 271–283.

[14] S. Ducasse, M. Lanza, R. Bertuli, High-level polymetric views of condensed run-time information, in: Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on, IEEE, 2004, pp. 309–318.

[15] M. M. Peck, N. Bouraqadi, M. Denker, S. Ducasse, L. Fabresse, Visualizing objects and memory usage, in: Smalltalks' 2010, 2010.

[16] F. Duseau, B. Dufour, H. Sahraoui, Vasco: A visual approach to explore object churn in framework-intensive applications, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 15–24.

[17] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, Visualizing the execution of java programs, Software Visualization (2002) 647–650.

[18] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, S. Z. Guyer, Heapviz: interactive heap visualization for program understanding and debugging, in: Proceedings of the 5th international symposium on Software visualization, ACM, 2010, pp. 53–62.

[19] S. P. Reiss, Visualizing the java heap to detect memory problems, in: Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on, IEEE, 2009, pp. 73–80.

[20] S. P. Reiss, Visualizing java in action, in: Proceedings of the 2003 ACM symposium on Software visualization, ACM, 2003, pp. 57–ff.

[21] S. P. Reiss, M. Renieris, Jove: Java as it happens, in: Proceedings of the 2005 ACM symposium on Software visualization, ACM, 2005, pp. 115–124.