Visual Loop: Bridging the Cognitive Gap in Software Development Through Visual-AI Collaboration

Luis F. Gomes
Carnegie Mellon University
Pittsburgh, PA, USA
University of Porto
Porto, Portugal
lfgomes@andrew.cmu.edu

Xin Zhou
Singapore Management University
Singapore City, Singapore
xinzhou.2020@phdcs.smu.edu.sg

David Lo
Singapore Management University
Singapore City, Singapore
davidlo@smu.edu.sg

Rui Abreu
Faculty of Engineering, University of Porto
Porto, Portugal
rui@computer.org

Abstract

Software development remains predominantly text-centric, despite decades of evidence showing that developers think and communicate visually. While sketches and diagrams externalize developers' mental models, they remain disconnected from source code and quickly become outdated. Recent advances in foundation models, capable of both code and visual reasoning, create an opportunity to unify these representations. In this vision paper, we introduce *Visual Loop*, a continuous visual development environment that keeps code and informal sketches in bidirectional synchronization. Our prototype connects a code editor with a tablet-based visualization workspace, allowing developers to explore, annotate, and modify systems through freehand sketches interpreted by multimodal LLMs. By grounding AI reasoning in static analysis and visual context, *Visual Loop* transforms sketching from passive documentation into an active interface for software evolution. We present illustrative scenarios, discuss cognitive and comprehension benefits, and outline directions for user studies and real-world integration.

CCS Concepts

• **Software and its engineering** → **Integrated and visual development environments**; • **Human-centered computing** → **Visualization toolkits**; **Interactive systems and tools**.

Keywords

Code Visualization, Sketch Recognition, LLMs, Developer Tools

ACM Reference Format:
Luis F. Gomes, Xin Zhou, David Lo, and Rui Abreu. 2026. Visual Loop: Bridging the Cognitive Gap in Software Development Through Visual-AI Collaboration. In *2026 IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (FORGE '26), April 12–13, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3793655.3793729

FORGE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2477-0/2026/04
https://doi.org/10.1145/3793655.3793729

1 Introduction

Software development faces a fundamental cognitive mismatch. Developers naturally think and reason in visual and spatial terms: sketching architectures on whiteboards, mapping data flows, and mentally simulating system interactions. These informal visualizations are not merely aesthetic; they are external representations of developers' mental models, helping them comprehend, communicate, and reason about complex systems [2, 6, 15]. Despite this visual nature of human cognition, developers are constrained to express and manipulate software purely through text.

Research in cognitive psychology consistently demonstrates that people process and remember visual information more effectively than textual information, a phenomenon known as the picture superiority effect [8, 27]. Dual-coding theory explains that visuals engage both verbal and visual memory systems, strengthening understanding and recall [18, 19]. Software developers, whose work inherently involves abstract, multi-level structures, are expected to perform such reasoning through text-based interfaces that provide little visual grounding [13]. This cognitive disconnect increases mental effort and makes it difficult for developers to maintain coherent mental models as software evolves [5, 13, 20]. After decades of IDE innovation, the dominant paradigm remains text-centric, ignoring the way human cognition naturally operates [29].

Over the years, multiple research directions have attempted to bridge the gap between code and visual reasoning. Traditional visualizations such UML, call graphs, and dependency graphs aim to clarify software structure but offer little flexibility for expressing design intent or evolving ideas [16, 22]. In practice, they quickly fall out of sync with code, becoming static documentation rather than active reasoning tools. Studies show that UML remains rarely used in real-world projects due to becoming outdated, insufficient detail, and weak integration with development workflows [23].

Other visualization systems, such as Software Cities [24–26], map code structures to spatial metaphors that helped developers explore complex systems. These tools effectively supported comprehension but remained read-only: developers can observe architectures but not manipulate or express design intent. At the opposite end, low-code and visual programming platforms, provide graphical interfaces for software creation [14, 17, 21]. However, they rely

on rigid, predefined visual vocabularies that constrain creativity and do not reflect how developers conceptualize systems [1, 12].

Other lines of work, such as SketchLink [4] and LivelySketches [3], connected analog sketches to source code, addressing archival and retrieval challenges. Yet, these systems did not maintain live synchronization, since changes in the drawings never propagated to code, and evolving codebases quickly broke the visual connection. Recent advances in large language models (LLMs) have reignited this vision, enabling sketch-to-code generation concepts [7, 9, 10] and automatic visual documentation from code [11, 28]. These approaches show that LLMs can translate between modalities, but they remain one-directional, lacking a continuous round-trip between code and visuals. A development environment where visual and code artifacts co-evolve is still missing.

