ZION: System Exploration With Class Contours

Mattia Giannaccari
REVEAL @ Software Institute – USI, Lugano, Switzerland
mattia.giannaccari@usi.ch

Marco Raglianti
REVEAL @ Software Institute – USI, Lugano, Switzerland
marco.raglianti@usi.ch

Michele Lanza
REVEAL @ Software Institute – USI, Lugano, Switzerland
michele.lanza@usi.ch

Abstract

Understanding the structure and behavior of classes and their roles in a software system is essential for maintaining and evolving object-oriented software. Reading source code provides limited support for quick comprehension or pattern identification. A class as a long stream of text in the IDE fails to communicate crucial information about its different components (*e.g.,* attributes, methods, instance vs. class side code), while IDE outlines are just a band-aid solution.

We present ZION, our tool to visualize the source code by representing classes as 2D architectural structures. ZION defines *Class Contours* by encoding class components (*e.g.,* attributes, methods) into customizable building features (*e.g.,* doors, windows). This allows tailored visualizations for identifying class roles and patterns, common archetypes, and hotspots for application logic. We present ZION's design, its features, alongside examples to highlight how the emerging visual patterns and the interactive visualization navigation enhance understanding complex software systems.

Demo video: https://youtu.be/5jS8nqpe5KM
Tool image: https://doi.org/10.6084/m9.figshare.30245659

CCS Concepts

• **Software and its engineering** → *Object oriented architectures*; **Software maintenance tools**; Software reverse engineering.

Keywords

Class Contours, System Comprehension, Software Visualization

ACM Reference Format:
Mattia Giannaccari, Marco Raglianti, and Michele Lanza. 2026. ZION: System Exploration With Class Contours. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 5–9, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3806412

1 Introduction

Maintaining and evolving object-oriented systems requires understanding the structure and behavior of its classes [2]. Traditional methods of source code navigation, and particularly reading code as long, linear streams of text in an Integrated Development Environment (IDE) is not sufficient [6, 10]. The textual representation of classes obscures key elements, such as the distinction between instance and class side code or the role of a class in the system [11].

Modern IDEs provide outline views (*i.e.,* a navigable tree representation of the top-level structure of the file, with methods and attributes) and package explorers to help mitigate these issues. Such solutions are suboptimal for developing a holistic understanding of the system's architecture [13, 15, 16].

To address these limitations, we introduce ZION, a visualization tool that represents classes as customizable 2D building glyphs called *Class Contours* [7]. Building on evidence that city-metaphor visualizations improve program comprehension [21, 22], ZION allows the interactive exploration of a codebase, encoding software components (*e.g.,* attributes, methods, documentation) into architectural elements (*e.g.,* windows, doors, flowerpots) with metrics mapped on the properties of the visual components (*e.g.,* length of method on width of window), and highlighting their relationships (*e.g.,* containment, inheritance). Class Contours communicate roles and characteristics of classes within a system, creating visual architectural patterns (Figure 1). ZION enables to focus on specific tasks, such as revealing class archetypes, identifying application logic hot-spots and anomalies, and exposing design patterns.

