Bridging the Gap: From Traditional to Futuristic Software visualisation Tools

Corvin Kögler

***Abstract*—The evolution of software development methodologies from traditional, waterfall approaches to agile practices has profoundly impacted the tools and techniques used for software visualisation. Historically, software development paralleled construction, with distinct phases for design, development, and maintenance, heavily relying on visualisation tools such as UML diagrams, flowcharts, and Gantt diagrams. These tools facilitated detailed upfront planning, essential when changes to a project were costly and time-consuming.**

However, the advent of continuous integration and continuous delivery (CICD), along with agile methodologies, has shifted the focus towards rapid, iterative development and close collaboration with customers. This shift has reduced the relevance of traditional visualisation tools, as the dynamic nature of modern software projects demands more flexible and less labour-intensive approaches.

Modern visualisation tools like Code Park and Gource offer innovative 3D and dynamic representations of codebases and the project's evolution. Despite their potential to enhance understanding and collaboration, these tools face challenges in performance, integration with popular Integrated Development Environments (IDEs), and practical limitations such as screen space requirements and the need for additional equipment. Consequently, their adoption in industry settings remains limited.

The rise of AI tools in software development further diminishes the need for traditional visualisation techniques. AI can automate various aspects of coding, including code generation, bug detection, and architectural planning, providing real-time feedback and reducing the need for manual visualisations. As the industry continues to prioritise speed, flexibility, and automation, traditional visualisation tools may become increasingly obsolete, and modern tools will need to adapt to find their place in the evolving landscape of software development.

***Index Terms*—software visualisation**

I. Introduction

Software development and construction share many parallels and utilise similar design methodologies. This historically extends to the project lifecycles. In the early days, software developers had to write significant amounts of their code on paper because computer time was scarce and machines were still slow. [1] Just like construction, the job of the architect and the construction worker (programmer) were separated because both jobs called for a different skill set. [2] The project was constructed on paper iterated on by multiple experts in different phases and ultimately constructed with the difference being that one industry worked with steel and concrete and the other with text inputs.

In construction one of the most popular design approaches is the waterfall concept where the project flows down the different steps and then ends in a maintenance loop. [3] Until recently software had to be treated similarly because once shipped updates were bound to significant effort. With the internet, the need for fully thought-out products in software mostly vanished reducing the time from the drawing board to the customer significantly. While software and construction still share similarities the field of software engineering has evolved in a different direction. With CICD it's becoming increasingly popular to develop the product with the customer, continuously iterating and refining it [4].

Companies picked up on this and largely switched to agile development techniques that are specifically designed to accommodate the new dynamic style of development. [3] It is now significantly cheaper to bring a product to the customer and it's normalised to find beta-state applications that are intentionally rapidly developing and unstable. Because of software frameworks and libraries, most basic functionality is ready for production reducing the need for writing new software to the unique aspects of the project.

Somewhere in this transition, the traditional visualisation tools lost most of their relevance. It's a significant overhead to update a UML diagram every time a library gets updated. Automatic tools never really gained traction in industry applications. While most traditional developers have to learn about UML diagrams and flowcharts during their education a significant chunk self self-thought developers don't. With modern tools, it's not important enough to take the time to visualise the codebase until there is a good enough reason for it.

This report researches modern alternatives to the traditional visualisation tools for software architecture discussing their advantages and drawbacks in comparison to traditional approaches and if there even is the need to use them anymore.

II. Traditional visualisations

Traditional visualisation tools for software architecture are manually created. [5] They specifically serve as a previous step to actually writing code and help pick up unwanted dependencies, architectural code smells, and other issues. For example, it's simple to see if Demeter's law was followed because in UML Class diagrams it's easy to spot violating connections between classes. Especially in the early days, it was significantly faster to work out the rough code structure before writing the first line of code. [1] The owner of the product specified what the code should do and developers constructed the architecture around that. Lastly, programmers typed the code and fixed any errors emerging. [1] If the code worked as intended and was tested enough it was ready to ship on a physical medium. If afterwards issues emerged it was logistically hard to bring a patch to the customer.

III. UML Diagrams

UML (Class) diagrams are a great tool to design the architecture of an application. [5] While unnecessary on smaller projects it's a great way to iterate on it before existing code prevents changes. In UML Diagrams the future code gets structured in classes planning out architectural patterns, needed methods, coupling, and communication between classes and instances. Afterwards, most of the code writing is simple enough to be handled asynchronously because the interface and shape are already defined. They are a great tool for teachers to test the understanding of students about architectural patterns without having to make them write large amounts of code.

