Glyph-Based Software Component Identification

Ignacio Fernandez[1], Alexandre Bergel[2], Juan Pablo Sandoval Alcocer[2], Alejandro Infante[2], Tudor Gîrba[3]

[1]MIT Computer Science '18, USA
[2]Pleiad Lab, DCC, University of Chile
[3]Feenk GmbH, Switzerland

***Abstract*—Glyphs are automatically generated visual icons, commonly employed as an object identification technique. Although popular in the Human Computer Interaction community, glyphs are rarely employed to address software engineering problems.**

We extended the VisualID glyph technique to cope with structural software elements and used it to address two issues in software maintenance: identify classes with the same dependencies and classes with a similar set of methods. We have compared VisualID against three visual representations: textual, graph (nodes and edges), and dependency structural matrix. Our experiments indicate that VisualID significantly helps identify classes with the same dependencies and classes with similar methods when compared with visual techniques commonly used in software maintenance.

I. Introduction

VisualID [1] is a glyph-based object identification technique. VisualID was originally proposed as a technique to generate desktop file icons based on filenames. Automatically generated glyphs help identify files having similar names [1]. Unfortunately, despite some attempts [2], [3], glyphs are rarely used within the software engineering community even though glyphs have positive effect on improving cognitive abilities.

Figure 1 represents six classes from an object-oriented system. Each class is visually represented using a glyph. The seed producing the glyph visual pattern is taken from the class name, cut down into camel case pieces. For example, the class named `RTMultiColoredLine` is cut down in {`RT`, `Multi`, `Colored`, `Line`}. Visually, we see similarities between `RTMultiLine`, `RTMultiColoredLine` and `RTDirectedLine`. The remaining glyphs have a low similarity which makes their glyphs different.

We have extended VisualID to exercise pattern matching and similarity computation on any arbitrary data structure, and not only on character strings as originally formulated. This extension enabled us to carry out controlled experiments to measure the effectiveness of the glyphs to address some software maintenance issues. In particular, we studied the following two research questions:

Q1 *Does VisualID help identify classes with the exact same dependencies?* It has been shown that identifying classes with the same dependencies is a recurrent need in programming and maintenance activities. Dependencies are often represented as a graph or a dependency structural matrix (DSM) [4]. We have compared VisualID against these two representations.

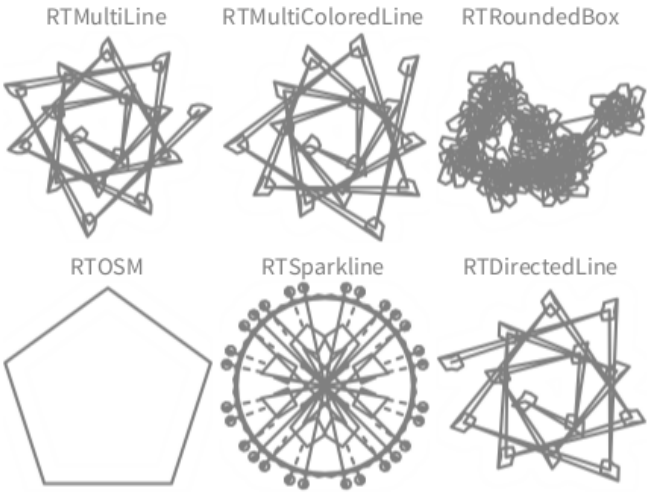

Fig. 1: VisualID glyphs example

Q2 *Does VisualID help identify classes with similar method names?* Identifying behavior similarities offered by class hierarchies is often key to spotting opportunities for code quality improvement by refactoring out common behavior. We compared the ability of VisualID against SystemBrowser to spot classes with a similar set of method names. SystemBrowser is the standard code editing and navigation tool in the Pharo programming environment.

We have carried out two controlled experiments [5] in which the measured and tested dependent variable is the productivity of a particular task and the independent variable reflects the context or tool used to perform the task.

For Q1, our results shows that VisualID is significantly more precise than DSM and Graph. In addition, it has a significantly higher recall (*i.e.,* the relevant pairs indicated by the oracle is close to the participant's answer). This result indicates that participants gives many more matching class pairs when using VisualID.

For Q2, we found that participants have a significantly higher recall using VisualID. This indicates that VisualID helps identify group of classes with similar behavior, though with low precision.

These two research questions complement each other since answering Q1 involves the ability of VisualID to identifying

exact matching pairs while Q2 is about identifying *similar pairs*. The two experiments we have carried out indicate that VisualID significantly helps identify matching or similar pairs. Although VisualID has been originally designed to evaluate similarity, we employ it to identify exact matching by answering Q1.

The two research questions shows that the visual representation of exact matching pairs and similar pairs are the key enablers to positively answers the two research questions. Although we do not verify whether the algorithm behind VisualID glyphs is the optimal algorithm to answer the research questions, our results suggests that it is robust enough to support two different software engineering-related tasks.

This paper is organized as follows. Section II informally describes the VisualID glyph technique. Section III describes our first controlled experiment to answer the Q1 research question. Section IV describes our second controlled experiment to answer the Q2 research question. Section V briefly presents our implementation and gives some of the lesson we have learnt when implementing VisualID. Section VI gives a brief overview of the related work. Section VII concludes and outlines our future work.

II. Glyph-based object identification

VisualID is a glyph-based object identification technique [1]. It generates a unique glyph for each object that is input. Similarities may exist between two glyphs depending on how similar their respective objects are: two glyphs that are visually similar are therefore likely to present two similar objects. In its original formulation, VisualID aims to automatically generate desktop icons based on filenames. We extend the VisualID technique to generate glyphs from any arbitrary data structure and a comparison function. For example, providing (i) a set of software classes and (ii) a comparison function over method names between two classes, then two classes with a large portion of common method names will be represented with visually similar glyphs.

A. VisualID Algorithm

The complete algorithm may be found in the original publication from Lewis *et al.* [1]. We summarize the approach here and describe our extensions to handle any arbitrary structures.

