IDE-assisted visualization of indebted OO variability implementations

Johann Mortara, Philippe Collet, Anne-Marie Dery-Pinna

IDE-assisted visualization of indebted OO variability implementations

Johann Mortara
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
johann.mortara@univ-cotedazur.fr

Philippe Collet
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
philippe.collet@univ-cotedazur.fr

Anne-Marie Pinna-Dery
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
anne-marie.pinna@univ-cotedazur.fr

ABSTRACT

Object-Oriented (OO) variability-rich software systems often implement their variability in a single codebase, using the mechanisms provided by the host language (*i.e.*, inheritance, overloading, design patterns). This variability is not documented and buried deep down in the code, thus impeding its identification and making it especially prone to variability debt at the code level. While this kind of variability implementation can now be detected, visualization support such as *VariCity* helps architects and developers understand the implemented variability using a city metaphor. In this paper, we demonstrate *VariMetrics-IDE*, an extension of *VariCity* that allows to visualize multiple quality metrics (*e.g.*, code complexity, test coverage) together with the variability implementations, while supporting navigation between the source code and the visualization in an IDE. This extension thus facilitates the identification of zones of variability implementations with variability debt.

CCS CONCEPTS

• **Software and its engineering** → **Software product lines**; **Software reverse engineering**; *Object oriented architectures*; • **Human-centered computing** → *Visualization systems and tools.*

KEYWORDS

software variability, technical debt, software visualization, quality metrics, object-oriented systems, reverse-engineering

ACM Reference Format:
Johann Mortara, Philippe Collet, and Anne-Marie Pinna-Dery. 2022. IDE-assisted visualization of indebted OO variability implementations. In *26th ACM International Systems and Software Product Line Conference - Volume B (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3503229.3547066

1 INTRODUCTION

When they reach a larger scale, sofware systems mainly become variability-rich [3] and use various mechanisms to implement their variability. Introducing additional complexity, these variability implementations are difficult to understand, maintain and test, which leads to technical debt [4]. The debt caused by variability implementations has been defined as *variability debt*, while its different forms are still to be determined Wolfart et al. [18]. Not following the Software Product Line (SPL) paradigm, many object-oriented (OO) systems implement their variability in a single codebase, using the traditional OO mechanisms (*i.e.*, inheritance, overloading of methods and constructors, design patterns) [1]. The variability is then intertwined with the structure and implementation of the business code, and without traceability to the domain knowledge, the identification and understanding of these variability implementations are hindered [13, 14]. Moreover, as the OO mechanisms may cause technical debt, variability debt [18] is likely to be introduced at the code level.

Some recent approaches such as *symfinder* and its extensions [8, 9, 13] detect potential variation points (*vp*-s) with variants in the targeted systems. Dense zones of detected symmetries have also be shown to represent interesting locations in terms of variability implementations. As identification is quite cumbersome on large systems [9] the *VariCity* visualization, based on the city metaphor, was proposed and validated on large Java variability-rich systems [6]. While this helps architects and developers understand the implemented variability, no support was given to ease the browsing between the code and this visualization. This is quite a strong limitation as developers use Integrated Development Environments (IDEs) as tools support to assist development and program comprehension activities [5]. Moreover, with some better knowledge of the variability implementations, the need for determining their technical debt naturally arises. In a companion paper [7], the authors have proposed *VariMetrics*, an extension of *VariCity* to display zones of OO variability implementations with technical debt. It provides additional visual properties on the buildings representing classes to display some quality metrics, such as code duplication, code complexity, and test coverage. *VariMetrics* was successfully applied to several large open-source projects.

In this tool demonstration paper[1] we present *VariMetrics-IDE*, an IDE integration for *VariMetrics* that embeds the visualization in the JetBrains IntelliJ IDE, enables its configuration through the IDE's menus. The developers can then use all the capabilities of *VariMetrics*, including the display of OO quality metrics, as any other IntelliJ plugin without getting out of the development environment. Both configuration of the view and bi-directional navigation between the code and the buildings in the visualization are supported. We expect this extension to facilitate the identification of zones of variability implementations with variability debt.

SPLC '22, September 12–16, 2022, Graz, Austria

[1] External information about *VariMetrics-IDE* consists in a video available at https://youtu.be/lRPK8nS5JMc and the experimental results from [7] available at https://deathstar3.github.io/varimetrics-demo/.

