Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations

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

Customizable Visualization of Quality Metrics for Object-Oriented 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

Many large-scale software systems intensively implement variability to reuse software and speed up development. Such mechanisms, however, bring additional complexity, which eventually leads to technical debt, threatening the software quality, and hampering maintenance and evolution. This is especially the case for variability-rich object-oriented (OO) systems that implement variability in a single codebase. They heavily rely on existing OO mechanisms to implement their variability, making them especially prone to variability debt at the code level. In this paper, we propose *VariMetrics*, an extension of a visualization relying on the city metaphor to reveal such zones of indebted OO variability implementations. *VariMetrics* extends the *VariCity* visualization and displays standard OO quality metrics, such as code duplication, code complexity, or test coverage, as additional visual properties on the buildings representing classes. Extended configuration options allow the user to choose and combine quality metrics, uncovering the critical zones of OO variability implementations. We evaluate *VariMetrics* both by reporting on the exposed quality-critical zones found on multiple large open-source projects, and by correcting the reported issues in such zones of one project, showing an improvement in quality.

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. Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations. In *26th ACM International Systems and Software Product Line Conference - Volume A (SPLC '22), September 12–16, 2022, Graz, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3546932.3547073

1 INTRODUCTION

The constantly increasing demand for software solutions constraints software practitioners to develop and maintain customizable software systems that can be delivered at high-rate while assuring an optimal level of quality and security. In this context, monitoring quality is crucial for the maintenance and evolution of such systems [53]. For example, projects managed using Agile methodologies define specific requirements to describe the desired qualities of the system [9, 94] and limit technical debt (*i.e.*, the impact on the system's maintainability and evolution [5]). While technical debt covers diverse aspects of the software and its development ecosystem [46], its identification at the implementation level is mainly done through code analysis (*e.g.*, by computing metrics or identifying a lack of tests [50]).

Such large-scale configurable systems are variability-rich [28, 29, 35] and make use of various mechanisms to implement their variability, for instance, annotative approaches (*e.g.*, preprocessor directives [51]) or aspects [56]. Most annotative mechanisms, however, are known to impede the quality of the software in multiple aspects, especially by bringing additional complexity [30] and polluting the code [47, 55], thus making the code difficult to understand, maintain and test [54], and leading to technical debt [50]. The studies of technical debt due to variability implementations led to new definitions [1, 58] and adaptations of standard definitions [24, 78] to consider variability mechanisms. Very recently, Wolfart et al. [95] reformulated the technical debt caused by variability implementations under the definition of *variability debt.*

Many variability-rich systems, however, do not follow a complete SPL approach and do not rely on the previously cited mechanisms to implement their variability. This is especially the case of object-oriented (OO) systems that often implement their variability in a single codebase, using the traditional OO mechanisms (*i.e.*, inheritance, overloading of methods and constructors, design patterns) [12, 27, 82]. This absence of dedicated implementation mechanisms causes the variability to be intertwined with the implementation, hampering its identification, analysis, and understanding as there is no traceability with domain information [83, 85]. Being completely dependent on mechanisms causing technical debt, such systems are prone to introduce variability debt [95] at the code level, calling for a solution to better identify and understand it.

On one side, multiple tools and approaches exist to compute metrics on an OO codebase, analyze its quality [50, 72], and determine technical debt [6]. Such metrics are often exploited in visualizations [14, 50], such as CodeCity [90] and Evo-Streets [80] that are now bundled in reference code analysis tools such as SonarQube[1]. Such visualizations, however, do not allow displaying the use of OO variability implementations mechanisms. Even in case some experts have good knowledge of the implemented variability of their system, they will need to observe the quality of the concerned classes

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

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

one by one. On the other side, a first approach to identify OO variability implementations has been proposed by Tërnava et al. [86], abstracting the OO mechanisms in terms of *variation points* (*vp*-s) and *variants* relying on the notion of symmetry [18, 96] to automatically identify zones with a high density of potential variability implementations [62]. Mortara et al. [59] then proposed *VariCity*, a city-based visualization to ease their identification. However, this visualization does not provide information on the quality of the system's classes and experts must rely on other tools to observe more closely the quality of the classes highlighted by *VariCity*. Furthermore, navigating between *VariCity* and a metric-specific tool would be cumbersome as it would require manually finding and mapping information having heterogeneous representations. Therefore, to the extent of our knowledge, no solution exists to visualize technical debt in OO variability-rich systems.

