City of Clones

Rainer Koschke
University of Bremen, Germany
orcid.org/0000-0003-4094-3444

Marcel Steinbeck
University of Bremen, Germany
marcel@informatik.uni-bremen.de

2022 IEEE 16th International Workshop on Software Clones (IWSC) | 978-1-6654-8447-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/IWSC55060.2022.00018

***Abstract*—Code Cities are used to visualize various aspects of software in 3D including clone information. Software entities such as methods or classes are often depicted by simple regular shapes, for instance, rectangular cuboids or cylinders, where different code metrics can be used to determine their height, width, and depth. Clone information is often represented by coloring or by edges connecting similar code entities. Because Code Cities may be used for purposes other than showing clone relations, the metrics for height, width, depth, and color of a shape may already be used for information not related to cloning and if so, cannot simply be re-assigned differently because that would lead to drastic changes in the visualization causing disorientation of human beholders.**

As an alternative, we propose to use the shape of the code entities to express their similarity. In this paper, we will discuss four ways to generate similar shapes for similar code entities and investigate whether dissimilar code entities have dissimilar shapes, too, to prevent false visual impressions. These approaches might even enable visual clone detection where clones are immediately recognizable by a human beholder not requiring a clone detector.

***Index Terms*—software visualization, code cities, code clones**

I. Introduction

To assist developers in the detection, prevention, and elimination of cloned code, a variety of tools have been developed in the last years. However, the number of clones within a software is typically many times higher than what developers can mentally grasp, making it necessary to present findings in a structured way. Tables are a simple, yet effective option to accomplish this. That said, tables do not take full advantage of the human visual perception. Based on the research field of information visualization, a wide range of techniques to visualize software and the clones contained therein were proposed [1]. These techniques, unlike tabular visualizations, allow humans to quickly get an overview of the characteristics of a software and to identify atypical patterns. A software visualization technique that has gained popularity a long time ago and which is still subject to current research is the *Code City* visualization [2]–[4]. The underlying data for Code Cities are typically hierarchical graphs consisting of nodes and edges. A hierarchical graph is one in which nodes can be decomposed into subnodes forming a tree hierarchy. Nodes in Code Cities are depicted as three-dimensional objects where leaf nodes in the hierarchy are often rendered as simple uniform shapes such as blocks or cylinders and inner nodes as circles or rectangles spatially enclosing their descendants.

To express relational data, edges visually connect related nodes. In the context of clone analysis, edges are often used to emphasize the fact that two source-code entities (represented as nodes) share cloned fragments. While edges are well suited to depict cloning, they can also be used to express other kinds of relations, such as, for example, import dependencies. If multiple relations must be visualized at once, the kind of relation can be encoded using colors. However, the number of colors humans can easily distinguish is limited. Moreover, too many edges create visual clutter even if they are bundled [5]. Last but not least, it may not be intuitive that two code entities that are not clones look alike if all nodes have the same uniform shape. If two code entities that are not clones had very different shapes, but two cloned entities a very similar shape, a human beholder of a Code City could detect clones by just looking at those. A clone detector needed to create the data required to add edges connecting clones may not even be needed then.

Contributions: In this paper we study alternative visualizations of cloning in Code Cities by providing more distinct individual shapes for code entities. The goal is to create similar shapes for clones, but different shapes for entities that are not clones. We experiment with three approaches to create more distinct kinds of shapes and one approach selecting textures imposed on a standard block shape. All these approaches generate their visualizations automatically based on metrics related to similarity of code entities. This paper is intended to present early ideas and initial results. The study is still explorative and preliminary, a more thorough evaluation is required in the future.

Outline: The next section discusses related research on Code Cities. Section III presents the four different alternative visualization techniques for leaf nodes and Section IV presents and discusses these in the light of examples. Section V concludes.

II. Related Research

This paper focuses on the visualization of code clones in software systems using Code Cities. For a comprehensive overview on clone visualization in general, we refer the interested reader to the work of Hamad et. al. [1]. In the following, we present related research on the topic of Code Cities.