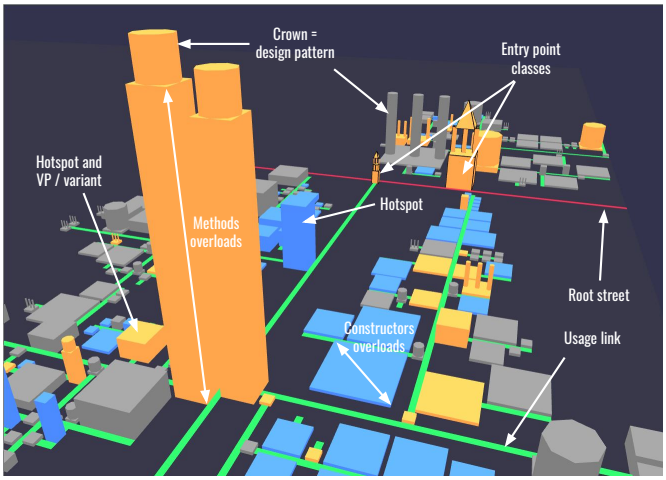

Figure 1: Main visual properties of *VariCity*, taken from [6].

2 IDENTIFYING OO VARIABILITY IMPLEMENTATIONS

In variability-rich OO systems, the mechanisms used to implement variability in a single code base , namely inheritance, overloading, and design patterns, do not align well with domain features, making their identification difficult. When these mechanisms are used to implement some variability, they also exhibit a property of symmetry [2, 15] since the commonality and variability addressed by the mechanism also represent the unchangeable and changeable parts in code assets. For example, inheritance allows us to factorize common parts (unchangeable) of multiple subclasses (the changeable parts) into a superclass.

As the unchanged and changed parts can be abstracted in terms of variation points (*vp*-s) with variants [12], a symmetry-based identification approach has been previously proposed [15] and tooled with the the *symfinder* toolchain [8]. The approach also relies on the notion of density of symmetries in the code to reveal the variability organization (*i.e.*, *vp*-s with an important number of variants at class or method level) [13] and display it in the shape of a graph. To fix the lack of precision of this first approach in *symfinder* [13], the usage relationships (type used as an attribute or method parameter) has been taken into account with the definition of a parameterized density measure that enables to automatically reveal fewer but more relevant *hotspot* zones concentrating variability implementations [9]. However, the resulting *symfinder*'s graph visualization is then harder to interpret on large systems with two types of relationships being displayed at the same time.

3 A CITY-BASED VISUALIZATION

3.1 *VariCity*

The city metaphor [16] has been applied to multiple types of metrics on software systems. In particular, CodeCity [17] and Evo-Streets [11] rely on the city metaphor to represent quality metrics on classes of an OO system. *VariCity* adapts the city metaphor to exhibit density of variability implementations [6] (*cf.* fig. 1). Classes are buildings whose dimensions evolve according to their method level variability metrics (method variants for the height and constructor variants for the width), exhibiting variability concentration at method level inside a class. As for design patterns, they usually involve several classes, but only the identified *vp* is highlighted in the visualization through a specific crown on the building.

The streets in the city shape it according to the usage relationships, starting from *entry point* classes, representing interesting points to explore the systems. These classes are positioned on a red street and other streets represent usage relationships. Buildings are placed by decreasing order of width on both sides of the street to exhibit density between classes. Additional usage relationships are represented as underground streets, and inheritance relationships as aerial links, both displayed when hovering a building. To make the *hotspot* zones of variability detected by *symfinder* (see previous section) noticeable, they are displayed in color (*vp*-s in yellow and variants in blue).

The visualization is itself configurable through several parameters, the first one being the *entry point* classes, more entry points tends to generate a larger city to explore. The classes displayed are also dependent on the usage orientation (`IN` to include only classes using the entry points, `OUT` for the reverse usage, `IN/OUT` for both ways) and the usage level (the number of hops from the entry points over the usage links).

3.2 Limitations

The visualization provided by *VariCity* shows multiple improvements compared to the graph visualization of *symfinder-2*. Using a navigable 3D representation of a city where the buildings exhibit the different variability metrics and are grouped by usage allows to better grasp the organization of the classes [6]. Inheritance and additional usage relationships are displayed on hover not to overload the visualization. However, it still exhibits limitations to be used in practice on very large systems.