The diagrams fit perfectly within the waterfall design approach because they allow building the whole structure of the project from the ground up and are fast to change. On larger projects with lots of connections, they tend to become convoluted fast but do a good job of showing spatial relations between modules. In turn, this clashes with rapid development techniques because prototypes are applied when the solution is not yet fully found. In this case, they have to be frequently altered manually creating a significant work overhead. While it is possible to automatically generate them from existing code applications lies within planning new code and less serving as a visualisation of existing ones. [5] Modern technology makes it possible to enhance them at least slightly by introducing interactivity and collapsing unwanted sections to clear some space. [6]

Creating a UML diagram involves several steps to visually represent the architecture and design of a software application. Beginning by identifying the key classes and components that will make up the system and determining the relationships between these classes, such as inheritance, associations, and dependencies. Defining the attributes and methods for each class, ensuring that they align with the planned functionality and design patterns. Arranging the classes and components spatially to reflect their relationships and interactions, using lines to connect them and arrows to indicate the direction of relationships. Labelling each class, attribute, method, and relationship clearly to provide a comprehensive understanding. Regularly reviews and iterations on the diagram, making adjustments as needed to accommodate new insights or changes in the design.

A. Flow Charts

Flow charts are a great tool to describe what an application should feel like to the end user. By visualising cause and effect they tend to be only marginally responsible for the architecture of the application. On the flip side, the architecture is very influential on what flows are possible for the end product. Often legacy code can act blocking to desired flows in the application. Well-structured code can mitigate this somewhat but building code with all possible changes in mind adds a significant amount of overhead to the code. In general, it's the fastest to write static code that relies on rigid systems to work. [7] This is often used for prototypes because there is no real benefit from making explorative code well-engineered if it has a high chance of being scrapped.

While traditionally one application was built for a single purpose due to resource limitations, popular modern applications tend to have a greater scope and therefore often no singular global user flow can be found. As an example, the first versions of writing applications had a clear workflow that they inherited from typewriters. The user was supposed to either create a new document or open an existing one, make changes to it, touch up the format and aesthetics, and then export it. Modern versions of Word include extensive graphics libraries, AI interfaces plugin systems, and basic calculation frameworks. Within this, a magnitude of different user flows is possible that the underlying architecture has to support.

Creating a flowchart involves a systematic approach to visually represent the sequence of actions or decisions within a process. Beginning with identifying all the tasks, actions, or decisions that need to be mapped out and determining the sequence of these tasks, noting any dependencies or conditions that influence the flow. Selecting appropriate symbols to represent different types of actions, such as rectangles for processes, diamonds for decision points, and arrows for the flow direction. Arranging these symbols on a canvas while ensuring a logical progression from the entry point to the endpoint. Drawing arrows to connect the symbols, clearly indicating the flow from one step to the next and labelling each step and decision to provide clarity. If the process includes complex decision points or loops, ensure these are accurately represented to reflect potential variations in the flow. Regularly review and update the flowchart to accommodate any changes in the process or to incorporate additional details. While flowcharts can be generated automatically from code, they often require careful interpretation to ensure accuracy, especially when representing flexible and well-engineered architectures designed to support a variety of user interactions and future changes.

B. Gantt Diagrams

Gantt diagrams, or Gantt charts, are project management tools that represent the timeline of tasks or activities against a calendar. [8] They illustrate the start and end dates of individual tasks, their duration, and their relationships with other tasks. In software development, Gantt charts are used to visualise the execution sequence of code, identifying which parts of the code run at specific times. This time-based visualisation helps in finding bottlenecks by highlighting areas where tasks overlap excessively or wait for dependencies, indicating potential inefficiencies. By revealing these patterns, Gantt charts offer insights into possible sources of misconfiguration. Additionally, they can highlight indirect dependencies that may not be immediately apparent in the code structure but impact performance and execution. This information is crucial for optimising both the architecture and the scheduling of tasks to improve overall system efficiency.