Glyph generation. Each glyph is a recursive structure. VisualID draws each glyph through the following process:

1) Establishing a parent glyph and generating it,
2) Generating children for the parent glyph, and
3) Generating more children until the glyph reaches a maximum complexity.

When the glyph reaches a set maximum complexity, the rendering simply stops. The complexity is a number between 0 and 5,000, an arbitrary value. A complexity of 0 means there is no glyph while a complexity of 5,000 means the glyph is at its most complex. Each glyph has its own way of calculating its complexity, which depends on the number of children it spawns. When a new glyph is added, its complexity is calculated and added to the total sum complexity of the figure.

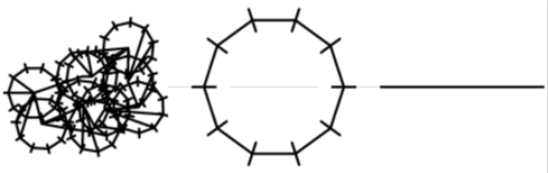

Fig. 2: Three-level recursive glyph rendering

VisualID begins its rendering by setting a basic glyph as the parent glyph. There are eight different primitive glyphs (line, figure, path, shape, radial, spiral, symmetry, and a null glyph). VisualID picks one glyph at random and establishes it as the parent glyph. Each glyph has its own set of variables that control different properties. These properties can include scaling, angles, points, number of vertices, and separation between subglyphs. These variables are calculated using pseudo-random generators. Once these properties are established the parent glyph is rendered. VisualID then decides at random which children to spawn for the parent glyph.

The algorithm for rendering complex glyphs uses a three-level recursive method. The first glyph is rendered in the first level. It is the most complex. As each level progresses, the rendered subglyph becomes less and less complex.

A child, or subglyph, is any glyph class that is not the parent glyph class. Each glyph class has its own set of children. When a parent spawns a child, a new glyph is generated using the same process as the parent glyph. Each child created can then generate children of its own, leading to higher levels of glyphs. Each glyph adds to the total complexity of the figure. The more glyphs present, the more complex the figure.

Recursion. Figure 2 demonstrates the algorithm for rendering glyphs. The first level contains the main glyph; it is the most complex. Here we have a VPath as the parent glyph with a VRadial as an outline child. The outline child is rendered in the second level, which is not as complex as the first level. In this level, the VRadial is rendered. It is a polygon with vertex children. The third level contains the VRadial's vertex children and the simplest glyph: a VLine.

Glyph complexity. The glyphs need to be complicated enough to be unique for a large number of objects, but they also need to be simple enough to be easily learned. Thus, the maximum level at which a child is spawned is three and the glyph's complexity ranges from 0 to 5,000. If the glyph is too simple, then we would have repeated glyphs. If the glyph is too complicated, then they would not be recognizable. While VisualID is capable of creating unique glyphs, it is also capable of creating similar glyphs for similar objects. VisualID's ability to recognize similar objects and draw similar glyphs for them is imperative for user identification.

Glyph mutation. VisualID generates similar objects through mutation. The input, if absent, is kept for future comparison. VisualID compares the new object with the other existing objects in the dictionary. If they are similar enough, the glyph of the original object is cloned and given to the new object as

Fig. 3: Left to right: The original, the clone, and the mutated figure

a glyph. The new glyph is then mutated. This mutation process is illustrated in Figure 3. The new figure keeps the same parent and child glyphs. Its variables are recalculated pseudo randomly. The end result is a similar but easily distinguishable figure.

Recognition. Through the process given above, VisualID generates distinct glyphs for a large number of objects. As a result, the user can easily identify and categorize different objects based on their respective glyphs. Lewis *et al.* [1] have shown that glyphs are effective for searching and relating elements when compared with generic icons. Unfortunately, VisualID has not been employed to assist software engineering tasks, which is the topic of this article.

III. Controlled Experiment: Class Dependencies

Understanding dependencies between classes is a recurrent need in programming activity. For example, class diagrams, which is the most frequently used UML diagram [6], put class dependencies at the heart of the diagram, expressed as associations.

Glyphs help identify classes having similar dependencies. To give an intuition of this effect, we draw two graphs showing class dependencies. Figure 4 gives the two versions of the graph. The graph on the left-hand side represents a circle as a class and uses non-directed edges as dependencies. The graph on the right hand side uses glyphs to represent classes. Interestingly, having glyphs clearly highlights comparable nodes, without having directed edges.

We will therefore measure the effect of VisualID against other representations. This section answers Research Question Q1. This section employs glyphs to identify exact matching pairs.

A. Experiment design

Baseline. The software engineering community has produced numerous different ways to assess class dependencies. In particular, we will focus on two representations, graph and DSM, as a baseline to evaluate VisualID:

- *Graph*: UML class diagrams represent classes as a box and dependencies using connecting lines and arrows. We designate as *Graph* the metaphor in which a class is represented as a node and a dependency as an arrow. In the graph given in Figure 5, Class A depends on Class B and C. Class B depends on A. Class D depends on Class E. Each node has a label to identify the corresponding class.
- *DSM*: A Dependency Structural Matrix [7], [4] is a popular way to represent dependencies between structural software entities. A DSM is a square matrix in which each column and row corresponds to a class. Columns and rows use the same ordered set of classes. Dependencies are indicated with a colored cell. Figure 5 shows a small DSM representing the dependencies of the same small system.

These two visual metaphors are motivated by current practice in industry. Many UML diagrams use graphs to represent dependencies, and DSM are frequently employed in commercial products (*e.g.,* Lattix[1] and Sonar[2]).

Code to evaluate. We will consider a set of 40 classes taken from a complex graphical user interface library. These 40 classes have 236 dependencies in total. We consider as dependencies all the class name references. For example, consider the following class:

```
public class Color extends Object {
  int r, g, b;
  public Color(int _r, int _g, int _b) {
    r = _r; g = _g; b = _b;
  }
  public String toString() {
    return super.toString() + "<"+r+","+g+","+b+">";
  }
}
```

The class `Color` depends on `Object`, its superclass, and on `String`, because of the return type of `toString()` and the call to `super`.

