Visually Exploring Object Mutation

Rodrigo Schulz
DCC,
University of Chile
Chile

Fabian Beck
VISUS,
University of Stuttgart
Germany

Jhonny Wilder Cerezo Felipez
Universidad Mayor
de San Simón
Bolivia

Alexandre Bergel
Pleiad Lab, DCC
University of Chile
Chile

***Abstract*—Object-oriented programming supports object mutation during a program execution. A mutation occurs whenever a value is assigned to an object field. Analyzing the evolution of object mutation is known to be difficult. Unfortunately, classical code debuggers painfully support the analysis of object mutations.**

***Object Evolution Blueprint* is a visualization dedicated to exploring object mutation over time. Our blueprint visually and concisely represents sequences of field mutations. The history of each field is adequately shown with respect to the dynamic value types. We have observed the use of our blueprint with three practitioners. Our visualization has been well received and accepted to complete two different software comprehension tasks. Moreover, our user study shows that the visualization is both intuitive and simple to learn.**

I. Introduction

The execution of an application written in an object-oriented programming language may produce a large number of objects [1]. These objects interact with each other by sending messages. Whenever an object receives a message, the values of the instance variables may change. While a message may produce a state transition, the set of instance variables and their current values represent the current state of an object. A wrong value stored in a variable is often the cause of unexpected behavior: object states are typically investigated when a bug occurs. Understanding object states and their changes over time is a central part of program comprehension tasks when debugging, extending, or maintaining the implementation of a software system [2]. Although object state analysis is a crucial step to comfortably carrying out programming activities, efficiently supporting object state reasoning has received scant interest from the software engineering research community [3].

In this paper we present a novel approach to visually exploring object state changes during a program execution. This analysis occurs by "snapshotting" objects during a program execution, and after the application execution, these object revisions are visualized and analyzed. The Object Evolution Blueprint represents each object as a list of contained attributes. Figure 1 gives the blueprint of an object of the class `RTKiviatBuilder`. In total, 153 revisions of that object have been produced. Each of the three variables of the class `RTKiviatBuilder` is represented as a diagram showing the evolution of the variable along the object revision from left to right. Figure 1 shows, encoded in the colored boxes, that the `view` variable was null in the early object revisions and was then sporadically modified. The variable `kiviatMetrics` held a collection which has grown to reach a size of 8. Similarly, the collection held by the variable `kiviatNodes` has grown to reach a size of 3. We also see that the filling of `kiviatNodes` occurred after the filling of `kiviatMetrics`. The blueprint supports three kinds of variable diagrams: numeric variables, collections, and a generic diagram for all other types. Interactions allow for further investigating details of the object state and focusing the analysis on time spans of interest. Our blueprint is implemented in Pharo [4], an object-oriented language and programming environment, and is seamlessly integrated with other programming tools. The graphics are rendered with the Roassal visualization engine[1]. We observed three participants to solve two exercises. We also analyzed answers of some questions regarding the usability and practicability of the blueprint.

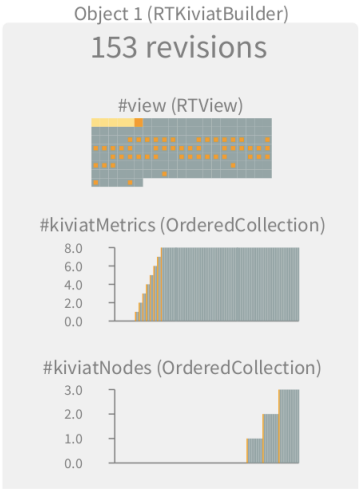

Fig. 1: Object Evolution Blueprint example

II. Object Mutation Meta-Model

This section presents the meta-model we use to create object revisions and represents the evolution of their values. During a program execution, a model is created and filled with information described by our meta-model. After the program execution, the Object Evolution Blueprint visualizes the model describing the evolution of object variables.

