Pervasive Software Visualizations

(Keynote)

Tudor Gîrba
tudorgirba.com
Keynote speaker

Andrei Chiş
University of Bern, Switzerland

***Abstract*—A picture tells a thousand words. We all know that. Then why are our development tools showing mainly text with so much obstinacy? Even when visualizations do make it into our tools, they typically do not make it past the periphery. Something is deeply wrong. We argue that visualizations must become pervasive in software development, and to accommodate this goal, the integrated development environments must change significantly.**

I. IDEs and the Problem of Code Reading

A significant part of the problem stems from the perception that software development is an activity of producing code. Yet, we know since a long time that developers spend the largest chunk of their time on understanding the existing systems [1].

Nevertheless, the integrated development environment (IDE) still favors the creation part. Just consider this: the central and largest part of the state-of-the-art development environments is taken by an editor. An editor! This is a tool used to enter or alter text. The IDE highly favors the typing part, but it overlooks the understanding needs. The I in the IDE is not as integrated as we might want to believe.

Let us now consider the impact of this choice.

A. The Magnifier Effect

An editor exposes the developer to some 50 lines of code at a time. This represents a tiny fraction of the overall system. It is as if we would hire someone to build a city and we would provide a magnifier glass as the only available inspection tool. The magnifier glass is a fine tool when we deal with details, but it is a terrible choice when it comes to understanding the big picture.

B. The Misleading Sense of Control

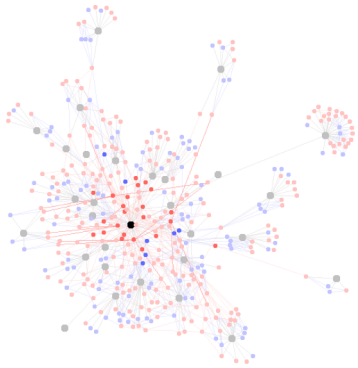
Looking at a class in an editor, makes us think of it in isolation. Its shape is well defined, with a clear begin and a definite end. Yet, code lives in a context.

To materialize the abstract notion of context, let us look to the right at an example of a class depicted through the lens of a System Attraction visualization[1]. In this visualization, classes are black, methods are red and attributes are blue. Method containment, attribute containment, and class inheritance are gray. Invocations are red. Attribute accesses are blue. Everything is arranged through a force based layout where each edge has equal weight.

[1]http://www.humane-assessment.com/blog/system-attraction/

When observed in isolation, our class, has a clear shape that looks controllable.

But, when we look at the class in a broader context and consider the collaborator classes as well, the shape of our class gets radically different, as the inner parts of our class are pulled away towards other points of interest.



This is a simple visualization that builds on the premise that code pieces should go next to their connections. While the design choices of the visualization, such as the weight allocated to different edges, can still be debated, the presence and impact of context is evident.

The simplicity provided by the physical locality of code that an editor offers is suddenly less suitable for understanding the complex nature of software systems. The code editor interface ignores the context altogether, but context matters and ignoring it will not make its impact go away.

C. The Implicit Nature of Code Reading

Developers approach understanding the system typically by reading code [2]. This is the most manual way to approach data, and data is what systems are made of. This needs to change. It does not scale, it is inaccurate and it is much too expensive.

Then why do developers read code? One reason is that reading is the inverse of writing. It is simply what you do with text. As long as developers will perceive software systems as being made out of text, reading will continue to be the most straightforward approach. Of course, developers rarely want

978-1-4673-7526-9/15 © 2015 IEEE

VISSOFT 2015, Bremen, Germany

to read code. They mostly want to understand it enough to be able to act. Reading is just the employed strategy.

Summarizing McLuhan's work [3], [4], Culkin warned us since half a century ago that *we shape our tools and thereafter our tools shape us* [5]. While he was referring to media in general, this unwritten law acts silently in software development as well, with or without our explicit approval.

For example, developers read code all day long, but they do not talk about it. They talk about the code they read, but not about how they read that code. The subject is not considered worthy of attention mostly because reading is something taken for granted. Essentially, this means that, as an industry, we are spending the largest chunk of the development budget on something we never talk about. It does not make sense, and yet it is a sensible consequence of McLuhan's prediction. Developers read because this is what the tool makes it easy for them to do.