In this paper, we propose *VariMetrics*, an extension of *VariCity* to support software quality metrics and reveal critical zones concentrating variability implementations prone to cause variability debt in the context of a single OO codebase. As determining a relevant quality measure relies on numerous factors [44], practitioners need to define relevant indicators for each system relying on the profusion of existing metrics [15]. Thus, *VariCity*'s configuration capabilities have been extended to enable one to compose state-of-the-art OO quality metrics as visual properties on the buildings, which are the classes of the project. We report on the evaluation of *VariMetrics*, which was first applied to seven open-source software systems to show that it reveals quality-critical zones of variability implementations (section 5.1). We also assess the relevance of the indebted classes identified in one project by improving a subset of these classes and their tests, thus showing a global improvement of the project's quality (section 5.2).

The rest of the paper is organized as follows. Section 2.1 introduces the motivations for our work. Section 2.2 details related work on OO variability implementations and quality metrics, as well as their associated visualizations (*VariCity* and CodeCity). Section 3 gives background on the identification of OO variability implementations and on how *VariCity* uses them to build its visualization. We then present *VariMetrics* and how it extends *VariCity* to support quality metrics in section 4. We evaluate our approach (section 5) and discuss threats to the validity and limitations in section 6. Finally, section 7 concludes the paper while presenting future work.

2 MOTIVATIONS

Software quality is an important field of research due to its broad impact on the software development cost [77]. In the domain of OO systems, multiple works focus on determining software quality metrics [11, 26, 41, 54, 57, 73], measuring the system evolution [34, 75], and validating the relevance of these metrics [43, 66]. Quality metrics have been recognized as useful for determining *technical debt* at the code level, *i.e.*, expedient but costly on the long term implementation constructs, primarily hampering maintainability and evolvability [5, 50].

2.1 Problem statement

Wolfart et al. [95] defined variability debt as *"Technical debt caused by defects and sub-optimal solutions in the implementation of variability management in software systems"*. They studied 52 industrial case studies reporting technical debt issues on variable software systems with the following main results:
1. the lack of knowledge of the implemented variability, as well as the absence of traceability, causes variability debt;
2. the absence of known variability implementation mechanisms is prone to cause artifact duplication, an increase of code complexity, and a "disappearance of links between implementation artifacts to business values" [22];
3. variability debt mainly impacts source code artifacts;
4. variability debt causes inability to systematically deal with customization and poor overall internal quality, complicating maintenance for the development team.

This work focuses on the identification of the part of variability debt dedicated to object-oriented variability implementations. Many large object-oriented systems are naturally variability-rich but they do not follow a systematic approach to manage variability as in the SPL paradigm [4, 70]. Consequently, they do not define features at a domain level in a formal model, and these features are not consistently documented or made explicit in the code assets. While some organizations adopt a *clone-and-own* approach [31, 74] to handle variability, with many disadvantages [23, 32], our work focuses on object-oriented variability-rich systems that manage variability in a single codebase.

In such systems, code assets are structured into three distinct parts, the core being assets included in all software products, commonalities being the common part between the related variations of code assets, and variations representing how and when code assets vary [7, 17, 35, 87]. Variation points (*vp*-s) and variants are concrete constructions in the code assets that usually abstract respectively the commonality and variation parts [20, 39, 40, 71]. A *vp* references one or more locations at which the variation is going to happen, while the variants express how the variation point varies [39].

In a single OO codebase, *vp*-s and variants can be implemented through diverse mechanisms already present in the language, such as inheritance, parameters, constructor and method overloading, or variability-related design patterns (*e.g.*, strategy, factory, template) [12, 27, 82, 84]. Recently, an approach based on detecting symmetries in OO mechanisms [18, 96] was proposed to identify these variability implementations without prior explicit knowledge of features [86]. Although it can abstract potential *vp*-s and *variants*, Mortara et al. [62] extended it to identify interesting zones of a high density of variability implementations. In a follow-up work, they rely on a city metaphor, known to help in software understanding [45], and provide *VariCity*, a visualization to ease identification (see section 3 for more details).

While the object-oriented variability implementations can be more easily identified, they are especially prone to technical debt at the source code level. They are directly reusing traditional mechanisms and variability code is then intertwined with the rest of the implementation code [83, 85]. Measuring their quality is thus crucial, and using quality metrics is then a natural way [50, 72] to determine technical debt [6], especially through visualizations [14, 50].

