A New Generation of CLASS BLUEPRINT

Accepted at VISSOFT 2022

Nour Jihene Agouf[1], Stéphane Ducasse[2], Anne Etien[2], Michele Lanza[3]
1: Arolla and Univ. Lille, CNRS, Inria, Centrale Lille
2: Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL — 3: Software Institute, USI Lugano, Switzerland

***Abstract*—In object-oriented programming, classes are the primary abstraction mechanism used by and exposed to developers. Understanding classes is key for the development and evolution of object-oriented applications. The fundamental problem faced by developers is that while classes are intrinsically structured entities, in IDEs they are represented as a blob of text. The idea behind the original CLASS BLUEPRINT visualization was to represent the internal structure of classes in terms of fields, their accesses, and the method call flow. Additional information was depicted using colors. The thus created visualization proved to be an effective means to support program comprehension. However, a number of omissions rendered it only partially useful.**

We propose CLASS BLUEPRINT V2 (in short BLUEPRINTV2), which in addition to the information depicted by CLASS BLUEPRINT also supports dead code identification, methods under tests, and calling relationships between class and instance level methods. In addition, BLUEPRINTV2 enhances the understanding of fields by showing how fields of super/subclasses are accessed. We present the enhanced visualization and report on a first validation with 26 developers and 18 projects.

***Index Terms*—Visualization, program comprehension, code quality.**

I. INTRODUCTION

Understanding application logic is a time-consuming task during maintenance and software evolution. Researchers report that over half of the maintenance time is spent on reading and understanding source code [1], [2],where developers pore over source code, looking for clues that help them to construct a coherent mental model of a system [3], so as to make appropriate changes while ensuring its quality [4]–[6].

This is a difficult undertaking for any programming language, however maintaining and monitoring the quality of an object-oriented system is more complex than for procedural programs [7], [8], due to several reasons, such as inheritance and polymorphism [9]–[12]: Inheritance and polymorphism increase the flexibility of programs by allowing dynamic binding of messages. Inheritance allows the extension of an existing behavior through an inheritance hierarchy; polymorphism the performance of a task in multiple forms, with different objects responding to messages with the same name but different implementations. What can and should be considered a strength of object-oriented languages de facto hinders program comprehension [11], [12]. Dynamic binding of messages leads to more complex traceability of the call flow of a program since the type of the object receiving the message is determined at runtime. To follow the call flow of a program, developers proposed several approaches, including integrated development environments (IDE) and debuggers [13].

However, the usage of such tools is often too fine-grained, and thus time-consuming. Software visualization can provide a graphical view of a piece of software rather than a sequence of source code text. To this end, researchers proposed several visualization approaches, both in 2D [14] and 3D [15], [16]. Lanza and Ducasse [17] proposed the CLASS BLUEPRINT visualization to help developers get a "taste" of the class. CLASS BLUEPRINT presents the internal structure of classes in terms of fields, their accesses, and the method call-flow. Additional information was represented using colors. The authors classified classes based on their internal structure [18]. Regardless of its effectiveness, it did not display some up-to-date properties of object-oriented programming.

We present BLUEPRINTV2, an extension of the CLASS BLUEPRINT visualization based on updated requirements for program understanding. This approach discerns it from other (visualization) techniques that focus on views of sequential text, by offering a technical portrayal of the class per se. BLUEPRINTV2 supports the identification of dead code (single and branch), methods under tests, and call flow between instance and class (static) methods. It also enhances fields understanding by showing how fields of super/sub-classes are accessed, as well as lazy initialization in a compact form. In addition to hook understanding from a superclass point of view. After detailing the principles behind BLUEPRINTV2, we discuss its in-vivo validation with developers.

II. LIMITS OF CLASS BLUEPRINT

The CLASS BLUEPRINT visualization was created to help developers understand class structures [17], [18]. It decomposes classes into layers representing the invocation sequence going through external, internal, and accessor methods. This decomposition into layers organizes the method call-graph, and allows one to see which attributes are accessed by which methods, directly or through their accessors (see Figure 1).

CLASS BLUEPRINT has a number of limitations:

1) *Tests*: does not show whether a method is covered by a test: When CLASS BLUEPRINT was initially developed, continuous integration, version control, and tests were not common development practices. Hence, CLASS BLUEPRINT did not take into consideration test classes or test methods. Today, tests are the gate towards a better and faster understanding of the software and ensure software robustness. Easily distinguishing tested and untested methods is an important view of a class.

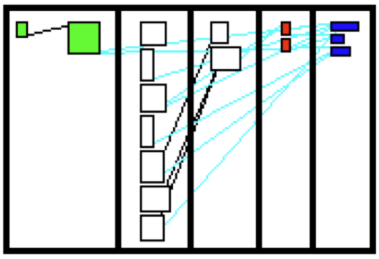

Fig. 1. A class blueprint from [18] with 5 layers: initialization, interface, internal implementation, accessor and attribute.

2) *Dead Code*: It does not heed dead code, *i.e.,* if a method is used or not: CLASS BLUEPRINT represents all methods of a class; even if these methods are dead. However, dead code hinders the understanding of classes [19], [20]. Yamashita *et al.,* report that dead code detection is desired by software professionals [21]. It avoids project practitioners to waste time on reading, understanding, and maintaining irrelevant code [22]. The previous version of the CLASS BLUEPRINT did not identify nor separate dead code from the used code. As such it misses an opportunity to provide information about effective class functionality.
3) *Instance vs Static*: The interplay between instance and class side (static) is not well supported. This is because all methods are classified into four layers only, concealing the details about such property in the class structure.
4) *Cyclomatic Complexity*: A method cyclomatic complexity is not revealed: Beside lines of code, CLASS BLUEPRINT gave limited information about the code quality such as method complexity [23], [24]. The cyclomatic complexity measures the number of linearly independent paths of a method [25]. It is of interest to practitioners responsible for the maintenance activities since methods with high cyclomatic complexity tend to be more difficult to understand and thus to test and maintain.
5) *Layer*: The accessors layer takes an unnecessary amount of space for a poor return of information. This layer covers the direct accesses of methods to attributes since the connection from a method layer to an attribute layer passes by the accessors layer.
6) *Hooks*: The reader has no clue if a given method is a hook in the system (being already defined in the subclasses), and in general CLASS BLUEPRINT largely disregarded the inheritance context. CLASS BLUEPRINT did provide an inheritance perspective to highlight the inheritance relationships of a given class with its ancestors and descendants: The class blueprints of the different classes of the hierarchy are gathered inside a single visualization. They are linked through inheritance, but also access, when methods of the subclasses use attributes defined in a superclass or invocation in case a method of a subclass, invokes one inherited method for example. In practice, this representation with several CLASS BLUEPRINT visualizations leads to complex visual elements and makes the understanding more challenging.
   Moreover, on the one hand, CLASS BLUEPRINT succeeded in conveying the information about method redefinition, using colors to distinguish between extending and overriding methods. On the other hand, the CLASS BLUEPRINT failed to stress out the methods redefined in the subclasses (overridden methods). Such information is important as it indicates the redefined methods and shares the global interior structure of the class and its relation with its subclasses. Spotting overridden methods allows the user to detect interface classes defining a generic behavior. This makes it easy for developers to understand the hierarchy from the studied class without having to read the source code of each and every subclass.