In their simplest form, Code Cities are an extension of *Tree-maps* [6] in 3D space, which, in turn, are an early attempt to express quantitative data (i.e., metrics) over a hierarchy. With Tree-maps, a hierarchy (whatever it is composed of) is depicted by recursively subdividing a planar shape (e.g., a rectangle) into smaller nested planar shapes—that is, the hierarchy is expressed by spatial inclusion. The area of the

2572-6587/22/$31.00 ©2022 IEEE
DOI 10.1109/IWSC55060.2022.00018

innermost shapes corresponds to a certain metric (e.g., lines of code, complexity, etc. for software), expressing the metric as a proportion of the available space. This approach allows humans to easily identify atypical patterns in a special context, and so it is not surprising that quickly the idea came up to express an additional metric by mapping its value to the height of the shapes. If the underlying shape is a rectangle, resulting in cuboids in 3D space (which is generally the most commonly used shape), the obtained visualization reminds of North American downtowns with buildings arranged in grids of blocks. Accordingly, such three-dimensional Tree-maps were called *Code Cities* [7]. Researchers quickly adopted the idea of Code Cities and have been using them to visualize a variety of aspects of software [2]–[4], [8]–[15].

In addition to the geometric extensions (width, height, and depth) of the buildings of a Code City, colors can be used to express yet another metric by mapping the metric values to color gradients (e.g., from green to red) and applying the gradients on the surface of the buildings [13]. Sizing and coloring the representation of the elements of a software to be visualized is well suited to express several metrics—up to four—for individual elements at once. However, relations between elements (e.g., include dependencies) cannot be represented appropriately by these means. To depict relations, edges connecting the related elements are therefore often used [16]. To diminish the visual clutter that can occur when visualizing a lot of (overlapping) edges, hierarchical edge bundles have proven themselves suitable [5].

As already mentioned, Code Cities, in their simplest form, are Tree-maps in 3D space. That said, Code Cities today vary also in their layout. Examples for other Code City layouts include *EvoStreets* (a layout with a special emphasis on the visualization of software evolution) [17], *Circular Balloon* (a layout which, due to the more generous use of space, makes it easier to grasp the hierarchical structure of the visualized elements) [18], and *Circle Packing* [18] or *Rectangular Packing* [3]—two layouts where the hierarchy is expressed by nested circles or rectangles, respectively.

III. Visual Design

This sections introduces the different types of shapes and textures to represent leaves of a hierarchical dependency graph represented as Code City. The goal is to generate similar shapes and textures for similar code entities that are leaves in the hierarchy, but distinct shapes and textures for code entities that are not clones of each other. The primary goal is to meet a human beholder's intuitive expectation that visual similiarity suggests relatedness. This expectation is expressed in one of the laws of Gestalt, namely, the principle of similarity. This principle states that humans group things together if they appear to be similar to each other [19].

A secondary goal may be to even enable a "visual clone detection", where a human beholder may spot clones without the necessity for a clone-detection tool. We note, however, that this goal is really secondary as clone-detector tools are available, fast, mostly accurate, and may find clones at a lower granularity that is depicted in a Code City (generally, methods or classes are visualized, while many clone detectors find nested cloned code fragments at the statement level [20]).

As presented in Section II, the dimensions and color of shapes for leaves are determined by metrics in Code Cities, but the shapes themselves are generally uniform, for instance, cuboids or cylinders. Thus, these shapes may differ in their dimensions and colors, but otherwise look alike. We keep the principal idea of using metrics for the dimensions and color and just extend it by using additional metrics to control the shape or texture. In the following subsections, we will describe three approaches how to depict metrics such that different shapes are constructed. After that we will present an approach that uses the same shape, but provides differences by way of different textures put on that form.

The metrics to be used to determine the shape should be related to cloning, that is, similar metric values should indicate similar code entities. Multiple clone detection techniques are in fact based on similar metric values. Metric-based techniques to detect function clones, for instance, use the number of calls nested in a function, the number of the nodes and edges of the control-flow graph corresponding to a function and others [21], [22]. There are also syntax-based techniques using metrics. For instance, the clone detector *Deckard* represents subtrees of the syntax tree as vectors whose elements count the frequency of node types of a syntax subtree [23]. These techniques work under the assumption that similar metric values indicate similar code. Along this line of thought, we recommend to use only metrics for the visualization that are also meaningful to clone detection. Specifically, in Section IV, where we present and discuss concrete example results, we will mimic the approach by *Deckard* and count the number of syntactic constructs contained in a code entity for which to create a shape or texture.

All the approaches we describe below have in common that the difference or similarity, respectively, is presented from only one perspective in a 3D virtual world. The objects in Code Cities are generally put on a plane so that their ground area is invisible, thus, their ground area cannot be used to convey any information. Their sides are in most cases difficult to see because of other objects standing in front of them, known as the occlusion problem in 3D visualization. For this reason, we only use the "roof" of the objects to present the information. The roof is always visible from above because all objects are on the same plane and leaves are not stacked onto each other.