We have chosen a codebase of size 40 classes because these classes can be comfortably represented on screen using any of the three representations without using scrollbars, a magnifier or other mechanisms that may introduce a bias in our experiment.

Work session. Each participant evaluated the codebase three times, using the Graph, VisualID, and DSM representations. Our session was designed to be relatively short, taking around 20 minutes in total. The activity of a participant is structured as follows:

- Each participant begins with four questions about their personal experience.
- Graph, VisualID, and DSM are then described and illustrated with a short and concise example.
- Exercise: The Graph representation of the codebase is evaluated. A large graph showing all the dependencies is presented. Each participant indicates groups of classes with the same dependencies.
- Exercise: The VisualID representation is evaluated. A VisualID's glyph is produced for each class of the codebase. The seed used for each glyph is the alphanumerically ordered list of dependencies. For example, to represent the class `Color` given above, the seed {`Object`, `String`} is used. Participants were asked to distinguish classes with the *same dependencies*.
- Exercise: The DSM representation is evaluated. Similarly as for Graph, each participant has to indicate classes with the same dependencies.

[1] http://lattix.com

[2] http://www.sonarqube.org/sonar-2-0-in-screenshots/

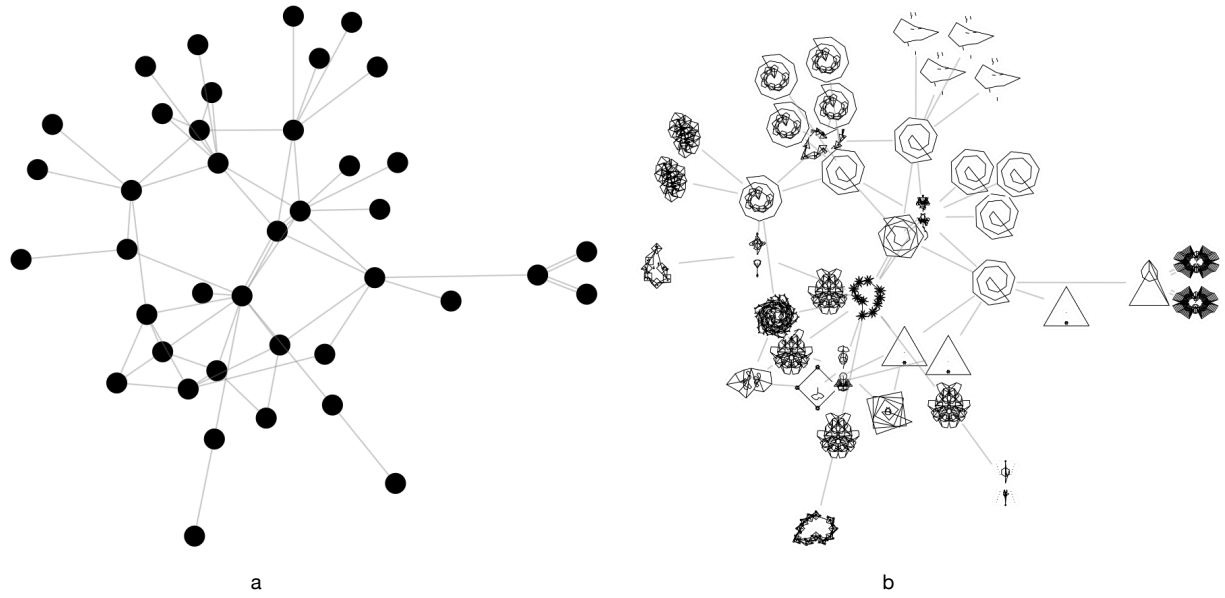

Fig. 4: Comparing Graph and VisualID

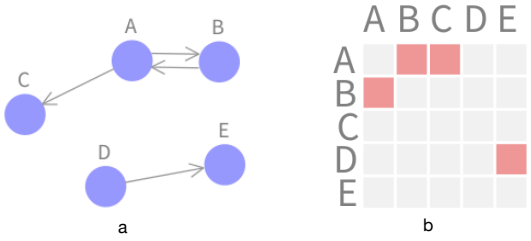

Fig. 5: The Graph (a) and DSM (b) visual representation of class dependencies of the same toy system

The experiment was carried out on paper sheets and in a dedicated room (to minimize interruption that may occur when being in front of a computer). We use the very same codebase for each of the three exercises. This motivation of this choice is based on avoiding bias that may occur from having different systems. Classes have been anonymized: instead of the name, each class was associated a number as identifier. Anonymizing classes is important since a name may be perceived as being related to other names, thus overriding our intent to evaluate the effectiveness of the visual representation.

Furthermore, each of the three exercises uses a distinct scheme to map numerical identifiers to classes to ensure that a class has different numerical identifiers across exercises. This is to make sure that participants do not see any similarities of the codebase between each evaluation.

Oracle. An oracle is needed to rank answers provided by the participants and which answers are considered as correct. To this end, for class pair (c_1, c_2) we compute automatically the similarity of dependencies: if the two classes have exactly the same dependencies we mark $S(c_1, c_2) = 1$, otherwise $S(c_1, c_2) = 0$. The VisualID algorithm is fed with the S comparison function.

The oracle indicates the dependency similarity score per couple of classes, and the number of perfect matches. From the 40 classes, only 42 pairs of classes have an exact match, within 800 different pair combinations ($40^2/2$).

Scoring. For each of the three exercises, a participant's answers groups of classes with common external dependencies $\{g_1, ..., g_n\}$, where each g_i is a set $\{c_1, ..., c_m\}$ of classes.

A score (p, r) is given for each exercise answer: p corresponds to the precision, and r to the recall. Both p and r are metrics commonly employed in information retrieval: precision measures the fraction of retrieved class pairs that have the same external dependencies and recall the fraction of matching class pairs that are retrieved. Both metrics range from 0.0 to 1.0:

- A perfect precision, $p = 1.0$, means that every class pair identified by the participant is indeed matching the criteria.
- A perfect recall, $r = 1.0$ means that all the matching class pairs are identified by the participant.