Our goal was to make up for all the above limitations, proposing a revisited and modern class blueprint visualization.

III. CLASS BLUEPRINT V2

As its ancestor, BLUEPRINTV2[1] focuses on individual class structure visual representation. It supports the understanding of each class separately. The focus is put on methods call-graph and how methods access to the studied class attributes in addition to the superclass attributes. Methods and attributes are represented as nodes whose color maps some semantic information. In addition, the node size conveys some properties: The height of a method node reflects the number of lines of code, and the width the number of outgoing invocations. As for the attributes, the node height corresponds to the number of direct accesses from methods within the class, the width concerns the number of external accesses from methods of the same hierarchy of classes. To ease the understanding, methods are positioned in different columns depending on their interactions with methods of other classes, or their nature (such as initializer, internal, getter, etc). BLUEPRINTV2 adds the following features (see Figures 3 & 4):

- *Static (or class side) entities* are now displayed. They are placed in a dedicated area on top of the visualization and separated from the instance side methods. The calls between the class and instance sides are represented. Class attributes are also represented in a distinct area in the top right corner.
- *Dead branches* are identified and separated in a specific layer at the bottom of the visualization (the cemetery). Dead attributes as well as their accessors are also separated at the bottom right corner.
- *Getters and setters* are merged into annotations around attributes to gain space. In addition, lazy initializers are handled as a kind of accessor.

[1]The link to the GitHub repository of the visualization can be found here: https://github.com/NourDjihan/ClassBlueprint. All the instructions on how to use the visualization are explained in the Readme.

- *Superclass state usage*: access to superclass attributes is represented in a separated area on top of the one of instance side attributes layer. Accesses to superclass state from local methods are represented.

These points are described in the following subsections.

A. Layers

As its ancestor, BLUEPRINTV2 classifies methods in different layers represented in columns.

a) Vertical Layers: First, on the left, the *initialization* layer gathers the methods responsible for object creation and initialization (*e.g.* initialize methods in Pharo and constructors in Java). Then comes the *external* layer containing methods that compose the external interface of the class. Such methods are either invoked by methods of the initialization layer or declared public or protected in languages supporting modifiers, or invoked by methods outside the class. In third comes the *internal implementation* layer representing the core of the class, *i.e.,* methods that are not supposed to be exposed to the outside of the class. It contains for example private methods or methods invoked by other methods of the same class. We removed the *accessors layer*: the setter and getter methods are on top and below their attribute, respectively (see Figure 2). It compacts the visualization without losing information: the developer can, at a glance, see if the attribute has a getter or a setter, is directly accessed, or is used through its accessors (if present).

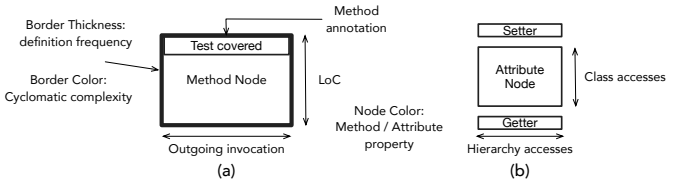

Fig. 2. Sketch of nodes: (a) For methods, size, border thickness and color convey information. (b) For attributes, size and color.

b) Horizontal Layers: In addition, in BLUEPRINTV2 the visualization has been split into 3 horizontal layers.

The *top* layer corresponds to the *class* side (static in Java). It gathers on the left the methods connected within a call-graph and on the right the *class side attributes*. In Java, the class side is specified with the static keyword. In Pharo, these are the entities of the metaclass.

The *bottom* layer corresponds to dead code. Once again, methods and attributes are separated (methods on the left, attributes on the right). It is possible to see a call-graph in the dead code layer. Indeed, a method is considered dead if it is not called in the project or if it is called only by dead methods. Consequently, dead branches can also be identified. This definition of dead code may lead to false positives, in particular in the case where API methods are called by external projects, not under analysis.

The *middle* layer corresponds to instance side methods and not dead code. It follows the vertical layer decomposition.

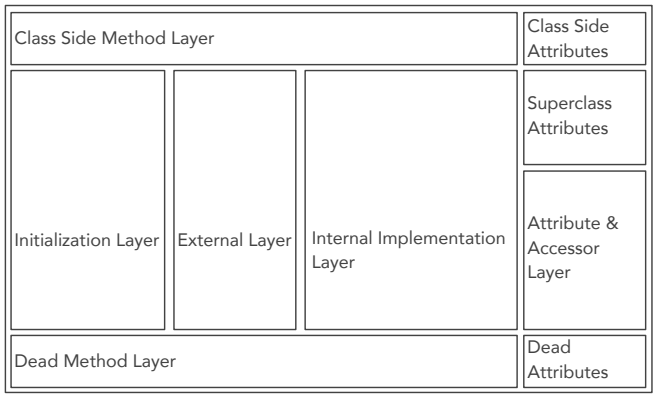

Fig. 3. Layout of CLASS BLUEPRINT V2: class level is on the top, instance level in the middle and dead entity on the bottom. The middle layer presents information via layers.

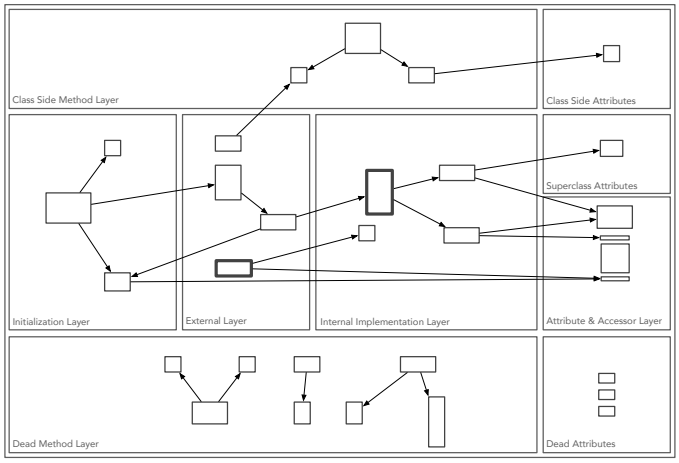

Fig. 4. A colorless BLUEPRINTV2 with the graphical representations of methods, attributes, and accessors using height and width metrics. The arcs represent calls between methods and accesses to attributes.