A. Circular Polygons

If cylinders are used, the diameter is the determining factor of the roof. In case two metrics need to be shown, the width and depth of a cylinder could be stretched accordingly, turning the circular roof into an ellipse. A more "readable" shape for two metrics, however, is a cuboid. Its root then forms a rectangle. If there are three metrics, a triangular roof may be formed. Thus, the general concept connecting this line of thought is a circular polygon, where the length of each polygon's line segment is proportional to a given metric. A circular polygon

is one in which vertices of the polygon lie on a circle. Such circular polygons are a natural continuation of the idea leading to cylinders and cuboids in the first place.

A circular n-sided polygon is determined by a set L of lengths l_i $(1 \leq i \leq n)$ for one of the polygon's line segments (or sides), i, where each l_i is the value of a metric. If there are two lengths, l_j and l_k, with $l_j \neq l_k$, the polygon is said to be *irregular*. To construct such a circular (generally irregular) polygon, given L, the radius of a circle must be found so that the starting vertex and ending vertex of each polygon side representing an l_i is on the circle and the distance between those two vertices is l_i.

Unfortunately, such a polygon exists only if there is no length l_j longer than the sum of the remaining lengths l_k with $k \neq j$. Otherwise all lengths l_k would lie on the line for l_j. Another difficulty is that the radius of the circle for arbitrary many numbers of sides, n, cannot simply be inferred by solving mathematical equations. Instead it needs to be found numerically by trial and error. Our implementation uses binary search in which two circles are iteratively increased or decreased, respectively, until a radius is found such that the distance of the vertices is close enough to the circle—as specified by a user-defined threshold.

B. Spiders

Spider charts (also known as web or radar charts) are often used to visualize multivariate data for a given entity. They have also been used to show clone metrics in particular [1]. Here a multi-dimensional co-ordinate system is rendered such that the center of each axis is at the center of a circle, C, and all axes are evenly distributed in two dimensions within C and end at a point on C. Each axis shows the value of one particular metric, where the distance from the center of C is a (generally) linear interpolation of the minimum and maximum of the metric's value range. The vertices of neighboring axes can be connected by lines forming a closed area. This area forms the roof of the shape for an entity in our approach. We do not actually show the circle of the spider chart because that might create the impression of a halo and is neither needed to determine a distinct shape.

The metric values must be normalized such that they fit into the circle, which is already done by the linear interpolation mentioned above. We then stretch all axes such that the axes with the maximal distance to the origin (there may be multiple such axes) reach the circle. This is analogous to cylinders where one metric determines the radius. The metrics are given by the user in a particular order. That order determines the order of the axes consistently for all nodes showing those metrics. To ease interpreting those charts, we follow the concept of a clock. The first metric is put on twelve o'clock and all others are then distributed clockwise. Unlike the circular irregular polygons described in the previous section, rendering these spider charts is straightforward.

A single axis with a value of zero is not a problem for these spider charts. In fact, it may even be considered as an advantage for our purpose because it creates a distinct saw kerf. If there are two or more neighboring axes with a null value, however, deciphering those spider charts visually becomes difficult.

Spider charts have been criticized for being difficult to read because the axes point into different directions. If the purpose is not only to create distinct shapes but also to be able to interpret these shapes, that is a valid argument. Yet, all axes for one particular metric are pointing into the same direction and we expect that comparing the values of the same metric for different nodes is more important than comparing the values of different metrics for the same node.

C. Bars

Bar charts are more easily to read than spider charts because all bars are oriented towards the same direction. For this reason, we also wanted to experiment with bar-chart like shapes. Here we use a baseline at which all bars are lined up. Any line parallel to the plane on which the Code City is placed could be used because the human beholder can move freely in the 3D space and look at the shapes from all angles. It makes sense, however, to select an angle such that the beholder looks a the full stretch of that line when he or she enters the scenery if this is not above the Code City. At any rate, the orientation of the baselines of all bar charts must be consistent.

The length of each bar represents a particular metric. The width of a bar could depict another metric as in polymetric views [24]. But then it is becomes more difficult to compare metrics put on the y axis with metrics put on the x axis. Arguably even worse, the combination of two large metric values in one bar will increase the area, which may be misleading. For this reason, we decided to have the same width for all bars. For the same reason, we use a fixed size for the baseline, in other words, the same width for all bars across all nodes. Because all such bar charts are equally scaled, corresponding bars can be compared across nodes, too. To allow that, the order of the bars is consistently again the order by which a user specified the metrics to be used.