To close this gap, we envision a new path for software development: a continuous visual development system. We call this vision *Visual Loop*, an environment where humans and AI share a common, evolving understanding of software through informal visuals. Rather than translating code into rigid diagrams or generating code from fixed visual vocabularies, *Visual Loop* enables a bidirectional dialogue between sketches and source code. In this new idea paper, we focus on demonstrating this concept, and we present a proof-of-concept example with a prototype that operationalizes the *Visual Loop* paradigm.

We demonstrate the concept through illustrative scenarios and discuss how shared visual understanding between humans and AI can reduce cognitive load, preserve intent, and improve comprehension. Finally, we outline future directions for evaluating this paradigm through user studies and extending foundation model reasoning to real-world development environments.

2 Visual Loop: The Vision

In *Visual Loop*, developers work with a tablet companion to their code editor, maintaining their mental model of the system as a living, manipulable artifact. They can express design intentions or modifications through freehand sketches, much as they would on a whiteboard, and the AI interprets these sketches contextually, reasoning about how they relate to the existing codebase and proposing corresponding changes. Unlike traditional visual languages, *Visual Loop* does not require a predefined syntax. Instead, the AI understands how to interpret developers' informal visuals in the same intuitive way that other humans would. This shared understanding transforms the development process from a sequence of textual commands into a continuous visual conversation between human cognition and machine reasoning.

2.1 Motivating Examples

This section motivates *Visual Loop* with examples of situations commonly encountered by developers.

Example 1: Onboarding Challenge. Alice joins a team working on an e-commerce platform. On her first day, she is asked to fix a bug in the checkout service. She spends hours searching through all files to understand the service architecture, drawing her own diagram on paper to map component relationships, and asking teammates. Her hand-drawn diagram becomes *outdated* after a refactoring PR is merged.

Problem. Alice must reconstruct the system's mental model from scratch using only text. The architectural knowledge exists only in senior developers' heads.

With *Visual Loop*

Alice opens the project in *Visual Loop*. The tablet immediately shows an auto-generated architecture diagram with all microservices.

She taps the "Payment Service" component on the tablet, and VS Code jumps to the relevant file. She circles three related components with her stylus and writes *"How to validate?"*

> **AI Assistant:** *"These three services coordinate through the ValidationQueue: (1) CheckoutService initiates, (2) PaymentGateway processes, (3) FraudDetector validates. See highlighted code."*

When the refactoring PR merges, the diagram updates automatically. Alice's annotations remain, now correctly aligned with the new structure.

Result. Alice saves times by understanding the system visually, navigates through diagrams, and maintains an always-current mental model.

Example 2: Design Discussion. A team gathers for an architecture review. The tech lead explains a proposed refactoring on the whiteboard, sketching both the current and new architectures while team members discuss and suggest changes. Someone takes a photo of the board before erasing it. Later, a developer manually translates the sketch into code but discovers inconsistencies during implementation. The photo shared in Slack becomes the only record, quickly outdated and not synced with the evolving code.

Problem. Rich visual design discussions rely on manual translation to code, leading to implementation drift and lost design rationale.

With *Visual Loop*

The team connects a tablet running *Visual Loop* to the projector, displaying the current architecture diagram. The tech lead sketches directly on the diagram—adding a new `CacheLayer`, drawing arrows from the API Gateway and Database, and crossing out the old direct connection. Next to the cache, she writes: *"Redis? What size?"*

> **AI Assistant:** "Proposed changes detected: add Redis-based CacheLayer, route API Gateway through CacheLayer, remove direct database calls. Cache configuration unresolved."

As the team debates alternatives, *Visual Loop* displays both options side by side for comparison. Once the team agrees, the AI applies the chosen design: it updates routing logic, adds TODO comments for configuration, and commits the synchronized architecture to the shared documentation.

Result. The design discussion directly becomes implementation. Less manual translation and no lost knowledge.