Exact matching. This experiment aims at assessing VisualID to identify classes with the very same external dependencies. We therefore have set a threshold of 1.0, the highest possible score that two classes can produce. This high threshold has the effect that glyphs do not mutate. A new glyph is associated to each class that does not have the same external dependencies as a previous class.

B. Results

Participant profile. We analyzed the answer from 18 participants. Among these participants, there are (i) 5 engineers from three different Chilean companies, (ii) 10 PhD students

Par.	Graph-p	Graph-r	Vid-p	Vid-r	Dsm-p	Dsm-r
1	1	0.36	1	0.83	0.75	0.36
2	0.65	0.26	1	1	0.36	0.31
3	0.67	0.33	1	0.93	1	0.21
4	1	0.29	0.98	0.98	1	0.33
5	0.53	0.55	1	0.95	0.81	0.71
6	1	0.48	0.55	0.43	1	0.36
7	0.25	0.02	0.93	0.93	0.71	0.29
8	0.33	0.02	1	1	0.45	0.43
9	0.78	1	0.88	1	0.58	0.81
10	0.39	0.98	1	1	0.43	0.86
11	0.87	0.31	1	1	0.81	0.4
12	0.89	0.38	1	1	0.67	0.29
13	0.48	0.26	1	1	0.71	0.12
14	1	0.36	1	0.98	1	0.36
15	1	0.29	0.9	0.86	0.6	0.29
16	0.73	0.38	0.95	0.93	0.72	0.43
17	0.91	0.5	1	0.93	0.76	0.31
18	0.87	0.79	1	1	0	0

Fig. 6: Result of the controlled experiment

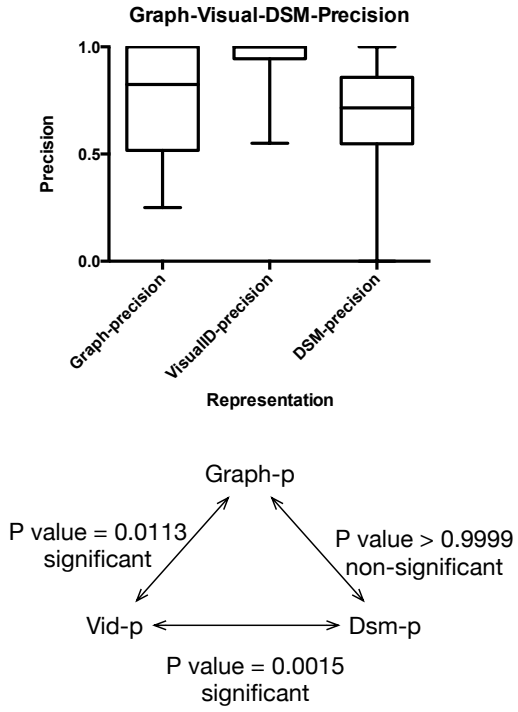

Fig. 7: Statistical analysis of the precision of the three techniques

having a strong background in programming, (iii) 3 professors in Software Engineering. One of the participants is female.

Intermediary check. We have deliberately used the same codebase for all the exercises. This decision simplifies the experiment by not considering the codebase, which would be represented as an additional independent variable.

When running the experiment, we asked the first four participants whether they discovered or even suspected that they were analyzing the same codebase at each exercise. We asked the participants orally (*i.e.,* this question was not written in the questionnaire sheet) to not influence them. They all answered they that did not suspect that the same codebase was used.

We also could not spot any indication that there is a learning effect between the exercises. We asked the first four participants whether they have felt the need to correlate the result of an exercise with participant's previous answers (*e.g.,* look for missing matching classes). All four participants answered that they did not even realized this possibility. This gave us confidence that using a within-subject design in our experiment is indeed the right choice given we had an available pool of participants of 18.

Raw data. Figure 6 shows the result of the controlled experiment we have carried out. The first column gives the participant number. Graph-p refers to the precision of using the Graph representation. Graph-r gives the recall value for the same representation. Similarly, Vid-p / Vid-r gives the result of the VisualID representation and Dsm-p / Dsm-r for the DSM representation.

Statistical tests. We first analyze the normality of the data. The Shapiro-Wilk test indicates that none of the samples is normal except Dsm-p. Kruskal-Wallis is a statistical test to analyze differences in median values between samples [8]. It is non-parametric (implying that it operates on data that are not normal as we have) and evaluates one particular factor. We apply this test to measure the effect of the representation on both the precision and the recall.

Kruskal-Wallis is a non-parametric version of ANOVA. ANOVA can be extended with the Bonferroni-Dunn test to support multiple comparisons, thus controlling the group error rate.

Analysis. Figure 7 gives the distribution of the precision for the three presentations we are considering. The graph indicates that the precision for the VisualID, with an average of 0.95, is greater than for Graph (0.74) and DSM (0.68). The multiple comparisons of the Kruskal-Wallis test indicates a significant difference when comparing VisualID for the precision. The Dunn's multiple comparisons test supports a significant difference between the samples Graph-p and Vid-p with a P value = 0.0113 and between Dsm-p and Vid-p with a P value = 0.0015. We used a confidence level threshold $\alpha = 0.05$.

Figure 8 gives the distribution of the recall for the three representations. The average recall for VisualID is 0.93, which is greater than for Graph (0.42) and DSM (0.38). The multiple comparison indicates that recall for VisualID is significantly different than for Graph and DSM. However, there is no significant difference between Graph and DSM as the P value is well above the α confidence level.