Consider the trivial program in SmalltalkLite syntax [5]:

[1] http://agilevisualization.com

```
class RTMondrian {
  view
  addElement(e) { view.add(e) }
  reset() { view = new RTView }
}

c = new RTMondrian
c.reset()
c.addElement(5)
c.addElement(7)
```

The class `RTMondrian` has one variable `view` and two methods, `addElement(e)` and `reset()`. The execution of the script creates an instance of `RTMondrian` and sends three messages to it: `reset()`, `addElement(5)`, and `addElement(7)`. Each message modifies the `view` variable: the message `reset()` initializes the variable `view` with an instance of the class `RTView`, and the two calls of `addElement(...)` fill `view` with two elements. Formally, we represent the succession of these three messages as an ordered collection of three tuples:

$\{c,\ view,\ nil,\ v_1,\ reset(),\ t_1\}$,
$\{c,\ view,\ v_1,\ v_2,\ addElement(5),\ t_2\}$,
$\{c,\ view,\ v_2,\ v_3,\ addElement(7),\ t_3\}$

The c variable refers to an instance of `RTMondrian`. The three values v_1, v_2, and v_3 represent the three different states of the variable `view`: v_1 is the empty view, v_2 is the view with the element 5, and v_3 is the view with elements 5 and 7.

Each tuple $\{o,\ variable,\ previousValue,\ newValue,\ message,\ time\}$ describes a revision of the execution that captures the change of *variable* for the object o. A complete object snapshot is a set of tuples. Before sending *message*, *variable* had the value *previousValue*. *Message* changes the value of *o.variable* to *newValue*. If *message* changes several variables, then several tuples are generated. A tuple represents a timestamped revision. The timestamp is given by the component *time*. A global repository contains all the object revisions and may be queried.

The previous example illustrated three mutations of the object c. We say that an object o mutates when receiving the message m if at least one variable is modified. An object mutation reflects a state modification in that object. Specifically, we consider a variable v is modified if a new value is given to v or the object referenced by v has a new value in one of its variables. This means that, if `v = 5`, then executing the expression `v = 6` modifies the variable `v` by giving a new value to it. Further, if `v = new OrderedCollection`, then `v.add(5)` modifies `v`. Note that this second example does not assign a new value to `v`, instead it modifies the object referenced by `v`. Note that an object revision may contain mutation from its previous revision. After the program execution, Object Evolution Blueprint visually represents object and variable mutations.

III. Object Evolution Blueprint

Object Evolution Blueprint is a visual representation of a particular object, based on the set of tuples previously described. Figure 2 gives an example of one object represented as a gray box. The object is an instance of the class `RTMondrian` (Mark **A** on the figure). The blueprint represents 42 different revisions of

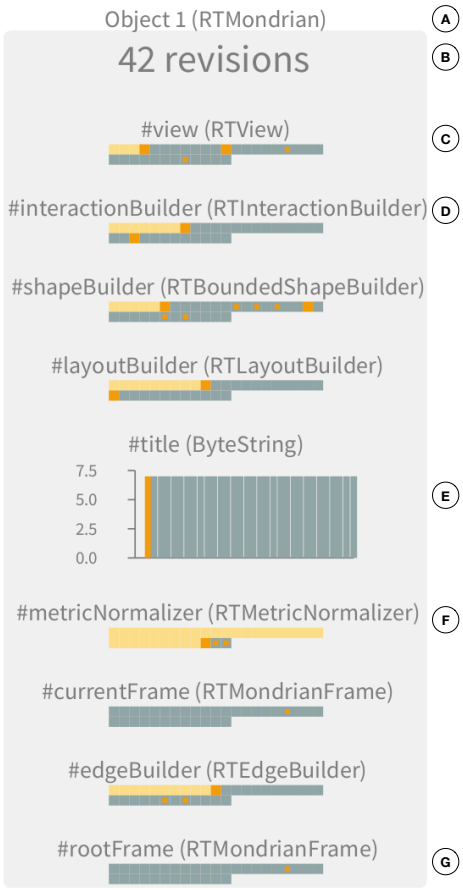

Fig. 2: Generic diagram example

the object (Mark **B**). The object has 9 instance variables in total. Variables are vertically lined up, ordered according to the class definition. The evolution of a variable over time is represented by a diagram. The type of the variable is used to select a suitable diagram. Three different diagrams are supported in our blueprint.