Firstly, variability implementations hamper the quality of a system and potentially create *variability debt* [18], especially since OO variability implementations have no dedicated mechanism and are therefore hidden in the codebase. With *VariCity*, understanding this debt would require to use additional tools extracting and visualizing quality metrics (such as SonarQube[2] that embeds the CodeCity [17] and Evo-Streets [11] visualizations) on the side and manually map those two sources of information. As these cities are shaped differently (the classic ones being shaped through package decomposition), the simultaneous usage of *VariCity* and one of these visualizations would be cumbersome, especially when experimenting with multiple metrics.

Secondly, *VariCity* is configured relying on knowledge from the codebase (*e.g.*, names of the entrypoint classes). Developers usually rely on an Integrated Development Environment (IDE) as tool support for assisting development and program comprehension activities [5]. Therefore, a user would need to manually input information from their IDE in *VariCity* and then go back to it to explore the classes exhibited by the visualization, implying important context switching. According to these limitations, we advocate that *VariCity*'s usability would be improved by *(i)* displaying quality metrics and *(ii)* being embedded into an IDE.

[2] https://www.sonarqube.org/

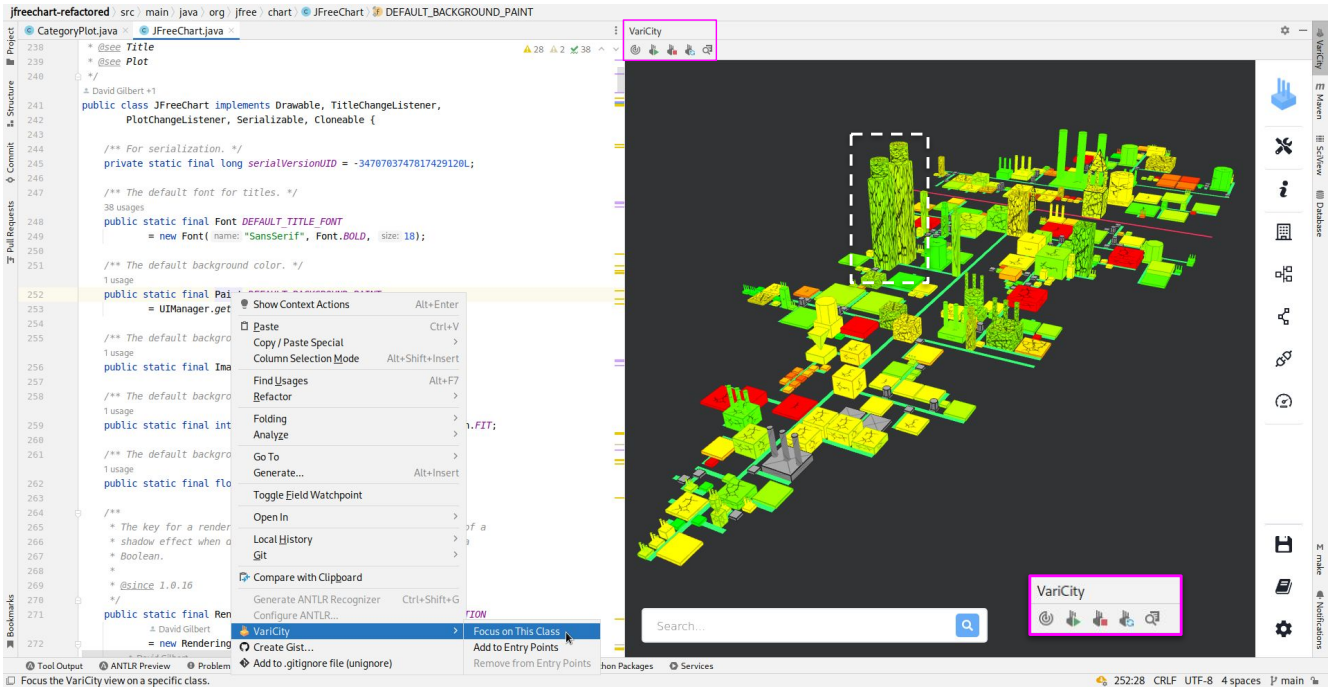

Figure 2: *VariMetrics-IDE* visualization of JFreeChart. The white and violet boxes have been manually added on the figure.