Creating a Gantt chart involves several systematic steps to visually represent the timeline and relationships of tasks or events. Beginning with listing all tasks, activities, or events that need to be tracked. For each task, determine the sequence and dependencies, identifying which tasks must precede or follow others. Assign start and end dates, or periods, to each task, estimating the duration required for completion. Set up a timeline along the horizontal axis and list the tasks along the vertical axis. Draw bars to represent the duration of each task, with the position and length of each bar indicating the start time, duration, and end time. If necessary, connect tasks with arrows or lines to denote dependencies and relationships. Regularly update the chart to reflect any changes in the schedule or task progression.

IV. Modern visualisations

Modern visualisation approaches mostly failed to gain traction in industry applications. There seems to be a void between severely outdated tools from the beginning of the 2000s and futuristic ones made for augmented and virtual reality. While visualisation is present in most modern development toolkits none aim primarily to show the architecture or relation between files apart from the file structure.

Most developers use an IDE to write their code. These come loaded with tools and plugins that sometimes leverage visualisations. For example, most IDEs bring a Git integration that visualises evolutionary data about the project in a graph structure. While this is useful the lack of architectural visualisation is surprising. Most architectural and structural visualisations aim to grant researchers insights into open-source repositories for analytics. [9] Therefore many tools are one-off prototypes that serve in the research context and get dropped afterwards. A significant amount of them were developed in the early 2000s when UML and waterfall approaches were vastly more popular than today. [3]

A. Code Park

Code Park is a 3D software visualisation tool that builds on the human understanding of spatial relations in the environment. [10] It uses cities as metaphors for making the code approachable even in large project sizes. For example, buildings represent classes while districts represent packages/modules. The properties of the buildings, such as height and base size, can describe different metrics like lines of code in the class, number of methods, or complexity. Using these parameters, architectural flaws can show in overly big buildings or district sizes. The model is designed to allow exploration of existing code and integration with plugins that deepen the understanding of the user.

Code Park aims to improve a programmer's understanding of an existing codebase in a manner that is both engaging and intuitive, appealing to novice users such as students. [10] It achieves these goals by laying out the codebase in a 3D park-like environment. Each class in the codebase is represented as a 3D room-like structure. Constituent parts of the class (variables, member functions, etc.) are laid out on the walls, resembling a syntax-aware wallpaper. Users can interact with the codebase using an overview and a first-person viewer mode.

The tool is designed to explore the effects of spatial recognition when interacting with the codebase, featuring multiple view modes. These include an exocentric (bird's eye) view and an egocentric (first-person) view, allowing users to examine the codebase at different granularities. The bird's eye view provides a holistic understanding of the codebase, while the first-person view enables a detailed exploration of individual classes. This dual-view approach helps users become familiar with and memorise the code structure more effectively. [10]

Additionally, Code Park supports syntax parsing, enabling features like go-to definition, which allows users to quickly jump to the location where a variable or function is defined. This helps in mentally and visually connecting the disparate parts of the code. The tool also incorporates animated transitions to maintain the user's spatial awareness, making navigation intuitive and engaging. [10]

While Code Park is a powerful tool in a research context, its application in a business environment presents several challenges. The use of the Unity game engine, while providing a rich and interactive 3D environment, also introduces significant resource overhead. This can lead to performance issues, particularly on lower-end hardware or when dealing with large codebases. Furthermore, the inability to integrate Code Park into existing Integrated Development Environments (IDEs) is a significant drawback. Developers often rely on the robust toolsets provided by their IDEs, and the lack of integration means they would have to constantly switch between the IDE and Code Park, disrupting their workflow. Another consideration is the portability of work. One of the appealing aspects of software development is the ability to work from anywhere, often on a laptop. However, the 3D visualisation of Code Park requires a considerable amount of screen space, which is a luxury not always available on smaller laptop screens. This could potentially limit the usability of the tool for developers on the go. Lastly, the requirement for additional equipment, such as high-resolution monitors or even VR headsets for the best experience, may not be feasible or cost-effective for many organisations. This adds an extra layer of complexity and cost in a business setting, where budget constraints are a significant consideration.