D. Icons

The approaches introduced in Sections III-A–III-C create visual representations that directly reflect the metrics of the leaf nodes of the Code City in the form of shapes. Our fourth approach, *icons*, takes a different path and is based on the ideas of: i) *locality-sensitive hashing* (*LSH*) and ii) the visualization of hashes as colored icons. Unlike conventional hashing methods, where hash collisions are minimized, LSH defines a special family of hash functions that maximize hash collisions in that similar objects (code clones in our case) are hashed into the same bucket—i.e., they have the same hash value. There are a number of different LSH implementations, each of which focusing on certain aspects of how to measure similarity. For example, *Nilsimsa* is a hashing function which is used in email spam detection [25]. Another example is the clone detection tool *Deckard* [23], which uses LSH to check the generated syntax tree vectors (cf. Section III) for similarity. Thus, LSH can be considered an automated clone detection. The hash values generated by the LSH function used

can then be visualized as icons mapped onto the surface of the blocks of the Code City using *hash visualization algorithms* (*HVA*). HVA find many applications in computer science, for example, in security contexts [26], [27], or by services such as *GitHub* where user avatars are auto-generated from the user's ID. Unfortunately, there may also be unwanted LSH collisions, that is, entities that are not clones receive the same hash resulting in a misleading equal icon.

IV. Preliminary Study

As stated in the introduction, the approaches we presented in the previous section are still at the early stage. Our current research is still exploring the design space of possible solutions. To get first preliminary insights, we will exemplify here how those visualizations may look like. After all, visualizations need to be viewed. To this end, we will show and discuss four visualizations based on the ideas sketched above for the same graph.

A. Preparations

To explore our ideas, we implemented a random graph generator that allows us to create as many graphs as we want based on multiple constraints. This way we can investigate arbitrary configurations systematically. Here we present visualizations for only one particular graph because of limited space. We chose one that is, on one hand, small enough to fit into this paper and, on the other hand, large enough to be far from trivial. The example graph we chose and present here shows the relevant aspects we want to convey to summarize our observations for graphs of different characteristics.

The random graph generator is parameterized by the number of inner nodes and leaves. For the graph we use here, we specified 90 leaves and 8 inner nodes. In addition, an arbitrary number of metrics can be generated randomly for each node. Each metric can be configured by the mean and the standard deviation. The actual value of a metric is then randomly selected from the normal distribution parameterized by those two configuration settings. We chose twelve metrics mimicking the frequency of syntactic constructs in a method, such as the number of `while`, `for`, and `do` loops, `if` statements, and more. Selecting those kinds of metrics resembles the approach by Deckard [23]. The mean and standard deviation to form the normal distribution from which the values are selected randomly were—in the absence of any real statistics—set by plausible guesses. For instance, the mean for `for` was chosen higher than for `while` loops, and those for `while` loops higher than for `do` loops. Whether our guesses are accurate or not is not really relevant for our purpose here because already the choice of metrics is arbitrary. Other metric-based clone detectors use very different kinds of metrics. The only fundamental assumption our visualization variants make is that the metrics chosen yield different values for different entities and similar values for similar entities. This assumption is made by all metric-based clone detectors.

The results of the different visualization approaches for our random graph can be seen in Figure 1 side by side. All of these Code Cities are laid out by the same layout algorithm, namely, EvoStreets [28] (the exact layout may be different due to different node sizes). For the first three approaches generating different shapes, the color of the nodes is the same in order to not distract the beholder. All screenshots are taken from above the Code City so that there is no depth. The focus is on the shape or texture, respectively. We suppress all other visual distraction as much as possible.

As said above, the graph has 90 leaf nodes. This number makes it difficult for a human beholder to exhaustively compare shapes and textures pairwise (for this reason, we would never suggest to replace automated clone detectors by this visual approach). To help the reader grasp this pictures, we decided to add edges connecting two nodes that are possible candidates for clones. These edges are only illustrative and intended as a visual aid for the reader of our paper and may or may not be shown in a real use of those kinds of visualization. To provide these edges, we again followed the scheme of a metric-based clone detector. We treat the metrics of each node as a vector and then compute the Euclidean distance. All pairs of nodes whose Euclidean distance is equal to or below a certain threshold will be connected by a line. This threshold was determined by experimentation such that the clone rate reached 28 %, that is, 25 out of 90 nodes were considered clones. If those nodes were methods in real world, this clone rate would be above the clone rate Roy and Cordy found for function clones in C programs; they reported about 15 % depending on their similarity threshold [29]. We wanted to be a bit more tolerant regarding the question whether two nodes are similar so that the reader can look at many suggestions and form his or her own opinion. Once again, the edges are only here for assisting the reader; we are not claiming that only connected nodes should be considered clones, not even that connected nodes are clones. They serve here that we can contrast our intuitive point of view with an objective distance measure.