4 VISUALIZING QUALITY METRICS IN *VARIMETRICS-IDE*

As OO quality metrics are usually individual measures on the classes, we have to determine how to improve the *VariCity* visualization. *VariCity* displays in yellow *vp*-s being hotspots, in blue variants being hotspots, and in grey classes not being hotspots (fig. 1). However classic quality-centric cities, such as CodeCity and Evo-Streets, simply color the buildings to expose properties inherent to the classes [10, 17]. In *VariMetrics*, three display strategies are thus introduced to handle quality metrics. First, buildings can be colored following a red-to-green sequence (*fade*). Second, a saturation can be applied for keeping the original colors of the buildings and lightening or darkening them (*intensity*). Finally, a crackled texture (*cracks*) can be applied over the building with different *levels* of cracks, allowing to combine metrics on a single view.

Since there is no single definition of quality and each user might be interested in different metrics, *VariMetrics-IDE* allows to configure the view. The user can select and combine the desired quality metrics on the different visual axes to visualize them simultaneously. Combining adequate metrics allows to exhibit critical classes concentrating variability implementations. Relying on work from Wolfart et al. [18], it results that OO systems implementing their variability in a single codebase are prone to exhibit variability debt as code duplication, lack of tests, and increased cognitive complexity.

The *VariMetrics* visualization has been quantitatively evaluated on seven industrial open-source projects from several thousands to 2.4MLoC. Using these metrics, the exhibited classes were both critical and concentrating variability implementations [7]. A qualitative evaluation was also conducted on the JFreeChart project, with the refactoring of the indebted zones detected by *VariMetrics*. An example of exhibited classes is visible on fig. 2, where the two classes in the white dotted box, `CategoryPlot` and `XYPlot`, exhibit multiple method overloads (due to their height) and little test coverage (due to their cracks). Refactoring those classes led to an improvement of those classes and the project's overall quality [7]. More details on this evaluation are available on *VariMetrics* webpage[3].

5 SEAMLESS NAVIGATION WITH THE CODE

Although *VariMetrics-IDE* can be used as a standalone application in a web browser, an integration is provided for the popular IDE JetBrains IntelliJ IDEA[4]. As the goal of this integration is to minimize the interactions out of the development environment, the integration embeds all the interactions needed to configure and use *VariMetrics-IDE*. As shown on fig. 2, the visualization is embedded in a panel in the IDE window and provides controls over *symfinder* and *VariMetrics-IDE* (see violet boxes in fig. 2). It also allows the execution of the whole toolchain as detailed in section 4 from the editor's window.

Additionally, bidirectional navigation between the visualization and the code is provided to ease transitioning between the code and its visual representation. On one side, it is possible to select multiple buildings in the visualization and to open their sources as tabs in the IDE. On the other side, IntelliJ's context menu has been enriched, and right-clicking on a class in the `Project` sidebar or on the name of the class in the editor panel proposes a `Focus on This Class` button, zooming the visualization on the desired class, and another entry to add or remove a class from the entry

[3] https://deathstar3.github.io/varimetrics-demo/
[4] https://www.jetbrains.com/idea/

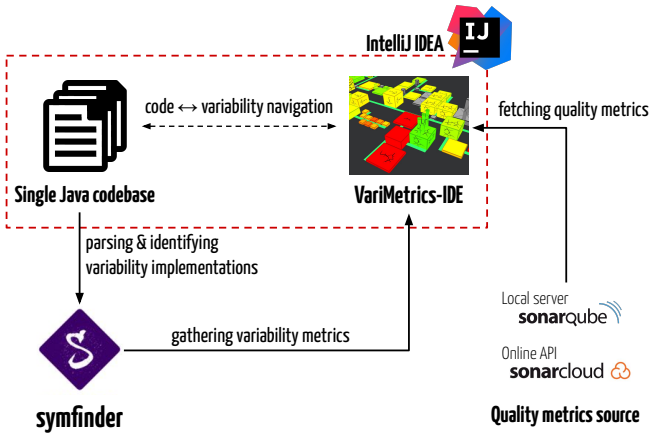

Figure 3: Description of *VariMetrics-IDE*.

points list. Finally, the plugin settings window allows to configure the usage level and usage orientation, and to browse the classes of the project to add them as entry points. It also embeds all the configuration capabilities of *VariMetrics*.

Although we did not conduct an experiment with real users to assert the quality of the integration, interactions have been designed following the IntelliJ Platform UI Guidelines[5] to ensure visual homogeneity in the editor, thus giving us confidence in its usability. In our own experience with the IDE integration, all exploration tasks conducted in the validation of the metrics part described in the previous section were heavily facilitated.