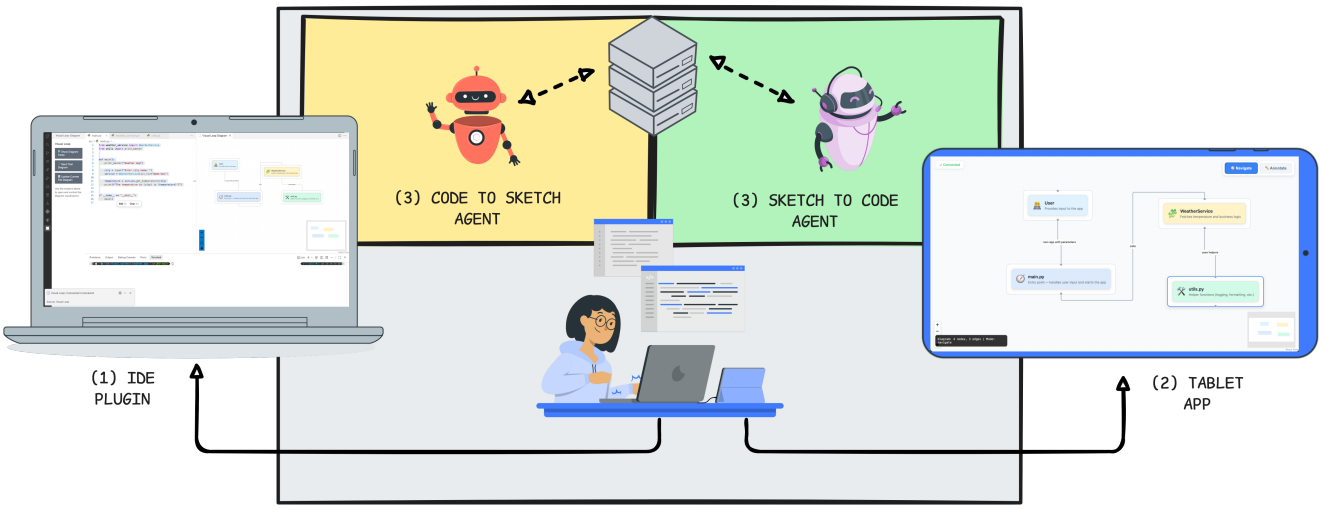

Figure 1: Visual Loop: a continuous visual development environment where code and sketches co-evolve through AI synchronization. Our prototype connects the IDE with a tablet interface, enabling developers to visualize, navigate, and modify systems through informal sketches: (1) desktop *IDE plugin* with static analysis and visualization hooks, (2) a tablet-based *Visualization Engine* for sketching and interaction, and (3) an *AI Synchronization Layer* powered by foundation models that maintain consistency between code and visuals in real time.

Example 3: Bug Hunt. Bob is debugging a race condition. He sets breakpoints in several files, reruns the program, and sketches the execution flow on paper. He discovers a timing issue between three asynchronous operations. The sketch helps him reason about the bug but stays informal and hard to share with teammates.

Problem. Debugging demands understanding dynamic behavior, and existing tools show only static code, leaving developers' mental models of execution unshared and disconnected from their environment.

With *Visual Loop*

It generates a live diagram of the payment flow, updating in real time as Bob steps through the debugger. When he notices three operations running in parallel, he circles them and writes "Race condition here?" on the tablet.

> **AI Assistant:** "Detected unsynchronized async calls: `UpdateInventory`, `ChargeCard`, and `SendConfirmation`. Suggest adding `await` on `ChargeCard` before `SendConfirmation`."

Bob draws an arrow from `ChargeCard` to `SendConfirmation` to enforce the correct order. The AI updates the code and synchronizes the diagram, showing the fixed flow. He then attaches the annotated visualization to the pull request, clearly illustrating the bug and its resolution. **Result.** Complex asynchronous debugging becomes intuitive and collaborative. Bob solves the issue visually before coding, and his team learns from the automatically generated documentation.

2.2 Prototype

The *Visual Loop* prototype exemplifies a new interaction direction where foundation models bridge the gap between code and human *visual* reasoning. Built on top of multi-modal LLMs, the system connects the developer's coding workspace with an intelligent visualization environment. Our prototype integrates three main components (Figure 1):

(1) **Code Editor (VSCode extension).** A standard development environment augmented with static analysis and contextual extraction. It feeds structured information—such as function hierarchies, data flow, and dependencies—into the foundation model, grounding its visual reasoning in the actual code semantics.