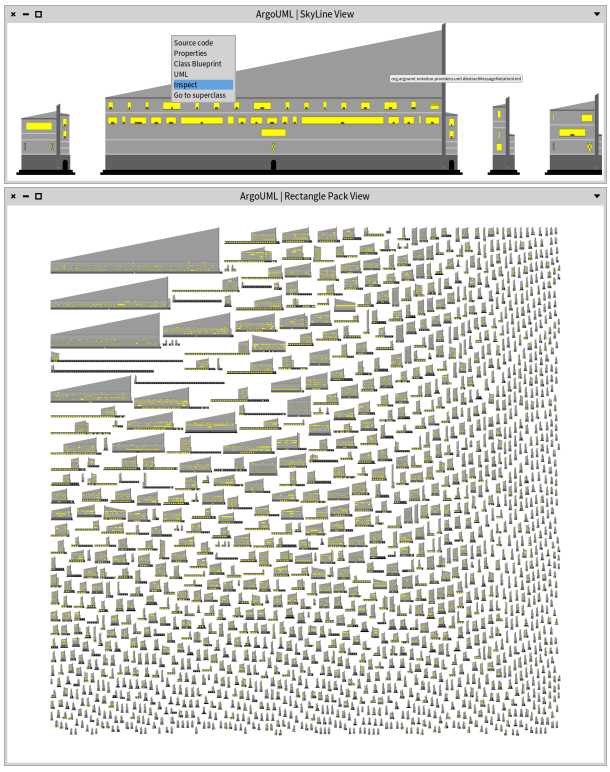

Figure 1: Zoom-in on Class Contours (top) and example overview of a complete system (bottom) in ZION.

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/26/07
https://doi.org/10.1145/3803437.3806412

2 Tool Design and Implementation

The core model of ZION is a graph composed of three interconnected sub-graphs (Figure 2): The software, the mapping, and the visual domain sub-graphs. The software and visual domain sub-graphs have two components each: A model and a meta-model sub-graphs. Meta-models contain information about the type of entities (*i.e.,* type nodes) and their properties (*i.e.,* property nodes). Model sub-graphs contain specific instances of the software system under study and their visualization (*i.e.,* instance nodes). The mapping sub-graph defines how the software domain is mapped onto the visual domain.

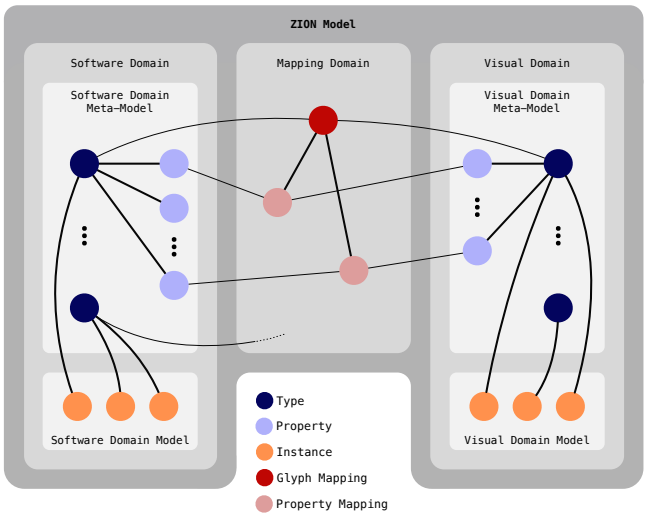

Figure 2: Overview of the core model of ZION.

Property mapping nodes connect domain properties to visual properties, while *glyph mapping nodes* connect domain type nodes (*e.g.,* class, method, comment) to visual type nodes (*e.g.,* building, window, flower pot). Creating a mapping from the software to the visual domain meta-model equals adding mapping nodes to the graph and connecting them to the types and properties to map. When a software system is imported, its model and meta-model graphs are populated. While mapping a software entity to a visual entity, domain properties can optionally run through transformations to adapt them to the visual properties they are mapped to.

The model importer (Figure 3) takes a software system as input and instantiates its ZION's model as follows: (A) It runs a set of custom CodeQL[1] queries to extract entities, relationships and properties from source code; (B) It uses the software domain meta-model to instantiate the software domain model; (C) It uses the mapping provided by the user and the software domain model to instantiate the visual domain model.

All importer components (*i.e.,* the set of custom CodeQL queries, the software domain meta-model, and the visual meta-model) are designed for modularity and extensibility. This architecture enables adding new custom properties, as well as easy adaptation of the meta-model to support other languages beyond Java.