B. A class inside a hierarchy

BLUEPRINTV2 puts the studied class in the perspective of its hierarchy. For this purpose, first, the inherited attributes are separated from the ones defined in the studied class and put in a different layer than the local attributes layer. In addition, attributes accessed by at least one method in the subclasses of the studied class are colored in dark green whereas the other ones are colored in dark blue. This allows developers to see how the class is using a superclass state and if their state is directly used in subclasses.

Second and as in the initial CLASS BLUEPRINT visualization, extending methods (performing a super invocation) are colored in orange and overriding methods (method redefinition without super invocation) are colored in brown. In addition, in BLUEPRINTV2 we introduced a color to spot *overridden* methods (*e.g.,* methods redefined in subclasses) are colored in pale orange. This information gives an idea of how the methods of the class are extended. In addition, to indicate if an abstract method is redefined in the subclasses, they are

marked with a pale orange square (In Pharo, classes may have abstract methods not redefined in the subclasses, moreover even concrete ones).

C. Additional Indications

In addition to the extensions presented above, BLUEPRINTV2 introduces other new features:

- *Cyclomatic complexity.* To complement the Lines of Code and fan-out of the method, we indicate the method with high cyclomatic complexity by coloring their border in red. Indeed when a method has a cyclomatic complexity higher than a given threshold fixed to five in Pharo, its border color is red.
  The thresholds have been defined following JIT compiler practices [26]–[28]. They can be changed if needed as discussed in Section V-D.
- *Tested methods*. To easily identify if a method is called from a test method, in BLUEPRINTV2, tested method nodes have their superior third in green. Note that we only use static information: it does not indicate whether the associated test passes or not.
- *Dead entities*. An *attribute* is dead if it has no incoming accesses, meaning that no external/internal method or accessor is accessing it. A *method* is considered as dead if *the method is not invoked by other methods*. Note that an abstract method is dead only if the method itself is not called and none of its reimplementations in the subsystem are invoked. In addition, initialization and test methods are not considered dead methods to limit false positives.

Properties	**Description**
Constant	Gray
* Dead code (attributes or methods)	Black
* Invoked from a test	Green annotation
Extending	Orange
Overriding	Brown
* Overridden	Pale orange
Abstract	Cyan
* Abstract and Reimplemented	Inside pale orange square
Delegating	Yellow
Internal Implementation	Purple
* Test Method	Pink
Setter or Getter	Magenta
* Lazy Initializer	Olive
Accessed locally	Blue
* Accessed by subclass	Green

TABLE I
METHODS AND ATTRIBUTE PROPERTIES. * MARK NEW COMPARED TO CLASS BLUEPRINT

D. Interactions

BLUEPRINTV2 is automatically computed and displayed for a class. It is an interactive visualization which is included in a tool to enable interactions.

a) Access to the code: A classic mouse hover displays the method name as depicted in Figure 5-(B). This example illustrates the event of a mouse hovering a method node colored in white in the externals layer named treatString. Additionally, a Shift pressed with a mouse hover displays the method corresponding code. Figure 5 illustrates three examples annotated with (A), (C), and (E) displaying the source code of methods contained in each class. The StartStopMarkupBlock-(A) visualization shows the source code of an overridden class side method named isAbstract. The PhraseLibrary-(C) class presents the source code of the phraseFor: delegating method colored in yellow. Finally, the AuthorFocusedDocBuilder-(E) class with the source code of an abstract initialization method colored in cyan named extension.

b) Call-graph: To better follow the call-graph, a left-click on a method displays in red its outgoing invocations as shown in Figure 5-(D). The InlineParser class visualization shows the outgoing calls from the linkOrFigureProcess: method (with red border) to three methods in the same internal implementation layer, and its access to most attributes of the instance side (eleven attributes). These remain red until clicking again on the selected method. Thus clicking on another method adds new invocations in red and so on. It enables the user to better see and understand the call graph of the studied method in the scope of the class.

Similarly, it is possible to highlight the incoming invocations of a method with a right-click as shown for the location-MonthYear method in Figure 5-(D) (second purple method of the first branch on the left of the class side methods). The incoming invocations of a method and the calling methods are highlighted in green. Adding new methods in the call graphs and deactivating is also possible.

IV. BLUEPRINTV2 IN PRACTICE

This section presents some examples of the BLUEPRINTV2. Each example calls attention to the internal structure of the class visualizations depicted in Figure 5. The visualizations are grouped in one figure to gain space and to easily compare different class structures since the visualizations are put close to each other. Each visualization is annotated with a letter to ease reference hereafter.

A. A simple class

The visualization of the StartStopMarkupBlock class (from Microdown project) shows that it has few methods. An initialization method is colored in orange showing that it extends its super implementation. It has a thick border because it is a megamorphic[2] method called initialize. Three overridden methods are colored in pale orange on the class side, external and internal implementation layers. Two overriding methods are colored in brown each defining a new behavior in the externals layer. Moreover, two abstract methods colored in cyan with a pale orange square indicate the presence of reimplementations in the subclasses. The class contains a dead method colored in black at the bottom of the visualization. Only three methods are under test (as identified by the green node annotation).

[2]A megamorphic method is a method frequently named in the system *i.e.,* several methods have the same name.

