Skylines: Visualizing Object-Oriented Software Systems Through Class Contours

Mattia Giannaccari, Marco Raglianti, Michele Lanza
REVEAL @ Software Institute – USI, Lugano, Switzerland

***Abstract*—Classes are the fundamental building blocks of object-oriented software systems, making their comprehension critical for effective software maintenance and evolution. Traditional source code views provide detailed information but often lack intuitive representations that reveal the structural and behavioral roles of a class at a glance. This is even harder for an overview of multiple classes in large and complex codebases. Moreover, identifying patterns and anomalies within classes remains challenging through conventional inspection.**

We propose Class Contours, a novel visualization metaphor that portrays individual classes as simple 2D architectural structures. Our approach visually encodes key class properties (*e.g.,* lines of code, attributes, accessors) into customizable building features (*e.g.,* windows, door frames, doors), supporting pattern recognition and task-specific visual exploration. With ZION, the tool we developed to exemplify our approach, we investigate how common class types correspond to recurring visual archetypes, allowing developers to swiftly recognize typical roles and structures within software systems.

Our initial findings suggest that the simple but effective metaphor can enhance the understanding of class semantics in large codebases and support the identification of design issues and code smells.

***Index Terms*—Class Contours, Visual Patterns, Visualization**

I. Introduction

Classes are the principal abstraction unit in object-oriented programming languages. They encapsulate state and behavior, providing a means to break down complex systems into smaller, self-contained units [1]. Essential for defining the structure of software systems through inheritance and composition [2], classes are not just mere code containers, they are integral to how software is designed and understood [3].

Comprehending classes is therefore critical for maintaining and evolving a codebase [4]. For example, a class may encapsulate the core logic of a specific part of the model, containing common behavior shared by its subclasses. Understanding its internal (*e.g.,* attributes, methods) and external structure (*e.g.,* clients, providers) is key to understanding its role in the system. In the context of software evolution, developers must make sense of existing class structures before extending, refactoring, or debugging them. Poor comprehension can lead to fragile changes, regressions, and architectural erosion [5].

Traditional source code views, such as those provided by Integrated Development Environments (IDEs), offer detailed insights into the internals of a class. Used as text editors "on steroids", modern IDEs allow developers to read and navigate class members, follow inheritance chains, and explore references, definitions, and usages. However, these views are inherently textual, linear, and hard to scale [6], [7].

Developers need to mentally reconstruct the role and behavior of a class from scattered fragments of code [8], presented as one page of text at a time. This process is cognitively demanding [9], slow, and error-prone, especially in large systems with many interdependent components [10]. Software visualization externalizes key information, leveraging visual perception to recognize structures, roles, and anomalies [11]. In the case of visualization of classes, there seems to be a tendency for the extremes. Some approaches, such as polymetric views and treemaps [12], [13] focus on scalability at the expense of details (usually provided on demand), while approaches like UML class diagrams and class blueprints [14]–[16] favor detailed views which however fall short, for example in terms of visual clutter [17], when it comes to visualizing large codebases. While some visualizations attempt to strike a balance, achieving an effective middle ground on large and complex codebases remains challenging [18]–[20].

We propose Class Contours, a novel visualization metaphor that represents individual classes using simple 2D architectural elements (Figure 1).

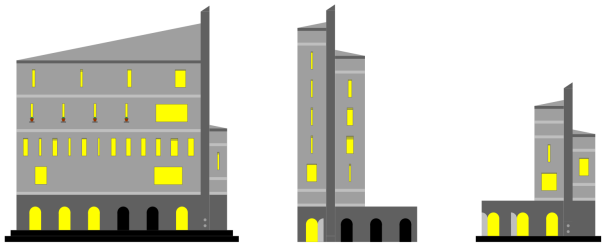

Fig. 1. Three classes visualized with Class Contours.