[1]See https://codeql.github.com

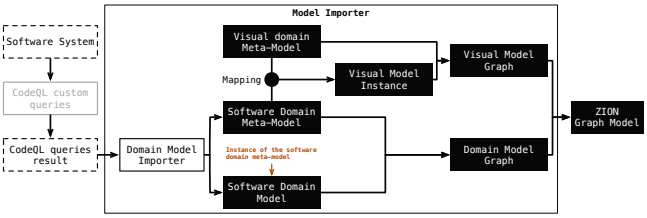

Figure 3: Importing a software and a visual domain in ZION.

A *viewspec* (*i.e.,* View Specification) is a way to describe and customize a visualization [17]. It consists of a way of selecting the entities to be visualized (*i.e.,* a filter), a way of sorting the selected entities, a set of glyph mappings to transform the software domain entities into glyphs, and a layout for arranging the glyphs into the visualization. Filtering and ordering can be optionally applied on both the domain entities and the visual entities. Figure 4 shows the full pipeline that ZION implements to build a visualization.

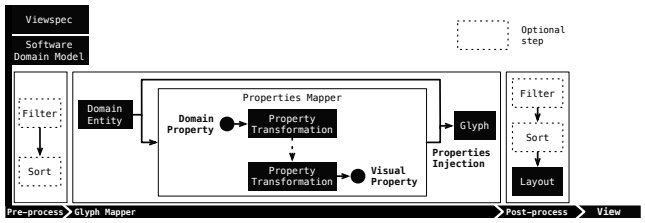

Figure 4: Building the visualization of a domain using a *viewspec*.

We implemented ZION in Pharo.[2] Thanks to Pharo's reflective capabilities, we can automatically extend the meta-models and identify methods that represent domain properties. By marking methods with custom *pragmas* (*i.e.,* annotations on Pharo's *CompiledMethod* objects), we enable system-wide property discovery, type-based grouping, and matching with visual properties of the same type.

ZION's modular architecture enables easy extension to new domains and visualization engines. Decoupling domain and visualization models allows the integration of new visual metaphors and domain-specific analyses with minimal effort. The architecture supports incremental extension, enabling developers to tailor both parsing and visualization pipelines to their specific needs.

ZION also provides a user interface for cloning repositories and parsing source code using CodeQL, a semantic code analysis engine developed by GitHub. The set of queries that we developed specifically for ZION allows extracting software entities (*e.g.,* classes, methods), relationships (*e.g.,* methods calls, attributes accesses) and properties (*e.g.,* number of lines of code, dead code) from a codebase. The result of the queries is then imported into ZION for visualization and analysis.

To render the visualization, ZION uses Roassal,[3] a visualization engine with interactive capabilities. It builds the visualization of *Class Contours*, with a set of adaptable glyphs, custom layouts, and viewspec presets that we exemplify in the following scenarios.