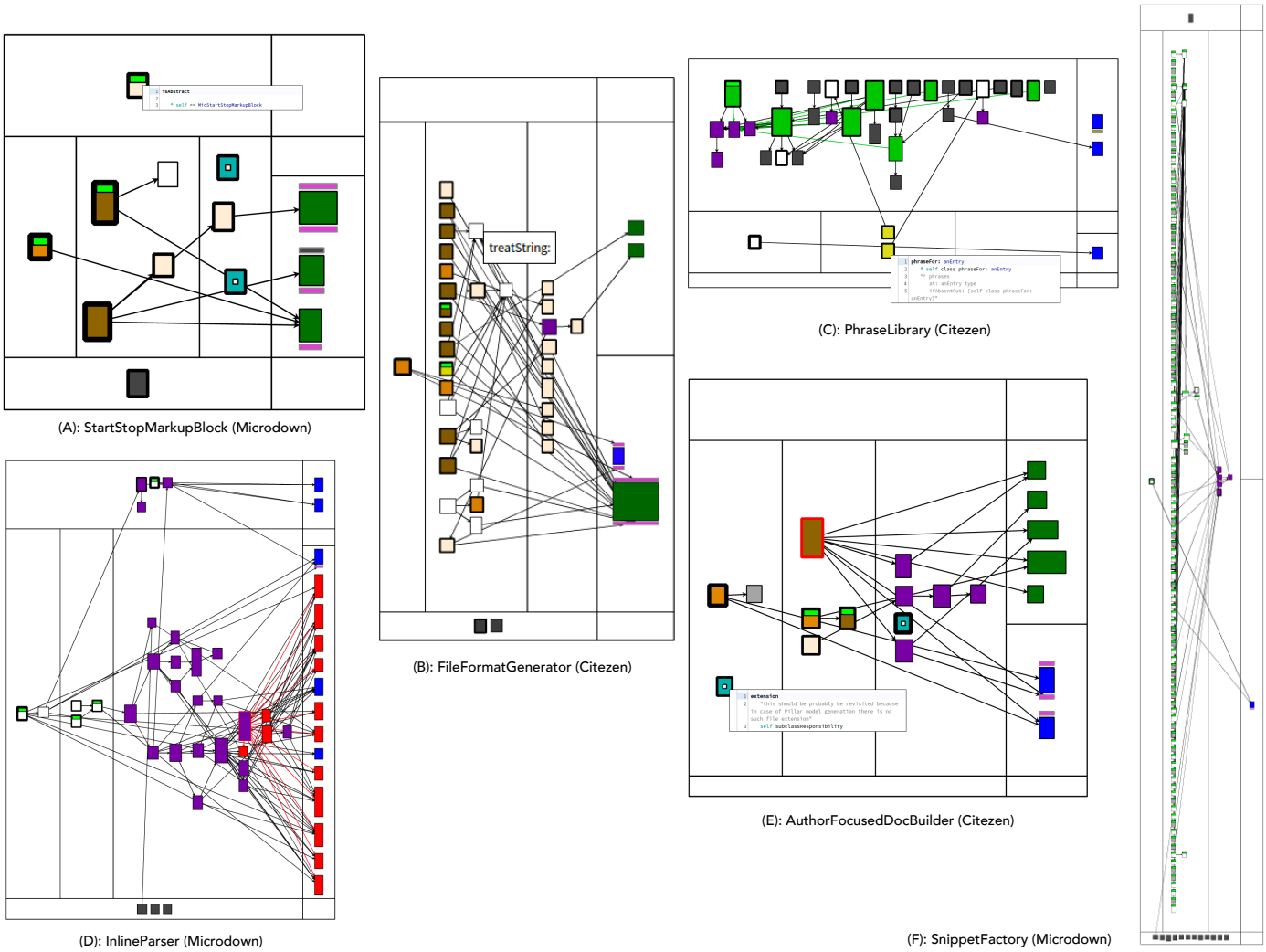

Fig. 5. Some examples of BLUEPRINTV2 on Citezen (a bib library) and Microdown (a markup language).

The class also contains three attributes each with both accessors however, these attributes are all accessed directly from the class methods without using their accessors, and colored in green meaning that they are accessed directly in the subclasses. Furthermore, the second attribute has a setter on top, colored in black indicating that the setter is not used in the class nor by methods of other classes, thus is dead. The other accessors are colored in magenta meaning that they are used by methods of other classes.

B. A class defining and redefining hooks

The FileFormatGenerator class (from the Citezen project) specializes in several superclass methods: it includes four methods extending an inherited behavior colored in orange and nine methods in brown overriding their superclass methods. Moreover, the class provides 13 new methods colored in pale orange overridden in the subclasses. With such data, we see that the class integrates well with its superclass and defines some important methods for its subclasses. Only two methods are tested (as indicated by the green annotation). The class accesses directly two attributes of its superclasses. One of its attributes (colored in green) is heavily used by its subclasses.

C. A static class: Playing a kind of factory/builder

The PhraseLibrary class from the Citezen project has an unconventional design. Most of its methods are defined on the class side. It acts as a kind of object factory composing kind of sentences. The PhraseLibrary has two instance side methods in the externals layer colored in yellow delegating to the class side methods for creational purposes. An example of a lazy initializer is depicted in the PhraseLibrary class visualization: the first attribute of the class side (on top), colored in olive under the attribute. The lazy initializer has no incoming calls inside the class however, its olive color indicates its use in other classes.

D. A large internal class with dead code

Classes such as the MicInlineParser class are referred to as *Single Entry* [18], indicating that the class has one or few entry points from its externals layer and a wide internal

implementation layer, in this case, composed of 22 methods and many invocations between these methods. Such classes are designed to deliver a one-complex functionality.

Yet, what draws attention to this class are the dead methods at the bottom layer of the visualization. Indeed, three methods inside this class are not invoked and not used in the project. Since dead code, when executed, is memory and time-consuming, this leads to believe that the class needs refactoring to improve its readability internally and reduce its complexity in the wholeness of the program.

E. Accessing superclass state

The AuthorFocusedDocBuilder class does not have class side nor dead nodes. Thereby, the call-graph of the methods inside the class appears clearly going from the left initializers, passing by the externals, the internals, and finally the attributes. This class contains many colored nodes, for instance in the initialization layer, a method colored in orange extends its superclass method and has direct access to both attributes of the class.

The externals layer contains a complex big method buildBody colored in brown with a red border, which directly accesses most attributes of the class except for the superclass attribute fieldOrder, and the instance side of attribute bodySpecification accessed through its getter. Such a case of direct access and through the accessor in the same class shows inconsistency in the design of the class. Nonetheless, the superclass attributes are all accessed by methods of the subsystem, hence their green color. The width of the first four superclass attributes gradually changes forming a cascade shape, this indicates which attribute is more accessed by methods of the subsystem.

F. A factory class with tested methods

Finally, the SnippetFactory class has an externals layer with 120 methods. This class is a holder of a ready to be used document elements that are essentially used by tests.

The externals layer contains constant methods in gray returning each predefined value and methods colored in white, meaning that they do not belong to the classifications defined in Table I. Such methods contain a small portion of source code, hence their small size where each is performing an operation and returning its result.

Moreover, almost all methods of the externals layer are annotated with green on top indicating that each is called by at least one test method. This class also has dead methods (bottom layer). Such dead methods are methods returning document elements that are not used by the tests. They are effectively dead methods.

V. Evaluation

This section explains the process of the evaluation of the BlueprintV2. We proceeded with both *qualitative* and *quantitative* evaluations. Such evaluations try to answer the following research questions:

- How does the visualization support the understanding of the class structure?
- How does the visualization help in assessing the quality of the class?
- How satisfied are the different profiles of participants with the proposed visualization?