Developers build tools to automate the tedious part of other people's decision making, but when it comes to understanding their own problems, they tend to resort to manual labour. While this situation is far from optimal, it also presents a great opportunity: developers already know how to approach data.

Let us take another example. If a developer needs to figure something out from 1 million database entries, the developer would likely start with a query. Yet, if the same developer receives a 1 million lines of code, the developer likely starts scrolling and navigating. The main difference between the two situations is the perception of the nature of the problem: the database problem is perceived as a data problem, while the code one is seen as a problem of text with occasional links.

Interestingly, the design of the tools matches that perception, too. The database tool typically presents a query field with a powerful enough language behind:

At the same time, the typical code tool focuses mainly on the editor and the navigation:

There certainly exist code query tools, but they are not at all as prevalent or as integrated as in the case of database tools. The lack of querying availability makes developers rely on something else. Nevertheless, when tools like NDepend[2] make it in the IDE, queries do get written.

[2]http://ndepend.com

We shape our tools and thereafter our tools shape us. This suggests that if we want to change the behavior, we should start from the tool.

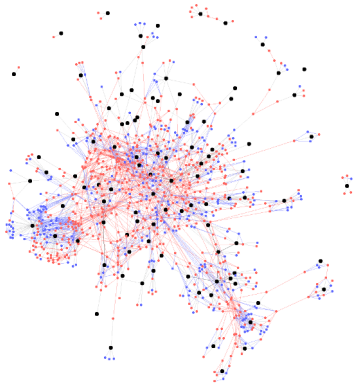
II. On the Lack of Visualizations in IDEs

Software visualization has come a long way from the initial explorations started in the previous century [6], [7] to the advanced techniques developed more recently [8], [9]. Yet, several decades after we first observed the problem of code reading, we still find ourselves starting papers, including this one, quoting the same mantra of how expensive code reading is [10]. This implies that all the effort poured in research and development in program comprehension has made little dent in the real world.

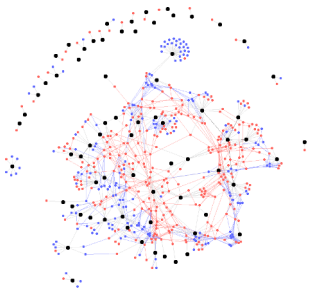
How are we to affect that trend? Simply providing sophisticated analyses does not seem to be enough.

We advance an extra reason for why this is the case: the typical alternatives to code reading are rigid, generic tools. In contrast, code reading is highly malleable. It deals with any context. And, software systems are highly contextual. To compete with code reading, we need tools that can be molded to that context.

Let us take an example. The class shown at the beginning of this article is part of a tiny student project created by a team of four students over the course of six weeks. Here is the whole system:



The project implements a basic calendar app for Android. The interesting thing about the example is that another team of four students developed an app with similar functionality within the same time frame. This second system looks as follows:



Even though both the functionality and the basic technology are highly similar, the resulting structure is radically different. This suggests that software structure is an emergent property that cannot be predicted from outside [11]. A further implication is that the understanding needs of the first team are different from those of the second team.

Our tools have to deal with that difference because it is a necessity inherent to the nature of software systems. If we want visualizations to become more pervasive, the underlying engines first have to become moldable.

III. Pervasive Software Visualizations

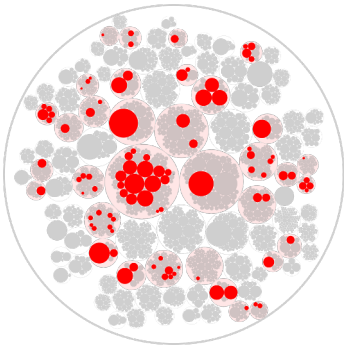
Visualizations must become prominent in the software development world. In our view, there are at least two distinct directions that have to be pursued in order to achieve this goal: (a) moldable visualizations, and (b) visual workflows.