V. Gource

Gource is not just a visualisation tool, but a powerful instrument for project management and team collaboration. By providing a real-time, interactive view of the project structure, it facilitates a deeper understanding of the codebase and its evolution over time. This dynamic representation allows project managers and team members to identify potential bottlenecks, track progress, and allocate resources more effectively. [11]

The unique feature of representing each developer as an icon on the graph fosters transparency and accountability within the team. It provides a visual record of each developer's contributions, making it easier to recognise individual efforts and promote effective collaboration. This feature can also aid in conflict resolution by clearly showing the areas of the codebase each developer has worked on. Moreover, Gource's open-source nature invites continuous improvement and adaptation. Developers can customise the tool to better suit their project's needs or contribute to its development, enhancing its functionality and usability. This flexibility makes Gource a versatile tool, adaptable to a wide range of projects, regardless of their size or complexity.

In conclusion, Gource goes beyond traditional software visualisation tools by offering a comprehensive, interactive, and engaging view of a project's structure and development process.

Despite its innovative nature Gource is not widely adopted. This could be because the focus on collaboration visualisation is largely irrelevant to well-organised teams. Additionally, it does not integrate with common IDEs and does not offer structural insights beyond the file structure.

VI. Conclusion

The evolution of software development methodologies from waterfall to agile has significantly altered the landscape of software visualisation tools. Traditional methods like UML diagrams, flowcharts, and Gantt charts served their purpose well during the early days of software engineering, providing clear and structured ways to plan, document, and manage projects. These tools were indispensable in a time when changes to a codebase were costly and time-consuming, making thorough upfront planning crucial.

However, as the field has shifted towards more dynamic and iterative approaches, the need for these static and often labour-intensive visualisations has diminished. CICD practices, combined with agile methodologies, emphasise quick iterations and direct collaboration with customers. This reduces the time from concept to deployment and increases the flexibility to make changes, rendering traditional visualisation tools less effective and relevant.

Modern visualisation tools, such as Code Park and Gource, offer innovative ways to interact with and understand codebases. Code Park leverages 3D visualisations to create an immersive experience, while Gource provides a dynamic representation of project evolution and collaboration. Despite their potential, these tools have not seen widespread adoption in industry settings. The primary reasons include performance issues, lack of integration with popular Integrated Development Environments (IDEs), and practical limitations like screen space requirements and the need for additional equipment.

The landscape of software development is further changing with the rise of AI tools. These tools promise to automate many aspects of coding, including code generation, bug detection, and even architectural planning. As a result, the necessity for developers to manually create and maintain detailed visualisations may continue to decrease. AI-driven insights and analytics could provide real-time feedback and suggestions, making traditional visual tools obsolete for many practical purposes.

In essence, while older visualisation techniques were once indispensable, their relevance has waned in the face of new methodologies and technologies. The industry now favours tools and practices that prioritise speed, flexibility, and automation. Modern visualisation tools, though innovative, have yet to find their place in the everyday toolkit of developers.

References

[1] N. Wirth, "A brief history of software engineering," vol. 30, no. 3, pp. 32–39, conference Name: IEEE Annals of the History of Computing. [Online]. Available: https://ieeexplore.ieee.org/document/4617912/?arnumber=4617912

[2] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," vol. 17, no. 4, pp. 40–52. [Online]. Available: https://dl.acm.org/doi/10.1145/141874.141884

[3] E. Kisling, "Transitioning from waterfall to agile: Shifting student thinking and doing from milestones to sprints."

[4] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "CI/CD pipelines evolution and restructuring: A qualitative and quantitative study," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 471–482, ISSN: 2576-3148. [Online]. Available: https://ieeexplore.ieee.org/document/9609201/?arnumber=9609201

[5] R. Pais, L. Gomes, and J. P. Barros, "From UML state machines to petri nets: History attribute translation strategies," in *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pp. 3776–3781, ISSN: 1553-572X. [Online]. Available: https://ieeexplore.ieee.org/document/6119924/?arnumber=6119924

[6] draw.io. [Online]. Available: https://app.diagrams.net/

[7] M. Srivastava and R. Brodersen, "Rapid-prototyping of hardware and software in a unified framework," in *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, pp. 152–155. [Online]. Available: https://ieeexplore.ieee.org/document/185217/?arnumber=185217

[8] Gantt.com. [Online]. Available: https://www.gantt.com

[9] C. Anslow, "Collaborative software visualization in co-located environments."

[10] P. Khaloo, M. Maghoumi, E. Taranta II, D. Bettner, and J. Laviola Jr, "Code park: A new 3d code visualization tool." [Online]. Available: http://arxiv.org/abs/1708.02174

[11] Gource - a software version control visualization tool. [Online]. Available: https://gource.io/