Conclusion. The experiment we have carried out measures

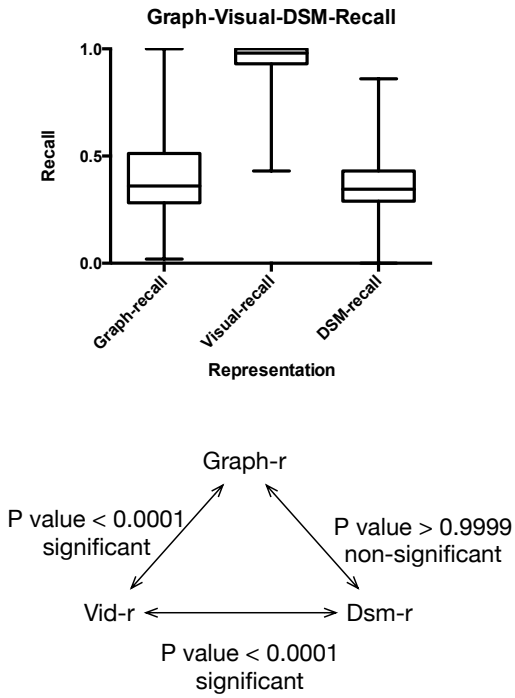

Fig. 8: Statistical analysis of the recall

the ability for the participants to identify classes with exact external dependencies using three different visual representations. Participants were able to retrieve a large portion of the matching classes, independently of the used representation: most of the indicated class pairs do indeed match. VisualID leads to significantly more *precise* results than when using Graph or DSM.

Regarding the *recall* metric, we have identified significant differences. VisualID results in a much higher recall than DSM or Graph. This means that a large portion of the matching class pairs (according to the oracle) are given by the participants when using VisualID. A significantly lower number of matching class pairs are given using DSM or Graph.

Informal post-experiment discussions with the participants indicate that the participants were not aware they have evaluated three times the same codebase.

Experiment relevance. The controlled experiment evaluates the productivity to perform a given task using a particular visual representation of class dependencies. The visual representations we are considering are not meant to be *solely* used to identify classes having the same dependencies. Software engineering tasks using class dependencies are often likely to be part of larger activities, such as software design or software maintenance. For example, graph representations are heavily used in UML (throughout all software development stages) and DSM is often used to identify dependencies cycles (often used in software maintenance activities).

As a consequence, we are not claiming that VisualID glyphs are the optimal representation for the considered task. Which is

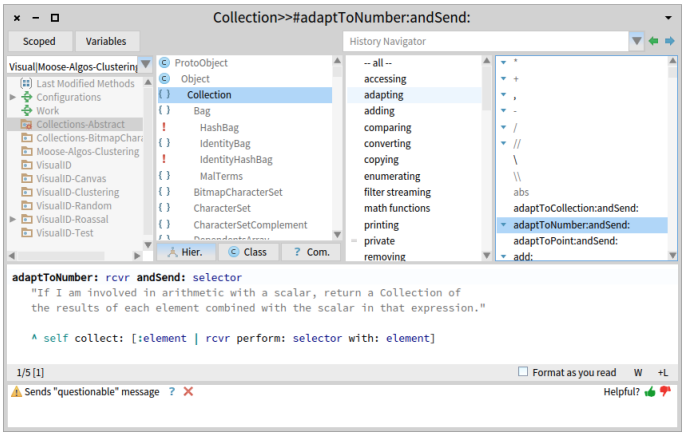

Fig. 9: The Pharo SystemBrowser

why we do not compare VisualID against a clustering algorithm performed on statically extracted dependencies. The fact that VisualID is significantly more precise and has a significant higher recall suggests that glyphs may be used in conjunction with other visual representations for which identifying classes having the same dependencies is relevant.

IV. Controlled Experiment: Class Behavior Similarity

Identifying similarities between class behavior is known to be a relevant issue in software engineering [9]. Silito *et al.* [10] identified a number of questions related to the need to identify similar class behavior. For example, it has been been shown that the following question is frequently asked by software engineers: *"Where is the code involved in the implementation of this specific behavior?"*. Identifying duplicated code behavior is a natural way to answer that question.

The assumption we will use in this experiment is that if two classes define the same set of methods, then they are likely to share a similar behavior. It has been shown that this assumption holds in the case of classes that belong to the same class hierarchy [11], [12]. This section answers Research Question Q2.

A. Experiment design

Baseline. The Pharo programming environment proposes the SystemBrowser as the main code programming tool [13]. Software engineers write and navigate through the source code within the SystemBrowser. Figure 9 depicts a browser open on the class `Collection`. The top four upper list panel indicates, from left to right, the list of packages, the list of classes, the list of method protocols, and the list of methods. The text panel located in the middle of the browser gives class and method source code.

The System Browser is not unlike other browsers in other IDEs. The code navigation supported by the System Browser is similar to the Eclipse's *Package Explorer*.

Classes are listed in such a way that the class hierarchy is explicitly apparent. The second upper panel in Figure 9 lists the classes as follows:

```
Collection
  Bag
    HashBag
    IdentityBag
    IdentityHashBag
    MalTerms
  BitmapCharacterSet
```

A superclass is located above its subclasses and the indentation indicates branches in the class hierarchy. The class `Collection` is a direct superclass of `Bag` and `BitmapCharacterSet`, and `Bag` has 4 subclasses. Structure of the class hierarchy will play an important role in our experiment since a cross-cutting concern often happens at the same level in a class hierarchy.

The baseline we consider in this experiment is to use the SystemBrowser to navigate through a class hierarchy. Participants can use all the navigation control used in a plain programming activity (*e.g.,* using the arrow keys on the keyboard to move the class selection, mouse clicking to select a particular class).

Code to evaluate. We consider the Pharo collection class hierarchy in our experiment. It is composed of 149 subclasses that are relevant for our experiment (we excluded classes with no method). The collection hierarchy is known to contain a fair amount of duplicated code among different branches. This large hierarchy therefore constitutes an appealing codebase that has already been extensively studied in the literature [11], [12].

Similarity. In this experiment, we are interested in measuring the ability of VisualID to represent classes with a similar set of methods. We have therefore lowered the threshold to produce a new glyph at 0.5, *i.e.,* two classes c_1 and c_2 having a Jaccard similarity on the methods names < 0.5 will have distinct glyphs. If their similarity is equal or greater than 0.5 then one of the glyphs will be a mutation of the other.

Pilot study and similarity threshold. This controlled experiment uses the ability of participants to measure similarity between pair of elements. Determining a relevant similarity threshold is therefore important since it determines the frontier between distinctness and similarity. We use as similarity threshold the average similarity of the pairs chosen by the participants in a pilot study.