Generic diagram. This diagram is used when the type of a variable is not a number or a collection. In Figure 2 all the variables except `title` are represented using a generic diagram. Each data point contained in the time series encodes an object revision. We use the following glyphs:

- : The variable has no value, it is therefore null.
- : The revision assigns a new value to the variable.
- : The object referenced by the variable is modified, but the object identity remains unmodified.
- : The variable and referenced object remain unmodified.

Consider the `view` variable (Mark **C**). During the first three object revisions, the variable was null. At the fourth revision, a value has been assigned to it. Another value is assigned a few revisions afterwards. The referenced value is then modified twice, without changing the value of `view`. Visualizing the evolution of variables allows one to compare different variables. For example, the initialization of variable

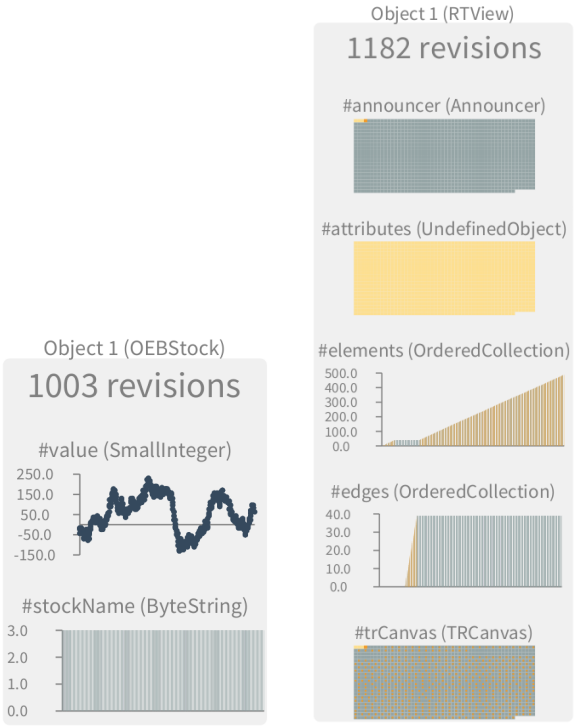

Fig. 3: Numerical and collection diagram example

`interactionBuilder` occurs after the initialization of variables `view` and `shapeBuilder` since the first ■ event happens later (*i.e.,* located on the right in the diagram).

Numerical diagram. In the case that the type of a variable is a number, a numerical diagram uses a chart to represent the evolution of the numerical values. Figure 3 (left) illustrates the evolution of an object of the class `OEBStock`, being part of a financial application. In total 1,003 revisions of the stock object were produced. The `OEBStock` class defines two variables, `value` and `stockName`.

Collection diagram. Analyzing the evolution in size of a collection is the purpose of the collection diagram. Figure 3 (right) represent 1,182 revisions of a `RTView` object. The variables `elements` and `edges` have the type `OrderedCollection`. The diagram shows the size of the collection over time and indicates changes of the collection itself, whether an object contained in that collection has been modified, replaced, added, or removed. Collection modifications are indicated using the color orange (■). The figure shows that the initialization of the `elements` and `edges` variables went through three distinct phases: (i) the collection `elements` is first filled while `edges` is empty, (ii) `edges` is then filled with 40 different entities while `elements` remains unmodified, (iii) `elements` is filled with 500 elements and `edges` is constant.

Interactive Exploration. Object Evolution Blueprint offers a range of interactions to support data navigation and obtain details on demand. *Brushing and linking* facilitates users to identify the exact variable value at a given moment in time. Locating the mouse above a diagram highlights all the variable value snapshots at the pointed time. This allows one to get a complete revision of the object at a given moment in time. The whole visualization may be zoomed in and out. This function is convenient in case many objects are simultaneously represented. Object Evolution Blueprint is meant to be used by practitioners to address some debugging- and comprehension-related issues. It is therefore crucial to have the visualization (i) easily accessible within the programming environment and (ii) any visualized entity should be one click away from the related source code.