A. Protocol

We invited developers from the Pharo community to participate in our research: We sent an email welcoming participants that would be interested in evaluating a new visualization. Twenty six developers joined our evaluation. Then, to explain and guide the participants through the experiment, we scheduled both group and individual meetings according to the participants' availabilities. If several participants were available at the same time we proceeded with a group meeting. On the opposite, we scheduled individual meetings. During those meetings, we first explained the visualization and its motivations, then described to the participants the process of the experiment:

1) Select a project they wish to analyze
2) Use the visualization on the selected project
3) Record their screen during the experiment
4) Write a report summarizing their findings
5) Fill the post-experiment survey

The meetings took from 10 minutes when individually, to 25 minutes in groups. This difference is principally due to the adaptation of the tool to the chosen project and the possible interactions between participants while in groups. Nonetheless, the content was the same for each participant.

a) Projects: The selected projects vary in number of packages, classes and methods. Table II summarizes this variety in projects. The same project could have been analyzed by

#Project	#Pkges	#Classes	#Median Methods	Domain
Avatar	2	18	6	Proxy
Sindarin	3	18	14	Debugging
MoTion	2	35	5	PM
Clap	5	47	8	Parsing
Slang	2	73	29	VM
Polyphemus	3	79	9	VM
AST-Core	3	101	21	DSL
Reflectivity	5	114	13	DSL
OpalCompiler	3	156	15	Compiling
Druid	1	170	12	VM
Seeker	2	236	9	Debugging
MooseIDE	16	250	8	Analysis
Polymath	60	309	11	Computing
Refactoring	12	378	6	Refactorings
AIPharo	85	424	6	AI
Roassal	39	445	12	Visualization
Iceberg	11	488	10	VCS
fylgja	73	941	15	Migration

TABLE II

THE NUMBER OF PACKAGES AND CLASSES IN EACH PROJECT, AND OF MEDIAN METHODS IN EACH CLASS, IN ADDITION TO PROJECT DOMAINS. IN THE TABLE, THE ORDER IS DESCENDING ACCORDING TO THE NUMBER OF CLASSES IN EACH PROJECT. THE ABBREVIATIONS REFER TO VM (VIRTUAL MACHINE), VCS (VERSION CONTROL), PM (PATTERN MACHINE), DSL (DOMAIN-SPECIFIC LANGUAGE).

more than one participant. For example, Iceberg was analyzed by two participants, MooseIDE by three, and OpalCompiler by two.

b) Participants: The participants of the experiment come from diverse backgrounds and have different statuses: interns, PhD students, developers and researchers. Consequently, their years of experience in programming also vary. Six participants have between [1-5] years of experience. Nine participants between [6-10] years of experience. Two participants have 14 and 15 years of experience each, and nine participants have over 20 years of experience.

In regards to the projects, the participants either used these projects before, developed them or are responsible for their maintenance. The expertise degree of the participants on the selected projects is let to their own appreciation. The survey showed that only 10% of the participants consider themselves newbies to the selected project, while 35% had some basic knowledge about the source code. Another 10% of the participants with advanced knowledge of the project and finally 45% of the participants are project experts.

B. Data Collection

Once we received the data (screen records, reports and survey filled by all participants), we watched the screen records looking for participants' behavior. Particularly, a focus has been put on (i) the participant's first impression in regards to the visualization, (ii) the most used features of the tool and (iii) if other tools have been used besides the BLUEPRINTV2 and what they are. We then, read the reports to have a better insight into their actual impressions and findings. Finally, we collected both data from the reports and the survey as explained in the next section.

C. Data Analysis

As mentioned prior, the evaluation of the BLUEPRINTV2 consists in two parts: qualitative and quantitative. The qualitative evaluation is based on the screen records and the reports sent by the participants. The quantitative evaluation is built upon the post-experiment survey.

a) Qualitative evaluation: When observing the screen records of the participants, it was brought to our attention that participants instinctively followed two complementary approaches: *Flight over* and *Plunge in*.

Flight over: The flight over consists of quickly navigating the visualizations of the classes one by one. It allows the user to visually detect the important classes and the less important ones based on the amount of information held by the visualizations (nodes and colors). It also allowed some users to detect:

- Empty classes: without methods nor attributes;
- Big classes: with lots of methods;
- Complex classes: with methods with high cyclomatic complexity *i.e.,* method nodes with red border;
- *Class-side* methods: with a considerable amount of method nodes placed in the class side layer;
- *Hook* classes: with noticeable hook nodes colored in pale orange;
- *Dying* classes: with several dead methods, or attributes, positioned at the bottom of the visualization;
- *Tested classes*: with tested methods *i.e.,* the green annotation in method nodes. More specifically, here, it is the proportion of tested methods that have been appreciated by the participants.

Plunge in: The plunge in consists of focusing on some classes for further investigations. This provides the user with a more in-depth analysis of the class internal structure, and the possibility to use more helpful interactions with the visualization such as the shift + mouse hover to quickly view the source code of the methods.

- *Code duplication*: some duplicated code was detected by several participants when method nodes have the same size and color, inside the same class or in other neighbor classes.
- *Complex classes*: most participants reported finding complex methods inside classes due to the size of the nodes or the red border color. They intend to divide them into small blocks of source code.
- *Dead method analysis*: some participants were surprised to find many dead methods. Such an outcome motivated them to inspect the senders of the "dead methods" using other tools than the visualization, (e.g., system browser and cross-referencer).
- *Dead code detection*: the detection of dead code was also surprisingly possible due to the length of the method nodes. One of the participants investigated long methods relatively used in his studied system and found a few which contained internal dead code which was not removed.
- *Commented methods*: some participants identified peculiarly long methods but not necessarily complex. Then, when navigating the source code of the method nodes (through a mouse hover), participants found that the length of the method node in fact reflects the long comment inside the method definition. One of the participants suggested adding an annotation at the bottom of the method node (since the top might be full with the test annotation), to demonstrate the presence of a comment.

b) Quantitative evaluation: Participants were asked to answer a post-experiment survey to give feedback about the visualization using a five Likert scale. We asked the participants if:

- *The visualization helps in understanding the code/state of a class is reused:* Most participants (46%) answered that they agree on this point, while others (15%) strongly agreed. However, over 23% said that they disagree, and 15% were undecided.
  Some participants in their reports mentioned that they appreciated the feature of adding the superclass attributes in the visualizations. This helped them understand which attributes of the superclass are used in the class under

analysis, and which methods are accessing them. Such a feature also reduces the time spent on understanding the relation with the superclass.