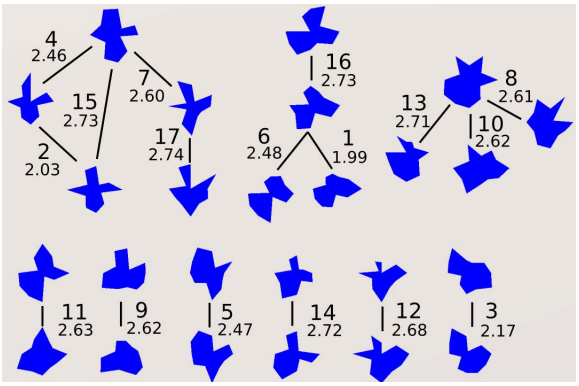
It is also important to note that we normalized the metric values using the Z score as requested to meaningfully compute the Euclidean distance. Be reminded that the Euclidean distance sums up the differences among the corresponding elements of two vectors. Without such normalization, metrics with a naturally wider range of values would dominate metrics with a very narrow range of values. The visualization, however, does not use this Z score.

B. Observations

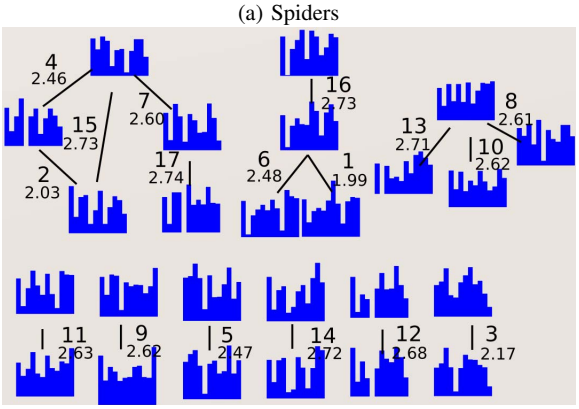
When we look at Figure 1, the first observation we can make is that the spider-like objects (Fig. 1c) are much more distinct than the two other shape variants, namely, circular polygons (Fig. 1a) and bar charts (Fig. 1b). The differences among the circular polygons are very subtle. The primary distinction among these is their area, but not their shape. The reason for that is that all vertices are on the same circle, so from far away they just look like slightly filed off cylinders. We consider ourselves unable to actually spot the similar shapes.

The bar charts are much more distinct than the circular polygons, but maybe even too distinct. They require more

cognitive processing and are harder to memorize. It is very difficult to search for a similar bar chart that is not directly in the neighborhood of a given one.



(a) Spiders



(b) Bars

Fig. 2: Clone clusters

The spider charts are both simple and distinct shapes. It is much easier to memorize them and to search for similar ones. There is one caveat, though, which is shared by all possible shape variants in general, but which becomes particularly effective for the spider charts: humans can tolerate certain transformations of shapes. If a figure is rotated, scaled, or even flipped, humans will still recognize it. So, there may be two spider charts that can be turned into each other by those transformations and, hence, may look similar to a human, but in fact there is no semantic ground for that because the meaning of the axes would change by these transformations. That is, human beholders need to be warned not to fall into this thought trap. The effect of those transformations runs into the void for circular polygons as they appear mostly as cylinders and can hardly be distinguished anyway (except for scaling). Our bar charts have a fixed baseline and by construction, there will be no other bar chart that looks like a rotation of another one—except for the special case when all bars are of equal length, that is, when two or more bar charts look like rectangles. The bar charts may, however, be affected by scaling (with respect to the height of the bars only because their width is the same for all) and possibly flipping.

Now we look at the pairs of shapes in Figure 1 conntected by edges. Once again, an edge connects two nodes if the Euclidean distance is not greater than a given threshold as discussed in the previous subsection. In order to ease the visual comparison, Figure 2 shows only the nodes connected by the edges. The reader might wonder why there are nodes in Figure 2—say A, B, C—where one edge connects A and B and another edge connects B and C, but there is no such edge connecting A and C. That is simply because the Euclidean distance between A and C is above the threshold; thus, this relation is not transitive.

The edges are labeled by two numbers in Figure 2: (1) the upper number is a label so that we can refer to the clone pair; this number is the rank in the ascending order formed by the Euclidean distance (the higher, the less similar are the nodes); (2) the number below the rank is the Euclidean distance between the two nodes connected by the edge. The reader may form his or her own opinion, but we think that the side-by-side comparison confirms the remarks we made above: The similarity is much more visible in the spider charts than in the bar charts. For instance, the two nodes of clone pair 1 with the lowest distance can more easily be related to each other for the spider chart than for the bar chart. At least we make the comparison for the bar charts by comparing each pair of corresponding bars to each other, whereas we look first at the overall shape of the spider charts and only then delve into the details of the corresponding axes. When we look at a cluster as a whole, we can more clearly see that nodes in the same cluster are more similar to each other than nodes in different clusters when presented as spider charts than when presented as bar charts. These remarks are derived from self observations. We plan to conduct eye tracking studies with multiple viewers to validate these preliminary subjective observations.