Usage. There are two ways to generate a blueprint. The first way to produce a visualization is to let the programmer locate snapshotting of objects in the source code. This is achieved by sending the message `oLog` to an object. Sending the `oLog` message to any object creates an object revision which is kept in a global repository, as described in Section II. The second way to generate a visualization is to use a dedicated code execution profiler. The profiler may be configured with classes for which their instances have to be snapshot: each object, instance of the specified class, that receives a message is snapshot. The profiler may be configured in several different ways to filter irrelevant data gathering. For example, a condition may be set to exclude particular objects from being recorded.

Multiple objects. Our blueprint is designed to support a detailed analysis of individual objects. Several objects may be represented and traditional navigation facilities are supported, including zooming in/out, searching, scrolling. Currently, our blueprint does not support analysis across a massive number of objects. Instead, it supports local and fine-grained analysis of individual objects. Note that our design concurs with traditional breakpoint debugging approaches to only consider a restricted number of objects.

IV. Evaluation

This section evaluates our blueprint used by three practitioners, including one PhD student and two professional software engineers, in a qualitative study.

A. User Study Design

Each participant evaluated two program executions and answered five open questions. Our session was designed to be relatively short, with 30 minutes as the maximum allowed time to complete the whole experiment. First, we asked some questions about the participant's personal experience. Then, we provided a description and some illustrations of Object Evolution Blueprint as learning material, also introducing how to manually create object revisions (we focused on this usage mode in the evaluation). The two tasks of the study were:

- Exercise 1: We provided a short executable script to render a visual treemap for a hierarchical data set. The objective of this exercise was to analyze some individual variable diagrams. A participant will therefore evaluate the evolution of some particular variables, independently. The questions we asked are: (Q1) Which variables have

TABLE I: Result of the exercises

Participant	Exercise 1				Exercise 2			
	Q1	Q2	Q3	Time	Q4	Q5	Q6	Time
P1	1	0	0	15	0	1	1	12
P2	1	1	1	13	0	1	1	12
P3	0	1	0	13	0	1	0	7
Total	**2**	**2**	**1**	13	**0**	**3**	**2**	10

no value assigned during the whole execution? (Q2) One variable always receives new values, which variable is it? (Q3) One variable is sometimes modified and sometimes receives a new value, which variable is it?

- Exercise 2: We provided another script that involves the Sugiyama graph layout algorithm [6]. The objective of this exercise is to compare particular individual diagrams. The questions we ask are: (Q4) How many times is the variable `x` modified? (Q5) Which of the 3 variables, `x`, `y`, and `z`, is initialized first? (Q6) Which are initialized at the same time?

Finally, we asked five questions to formalize the impression left by the blueprint. The answers assess the overall usefulness and the perception of the blueprint left to the participants.

We have considered two different exercises to cover two complementary aspects of the blueprint: characterizing an individual variable, and relating a group of variables. For each exercise we provide an entry point of the code. This entry point may be considered by the participant as the root of the execution call graph for the exercise.

B. Results: Exercises

On average, the participants took 13 minutes to complete the first exercise and 10 minutes to complete the second. Each exercise had three questions. We gave 1 point per correct answer. We determined the correct answers from a careful analysis of the code. Table I summarizes the result of the exercises. The user study shows that all but one question received at least one correct answer. Question 4 was incorrectly answered three times, which indicates that the participants were not able to identify the number of times variables were modified. A variable modification is represented with a particular color in a variable diagram. The participants were not able to identify the correct location where to insert the revision creation, which resulted in them missing the relevant object revisions. Question 5, about identifying the variable that is initialized first, was correctly answered by the three participants.

Participant P1. The participant started to execute the script to see its effect. The participant then used the debugger to follow the control flow and identify locations where to create the object revision (*i.e.,* inserting the call to `oLog`). These locations were essentially identified by operating the classical debugging operations to follow the program execution control flow. When deemed relevant, the participant captured the object revision. The fact that the participant naturally used the debugger indicates that the blueprint is not meant to replace the debugger.