Each class is portrayed as a stylized building, where visual elements correspond to class properties. For example, the number of lines of code is mapped to the width of the structure, attributes are represented as doors, and methods as windows. The mapping is customizable and supports the identification of class roles and patterns at a glance. We implemented our approach in a tool called ZION, which supports parsing Java projects, generating visualizations, customizing the mapping of properties, and interacting with rendered class glyphs. ZION provides single class and system-wide views for interactive exploration. We demonstrate the efficacy of Class Contours through two case studies, highlighting typical archetypes (*e.g.,* data classes, utility classes), outliers, design inconsistencies, and code smells that might otherwise go unnoticed, all without reading a single line of code.

 DOI: 10.1109/VISSOFT67405.2025.00015

II. Class Contours

Class Contours use schematic 2D architectural elements to visualize class properties, mapping relevant metrics on position and size of the elements (Figure 2). Classes are buildings, where the left and right parts contain instance and class side members, respectively. The width of the two sides corresponds to the number of lines of code of the specific side. The chimney in the middle divides the two sections and contains the initializers of the class (*e.g.,* constructors in Java). The building is elevated by steps, one for each of its superclasses.

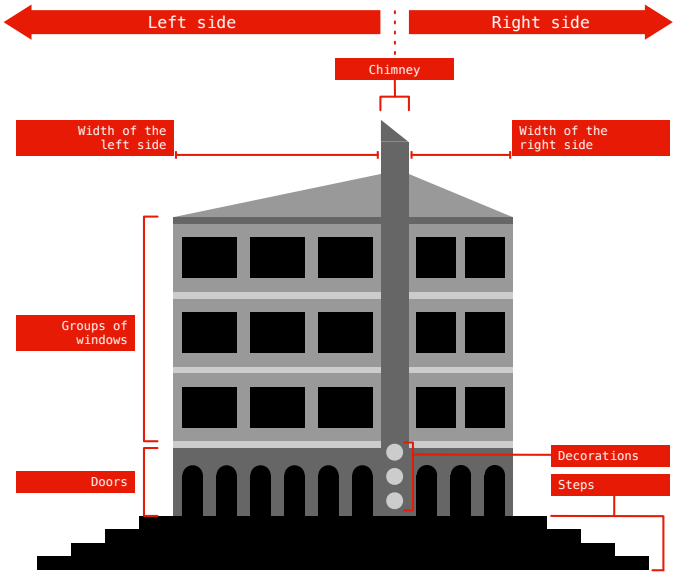

Fig. 2. Visual properties of a Class Contour.

Both sides of the building are divided into 4 floor sections. The ground floor contains the attributes, the upper floors the methods (public first floor, protected/package second floor, private methods third floor). The roof's inclination is a visual cue that helps distinguishing between the instance and the class side when looking at the building from a distance.

Methods (except accessors and initializers) are visualized as windows with the visual properties described in Figure 3.

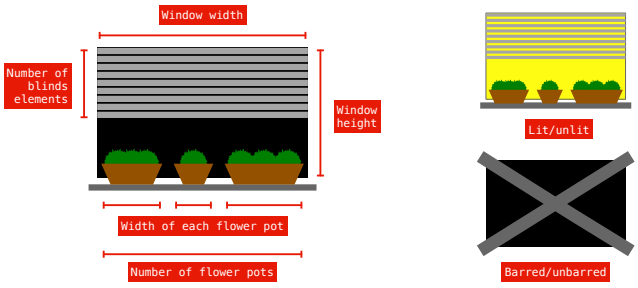

Fig. 3. Visual properties of a window in Class Contours.

Width and height of the window can represent the size of a method (*e.g.,* number of lines of code). The light inside the window can be on or off and the window can be barred or not. Light and bars can represent boolean properties of a method (*e.g.,* is abstract, is an override, is dead code).

The window blinds can be controlled in two ways: They can be used as a boolean property (*i.e.,* completely open or closed), or as a numerical attribute, counting the number of elements of the blinds. Flower pots can be used to decorate the window by specifying the number of flower pots and the width of each pot. Pots, following the idea of "decorating", can map properties such as documentation comments or annotations.