[2]See https://pharo.org
[3]See https://github.com/pharo-graphics/Roassal

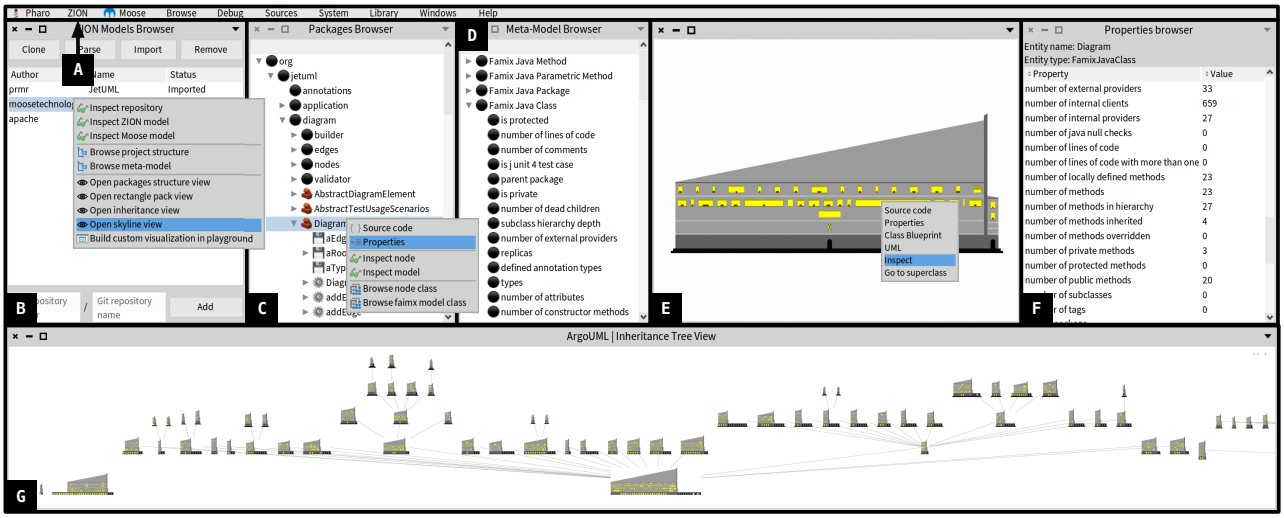

Figure 5: The user interface of the ZION toolset.

3 User Interface and Workflows

The user interface of ZION consists of a set of tools that enable importing a software system, navigating its entities and properties, and exploring preset and custom views. The main tools available in ZION are shown in Figure 5. The entry point is Pharo's "world menu" (A), which allows to open the model browser.

The **Model Browser (B)** allows to import new repositories in ZION, by cloning them and parsing their code. Right-clicking an imported repository opens a contextual menu with tools, visualization presets, and visualization customization.

The **Packages Browser (D)** of a model shows the same view of an IDE outline, offering a familiar navigation of the packages tree structure and classes content, possibly invoking other ZION tools by right clicking on any entity in the browser.

Visualization Presets (E, G) are tailored around specific tasks: *SkyLine View* (E) allows scrolling to each class and zooming-in up to single buildings; *Inheritance View* (G) highlights inheritance between classes to contextualize the shape of the building according to its superclasses (providing complementary information).

The **Properties Browser (F)** can be opened from any view and visualizes the value of each property of a domain entity.

Contextual Menus provide a way to directly interact with an entity. For example, it is possible to open a properties browser, an inspector on an entity's model, and to jump to the superclass of a class. Contextual menus also integrate ZION with familiar tools, such as opening the source code in VSCode or visualizing the UML diagram of a class.

Since ZION focuses on visualizing large codebases, each view is provided with a textual search bar that highlights selected results directly into the visualization, as shown in Figure 6.

Figure 6: Search bar to find and highlight domain entities.

4 Example Usage Scenarios

Class Contours let developers quickly grasp properties and roles of classes. Figure 7 summarizes the mapping we use in this visualization. Our prior work shows that visual patterns of Class Contours help identify data, utility, adapter, and god classes; constant definers; long methods; classes with a large interface or a single entry-point; and instance side methods that should be class side [7].

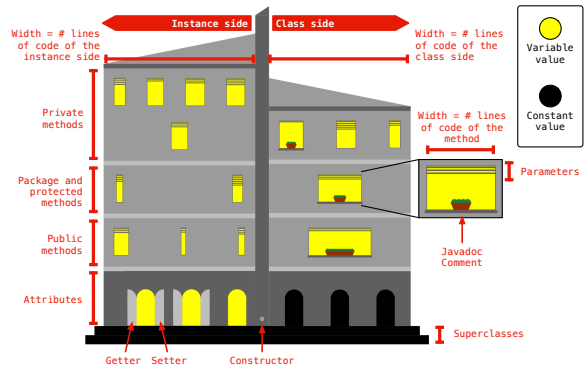

Figure 7: Visualizing a class with Class Contours using ZION.

Small cues (*e.g.*, the tilting of the roof, the tip of the chimney) are designed to make Class Contours distinguishable and informative even from a distance, as shown in Figure 8.