A. Moldable Visualizations

Developers represent a special class of users. They are not clickers. They are primarily programmers. They approach data through programs, and they should be given the opportunity to approach the data around their own software systems in the same way. This implies that the tools dedicated to developers should not focus on providing ready-made functionality, but on offering rich programmability. When viewed in this way, the IDE becomes a language and the question transforms from *how many predefined analyses exist?* to *how inexpensive is it to craft a new one?*[3].

Tools such as Mondrian [12], d3[4] [13] or Roassal[5] [14] offer models that make defining the visualization concise. Moreover, these visualization engines come with deep integration with the underlying runtime and offer rapid prototyping options. These are but a few examples of the recent trend towards programmable visualizations, and they show that indeed, the cost of an effective visualization can be dramatically low.

For example, the circular treemap below expose the extension points in the Glamorous Toolkit[6] — its most important feature. The visualization code has 38 lines, including data gathering, and was created within less than one hour[7].



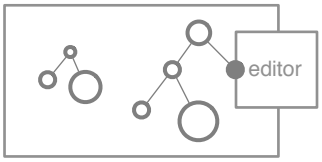
[3]http://humane-assessment.com
[4]http://d3js.org
[5]http://agilevisualization.com
[6]http://gt.moosetechnology.org
[7]http://www.humane-assessment.com/blog/communicating-pharo-4-0

This drop in cost opens up the option for developers to go as far as to craft a custom visualization, use it for some minutes, and then throw it away. This might sound far fetched, but spreadsheet programs made a similar workflow possible a long time ago. Such engines must become readily available in the IDE.

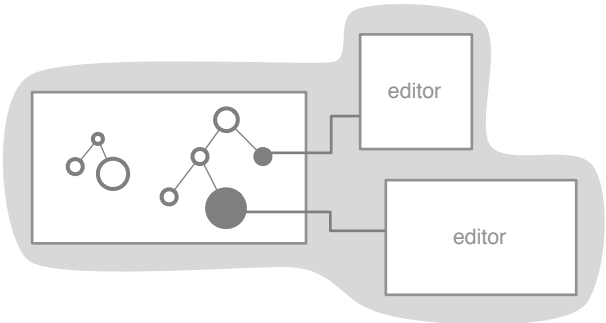
B. Visual Workflows

For these engines to be effective, the IDE has to welcome visualizations. The code editor must become secondary. It is important to edit code, but editing comes ideally after understanding. This challenges the design of current IDEs, as they are too centered around fixed navigation and code editing:

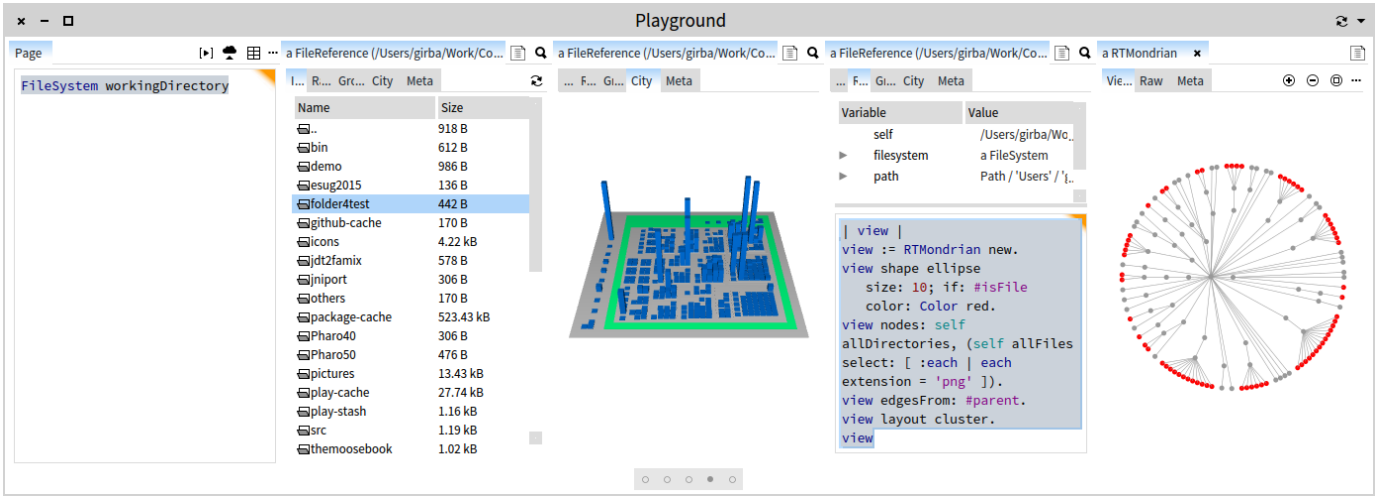
Visualizations must receive natural places inside the IDE. For example, if a visualization provides the overview, editing should be embedded within that experience and be driven by that overview. Classic, tree-like navigation becomes irrelevant in many situations, too:



An appealing example is provided by the design of Code Bubbles [15]. The layout itself is a visualization that can be affected during development to capture the current work focus and causality of navigation. Extending this model to ensure an even deeper connection between visualization and the manipulating panes can also be imagined:



While most IDEs focus on code, software systems are more than just code. Configurations, tests, documentation, or logs are equally important artifacts. Furthermore, the runtime of the system comes with yet another level of challenge for which textual representations do not fit at all. All these artifacts are important for understanding the system.



For example, an object inspector is typically a dry tool displaying every object through a generic tree widget. Through the Glamorous Toolkit project, we developed the concept of a moldable inspector that offers multiple views for each object and tracks the path of the inspection session [16].

The picture from on top shows an inspection session using a moldable inspector. In the first pane, we write a query, in this case a simple one retrieving the current folder. The second pane inspects the resulting object — an instance of FileReference. As the object corresponds to a folder, the developer sees the content of that folder. Selecting one subfolder spawns the third pane which displays a 3D CodeCity visualization [17] of the subfolder structure, where the blue leaves represent the files of different dimensions. Selecting one of the folder (depicted with green) from the visualization spawns another pane in which we handcraft a custom visualization using a dedicated Roassal graph builder [18]. Evaluating the script shows the resulting visualization object to the right with a view rendering its graphical representation.

The inspector offers different views for each object through tabs, and these views are defined as extensions of the classes of the respective objects. The circular treemap visualization from the previous page shows the amount of extensions that are distributed through the default Pharo 4.0 release[8]. Notably, these extensions have on average 8 lines of code.

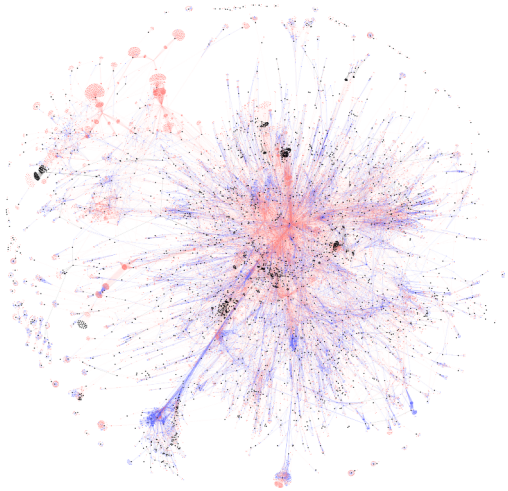
This simple example reveals that even basic tools such as an inspector can offer rich opportunities that are otherwise overlooked. Furthermore, it shows how standard and custom visualizations and queries can coexist in a uniform tool. This is possible because the tool was conceived to welcome this integration in one visual workflow. This is by no means the ultimate design, but it is an attestation for how such visual workflows are achievable when we design with intent.

IV. Final Words

The IDE has to change radically. Understanding software systems has to be supported as a critical activity that is integrated in the overall workflow. Visualizations cannot be ostracized anymore, and cramming visualizations in the typical design of current IDEs is not a promising strategy. We need to start from scratch and make visualizations first class citizens in the IDE because a picture tells a thousand words, and we need to tell effectively billions of such words.