- *The visualization helped in understanding the reused code from the superclasses:* Over half of the participants (53%) were undecided, where 19% disagreed, 23% agreed, and 3% strongly agreed. Because some projects did not use inheritance, hence the visualization was not answering to this question which justifies the amount of indecision in this case.
- *Did the visualization help in understanding class/instance side communication*: Over 53% answered that they agree, and 19% strongly agreed. Other participants of 19% answered that they were undecided about this point, and a few percentage of 7% answered that they disagree.
  We believe that the classification of the class side methods on top and the instance side methods in the middle reduces the cumbersome of links between method nodes in the middle layers. Moreover, the participant who analyzed the *Microdown* project mentioned: "The MicAbstractDelimiter class shows the nice interplay between class and instance side methods".
- *Is the design of the class well summarized in the visualization:* Over 7% and 38% of the participants strongly agreed, and agreed, respectively. Among the participants over 30% were undecided and 23% disagreed.
  This is also understandable because the visualization summarizes the structural relationships and not the design of the class itself. In some cases, indeed the design is well shown for instance visitor classes, factory classes, and builder classes, etc. But not all design is reflected by the class blueprint. Nonetheless, one participant mentioned: "I was able to follow for each method clearly what are the methods of the same class that are injected inside of it", referring to the outgoing invocations between methods.
- *Does the visualization hep in detecting dead code?* Over 26% and 46% of the participants answered that they strongly agree and agree, respectively. However, 11% answered that they strongly disagree, 3,8% just disagree, and another 11% were undecided.
  All participants reported finding dead code in their projects. The participants who analyzed the same projects on the one hand found the same dead methods and on the other hand different dead methods. This is one of the reasons which motivated us to agree on analyzing the same project by two or three participants. Including one participant who eventually refactored his code and justified in his report: "The visualization also helped me quickly identify dead code and eliminate it. As this is a new project (early stage of development) I didn't remove all dead methods or classes, but in other kinds of projects I would do it". Another participant expressed: "Dead methods correspond mostly to unused code that I forgot to remove".
- *The visualization helps in detecting complex methods:* Almost all participants appreciated this feature, including 46% who strongly agreed and 38% who agreed to this assertion. The other participants were 11% undecided, 3% did not agree. The participant responsible for maintaining the Roassal project mentioned and we cite: "With the height representing the lines of code and with the red border. It was easy to find the complex methods in this class. This is an anomaly because they are long examples that maybe should be split into classes". He found long complex methods that represent examples of how to use Roassal.
  When reading the reports not all participants found complex methods in their projects, which might explain such a decision. However, several others mentioned the presence of complex methods in their projects, including some who consider investigating the complexity of such methods for correction purposes.
- *The visualization helps in identifying tested/untested methods:* Over half of the participants also appreciated this feature, including 34% who agreed and 34% who equitability strongly agree. The other 19% were undecided and only 7% disagreed and very few of 3% strongly disagreed.
  Some participants in their reports mentioned that the visualization helped them identify the weak spots (not tested methods) in their source code, which they intend to reinforce. The *Microdown* participant found this feature useful and we quote: "In the MicHTMLDoc class we could exclusively see the tested and untested methods".
- *The visualization is scalable for large classes:* Among the participants, 3% strongly agreed, and 42% agreed that the visualization was scalable for their classes. The other 30% were undecided, 15% disagreed, and 7% strongly disagreed.
  This question is also relative to the project (in case the project contains big classes), hence the diversity of answers. As with any visualization including nodes and connections between those nodes, the bigger the class is the harder it becomes to display all the pieces of information at once with a clear classification of the layers and connections between the nodes. Nonetheless, other participants found that the classification of the methods on the right and attributes on the left in big classes helped them to (i) better see the attributes and (ii) more clearly identify the access to these attributes, and (iii) better distinguish the class side methods.
- *The visualization is easy to use:* Three-quarters of the participants (76%) agreed that the visualization was easy to navigate and 15% strongly agreed. For the disagreed participants of 7%, the difficulties mostly come from the first interaction with the visualization and how to start using it.
- *They would like to use the visualization in the future:* Half of the participants (50%) wish to re-use the visualization in their future work and over 15% strongly agreed. Other answers include indecision with 26%, disagreeing, and strongly disagreeing with the same percentage (3.8%).

When looking for clues in the reports to understand the reasons behind indecision and disagreement, it was mostly because of big large classes. One of the participants who analyzed Iceberg reported that some classes contained more than 120 methods with hundreds of connections between method nodes.
Nonetheless, the most appreciated feature (mentioned in all the reports) is the dead code detection. Others also found very useful the interactions with the visualization (mouse hover to see the name of the method, shift + mouse hover to see the source code of the method, the double click to open the method in the system browser, the right-click to highlight the outgoing invocations and left-click to highlight the incoming invocations). While others found the colors very useful to have an understanding of the methods inside its system and their relation with the super/sub implementations.

D. Discussion

Very few participants reported finding dead attributes in their projects. Furthermore, some participants reported finding some false positives in dead methods. Such false positives are often due to methods that belong to an API and that are not called in the system under analysis. Other cases were in extension[3] methods from a package that itself is not part of the analysis.

Finally, the absence of the green test annotation in method nodes allowed users to consider reinforcing tests in these parts of the source code. Especially since tests measure the confidence that certain features are adequately implemented. Some participants reported missing tests meaning that their projects were not as expectedly well covered by tests and others added tests to cover such methods.

E. Threats to Validity

a) Internal Validity: To what extent we can draw a causal link between the treatment in the experiment and the response? Regarding the projects, they were selected by the participants according to their previous experiences. The participants either used or developed these projects before. They considered their level of knowledge of the projects as debutants, intermediaries, advanced, and experts.

The level of expertise in software programming has been collected for each participant. Concretely, the expertise varies from two to thirty years. Our experiment was not dedicated to a specific audience but to diverse profiles. We want to report that we performed a first attempt to evaluate the use of the visualization to reverse engineer unknown software with internship students (3rd year). Such an attempt was unsuccessful since most of the students did not have enough concerns about quality and good object-oriented design.

The project sizes vary from 18 to 941 classes. As explained in Section V-C, most participants flight over the whole or lot of classes and only plunged in the classes they found worthy of the analysis (important classes, big classes, etc).