Participant P2. The participant did not use the debugger in both exercises. Instead, the participant solely navigated in the source code and used the Object Evolution Blueprint to answer the questions. He browsed the code and looked for the implementor and sender of message call to obtain a mental overview of the call graph. Calls to create object revisions are then inserted later on. Note that the participant qualified himself as experienced with the Pharo debugging facility.

Participant P3. Similarly to P2, P3 manually searched for the entry point of where to capture the object revision.

We did not find any indication that the participant misunderstood the blueprint. Pharo offers the inspector tool [7] to inspect the value of each instance variable. Participants P1 and P2 opened the inspector on previous object snapshots. This suggests that easily jumping from the blueprint to the actual object inspector is a valuable asset of our blueprint implementation. Participants regularly brushed diagram data points to obtain the name of the message in which the snapshot was taken, in particular, during the second exercise.

C. Results: Open Questions

Do you consider the tasks as realistic program comprehension tasks? We asked the participants to use a Likert-type scale to answer this question (*i.e.,* a scale from 1 to 5, where 1 means totally disagree and 5 means completely agree). Participant graded our set of exercises with 4, 4, 3. P1 commented that "the tasks are low level, so the participant is not challenged to have a wider comprehension."

Do you feel you can answer these questions using the classical textual logging and debugging facility of Pharo? Using the same scale, participants answered 3, 4, 2. Pharo offers an expressive API to reflect on the execution of an application. P1 had the feeling that fully using Pharo's reflective capabilities would probably help in completing the exercises. P3 said that the blueprint eases the answering of the questions.

Do you see one or more benefits in using the Object Evolution Blueprint? If yes, which ones? P1 saw a benefit to using the blueprint to analyze state changes. However, P1 is wondering about the scalability of the whole approach. P2 said that the evolution of each variable is apparent. There is therefore less information one has to remember when performing a comprehension task. In addition, several variables may be compared at once. P3 said that the visualization is useful for identifying singular points in the execution of an application.

For what purpose do you think the visualization is most useful? P1 commented that it is useful for addressing and identifying related events. P2 said that the visualization is useful for (i) identifying variables receiving unexpected values, (ii) crafting a cache for a variable, (iii) giving a visualization of a historical evolution of the state, being useful when the application has crashed.

Would you like to use a visualization like ours in your software development activities? P1 and P3 answered affirmatively. P2 is unsure what to answer since P2 does not feel familiar

with the visualization. It is therefore difficult to identify a recurrent situation where the blueprint is beneficial.

V. Related Work

To date there has been little work on visually exploring object mutations. Alsallakh et al. [8] developed an approach for Java code that, among other things, visualizes the evolution of generic variables in a timeline visualization discerning null values from others The approach also includes a specialized visualization for numeric variables. However, the approach does not highlight changes in the generic visualization and does not present variables in context of the respective object. Beck et al. [9] discuss how visualizations of evolving numeric variables can be integrated into the source code view as word-sized visualizations.

Other approaches focus on visualizing program execution as well, but from a structural or architectural perspective, for instance, recording and showing on a timeline UML-like sequence diagrams [10], [11], [12], stack traces [13], or dynamic call graphs [14]. General profiling tools represent runtime information on method levels in lists or call trees. Although these approaches offer insights into the temporal evolution of program executions, the state of variables and objects only play a secondary role if any at all.

Moreover, there exist specialized visualization approaches for debugging. Some show the program state at a breakpoint, such as DDD [15], ZStep 95 [16], deet [17], or reference patterns in Jinsight [10]. Exploring the evolution of state is possible by interactively stepping through the code. It might also help debugging to visually compare data structures of alternative program runs [18]. Typical means of visualization are linked nodes in a tree structure [15], [16], [17], [10] or nested boxes [15]. Visualizations can also be tailored to specific data structures like lists and arrays [19], [20], [21], trees [19], [22], [23], images and other media [22], [21], [7], or maps [7]. Most approaches are extensible by user-defined visualizations of other data structures. Since we want to give an overview of the evolution of program state, however, we do not use breakpoints and stepping.