Attributes of a class are visualized as door frames on the ground floor, with their getters and setters (*i.e.,* the left and right door) if present, and can be lit or unlit (Figure 4). This allows mapping a method property and an attribute property with the same semantic. For example, a constant method always returns the same value and a constant attribute's value cannot change. This can be mapped on an unlit window and door, respectively.

Fig. 4. Visual properties of a door in Class Contours.

Class Contours are implemented in ZION, the visualization tool we developed to refine and validate our approach on interesting case studies. Besides defining the mappings, ZION allows transforming domain properties to better fit the visualization (*e.g.,* binning, scaling, setting a threshold).

III. Case Studies

We present two case studies to show the capabilities of Class Contours, focusing first on individual classes, and then depicting complete systems. The mapping of Java properties on the contours for both case studies is shown in Table I.

TABLE I
Mapping of Java properties onto Class Contours properties.

Java Entity	Visual Entity	Visual Property	Java Property
Class	Class Contour	Building's left side	Instance side members
		Building's right side	Class side members
		Left side width	#LOC of the instance side
		Right side width	#LOC of the class side
		Floors	Visibility groups (public, package/protected, private)
		Decorations	Constructors
		Steps	Superclasses chain
Method	Window	Width	#LOC of the method
		Height	Fixed value (30px)
		# blinds elements	Number of parameters
		Lit/unlit	Method returns a constant?
		Barred/unbarred	Not used
		Flower pots	Javadoc comments
Attribute	Door	Frame width	Fixed value (30px)
		Frame/door height	Fixed value (60px)
		Left door	Has getter?
		Right door	Has setter?
		Lit/unlit	Attribute is constant?
Comment	Flower pot	Width	# of comment lines