We picked six classes c_i, i ranging from 1 to 6. We asked three participants to identify pair of classes that they consider similar using SystemBrowser. The participants identified four class pairs in total. The four participants reported the same pair and it has a similarity of 0.71. The fourth pair has a similarity of 0.35. The average similarity between the different class pairs is 0.53 ($= (0.71 + 0.35)/2$). We rounded down this value and used 0.5 as the similarity threshold for the glyph generation.

The six classes we picked belong to a code package in which similarity was expected to be found. Since we wanted the three participants to participate in the full study, the six classes we selected are not within the codebase used for the full experiment.

Work session. Each participant evaluates the same codebase twice: the first time using the Pharo IDE and the second time using VisualID. Each work session is organized as follows:

- Each participant begins with a couple of questions about their personal experience with Pharo and as a software engineer. To carry out the experiment, each participant must be familiar with the Pharo SystemBrowser.
- Exercise: The participant is offered a Pharo SystemBrowser that shows the collection class hierarchy. The method source code has been hidden to let the larger part of the screen show the class hierarchy and the list of methods. Only 5 minutes were allowed for this exercise.
- Description of VisualID. This description is the same as in the previous experiment.
- Exercise: Class similarities are evaluated using the VisualID representation. The 149 glyphs are located on a grid. Figure 10 illustrates the exercise set of a participant. Participants can zoom-in and out, drag-and-drop each glyph. This is intended to let a participant group glyphs. Again, 5 minutes were allowed for this activity. Each glyph has a numerical identifier, ranging from 1 to 149.

Each participant has a new set of randomly generated glyphs, and therefore a unique set of glyphs to evaluate. Similarly as in the previous experiment, participants were not aware that the same codebase is used for the two exercises.

B. Results

Participant profile. In total we analyzed the result of 12 participants and discarded one participant answer because one exercise answer was missing. The analysis presented below is therefore based on 11 participants. The participants are a subset from the participants we had in the first experiment. All the participants operated the very same computer setup, with same screen size, the same mouse and the same keyboard. We did so to avoid bias related to the use of the desktop computer. No female participated in the experiment.

Raw data. Figure 11 gives the result of the controlled experiment. The first column lists the participant identifiers. The second column (SB-p) gives the precision when using the SystemBrowser while the third column (SB-r) indicates the recall. The fourth column (Vid-p) indicates the precision of using VisualID and the last column (Vid-r) gives the recall.

Analyze. The two samples SB-p and Vid-p have a similar average (0.49 and 0.41). The Shapiro-Wilk test indicates that both SB-p and Vid-p are normally distributed. The standard t-test indicates no significant difference between these samples (P = 0.3301), implying that the SystemBrowser and VisualID perform equally for the precision when identifying classes with a similar method set.

The sample SB-r has an average of 0.06 while Vid-r has an average of 0.32. The Shapiro-Wilk test indicates that the sample SB-r is not normal, whereas Vid-r is normal. We therefore cannot use the t test. The Mann-Whitney U test is a nonparametric statistical test that operates over two samples of the same size. It does not assume normality of the samples.