(2) **Visualization Engine (Tablet app).** A companion workspace for sketch-based interaction, real-time diagram updates, and code navigation. Developers can freely draw and annotate while the underlying model interprets these gestures and symbols within the code context. This interface transforms the tablet into an intelligent visual processor that understands intent and syntax.

(3) **AI Synchronization (LLM).** This layer orchestrates two complementary sets of AI agents: *Code to Sketch* and *Sketch to Code*. These agents leverage the multi-modal reasoning (code, text, and images) to maintain semantic alignment between visuals and source code. Code2Sketch updates diagrams that reflect the evolving program structure, while Sketch2Code interprets developer visual annotations to propose code edits or refactorings.

3 Proof-of-Concept Example': Weather App Demonstration

To illustrate *Visual Loop*, we implemented a minimal prototype using a VS Code extension and a tablet app (Figure 1). For the proof-of-concept, we use a simple weather application composed of three files: `main.py`, `utils.py`, and `weather_service.py`. The WeatherService class fetches and processes data using helper functions in `utils.py`.

In this proof-of-concept example, the goal is to remove the *WeatherService* class, identified as a source of redundant logic and complexity, and refactor the application to call `utils.py` directly. It follows the workflow depicted in Figure2:

1. Initialization The developer opens the project in *Visual Loop*. The desktop editor displays the auto-generated diagram, which can be freely rearranged with drag-and-drop to match the developer's mental model.

2. Synchronization The connected tablet mirrors this updated layout in real time, maintaining consistent spatial organization between devices.

3. Exploration Using the tablet in `Navigation Mode`, we can tap on nodes, automatically highlighting the corresponding code segments in VS Code. This live linkage bridges high-level structure and implementation details.

4. Annotation Switching to `Drawing Mode`, the developer sketches arrows and notes directly on the diagram. For example, cross out WeatherService, draw new links from `main.py` to `utils.py`, and write "direct call" as an annotation.

5. Interpretation Upon pressing Save, the hand-drawn modifications are sent to the LLM-based reasoning module. The AI interprets the visual intent and summarizes it[1].

> **AI Answer:** The user wants to make two main changes based on the hand-drawn annotations: 1. **Remove the "WeatherService" Component**. 2. **Add a direct call to utils.py**.

The summary is presented to the developer for confirmation. In future iterations, these AI inferences can be applied directly as candidate code edits.

The prototype was evaluated across 30 drawing interactions of four types. The model correctly interpreted most annotations, achieving the following accuracies: (1) Remove components: 4/5 cases (80%), (2) Add components: 6/7 (85.7%), (3) Remove calls: 6/8 (75%), and (4) Add calls: 9/10 (90%).[2]

4 Discussion

Visual Loop proposes a paradigm shift rather: a move toward continuous, visual–textual co-evolution of code and visuals. This transition raises conceptual, technical, and practical questions worth discussing.

Visual Loop is not a replacement for programming languages. It does not aim to remove code but to expand how intent is expressed. Visual interaction is most beneficial when it mirrors how

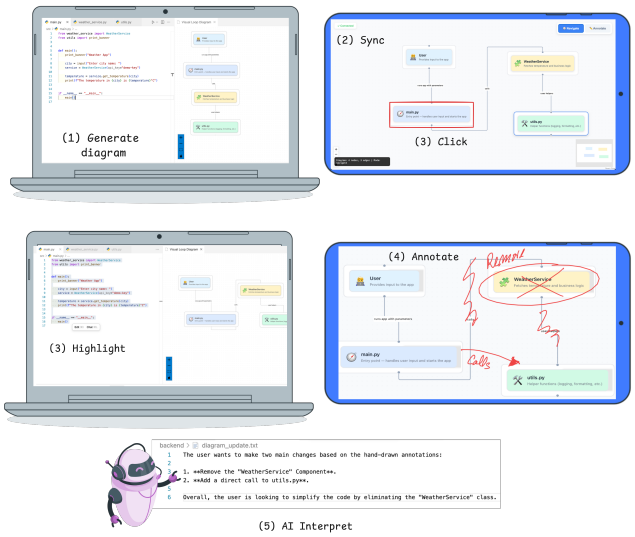

Figure 2: Proof-of-Concept Example: exploring code and rearranging diagrams to sketching changes, interpreting them through AI, and synchronizing updates between code and visuals.

developers reason. Text remains superior for precision and completeness, but visuals provide an intuitive medium for reasoning and for communicating structural changes. The two modalities are not competing; they form a complementary loop that aligns with human cognition.