6 IMPLEMENTATION

Figure 3 describes the integration of *VariMetrics* in the analysis toolchain based on *symfinder*. First, *symfinder* [8, 9] parses the codebase to automatically identify the symmetries and, relying on them, the *vp*-s and variants. Then, quality metrics are extracted either from the SonarCloud[6] profile page of the project if available, or else from a local SonarQube server that is run locally. *VariMetrics* then combines the variability and quality information to render the visualization using the Babylon.js 3D library. Outside an IDE, the resulting view is deployed with Webpack and requires only a web browser to be viewed. All the toolchain components (*symfinder*, metrics fetching and *VariMetrics*) are packed in several Docker images to ease reuse of all or parts of the toolchain. The IDE integration is developed using the IntelliJ Platform SDK[7] and is installed as any other plugin for the IDE.

7 CONCLUSION

VariMetrics-IDE enhances the *VariCity* visualization by allowing to exhibit quality metrics as additional and configurable visual properties on the buildings. Displaying in a single representation variability and quality information allows to spot zones concentrating variability implementations and being less reliable. The visualization and its configuration are embedded in the JetBrains IntelliJ IDEA IDE to limit the interactions outside the developers working environment. Bidirectional browsing between the visualization and classes in the code is supported.

As future work, we aim to conduct an empirical evaluation of *VariMetrics-IDE* with real architects and developers to validate our approach and tooling, but also to better understand the industry needs and expectations for such tools.

[5] https://jetbrains.github.io/ui/
[6] https://sonarcloud.io/
[7] https://plugins.jetbrains.com/docs/intellij/welcome.html

ACKNOWLEDGMENTS

We thank Patrick Anagonou, João Brilhante, Charly Ducrocq, Ludovic Marti, Guillaume Savornin and Anton van der Tuijn for their contribution in the development of *VariMetrics-IDE*.

REFERENCES

[1] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* (2013).
[2] James O. Coplien and Liping Zhao. 2000. Symmetry Breaking in Software Patterns. In *International Symposium on Generative and Component-Based Software Engineering (GCSE 2000)*. Springer, Springer, 37–54.
[3] Matthias Galster. 2019. Variability-Intensive Software Systems: Product Lines and Beyond. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '19)*. ACM, 1–1.
[4] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.
[5] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 25–35.
[6] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2021. Visualization of Object-Oriented Variability Implementations as Cities. In *2021 Working Conference on Software Visualization (VISSOFT)*. Luxembourg (virtual), Luxembourg, 76–87.
[7] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2022. Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations. In *Proceedings of the 26th International Systems and Software Product Line Conference (SPLC '22)*, Vol. Volume A. ACM, Graz, Austria, 1–12.
[8] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2019. symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations. In *the 23rd International Systems and Software Product Line Conference*, Vol. B. ACM Press, Paris, France, 5–8.
[9] Johann Mortara, Xhevahire Tërnava, Philippe Collet, and Anne-Marie Dery-Pinna. 2021. Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships. In *SPLC 2021 - 25th ACM International Systems and Software Product Line Conference*, Vol. Volume B. ACM, Leicester, United Kingdom, 1–8. https://doi.org/10.1145/3461002.3473943
[10] Frank Steinbrückner and Claus Lewerentz. 2010. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization*. 193–202.
[11] Frank Steinbrückner and Claus Lewerentz. 2013. Understanding software evolution with software cities. *Information Visualization* 12, 2 (2013), 200–216.
[12] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and experience* 35, 8 (2005), 705–754.
[13] Xhevahire Tërnava, Johann Mortara, Philippe Collet, and Daniel Le Berre. 2022. Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach. *Journal of Automated Software Engineering* (Feb. 2022), 1–52.
[14] Xhevahire Tërnava and Philippe Collet. 2017. Tracing Imperfectly Modular Variability in Software Product Line Implementation. In *International Conference on Software Reuse (ICSR '17)*. Springer, 112–120.
[15] Xhevahire Tërnava, Johann Mortara, and Philippe Collet. 2019. Identifying and visualizing variability in object-oriented variability-rich systems. In *the 23rd International Systems and Software Product Line Conference*. ACM Press, Paris, France, 231–243. https://doi.org/10.1145/3336294.3336311
[16] 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, 92–99.
[17] Richard Wettel and Michele Lanza. 2008. CodeCity: 3D visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*. 921–922.
[18] Daniele Wolfart, Wesley Klewerton Guez Assunção, and Jabier Martinez. 2021. Variability Debt: Characterization, Causes and Consequences. In *XX Brazilian Symposium on Software Quality*. 1–10.