Obviously, the answers depend on the project the participants choose and their expertise on the project or in software programming. However, the diversity of the projects and the diversity of experiences of the participants limit the threat of internal validity.

b) External Validity: Are our results generalizable for practice modernization? Even though the evaluation was only based on Pharo projects, the visualization also supports Java projects analysis. However, for this paper, we limited our choice to only Pharo projects since we have easy access to experts. We plan in future works to apply the visualization on Java projects when we have the possibility to interact with Java maintainers to better adapt the visualization or add new features according to their feedback if needed. Nonetheless, the approach itself can be applied to any object-oriented program. For other programming languages such as C++ and Python, the Pharo community does not have a parser yet.

c) Construct Validity: Are we asking the right questions? The answers to the questions, presented in the quantitative evaluation (Section V), may depend on the projects analyzed by the participants and their expertise. Some projects do not necessarily provide elements to answer all the questions. For instance, some projects do not commonly use inheritance, consequently, methods overriding, extending their super implementations, and methods overridden in the subclasses may be absent. Additionally, some classes do not have lots of class side methods. Thus, it may be difficult to observe the communication instance/class sides. However, we believe that the number of participants as well as the diversity of the projects and participants reduce the threats.

d) Reliability: To what extent can the results be reproduced when the research is repeated under the same conditions? The qualitative evaluation relies on the feedback of users according to their projects. The anonymity does not enable the reproduction of the research under the same conditions. Furthermore, we also suggest that the results of these evaluations vary with the projects and the experiences of the users even if we did our best to reduce this point by increasing the number and the diversity of users and projects.

Nevertheless, for reproducibility purposes, the tool and all the artefacts used in the evaluations are available online. With the tool, we provide full instructions on how to start using the visualization in the Readme. The legend of the visualization can also help the user through her navigation.

VI. Related Work

The visualization proposed in this paper is a patrimony work of the first Class Blueprint presented by Lanza *et al.,* in 2001. In Class Blueprint [18], the authors classified class methods into layers, where each method/attribute node is characterized by a color and a size. They, however, missed some important aspects in assessing software and detecting its vulnerabilities.

[3]In Pharo, a class can be extended by methods that are packaged in another package than the one of the class.

Visualizations facilitate program comprehension because they provide a graphical view of the software rather than an alphabetical sequence of source code text. To this end, researchers propose several visualizations, such as Hunter [29] which focuses on understanding the dependencies between software artefacts. In their article, Dias *et al.,* use colors to distinguish between file branches as well as the size of the node which corresponds to the number of lines of source code in each file. Such visual weight helps get an understanding of the software components and their relations more easily. Tamer's *et al.,* [30] propose a visualization to assess software quality and the dependencies of composed-based JavaScript React applications. Such visualizations also use the node-link diagram to demonstrate the connection between software components. Boccuzzo *et al.,* present CocoViz [31], a visualization inspired by daily life graphical elements (houses, spears, or tables) to convey information about the program components and the program vulnerabilities. However, these approaches focus on web-application components and not object-oriented programs.

Including visualizations that focus on object-oriented software quality and comprehension, the SeeSoft [32] tool visualization maps lines of code into thin rows, uses colors to display information about the timeframe where the code has changed. Scheibel *et al.,* [33] propose a treemap visualization depicting software data using areas, colors and nesting. Additionally, Lanza's *et al.,* present CodeCrawler [14] that supports the understanding of program structure via polymeric views. Wettel's *et al.,* present CodeCity [15] that displays software classes as buildings and packages as the ground foundation on which they are built [34]. Another city metaphor is the SArF map [16] clustering technique, which groups together classes with the same features on the same grounds, separated by a street representing the relevance between these features. Sazzadul *et al.,* propose EvoSpaces [35] which is a similar approach to the city metaphor but dedicated to C/C++ projects. It displays the architecture and metrics of software in 3D to help quickly understand the software under analysis and the relationship between files. They also offer a night view which displays the execution trace of the software. CityVR [36] is an interactive 3D visualization tool that implements the city metaphor technique using virtual reality. Another original metaphor for software visualizations is *Software Feather* [37] which maps class and interface metrics as feathers, one of its main purposes is for users to apprehend to recognize which feather corresponds to which class in the system. Another metaphor derived from nature is *Software Forest* where Atzberger *et al.,* [38] display software as a forest by mapping properties of entities such as the size and trend data. Furthermore, Ignacio *et al.,* [39] extend visualIDs as a glyph technique to cope with structural software elements. The authors use them to identify classes with the same dependencies and classes with a similar set of methods. Moreover, Churcher *et al.,* used 3D to visualize class cohesion [40]. Wuerthinger *et al.,* worked on a visualization that focuses on the dependency graph of Java projects [41]. Daniel *et al.,* [42] also worked on visualizing Java projects but focusing on the incremental exploration of a project. Dennie *et al.,* [43] present another approach called SolidSX, in contrast to the previous approaches, SolidSX offers a modular analysis of the structure, dependencies, and metrics. Erdemir *et al.,* [44] offer E-Quality, a graph-based visualization that extracts quality metrics and class relations from Java source code and thus does not extract quality metrics of the class body. Such solutions focus on the overall conceptual analysis of the program, on which they succeed, hence overlooking the deeper up-to-date analysis of class interior decor and the relationship between its components. Finally, Anslow *et al.,* propose the SourceVis visualization platform which is designed for multiple users supporting multiple visualization types and displaying such visualizations on large multi-touch tables [45]. SourceVis proposes a reimplementation CLASS BLUEPRINT based on its original layers. Since BLUEPRINTV2 is an extension of CLASS BLUEPRINT, it could be introduced also in SourceVis.

VII. CONCLUSION

Understanding classes is important since they are the key abstractions in object-oriented programming. Object-oriented programming late binding makes understanding more difficult than procedural one. In particular, there is no specific reading order that IDE could use to present information to developers.

CLASS BLUEPRINT [18] proposed a compact view of class call-graph based on layers. In this article, we identified the limits of CLASS BLUEPRINT and proposed a new version. BLUEPRINTV2 supports dead code identification, methods under tests, call-flow between instance and class (static) methods. It enhances fields understanding by showing how fields of super/sub-classes are accessed, as well as lazy initialization in a compact form. It also supports hook understanding from a superclass point of view.

We presented a first validation with developers. The evaluation is two fold qualitative and quantitative. The qualitative part enabled to highlight two complementary approaches to use the proposed visualization. The *Flight over* consists of quickly navigating the visualizations of the classes one by one, with a generalized purpose in mind for the user. This approach leads to the detection of some issues in existing classes or highlight the need for deeper analysis when needed. The *Plunge in* corresponds to this deeper analysis, like complex class or dead code detection. From a quantitative point of view, the feedbacks are mostly positive from the uses of the visualization.

For future works, we plan first to introduce new features such as annotating commented methods similar to the tested methods annotation feature. In addition to highlighting if the method test passes or fails. Secondly, we will evaluate our visualization on projects written in Java and other object-oriented languages.

ACKNOWLEDGMENT.

We thank Arolla[4] for the funding of Nour J. Agouf's research, and all the participants for their availability and willingness to perform our experiment.

REFERENCES

[1] I. Sommerville, *Software Engineering*, 6th ed. Addison Wesley, 2000.

[2] A. M. Davis, *201 Principles of Software Development*. McGraw-Hill, 1995.