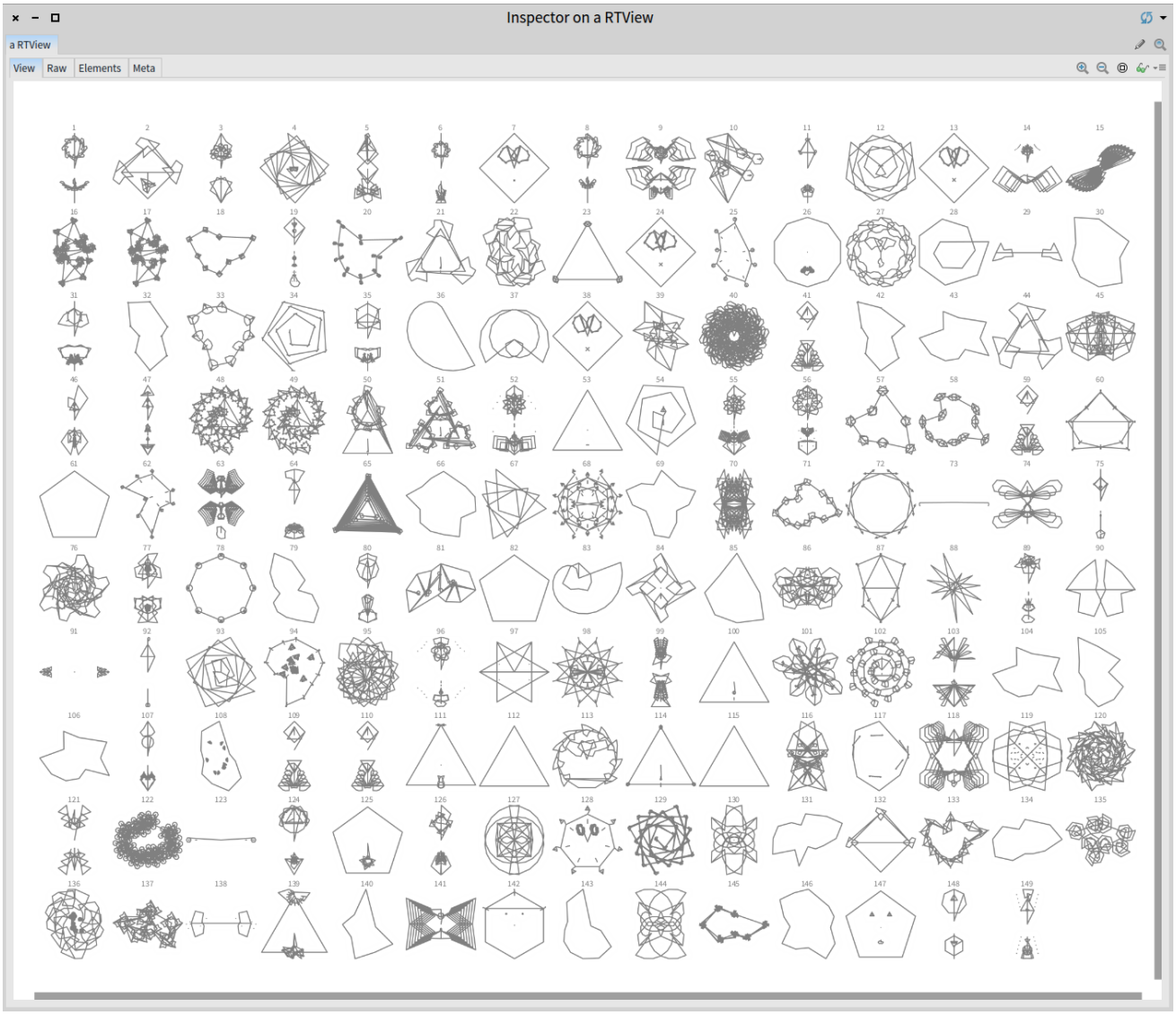

Fig. 10: Glyphs representing class methods

Par.	SB-p	SB-r	Vid-p	Vid-r
1	0.67	0.11	0.44	0.57
2	0.75	0.09	0.12	0.51
3	0.5	0.11	0.47	0.26
4	0	0	0.13	0.17
5	0.57	0.11	0.61	0.31
6	0.5	0.03	0.67	0.17
7	0.8	0.11	0.82	0.26
8	0.2	0.03	0.4	0.06
9	0.25	0.03	0.34	0.37
10	1	0.03	0.31	0.46
11	0.25	0.03	0.27	0.4

Fig. 11: Result of the controlled experiment 2

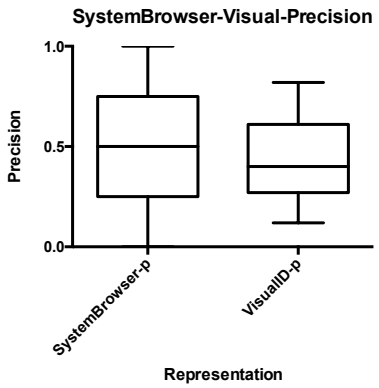

Fig. 12: Precision of SystemBrowser and VisualID

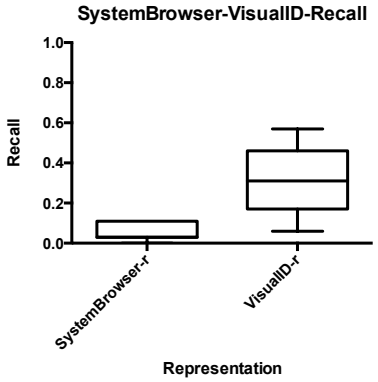

Fig. 13: Recall of SystemBrowser and VisualID

The test indicates a significant difference between SB-r and Vid-r ($P < 0.0001$).

Conclusion. Our analysis indicates that when using SystemBrowser, the participants were slightly more precise indicating classes with the same group of methods. However, this difference is not significative. On the other hand, participants had a significantly higher recall using VisualID.

Participants had 5 minutes per exercise. The number of answers provided for exercise using SystemBrowser is usually very low while for the exercise using glyphs participants gave many more similar groups of classes. Participants were much faster using VisualID than SystemBrowser. This difference in speed to carry out the exercises explains the difference of the recall: participants reported many more similar pairs using VisualID than when using SystemBrowser.

We gave complete freedom to the participants to interpret whether two classes were similar or not. We have seen that this notion of similarity varies across participants. Independently of the degree of similarity taken by each participant, the precision is comparable while the recall is significantly higher with VisualID.

Visually measuring the similarity inherently leads to a subjective interpretation. Some of the participants were unsure about whether two glyphs were *similar enough* to be reported during the experiment. One possibility to remove doubt about similarity would be to let the participant check the actual source code could in case of uncertainty about similar glyphs. However, this could be interpreted as a serious measurement bias in our experiment since the two exercises would not be that different anymore. To preserve our experiment from such a breach, we forbade participants to look at the code when using glyphs.

Glyphs are located on a grid using a randomly generated order. Using another order (*e.g.,* using a clustering algorithm) would favor the pattern visual recognition, and therefore likely to increase VisualID' score. We envision the use of VisualID as a visual aid to complement a larger software engineering activities that makes use of inferring behavior similarity. For example, the physical location of classes is often determined by a hierarchical layout involving packages and nested packages, as it often happens in a UML class-diagram. By not relying on a determined grid layout (*i.e.,* a layout that drives the spacial location of each glyph), our result shows that the visual representation of UML classes may be augmented with glyphs to enjoy its property, without affecting the overall spatial layout.

V. Implementation

Roassal. VisualID is implemented in the Roassal visualization engine [14][3]. Roassal has been extended with a new shape, called `RTVisualID`. The example given in Figure 1 is produced by the following script written in Pharo using Roassal:

```
shape := RTVisualID new.
shape basedOn:
      [ :aClass | aClass name piecesCutWhereCamelCase ].
shape score: 0.4.

c := RTCompositeShape new.
c add: RTLabel new.
c add: shape.
c vertical.

v := RTView new.
v addAll: (c elementsOn: {RTMultiLine . RTMultiColoredLine .
      RTRoundedBox . RTOSM . RTSparkline . RTDirectedLine} ).
RTGridLayout on: v elements.
v
```

Code marked in **bold** involves the use of VisualID. A glyph shape is configured with a given score, 0.4 in our example, and a function that produces a collection. This collection is used as a seed to generate the glyph. In our example, the function simply cuts down a name class into pieces. The glyph shape is instantiated into the actual visual elements using the message `elementsOn:`. We take a set of six Pharo classes as the represented object model. Classes written in Java or any other programming language would be equally relevant if coupled with the Moose platform[4].

Our implementation is available under the MIT License. The source code is accessible from the Pharo forge[5].