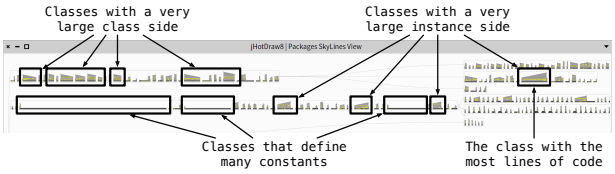

Figure 8: Patterns and insights from a zoomed-out view.

The power of ZION comes from the ability to progressively and interactively disclose additional information to understand the *semantic role* of a class and to detect anomalies. Figure 9 shows the Class Contour of the class `BezierCurves` from JHotDraw.[4]

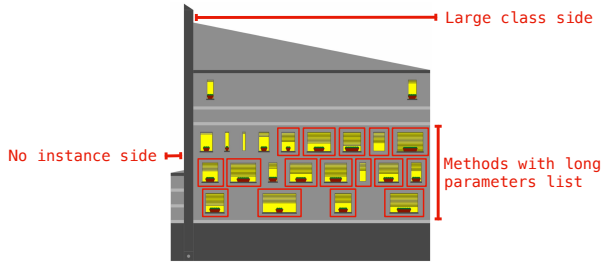

Figure 9: The Class Contour of the BezierCurves class.

The visualized class contains no instance-side members but provides many public, class-side methods. This suggests that the class acts as a utility class. The visualization makes this visible by separating instance-side and class-side elements, while the tooltip showing the class name suggests its context (*i.e.,* to operate on bézier curves). Figure 9 highlights in yellow the methods that expose long parameter lists, which appear as blinds covering the windows. Long parameter lists can be a cue for the Missing Abstraction code smell, which refers to representing an entity through a set of parameters instead of representing it through a dedicated abstraction (*i.e.,* a class). A subsequent inspection of the source code (by opening the class definition inside the IDE through ZION's contextual menu) confirms that the class repeatedly operates on raw parameter sets. The inspection also reveals that these parameters correspond to concepts that could be encapsulated as `CubicCurve` and `QuadCurve`. The class therefore re-implements curve operations through sequences of primitive parameters rather than through explicit abstractions. This example illustrates the tool's support for progressive disclosure and anomalies detection. The visualization provides high-level structural cues (*e.g.,* long parameter lists, clusters of large methods). The source code then provides the most fine-grained level of detail. Developers can move from the visualization to the code with minimal friction, which allows the tool to complement the IDE and integrate naturally with standard development workflows. The visualization also indicates that most methods are documented with Javadoc. Combined with structural information and method shapes, this offers a compact overview of the class design and its potential deficiencies.

5 Related Work

Understanding structure and behavior of classes in object-oriented software systems has been a longstanding concern in software visualization research [3, 14]. Traditional approaches, like UML class diagrams, provide a formal mechanism to represent object-oriented software [4]. However, when applied to large systems, they can become dense and unwieldy, limiting their usefulness in everyday development tasks [19]. To address this, we introduce a more lightweight and cognitively manageable abstraction that supports visual reasoning and pattern recognition without overwhelming detail.

Tools such as Class Blueprints [5], Polymetric Views [12], CodeCity [21], and Layered BubbleTea [18] visualize classes by focusing on either the internal structure of single classes or the overall software system organization. However, as software systems grow in size and complexity, the usefulness of these tools increasingly depends on the degree of interactivity they provide [1, 8]. Interactive capabilities (*e.g.,* zooming, filtering, contextual tooltips) enable users to navigate large models more effectively and avoid visual density. Moreover, progressive disclosure mechanisms allow the visualization to reveal information incrementally, showing only the most relevant details at a given moment [9, 20]. This helps users build understanding step by step, reduces cognitive load, and supports deeper exploration without sacrificing clarity.

Our approach aims at integrating fine-grained class views into a comprehensive visualization that supports insights on the architecture of the system and on local, single class details. Information about the role of the class can be inferred from the semantics conveyed by its Class Contour.