Foundation models play a central role in this vision. Traditional visualization systems relied on predefined grammars or rule-based mappings between code and diagrams [16, 21, 22]. In contrast, foundation models bring generalization: they can infer structure from code and interpret informal sketches without relying on fixed visual vocabularies [10, 28]. This opens a new research path where models learn to ground abstract visual reasoning in executable semantics. However, this flexibility also introduces challenges in controllability, explainability, and reproducibility that must be addressed.

The approach raises questions about authorship and collaboration. When AI agents interpret or evolve diagrams and code, who is responsible for the resulting changes? In collaborative environments, multiple developers might annotate or sketch simultaneously, requiring version control mechanisms for text but also for visual artifacts and AI inferences.

The integration of visual reasoning introduces evaluation challenges. Unlike traditional IDE extensions or visualization tools, the quality of *Visual Loop* interactions depends not only on output correctness but also on how well the system aligns with human cognitive processes. Empirical studies on usability, and cognitive load are crucial to determine when visual co-development improves or degrades problem-solving.

[1]The model used in the prototype is OpenAI's gpt-4o-mini

[2]Some interactions are implicit; for instance, removing a node also implies removing its associated calls. Full experimental data are available at: https://github.com/luisgomes24/Visual-Loop.

5 Conclusion and Future Work

Software development has assumed that programming should be primarily textual, although humans think visually. The emergence of multimodal foundation models challenges this assumption, enabling tools that align with human cognition rather than computer syntax. In this position paper, we introduce *Visual Loop*, taking a first step toward this vision: a development environment where code and visuals co-evolve. AI interprets freehand sketches as developer intent, allowing the transition between visual reasoning and code. This is not a replacement for coding but a cognitively aligned interface where text and visuals show synchronized views of the system.

In future work, we plan to evolve the prototype into a fully functional environment that supports development workflows. We aim to conduct user studies with professional developers to evaluate how continuous visual–textual synchronization impacts comprehension, task efficiency, and collaboration. These studies will also collect qualitative feedback on usability cognitive benefits, guiding iterative refinement of the design. We open the door to broader research directions: (1) reliability of AI-driven visual understanding; (2) collaborative sketching and shared visual reasoning among teams; and (3) metrics to quantify how effectively visual artifacts reflect evolving code structures.

Acknowledgments

This research/project was supported by the National Research Foundation, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore. Support for this research was also provided by the Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) through the CMU Portugal Program under Grant PRT/BD/152343/2021.

References

[1] Md Abdullah Al Alamin, Sanjay Malakar, Gias Uddin, Sadia Afroz, Tameem Bin Haider, and Anindya Iqbal. 2021. An Empirical Study of Developer Discussions on Low-Code Software Development Challenges. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, Madrid, 46–57. doi:10.1109/msr52588.2021.00018

[2] Sebastian Baltes and Stephan Diehl. 2014. Sketches and diagrams in practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 530–541. doi:10.1145/2635868.2635891

[3] Sebastian Baltes, Fabrice Hollerich, and Stephan Diehl. 2017. Round-Trip Sketches: Supporting the Lifecycle of Software Development Sketches from Analog to Digital and Back . In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE Computer Society, Los Alamitos, CA, USA, 94–98. doi:10.1109/VISSOFT.2017.24

[4] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking sketches and diagrams to source code artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 743–746. doi:10.1145/2635868.2661672

[5] Abir Bouraffa, Gian-Luca Fuhrmann, and Walid Maalej. 2023. Developers' Visuo-Spatial Mental Model and Program Comprehension. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 1920–1932. doi:10.1109/ICSE48619.2023.00163

[6] Mauro Cherubini, Gina Venolia, Rob DeLine, and Amy J. Ko. 2007. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 557–566. doi:10.1145/1240624.1240714

[7] Marcelo d'Amorim, Rui Abreu, and Carlos Mello. 2020. Visual Sketching: From Image Sketches to Code . In *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE Computer Society, Los Alamitos, CA, USA, 101–104. https://doi.ieeecomputersociety.org/