Tools are important, but ultimately, it is the behavior of developers that we need to affect. There certainly exists an educational side to the problem of adopting visualizations, or other comprehension tools [10]. From our experience of working with and training software development teams, the precondition to this adoption is to get developers to perceive their own problems as data problems. This can be achieved by treating developers for what they are: developers. They have to be empowered with the ability of crafting tools to match their own problems. Once that happens the alternatives to code reading become more apparent.

Finally, we should remind ourselves and our fellow engineers that software systems represent perhaps the most sophisticated creations we have ever built, and that beyond the challenge of constructing and understanding them effectively, there is tremendous beauty lying underneath.



[8]http://pharo.org

ACKNOWLEDGMENTS

We would like to thank the Glamorous Team and the communities from around Moose and Pharo for their amazing work. That work makes dreaming the future easy, and building that future possible.

REFERENCES

[1] V. Basili, "Evolving and packaging reading technologies," *Journal Systems and Software*, vol. 38, no. 1, pp. 3–12, 1997.

[2] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer – an investigation of how developers spend their time," in *Proceedings of ICPC 2015 (23rd IEEE International Conference on Program Comprehension)*, 2015.

[3] M. McLuhan, *Understanding Media: The Extensions of Man*. New York: Mentor, 1964.

[4] M. McLuhan and Q. Fiore, *The Medium is the Massage: An Inventory of Effects*. Penguin Books, 1967.

[5] J. M. Culkin, "A schoolman's guide to marshall mcluhan," *The Saturday Review*, Mar. 1967. [Online]. Available: http://www.unz.org/Pub/SaturdayRev-1967mar18-00051

[6] H. A. Müller and K. Klashinsky, "Rigi — a system for programming-in-the-large," in *ICSE '88: Proceedings of the 10th international conference on Software engineering*. IEEE Computer Society Press, 1988, pp. 80–86. [Online]. Available: http://portal.acm.org/citation.cfm?id=55832

[7] T. Ball and S. Eick, "Software visualization in the large," *IEEE Computer*, vol. 29, no. 4, pp. 33–43, 1996.

[8] S. Ducasse and M. Lanza, "The Class Blueprint: Visually supporting the understanding of classes," *Transactions on Software Engineering (TSE)*, vol. 31, no. 1, pp. 75–90, Jan. 2005. [Online]. Available: http://scg.unibe.ch/archive/papers/Duca05bTSEClassBlueprint.pdf

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

[10] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 255–265.

[11] T. Gîrba, "The emergent nature of software architecture," *NDC Magazine*, vol. 1, no. 1, pp. 50–55, May 2015.

[12] M. Meyer, T. Gîrba, and M. Lungu, "Mondrian: An agile visualization framework," in *ACM Symposium on Software Visualization (SoftVis'06)*. New York, NY, USA: ACM Press, 2006, pp. 135–144. [Online]. Available: http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf

[13] M. Bostock, V. Ogievetsky, and J. Heer, "D3 data-driven documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011. [Online]. Available: http://dx.doi.org/10.1109/TVCG.2011.185

[14] V. P. Araya, A. Bergel, D. Cassou, S. Ducasse, and J. Laval, "Agile visualization with Roassal," in *Deep Into Pharo*. Square Bracket Associates, Sep. 2013, pp. 209–239.

[15] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: a working set-based interface for code understanding and maintenance," in *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. New York, NY, USA: ACM, 2010, pp. 2503–2512.

[16] A. Chiş, T. Gîrba, O. Nierstrasz, and A. Syrel, "The moldable inspector," in *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2015. New York, NY, USA: ACM, 2015, p. to appear. [Online]. Available: http://dx.doi.org/10.1145/2814228.2814234

[17] R. Wettel and M. Lanza, "Codecity: 3d visualization of large-scale software," in *ICSE Companion '08: Companion of the 30th ACM/IEEE International Conference on Software Engineering*. ACM, 2008, pp. 921–922.

[18] A. Bergel, S. Maass, S. Ducasse, and T. Gîrba, "A domain-specific language for visualizing software dependencies as a graph," in *Proceedings of 2nd IEEE Working Conference on Software Visualization (VISSOFT NIER)*, 2014. [Online]. Available: https://dl.dropboxusercontent.com/u/31543901/MyPapers/Berg14c-Graph.pdf