The Code City shown in Figure 1d is based on our *icons* approach presented in Section III-D. Due to the lack of suitable LSH and HVA implementations for Unity and C#, which we use to create our visualizations, we mocked this approach as follows: Each node in the city is assigned a unique texture from a fixed pool of textures. The nodes connected by the clone edges introduced above are grouped to clone classes and are assigned the same texture (e.g., the clone pairs *2, 4, 7*, and *17* in Figure 2a are grouped together and obtain the same texture). Grouping similar nodes simulates the LSH part of our approach. Assigning unique textures, in turn, simulates the HVA part. From our observation, the textures in Figure 1d are visually easier to distinguish compared to the other styles; yet, the many different shapes and colors may have a distracting effect on human beholders. Moreover, this approach suffers from two major disadvantages compared to the others: Firstly, the icons do not allow any conclusions to be drawn about the underlying metrics. Secondly, similar nodes can only be recognized as such if they have the exact same texture. Should two nodes fall below the threshold of similarity (in case of LSH this means they are hashed into different buckets) their visual representation differs significantly. Hence, nearly missed clone

pairs and nodes that are in no way similar to each other cannot be distinguished by users. To overcome this, locality-preserving hashing (LPH) [30] might be used in favor of LSH. With LPH, the relative distance between objects is preserved. That is, in our case, the distance of the hash values of similar nodes is less than the distance of the hash values of dissimilar nodes. To take advantage of LPH, though, the hash visualization which is used to create the textures needs to generate similar images for similar hashes, too. However, this requirement is generally not in the focus of HVA.

V. Conclusions

In this paper, we have presented early ideas to render clones in Code Cities similarly to each other. Our main motivation for that is to meet an intuitive expectation of human beholders: clones should look similar to each other and nodes not clones of each other should look different. This kind of rendering may even lead to a visual approach to clone detection; although we do not really see this as a replacement of automated clone detectors—in fact, the approach using the same icon for all clones in a clone cluster relies on some kind of clone detector. Yet, the visualization may at least help to investigate the similarity and difference among the clones gathered by automated clone detectors.

Experimenting with different kinds of shapes, we found that spider charts are generally better suited than bar charts because they lead to simpler and more characteristic shapes. We would not recommend circular polygons because their visual differences are too subtle. How distinct and memorable icons are depends very much on the generation of icons of course. A limitation of our current icon generation is that it is unable to create similar icons for only similar entities.

Our work towards rendering clones is still at an early stage. Neither have we explored the space of visual objects exhaustively, nor have we conducted controlled experiments. We are not aware of any earlier attempts to render clones similarly [1] and we hope that our paper will inspire other researchers to contribute to this aspect of clone visualization.

References

[1] M. Hammad, H. A. Basit, S. Jarzabek, and R. Koschke, "A systematic mapping study of clone visualization," *Computer Science Review*, vol. 37, p. 100266, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574013719302679

[2] C. Knight and M. Munro, "Virtual but visible software," in *International Conference on Information Visualization*. IEEE, 2000, pp. 198–205.

[3] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Jun. 2007, pp. 92–99.

[4] R. Koschke and M. Steinbeck, "See your clones with your teammates," in *International Workshop on Software Clones*, 2021, pp. 15–21.

[5] D. H. R. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, Sep. 2006.

[6] B. Johnson and B. Shneiderman, "Tree-maps: A space-filling approach to the visualization of hierarchical information structures," in *Proceedings of the Conference on Visualization*. IEEE Computer Society Press, 1991, pp. 284–291.

[7] K. Andrews, J. Wolte, and M. Pichler, "Information pyramids: A new approach to visualising large hierarchies," in *IEEE Conference on Visualization*. IEEE Computer Society Press, 1997, pp. 49–52.

[8] F. Fittkau, S. Roth, and W. Hasselbring, "ExplorViz: visual runtime behavior analysis of enterprise application landscapes," in *European Conference on Information Systems*, 2015, pp. 1–13.

[9] G. o. Balogh, A. Szabolics, and A. Beszédes, "CodeMetropolis: Eclipse over the city of source code," in *Conference on Source Code Analysis and Manipulation*, Sep. 2015, pp. 271–276.