[8] Paul Fraisse. 1968. Motor and verbal reaction times to words and drawings. *Psychonomic Science* 12, 6 (June 1968), 235–236. doi:10.3758/bf03331287

[9] Luís Gomes. 2023. Transforming Ideas into Code: Visual Sketching for ML Development. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Cascais, Portugal) *(SPLASH 2023)*. Association for Computing Machinery, New York, NY, USA, 10–12. doi:10.1145/3618305.3623588

[10] Lúıs F. Gomes, Vincent J. Hellendoorn, Jonathan Aldrich, and Rui Abreu. 2025. *An Exploratory Study of ML Sketches and Visual Code Assistants*. IEEE Press, Ottawa, Canada, 1653–1664. https://doi.org/10.1109/ICSE55347.2025.00124

[11] Luís F. Gomes, Xin Zhou, David Lo, and Rui Abreu. 2025. VisDocSketcher: Towards Scalable Visual Documentation with Agentic Systems. arXiv:2509.11942 [cs.SE] https://arxiv.org/abs/2509.11942

[12] Martin Hirzel. 2023. Low-Code Programming Models. *Commun. ACM* 66, 10 (Sept. 2023), 76–85. doi:10.1145/3587691

[13] 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 Softw. Engg.* 25, 6 (Nov. 2020), 4427–4471. doi:10.1007/s10664-020-09835-6

[14] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/985692.985712

[15] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 492–501. doi:10.1145/1134285.1134355

[16] Zheng Li, Aidan McGowan, Yan Liu, and Abdelwahab Hamou-Lhadj. 2025. UML Crisis! An Educational Perspective. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering* (Clarion Hotel Trondheim, Trondheim, Norway) *(FSE Companion '25)*. Association for Computing Machinery, New York, NY, USA, 895–900. doi:10.1145/3696630.3727246

[17] OutSystems. 2025. OutSystems AI-Powered Low-Code Platform. Available at: https://www.outsystems.com/.

[18] Allan Paivio. 1990. *Mental Representations: A dual coding approach*. Oxford University Press, Oxford. doi:10.1093/acprof:oso/9780195066661.001.0001

[19] Allan Paivio and Kalman Csapo. 1973. Picture superiority in free recall: Imagery or dual coding? *Cognitive Psychology* 5, 2 (1973), 176–206. doi:10.1016/0010-0285(73)90032-7

[20] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*. IEEE Press, Madrid, Spain, 524–536. doi:10.1109/ICSE43902.2021.00056

[21] Nikitha Rao, Jason Tsay, Kiran Kate, Vincent Hellendoorn, and Martin Hirzel. 2024. AI for Low-Code for AI. In *Proceedings of the 29th International Conference on Intelligent User Interfaces* (Greenville, SC, USA) *(IUI '24)*. Association for Computing Machinery, New York, NY, USA, 837–852. doi:10.1145/3640543.3645203

[22] Rebecca Reuter, Theresa Stark, Yvonne Sedelmaier, Dieter Landes, Jurgen Mottok, and Christian Wolff. 2020. Insights in Students' Problems during UML Modeling. In *2020 IEEE Global Engineering Education Conference (EDUCON)*. IEEE, Porto, Portugal, 592–600. doi:10.1109/educon45650.2020.9125110

[23] Joseph Romeo, Marco Raglianti, Csaba Nagy, and Michele Lanza. 2025. UML is Back. Or is it? Investigating the Past, Present, and Future of UML in Open Source Software . In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2342–2354. doi:10.1109/ICSE55347.2025.00155

[24] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, Banff, AB, Canada, 92–99. doi:10.1109/VISSOF.2007.4290706

[25] Richard Wettel and Michele Lanza. 2008. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 921–922. doi:10.1145/1370175.1370188

[26] Richard Wettel, Michele Lanza, and Romain Robbes. 2011. Software systems as cities: a controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 551–560. doi:10.1145/1985793.1985868

[27] Andrew J. O. Whitehouse, Murray T. Maybery, and Kevin Durkin. 2006. The development of the picture-superiority effect. *British Journal of Developmental Psychology* 24, 4 (Nov. 2006), 767–773. doi:10.1348/026151005x74153

[28] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 119, 14 pages. doi:10.1145/3654777.3676374

[29] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 29 (March 2022), 47 pages. doi:10.1145/3487569