Lesson learnt. Some challenges we faced were in gathering sufficient background knowledge for implementation and development of a faster algorithm. JP Lewis *et al.* [1] explain very clearly what VisualID does and how it is beneficial. However, the essay does not explain in great detail how to draw the glyphs. It assumes the reader has some background in visual programming. To gain a clearer understanding of VisualID, we used the source code written by Joshua Rosen as reference[6]. Even then, we still had to modify it for our purpose.

VI. Related Work

Chuah and Eick [2] have applied glyphs to track software errors, isolate problems, and monitor development progress. The glyph is based on a face-like glyph in which software

[3]http://agilevisualization.com
[4]http://moosetechnology.org
[5]http://smalltalkhub.com/#!/~ObjectProfile/Roassal2
[6]http://www.hackerposse.com/~rozzin/VisualIDs/

metrics are mapped on different elements of the face such as the hair, nose, and mouth.

Stardinates [15] is a mechanism to visualize data described with a set of metrics. Each visualization looks like a start, similar to a kiviat. Axes originate from a unique points and are located at a regular angle. A stardinate is a visualization that combines geometric and glyph. A stardinate is interactive since axes may be added and removed, manipulate the "record line", changing the orientation of the axes, modifying scales.

Semanticons [16] generates a file icon that are both meaningful and easily distinguishable. This generation reflects an estimation of the semantics of the file, which depends on its name, location, and content. This semantics is then used to formulate a query performed on an image database. Obtained images are simplified by segmenting them and removing unimportant regions. A user study shows that search tasks may proceed faster in some situations.

Kolhoff *et al.* [17] propose a technique to generate icons for music files. The icon is automatically generated using a neural network to determine the graphical parameters from some acoustic features of the waveform stored in the represented audio file.

VII. Conclusion and Future Work

Advanced visualization techniques offered by the Human Computer Interaction community are unfortunately rarely applied to address software engineering problems. The work presented in this paper evaluates the use of randomly generated and mutated glyphs to address some common problems in software maintenance. Our result indicates that VisualID significantly helps reduce the number of false positives when identifying matching pairs.

The two controlled experiments evaluate the expressiveness of VisualID's glyphs to convey some software-related information. Our results suggest that glyphs are applicable to a wide range of usages involving similar and exact software element matching.

As future work, we plan to:

- Extend a programming environment, such as Eclipse or Pharo, with glyphs. Glyphs may then be used to visually tag software elements (*e.g.,* packages, classes, methods) to easy retrieval.
- Replace the random number generation by a particular metric to produce glyph *complexity*. Such a metric would then produce visually complex glyphs for complex elements.
- Coloring glyphs to add a new cognitive dimension. Colors will be defined as new rules that may be subject to mutation and cloning.

Acknowledgments. We thank Renato Cerro for his comments on an early draft of this paper. We gratefully thank Chung Ho Huang and Chris Thorgrimsson, from Lam Research, to support this work. Ignacio Fernandez's work is supported by the MISTI MIT-Chile program. We also thank the European Smalltalk User Group (www.esug.org) for the sponsoring. Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT, Chile, CONICYT-PCHA/Doctorado Nacional para extranjeros/2013-63130199. Alejandro Infante is supported by CONICYT-PCHA/MagsterNacional/2015-22150809.

References

[1] J. Lewis, R. Rosenholtz, N. Fong, U. Neumann, VisualIDs: automatic distinctive icons for desktop interfaces, ACM Transactions on Graphics 23 (3) (2004) 416–423.

[2] M. Chuah, S. Eick, Glyphs for software visualization, in: Program Comprehension, 1997. IWPC '97. Proceedings., Fifth Iternational Workshop on, 1997, pp. 183–191. doi:10.1109/WPC.1997.601291.

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

[4] J. Laval, S. Denier, S. Ducasse, A. Bergel, Identifying cycle causes with enriched dependency structural matrix, in: Proceedings of the 2009 16th Working Conference on Reverse Engineering, WCRE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 113–122. doi:10.1109/WCRE.2009.11.
URL http://dx.doi.org/10.1109/WCRE.2009.11

[5] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: F. Shull, J. Singer, D. Sjoberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer London, 2008, pp. 285–311.

[6] M. Petre, Uml in practice, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 722–731.

[7] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of OOPSLA'05, 2005, pp. 167–176.

[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Kluwer Academic Publishers, 2000. doi:10.1007/978-3-642-29044-2.

[9] R. Koschke, Identifying and removing software clones, in: Software Evolution, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, Ch. 2, pp. 15–36. doi:10.1007/978-3-540-76440-3_2.

[10] J. Sillito, G. C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 23–34. doi:10.1145/1181775.1181779.

[11] A. P. Black, N. Schärli, S. Ducasse, Applying traits to the Smalltalk collection hierarchy, in: OOPSLA'03: Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications, Vol. 38, 2003, pp. 47–64. doi:10.1145/949305.949311.

[12] D. Cassou, S. Ducasse, R. Wuyts, Traits at work: the design of a new trait-based stream library, Journal of Computer Languages, Systems and Structures 35 (1) (2009) 2–20. doi:10.1016/j.cl.2008.05.004.

[13] A. Bergel, S. Ducasse, C. Putney, R. Wuyts, Creating sophisticated development tools with OmniBrowser, Journal of Computer Languages, Systems and Structures 34 (2-3) (2008) 109–129. doi:10.1016/j.cl.2007.05.005.

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

[15] M. Lanzenberger, S. Miksch, M. Pohl, The stardinates - visualizing highly structured data, in: Proceedings of the Seventh International Conference on Information Visualization, IV '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 47–.
URL http://dl.acm.org/citation.cfm?id=938981.939690

[16] V. Setlur, C. Albrecht-Buehler, A. A. Gooch, S. Rossoff, B. Gooch, Semanticons: Visual metaphors as file icons, Computer Graphics Forum 24 (3) (2005) 647–656. doi:10.1111/j.1467-8659.2005.00889.x.

[17] P. Kolhoff, J. Preuss, J. Loviscach, Content-based icons for music files, Computers & Graphics 32 (5) (2008) 550–560. doi:10.1016/j.cag.2008.01.006.
URL http://www.sciencedirect.com/science/article/pii/S009784930800006X