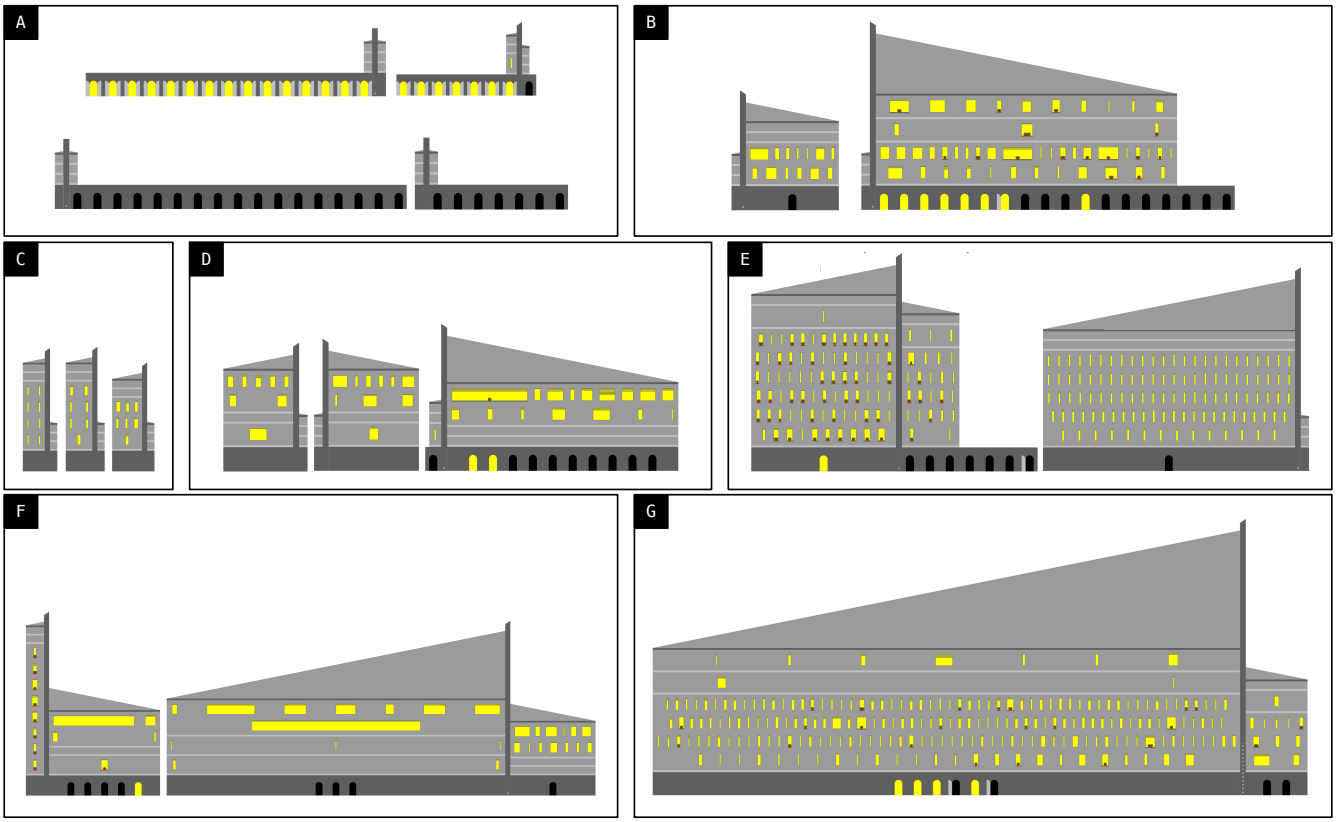

Fig. 5. Class Contours of classes extracted from the visualization of *Apache Dubbo*.

A. Individual Class Contours

We present 9 examples taken from *Apache Dubbo*.[1]

Data Class (Figure 5 A, top): A low building with many doors and few or no windows on the instance side, since it contains only attributes that make such classes simple *data holders*. Its class side is usually empty.

Constants Definer (Figure 5 A, bottom): A low building with only doors on the class side, as it only defines constants. The class side usually has no methods. All the doors are unlit because the constants do not change their value by definition.

Utility Class[2] (Figure 5 B): A building that comes in various sizes with an empty instance side. The class side has windows and possibly doors, as it only defines static methods (and constants, if needed). Projects often have one or more large utility classes, resulting in equally large Class Contours.

Instance Side Class Methods (Figure 5 C): A building with windows but no doors on the instance side. This usually indicates a code smell. Such methods do not depend on any state and should be moved to the class side. A notable exception are, for example, patterns like Java comparators and Spring Boot services (see Section IV).

Single Entry Point (Figure 5 D): A building with only one window in the public floor, no windows in the package/protected floor.

There is a variable number of windows in the private floor. The windows can be in the instance side, class side, or a mix of both. Similar to the entry point in class blueprints [14], the Class Contour also conveys the amount of state (lit doors on the instance side) and application logic (methods at the upper floors) hidden behind the entry point.

Adapter (Figure 5 E): A building with a large number of windows and only one door in the instance side. Its class side is usually empty but can also contain utility methods. In this case, thanks to Class Contours, we could find a class with the wrong name, since the Contour on the left of Figure 5 E is named `RpcContext` but, looking like an adapter, manual inspection confirmed that it does not really have any context inside and it is indeed an adapter.

Long Methods (Figure 5 F): A building of any size with very large windows (methods). This often indicates a code smell. Such methods should be refactored into smaller ones.

Large Interface or God Class (Figure 5 G): A huge building with many windows in the instance side. In the case of the Large Interface, it has many windows at the public floor. Such Class Contours highlight large hotspots of code of interest (potentially with public access, when the first floor is densely *windowed*), but also indicate a potential smell where one single class has too many responsibilities and should be broken up into smaller classes.

[1] See **`https://github.com/apache/dubbo`**

[2] A special case of helper class, usually non-instantiable, providing reusable functionalities (mainly) through static methods.