In general, object-centric debugging [3] is also related to our approach. It switches the focus from the runtime stack to objects because debugging questions of developers are often more related to objects—we apply the same paradigm here.

VI. Conclusion

Assessing the evolution of individual objects during a program execution is a difficult activity. This paper describes Object Evolution Blueprint as a visual support to characterize and comprehend object mutations during an object life-time. Our approach has been implemented in the Pharo programming environment. However, our design does not rely on any particularities of Pharo.

The blueprint has been evaluated using a user study, and its usefulness has been illustrated via three participants. The results of two exercises and the participant perceptions indicate that the blueprint is useful for addressing particular situations in program comprehension.

Acknowledgment

We thank Renato Cerro for his comments on an early draft. Fabian Beck is indebted to the Baden-Württemberg Stiftung for the financial support of this research project within the Postdoctoral Fellowship for Leading Early Career Researchers.

References

[1] D. Marinov and R. O'Callahan, "Object equality profiling," in *Proceedings of OOSPLA'03*.

[2] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings of FSE'06*.

[3] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *Proceedings of ICSE'12*.

[4] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Square Bracket Associates, 2013.

[5] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts, "Stateful traits and their formalization," *Journal of Computer Languages, Systems and Structures*, vol. 34, no. 2-3, pp. 83–108, 2008.

[6] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[7] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel, "The Moldable Inspector," in *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2015.

[8] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch, "Visual tracing for the Eclipse Java Debugger," in *Proceedings of CSMR'12*.

[9] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf, "Visual monitoring of numeric variables embedded in source code," in *Proceedings of VISSOFT'13*.

[10] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang, "Visualizing the execution of Java programs," in *Software Visualization*. 2002.

[11] W. De Pauw and S. Heisig, "Visual and algorithmic tooling for system trace analysis: a case study," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 1, pp. 97–102, 2010.

[12] ——, "Zinsight: A visual and analytic environment for exploring large event traces," in *Proceedings of SOFTVIS'10*.

[13] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multi-threaded software systems by using trace visualization," in *Proceedings of SOFTVIS'10*.

[14] F. Beck, M. Burch, C. Vehlow, S. Diehl, and D. Weiskopf, "Rapid serial visual presentation in dynamic graph visualization," in *Proceedings of VLHCC'12*.

[15] A. Zeller and D. Lütkehaus, "DDD—a free graphical front-end for UNIX debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[16] H. Lieberman and C. Fry, "ZStep 95: A reversible, animated source code stepper," in *Software Visualization: Programming as a Multimedia Experience*. Cambridge, MA, MIT Press, 1997, pp. 277–292.

[17] D. R. Hanson and J. L. Korn, "A simple and extensible graphical debugger," in *Proceedings of the USENIX Annual Technical Conference*, ser. ATEC. USENIX Association, 1997, pp. 183–174.

[18] D. Abramson and R. Sosic, "A debugging tool for software evolution," in *Proceedings of WCRE'95*.

[19] J. H. Cross II, T. D. Hendrix, D. A. Umphress, L. A. Barowski, J. Jain, and L. N. Montgomery, "Robust generation of dynamic data structure visualizations with multiple interaction approaches," *ACM Transactions on Computing Education*, vol. 9, no. 2, p. 13, 2009.

[20] B. Alsallakh, P. Bodesinsky, S. Miksch, and D. Nasseri, "Visualizing arrays in the Eclipse Java IDE," in *Proceedings of CSMR'12*.

[21] D. Rozenberg and I. Beschastnikh, "Templated visualization of object state with Vebugger," in *Proceedings of VISSOFT'14*.

[22] Y.-P. Cheng, J.-F. Chen, M.-C. Chiu, N.-W. Lai, and C.-C. Tseng, "xDIVA: a debugging visualization system with composable visualization metaphors," in *Companion to OOPSLA'08*.

[23] A. Chiş, T. Gîrba, and O. Nierstrasz, "The Moldable Debugger: A framework for developing domain-specific debuggers," in *Proceedings of SLE'14*.