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

[11] W. Scheibel, C. Weyand, and J. Döllner, "EvoCells - A treemap layout algorithm for evolving tree data," in *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, 2018, pp. 273–280.

[12] A. Schreiber, L. Nafeie, A. Baranowski, P. Seipel, and M. Misiak, "Visualization of software architectures in virtual reality and augmented reality," *IEEE Aerospace Conference*, pp. 1–12, 2019.

[13] D. Limberger, W. Scheibel, J. Döllner, and M. Trapp, "Advanced visual metaphors and techniques for software maps," in *International Symposium on Visual Information Communication and Interaction*, Sep. 2019, pp. 1–8.

[14] V. Dashuber, M. Philippsen, and J. Weigend, "A layered software city for dependency visualization," in *International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, vol. 3. SciTePress, 2021, pp. 15–26.

[15] R. Koschke and M. Steinbeck, "Modeling, visualizing, and checking software architectures collaboratively in shared virtual worlds," in *Workshop on Software Architecture and Architectural Consistency*, 2021.

[16] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," *Journal on Software Maintenance and Evolution*, vol. 15, no. 2, pp. 87–109, 2003.

[17] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *ACM Symposium on Software Visualization*. ACM, 2010, pp. 193–202.

[18] R. Koschke and M. Steinbeck, "Clustering paths with dynamic time warping," in *Working Conference on Software Visualization*, 2020, pp. 89–99.

[19] M. Wertheimer, *Festschrift für Carl Stumpf*, ser. Psychologische Forschung; Zeitschrift für Psychologie und ihre Grenzwissenschaften. Verlag von Julius Springer, 1923, vol. 4, ch. Untersuchungen zur Lehre von der Gestalt, pp. 301–350.

[20] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[21] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance*. Washington: IEEE Computer Society Press, Nov. 4–8 1996, pp. 244–254.

[22] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *IEEE Working Conference on Reverse Engineering*, 1997, pp. 44–53.

[23] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

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

[25] E. Damiani, S. Vimercati, S. Paraboschi, and P. Samarati, "An open digest-based technique for spam detection," in *ISCA PDCS*, 01 2004, pp. 559–564.

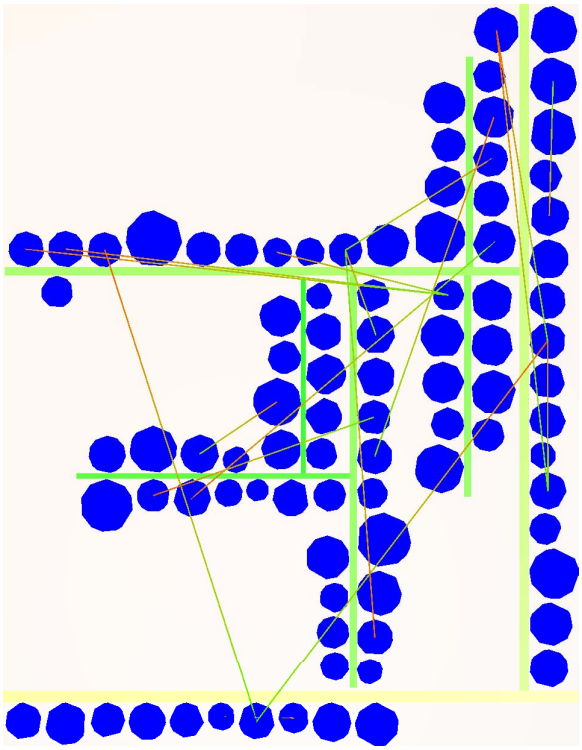
[26] A. Perrig and D. X. Song, "Hash visualization: a new technique to improve real-world security," in *International Workshop on Cryptographic Techniques and E-Commerce*, 1999.

[27] J. Fietkau and M. Balthasar, "Using hash visualization for real-time user-governed password validation," in *Mensch und Computer 2019 - Workshopband*. Bonn: Gesellschaft für Informatik e.V., 2019.