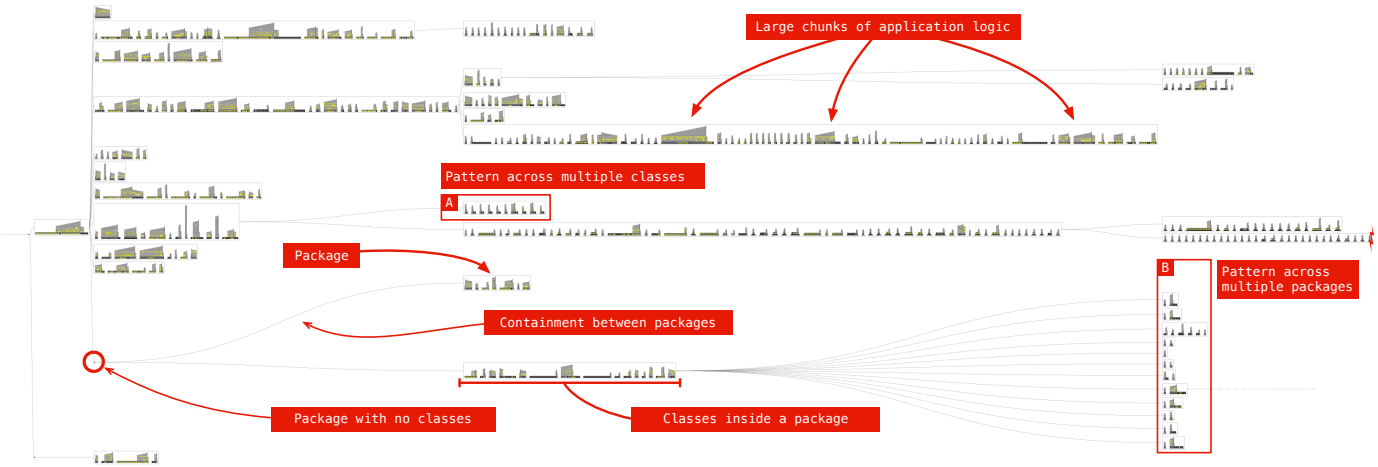

Fig. 6. Class Contours in a tree layout for visualizing the packages and classes of *antlr4* in a hierarchical view.

B. Class Contours at Scale

Figure 6 depicts the complete *ANTLRv4*[3] system. Recognizing patterns at this scale can provide information about entire parts of the system without having to explore all the classes individually. Due to space constraints we invite the reader to study the annotations in the figure, we exemplify two of them.

Figure 7 shows a zoom-in of Figure 6 A. All the classes have a similar Class Contour and provide similar functionalities. For comprehending the semantics of the whole package, it is sufficient to comprehend one class and then to generalize.

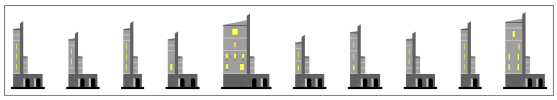

Fig. 7. Pattern of similar Class Contours in the same package.

Figure 8 shows a zoomed-in view of Figure 6 B.

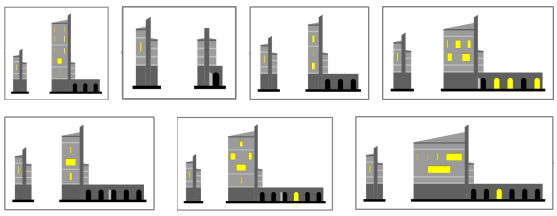

Fig. 8. Pairwise patterns of Class Contours in different packages.

All the packages have the same structure, with classes in different packages having, pairwise, a similar appearance. To comprehend the semantics of all the packages, it is sufficient to understand one single package and then generalize.

Even at the smallest scale of a system-wide overview visualization, some glyphs draw the attention highlighting key areas of interest (*e.g.,* large parts of application logic), while finer details are naturally hidden as not essential at such scale.

[3]See **`https://github.com/antlr/antlr4`**

IV. Discussion

Class Contours aim to summarize selected specific aspects of source code through an intuitive visual metaphor, reducing cognitive load during the comprehension of a software system. The visual cues we included provide insights at a glance that are relevant for specific tasks (*e.g.,* comprehension, refactoring, bug/smell fixing), guiding exploration and helping users maintain a clear mental map of the system.