6 Limitations and Future Development

The main limitation of ZION is its reliance on third-party parsers, which restricts control over data quality and often requires ad-hoc solutions for the assumptions, deficiencies, and errors introduced by these parsers. We plan to develop a built-in parser tailored to ZION to improve reliability and control. Moreover, ZION has only been evaluated on Java. As we improve our parsing infrastructure, we plan to support additional languages. This generalization should further validate our architectural metaphor based on Class Contours, broadening the applicability and impact of our tool.

While ZION scales to very large systems, visualization generation can be costly. For example, in our experiments with Apache Dubbo,[5] model import took about 20 minutes (mainly CodeQL database creation and query execution) and visualization generation roughly 10 minutes (due to graph traversal), both proportional to system size and number of relationships. We plan to mitigate these costs by integrating model building into CI/CD pipelines at commit time and extending the graph model caching to the visual domain.

Lastly, we plan to conduct a comprehensive user study to evaluate the usability of ZION and the effectiveness of Class Contours in practice, providing empirical feedback to guide further refinements.

7 Conclusion

Understanding the structure and behavior of object-oriented software is crucial for effective maintenance and evolution, yet conventional code navigation tools fall short in providing a clear, holistic view of a system's architecture. We introduced ZION, a novel visualization tool that represents classes as Class Contours, customizable 2D glyphs inspired by building facades. ZION empowers developers to visually explore and interpret classes and architectural patterns through intuitive, highly expressive, and interactive visualizations.

ZION facilitates the identification of class roles, design anomalies, and architectural (in)consistencies. The presented scenarios show how it enhances system comprehension, offering a flexible approach to developer onboarding on large codebases, complementing and going beyond what is actually possible within current IDEs.

[4] See https://github.com/wrandelshofer/jhotdraw

[5] See https://github.com/apache/dubbo

Acknowledgments

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

References

[1] Maxime André, Marco Raglianti, Anthony Cleve, and Michele Lanza. 2025. Visualizing and Exploring Data Access in Microservices Using Interactive Treemaps. In *Proceedings of VISSOFT 2025 (Working Conference on Software Visualization)*. IEEE, 36–46. https://doi.org/10.1109/VISSOFT67405.2025.00012

[2] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. 2004. *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison–Wesley.

[3] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. A Systematic Literature Review of Modern Software Visualization. *Journal of Visualization* 23 (2020), 539–558. Issue 4. https://doi.org/10.1007/s12650-020-00647-w

[4] Brian Dobing and Jeffrey Parsons. 2006. How UML is Used. *Commun. ACM* 49, 5 (2006), 109—-113. https://doi.org/10.1145/1125944.1125949

[5] Stéphane Ducasse and Michele Lanza. 2005. The Class Blueprint: Visually Supporting the Understanding of Classes. *Transactions on Software Engineering* 31, 1 (2005), 75–90. https://doi.org/10.1109/TSE.2005.14

[6] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S. Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan I. Maletic. 2021. gazel: Supporting Source Code Edits in Eye-Tracking Studies. In *Proceedings of ICSE 2021 (International Conference on Software Engineering)*. IEEE, 69–72. https://doi.org/10.1109/ICSE-Companion52605.2021.00038

[7] Mattia Giannaccari, Marco Raglianti, and Michele Lanza. 2025. Skylines: Visualizing Object-Oriented Software Systems Through Class Contours. In *Proceedings of VISSOFT 2025 (Working Conference on Software Visualization)*. IEEE, 64–68. https://doi.org/10.1109/VISSOFT67405.2025.00015

[8] Wilhelm Hasselbring, Alexander Krause, and Christian Zirkelbach. 2020. ExplorViz: Research on software visualization, comprehension and collaboration. *Software Impacts* 6 (2020), 100034. https://doi.org/10.1016/j.simpa.2020.100034