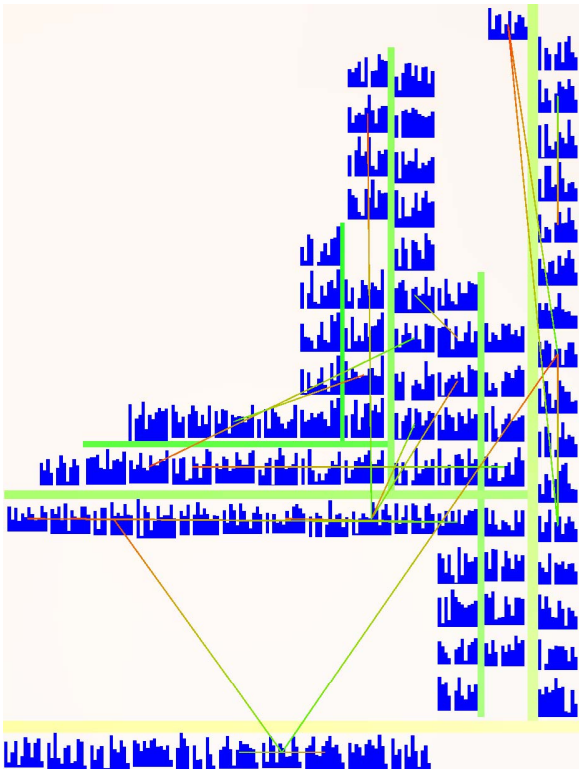
[28] F. Steinbrückner, "Consistent software cities: supporting comprehension of evolving software systems," Ph.D. dissertation, Brandenburgischen Technischen Universität Cottbus, Cottbus, 06 2013.

[29] C. Roy and J. Cordy, "An empirical study of function clones in open source software," in *IEEE Working Conference on Reverse Engineering*, 10 2008, pp. 81–90.

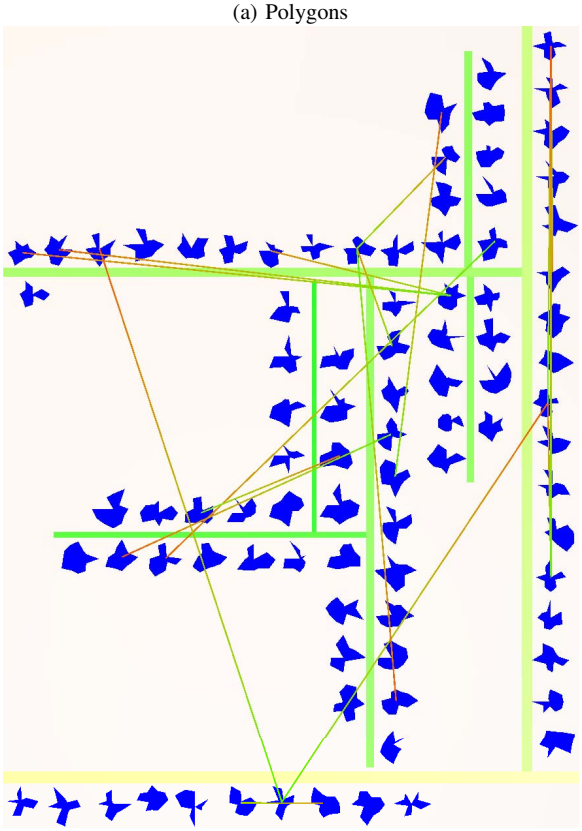
[30] Y.-H. Tsai and M.-H. Yang, "Locality preserving hashing," in *2014 IEEE International Conference on Image Processing (ICIP)*, Oct 2014, pp. 2988–2992.



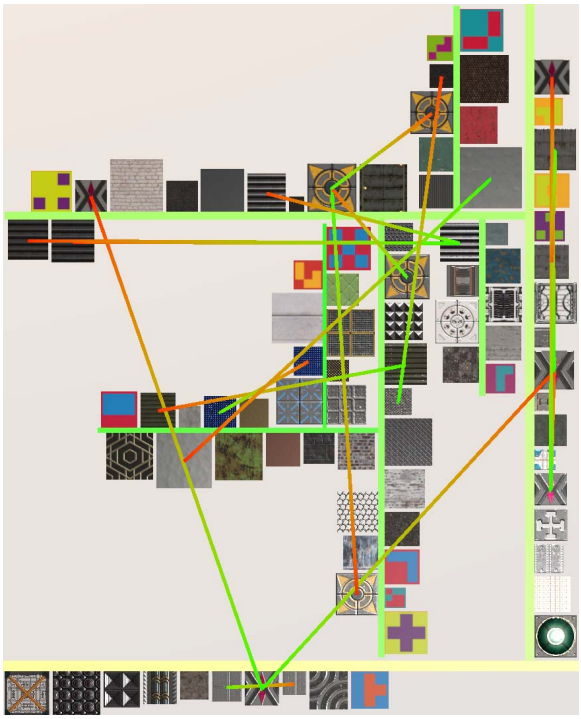
(a) Polygons



(b) Bars



(c) Spiders



(d) Icons

Fig. 1: Different styles. Edges highlight possible candidates for clones.