ZION provides a fully configurable mapping of domain (code) properties on visual properties. The information conveyed by the visualization is highly dependent on this mapping. We presented one possible mapping to highlight the findings of our two case studies. As can be seen by the unused visual properties and the default values in Table I, using all available visual mappings could cause overloading of the visualization and lead to a cluttered view. Further improving this balance is part of our future work.

The configurability takes into account that an anomalous class in a context could be the right implementation in another case. For example, *instance side class methods* are considered a code smell in pure object-oriented programming, while in specific cases (*e.g.,* Java comparator, Spring Boot services) this is the standard way of implementing such functionalities.

V. Related Work

Understanding the structure and behavior of classes in object-oriented systems has long been a central goal of software visualization [21]–[23]. Different techniques focus on comprehension of the evolution of large codebases [24]–[29].

UML's class diagrams represent the static structure of object-oriented software [30], offering a formalized and standardized visual language for classes, attributes, methods, and relationships. While useful for documenting precise structural information, UML often falls short when applied to large systems, where their level of detail can become overwhelming and hinder the ability to grasp higher level patterns [17].

Class blueprints use a layered metaphor to represent structural and behavioral elements of classes (*e.g.,* instance variables, methods, invocations) [14] to understand their internal composition. Although effective for analyzing single classes and identifying architectural patterns across classes, class blueprints are not well suited for system-level overviews.

Polymetric views encode software metrics into a lightweight yet expressive visualization [12]. Properties such as lines of code or number of attributes are mapped to visual attributes of shapes (*e.g.,* height, width, color). Polymetric views are effective at conveying system-wide metric-driven insights, but often lack details.

CodeCity [31] is a 3D visualization framework where software systems are represented as cities, packages as districts, and classes as buildings. It provides a metaphor similar to Class Contours, leveraging spatial cognition to allow developers to explore software landscapes and detect anomalies or structural patterns at scale with "buildings look-a-like". Code cities still lack the configurable fine-grained, yet carefully constrained representation of Class Contours.

VI. Conclusion

We presented Class Contours, a visualization technique that provides a balance between fine-grained detail and overview of the whole system. We showed how Class Contours can be used to grasp insightful information about a software system at a glance, such as identifying class roles, hotspots, and anomalies. We also discussed how tiny visual cues, like the inclination of the roof, can improve how the visualization is perceived "from a distance".

Class Contours do just hint at the actual functionality of a class, so they are not meant to replace source code navigation, but to complement and ease it. While many challenges remain, our approach is the starting point for exploring a new and promising metaphor. The catalogue of patterns is being expanded as we speak, our future work lies in trying to identify class archetypes and to assign them a unique, simple, and intuitive visual representation.

Acknowledgments: This work is supported by the Swiss National Science Foundation (SNSF) through the project "FORCE" (SNF Project No. 232141).

References

[1] J. Hunt, *Smalltalk and Object Orientation: An Introduction*. Springer, 1997.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2009.

[4] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Connallen, and K. A. Houston, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Addison Wesley, 2004.

[5] R. Li, P. Liang, M. Soliman, and P. Avgeriou, "Understanding architecture erosion: The practitioners' perceptive," in *Proceedings of ICPC 2021*. IEEE, 2021, pp. 311–322.

[6] A. Jbara and D. G. Feitelson, "How programmers read regular code: A controlled experiment using eye tracking," *Empirical Software Engineering*, vol. 22, no. 3, pp. 1440–1477, 2017.

[7] S. Fakhoury, D. Roy, H. Pines, T. Cleveland, C. S. Peterson, V. Arnaoudova, B. Sharif, and J. I. Maletic, "gazel: Supporting source code edits in eye-tracking studies," in *Proceedings of ICSE 2021*. IEEE, 2021, pp. 69–72.