[9] Devamardeep Hayatpur, Daniel Wigdor, and Haijun Xia. 2023. CrossCode: Multi-level Visualization of Program Execution. In *Proceedings of CHI 2023 (Conference on Human Factors in Computing Systems)*. ACM, 593:1–593:13. https://doi.org/10.1145/3544548.3581390

[10] Ahmad Jbara and Dror G. Feitelson. 2017. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. *Empirical Software Engineering* 22, 3 (2017), 1440–1477. https://doi.org/10.1007/s10664-016-9477-x

[11] Rodi Jolak, Maxime Savary-Leblanc, Manuela Dalibor, Andreas Wortmann, Regina Hebig, Juraj Vincur, Ivan Polasek, Xavier Le Pallec, Sébastien Gérard, and Michel R. V. Chaudron. 2020. Software Engineering Whispers: The Effect of Textual vs. Graphical Software Design Descriptions on Software Design Communication. *Empirical Software Engineering* 25, 6 (2020), 4427–4471. https://doi.org/10.1007/s10664-020-09835-6

[12] Michele Lanza and Stéphane Ducasse. 2003. Polymetric Views—A Lightweight Visual Approach to Reverse Engineering. *Transactions on Software Engineering* 29, 9 (2003), 782–795. https://doi.org/10.1109/TSE.2003.1232284

[13] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of ICSE 2006 (International Conference on Software Engineering)*. ACM, 492–501. https://doi.org/10.1145/1134285.1134355

[14] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. 2018. A Systematic Literature Review of Software Visualization Evaluation. *Journal of Systems and Software* 144 (2018), 165–180. https://doi.org/10.1016/j.jss.2018.06.027

[15] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. 2016. Taming the IDE with Fine-Grained Interaction Data. In *Proceedings of ICPC 2016 (International Conference on Program Comprehension)*. IEEE, 1–10. https://doi.org/10.1109/ICPC.2016.7503714

[16] Fernando Olivero, Michele Lanza, Marco D'Ambros, and Romain Robbes. 2011. Enabling Program Comprehension through a Visual Object-focused Development Environment. In *Proceedings of VL/HCC 2011 (Symposium on Visual Languages and Human-Centric Computing)*. IEEE, 127–134. https://doi.org/10.1109/VLHCC.2011.6070389

[17] Marco Raglianti, Csaba Nagy, Roberto Minelli, and Michele Lanza. 2022. DiscOrDance: Visualizing Software Developers Communities on Discord. In *Proceedings of ICSME 2022 (International Conference on Software Maintenance and Evolution)*. IEEE, 474–478. https://doi.org/10.1109/ICSME55016.2022.00062

[18] Satrio Adi Rukmono, Michel R. V. Chaudron, and Christopher Jeffrey. 2024. Layered BubbleTea Software Architecture Visualisation. In *Proceedings of VISSOFT 2024 (Working Conference on Software Visualization)*. IEEE, 122–126. https://doi.org/10.1109/VISSOFT64034.2024.00024

[19] Harald Störrle. 2012. On the Impact of Layout Quality to Understanding UML Diagrams: Diagram Type and Expertise. In *Proceedings of VL/HCC 2012 (Symposium on Visual Languages and Human-Centric Computing)*. IEEE, 49–56. https://doi.org/10.1109/VLHCC.2012.6344480

[20] Alex Ulmer, Marco Angelini, Jean-Daniel Fekete, J orn Kohlhammer, and Thorsten May. 2024. A Survey on Progressive Visualization. *Transactions on Visualization and Computer Graphics* 30, 9 (2024), 6447–6467. https://doi.org/10.1109/TVCG.2023.3346641

[21] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *Proceedings of VISSOFT 2007 (International Workshop on Visualizing Software for Understanding and Analysis)*. IEEE, 92–99. https://doi.org/10.1109/VISSOF.2007.4290706

[22] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software Systems as Cities: A Controlled Experiment. In *Proceedings of ICSE 2011 (International Conference on Software Engineering)*. ACM, 551–560. https://doi.org/10.1145/1985793.1985868