[3] M.-A. D. Storey, K. Wong, and H. A. Müller, "How do program understanding tools affect how programmers understand programs?" in *Proceedings of the 4th Working Conference on Reverse Engineering*. IEEE Computer Society, 1997, pp. 12–21.

[4] T. A. Corbi, "Program understanding: Challenge for the 1990's," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[5] T. Gilb and D. Graham, *Software Inspection*. Addison Wesley, 1993.

[6] V. Basili, "Evolving and packaging reading technologies," *Journal Systems and Software*, vol. 38, no. 1, pp. 3–12, 1997.

[7] U. Dekel, "Applications of concept lattices to code inspection and review," Department of Computer Science, Technion, Tech. Rep., 2002.

[8] M. Lanza and S. Ducasse, "Understanding software evolution using a combination of software visualization and software metrics," in *Proceedings of Langages et Modèles à Objets (LMO'02)*. Paris: Lavoisier, 2002, pp. 135–149.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[10] E. Casais and A. Taivalsaari, "Object-oriented software evolution and re-engineering (special issue)," *Theory and Practice of Object Systems (TAPOS)*, vol. 3, no. 4, pp. 233–301, 1997.

[11] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1038–1044, Dec. 1992.

[12] A. Dunsmore, M. Roper, and M. Wood, "Object-oriented inspection in the face of delocalisation," in *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*. ACM Press, 2000, pp. 467–476.

[13] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the comprehension of program comprehension," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, pp. 1–37, 2014.

[14] M. Lanza and S. Ducasse, "Polymetric views—a lightweight visual approach to reverse engineering," *Transactions on Software Engineering (TSE)*, vol. 29, no. 9, pp. 782–795, Sep. 2003.

[15] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *Companion of the 30th international conference on Software engineering*, 2008, pp. 921–922.

[16] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, and A. Matsuo, "Sarf map: Visualizing software architecture from feature and layer viewpoints," in *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 2013, pp. 43–52.

[17] S. Ducasse and M. Lanza, "The Class Blueprint: Visually supporting the understanding of classes," *Transactions on Software Engineering (TSE)*, vol. 31, no. 1, pp. 75–90, Jan. 2005.

[18] M. Lanza and S. Ducasse, "A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint," in *Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*. ACM Press, 2001, pp. 300–311.

[19] M. Mantyla, J. Vanhanen, and C. Lassenius, "A taxonomy and an initial empirical study of bad smells in code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.* IEEE, 2003, pp. 381–384.

[20] A. M. Fard and A. Mesbah, "Jsnose: Detecting javascript code smells," in *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2013, pp. 116–125.

[21] A. Yamashita and L. Moonen, "Do developers care about code smells? an exploratory survey," in *2013 20th working conference on reverse engineering (WCRE)*. IEEE, 2013, pp. 242–251.

[4]https://www.arolla.fr

[22] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K.-H. Prommer, "How much does unused code matter for maintenance?" in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1102–1111.

[23] B. Henderson-Sellers, *Object-oriented metrics: measures of complexity*. Prentice-Hall, Inc., 1995.

[24] M. Shepperd, "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal*, vol. 3, no. 2, pp. 30–36, 1988.

[25] A. Watson and T. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric," National Institute of Standards and Technology, Washington, D.C., Tech. Rep., 1996.

[26] E. Miranda, "The cog smalltalk virtual machine," in *Proceedings of VMIL 2011*, 2011.

[27] ——, "Brouhaha — A portable Smalltalk interpreter," in *Proceedings OOPSLA '87*, vol. 22, Dec. 1987, pp. 354–365.

[28] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the Smalltalk-80 system," in *Proceedings POPL '84*, Salt Lake City, Utah, Jan. 1984.

[29] M. Dias, D. Orellana, S. Vidal, L. Merino, and A. Bergel, "Evaluating a visual approach for understanding javascript source code," in *2020 IEEE/ACM 28th International Conference on Program Comprehension (ICPC)*, 2020.

[30] H. Tamer, D. van den Bongard, and F. Beck, "Visually analyzing the structure and code quality of component-based web applications," in *2021 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2021, pp. 160–164.

[31] S. Boccuzzo and H. Gall, "CocoViz: Towards cognitive software visualizations," *VISSOFT 2007. 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, vol. 0, pp. 72–79, 2007.

[32] S. G. Eick, J. L. Steffen, and S. Eric E., Jr., "SeeSoft—a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, Nov. 1992, depth.

[33] W. Scheibel, M. Trapp, D. Limberger, and J. Döllner, "A taxonomy of treemap visualization techniques." in *VISIGRAPP (3: IVAPP)*, 2020, pp. 273–280.

[34] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, 2007, pp. 92–99. [Online]. Available: http://dx.doi.org/10.1109/VISSOF.2007.4290706

[35] S. Alam and P. Dugerdil, "Evospaces visualization tool: Exploring software architecture in 3d," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 269–270.

[36] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "Cityvr: Gameful software visualization," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 633–637.

[37] F. Beck, "Software feathers figurative visualization of software metrics," in *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*. IEEE, 2014, pp. 5–16.

[38] D. Atzberger, T. Cech, M. de La Haye, M. Söchting, W. Scheibel, D. Limberger, and J. Döllner, "Software forest: A visualization of semantic similarities in source code using a tree metaphor." in *VISIGRAPP (3: IVAPP)*, 2021, pp. 112–122.

[39] I. Fernandez, A. Bergel, J. P. S. Alcocer, A. Infante, and T. Gîrba, "Glyph-based software component identification," in *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC '16)*, 2016.

[40] N. Churcher, W. Irwin, and R. Kriz, "Visualising class cohesion with virtual worlds," in *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2003, pp. 89–97.

[41] T. Würthinger, C. Wimmer, and H. Mössenböck, "Visualization of program dependence graphs," in *International Conference on Compiler Construction*. Springer, 2008, pp. 193–196.

[42] D. T. Daniel, E. Wuchner, K. Sokolov, M. Stal, and P. Liggesmeyer, "Polyptychon: A hierarchically-constrained classified dependencies visualization," in *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 2014, pp. 83–86.

[43] D. Reniers, L. Voinea, and A. Telea, "Visual exploration of program structure, dependencies and metrics with solidsx," in *2011 6th International workshop on visualizing software for understanding and analysis (VISSOFT)*. IEEE, 2011, pp. 1–4.

[44] U. Erdemir, U. Tekin, and F. Buzluca, "E-quality: A graph based object oriented software quality visualization tool," in *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 2011, pp. 1–8.

[45] C. Anslow, S. Marshall, J. Noble, and R. Biddle, "Sourcevis: Collaborative software visualization for co-located environments," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2013, pp. 1–10.