[8] M.-A. Storey, "Theories, methods and tools in program comprehension: Past, present and future," in *Proceedings of IWPC 2005*. IEEE, 2005, pp. 181–191.

[9] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proceedings FSE 2006*. ACM, 2006, pp. 23—34.

[10] G. C. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *Proceedings of FSE 1995*. ACM, 1995, pp. 18–28.

[11] S. Diehl, *Software visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[12] 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, 2003.

[13] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi treemaps for the visualization of software metrics," in *Proceedings of SoftVis 2005*. ACM, 2005, pp. 165–172.

[14] M. Lanza and S. Ducasse, "A categorization of classes based on the visualization of their internal structure: The class blueprint," in *Proceedings of OOPSLA 2001*. ACM, 2001, pp. 300–311.

[15] S. Ducasse and M. Lanza, "The class blueprint: Visually supporting the understanding of classes," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 75–90, 2005.

[16] N. J. Agouf, S. Ducasse, A. Etien, and M. Lanza, "A new generation of class blueprint," in *Proceedings of VISSOFT 2022*. IEEE, 2022, pp. 29–39.

[17] H. Störrle, "On the impact of layout quality to understanding UML diagrams: Diagram type and expertise," in *Proceedings of VL/HCC 2012*. IEEE, 2012, pp. 49–56.

[18] A. Hoff, C. Seidl, and M. Lanza, "Uniquifying architecture visualization through variable 3D model generation," in *Proceedings of VaMoS 2023*. ACM, 2023, pp. 77–81.

[19] S. A. Rukmono, M. R. V. Chaudron, and C. Jeffrey, "Layered BubbleTea software architecture visualisation," in *Proceedings of VISSOFT 2024*. IEEE, 2024, pp. 122–126.

[20] M. Risi and G. Scanniello, "MetricAttitude: A visualization tool for the reverse engineering of object oriented software," in *Proceedings of AVI 2012*. ACM, 2012, pp. 449—456.

[21] A. R. Teyseyre and M. R. Campo, "An overview of 3D software visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 87–05, 2009.

[22] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *Journal of Systems and Software*, vol. 144, pp. 165–180, 2018.

[23] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, and W. Chen, "A systematic literature review of modern software visualization," *Journal of Visualization*, vol. 23, no. 4, pp. 539–558, 2020.

[24] A. Hoff, L. Gerling, and C. Seidl, "Utilizing software architecture recovery to explore large-scale software systems in virtual reality," in *Proceedings of VISSOFT 2022*. IEEE, 2022, pp. 119–130.

[25] N. Hawes, S. Marshall, and C. Anslow, "CodeSurveyor: Mapping large-scale software to aid in code comprehension," in *Proceedings of VISSOFT 2015*. IEEE, 2015, pp. 96–105.

[26] C. Armenti and M. Lanza, "Using interactive animations to analyze fine-grained software evolution," in *Proceedings of VISSOFT 2024*. IEEE, 2024, pp. 36–47.

[27] M. Burch, S. Diehl, and P. Weißgerber, "EPOSee — A tool for visualizing software evolution," in *Proceedings of VISSOFT 2005*. IEEE, 2005, pp. 1–2.

[28] A. Bogart, E. A. AlOmar, M. W. Mkaouer, and A. Ouni, "Increasing the trust in refactoring through visualization," in *Proceedings of ICSEW 2020*. ACM, 2020, pp. 334–341.

[29] M. Giannaccari, M. Raglianti, and M. Lanza, "Code refactoring in virtual reality," in *Proceedings of IDE 2025*. IEEE, 2025, p. in press.

[30] H. Koç, A. M. Erdoğan, Y. Barjakly, and S. Peker, "UML diagrams in software engineering research: A systematic literature review," in *Proceedings of IMISC 2020*, vol. 74(1), no. 13. MDPI, 2021.

[31] R. Wettel and M. Lanza, "Visualizing software systems as cities," in *Proceedings of VISSOFT 2007*. IEEE, 2007, pp. 92–99.