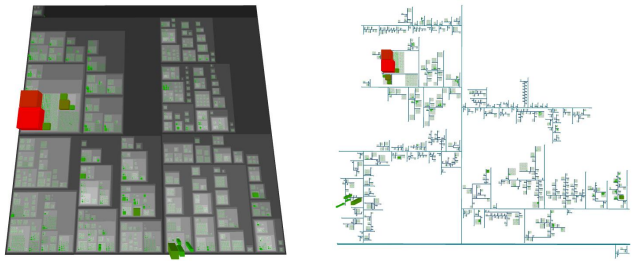
To structure our definition of the problem, we define a general usage scenario that will drive our studies and design choices in the remainder of the paper. From the quality point of view, it has been shown that, in an agile context, team members involved in quality requirements definition correspond to senior profiles [2, 9]. Therefore, they represent advanced developers that have enough knowledge of the system to have an overview of the implemented domain, as well as of quality to design quality requirements. In the following, we identify such people as "experts". According to our analysis above, we phrase the scenario as follows: *the expert wants to analyze the quality of the variability implementations to potentially identify OO variability debt.*

2.2 Related Work

Object-oriented metrics and visualization. Tools have been developed to automatically analyze OO codebases and extract quality metrics [49], such as SonarQube[2], one of the most frequently used open-source code analysis tools, adopted by more than 200K developer teams, including more than 250K public open-source projects on its cloud version SonarCloud[3]. Not only the metrics are extracted, but a set of customizable rules gives more precise insights into the defects detected, and how to correct them [48, 68]. Finally, a set of plugins complete the tool to provide improved exploitation of the extracted metrics, such as advanced visualization solutions. One of them is SoftVis3D[4], which embeds CodeCity [90] and Evo-Streets [79], two popular visualizations relying on the city metaphor [89] to represent the system and its quality metrics. Figure 1 illustrates the two visualizations on the GeoTools project[5], an open-source Java library for geospatial data management. Classes are represented as buildings and their width, height, and color are used to display the quality metrics, making discernible classes maximizing these metrics. Districts in CodeCity (fig. 1a) and streets in Evo-Streets (fig. 1b) represent the decomposition in packages. As such visualizations have proven to help the comprehension of a system's quality [93], multiple other city-based visualization approaches for quality have been proposed [25, 69, 91]. However, none of them allows displaying information on the system's variability.

Object-oriented variability visualization. While visualizations for properties of variable systems are focused on systems organized as an SPL or making use of annotative approaches for which features are known [3, 10, 33, 42, 52], little work exists on visualization of OO variability implementations. *symfinder* [62] proposes a graph visualization displaying the information output by its symmetry-based detection of variability implementations. Nodes represent classes, linked together by edges, being inheritance relationships. The color and size of a node evolve according to the number of constructor and method overloads respectively. *symfinder* was later extended to take into account usage relationships between classes with *symfinder-2* [64] and the visualization was also extended by displaying such relationships as dashed arrows.

[2] https://www.sonarqube.org/
[3] https://sonarcloud.io/explore/projects
[4] https://softvis3d.com/
[5] https://www.geotools.org



(a) CodeCity view **(b) Evo-Streets view**

Figure 1: Views of GeoTools, using cyclomatic complexity as footprint, # LoC as height, and complexity as color.

Recently Mortara et al. [59] proposed *VariCity*, a visualization relying on the city metaphor to display the information from *symfinder-2*. An example of generated visualization is shown in fig. 2. As with CodeCity and Evo-Streets, a class is represented by a building. The dimensions, however, represent the class-based metrics related to variability (*cf.* section 3.1). Streets departing from a building represent a usage relationship between this class and every other class whose building is on the street. Therefore, the discernible classes are the ones concentrating variability implementations. For example, `FilterFactoryImpl` is shaped as a skyscraper due to an important number of method overloads (141). Its goal is to create filters allowing to select zones from a map[6]. The large strategy is `Query` (10 constructors), which uses filters to query information from a data source. On the opposite, `FilterVisitor` is not very variable in itself but uses all the implemented filters, in the blue dotted box, noticeable by being a long street. Coloring the hotspot classes not only emphasizes the filters having more variants, but also exhibits some isolated classes, for example `NumberRange`, which implements a numerical range of values. On the opposite, the two red classes exhibited in fig. 1 because of their too high cyclomatic complexity (`gml311.DocumentRootImpl` and `gml311.Gml311PackageImpl`) are not visible in fig. 2 as they are not part of zones concentrating variability implementations. More details on the organization of the visualization are given in section 3.2. *VariCity*, however, does not display information related to the software quality of the displayed classes.

Summary. Consequently, to the extent of our knowledge, no solution exists to visualize at the same time, for an OO system, its variability implementations, and quality metrics over them. As the cities of *VariCity* and Evo-Streets are shaped differently, the simultaneous usage of both visualizations would be cumbersome, especially when experimenting with multiple metrics. This thus calls for a unified but customizable visualization and we propose to extend *VariCity* to incorporate quality metrics over a variability-centric visualization.

3 BACKGROUND

In this section, we give background details on how OO variability implementations are identified and how *VariCity*, which we extend in this work, exploits this information to provide a dedicated visualization.

[6] https://docs.geotools.org/latest/userguide/library/main/filter.html

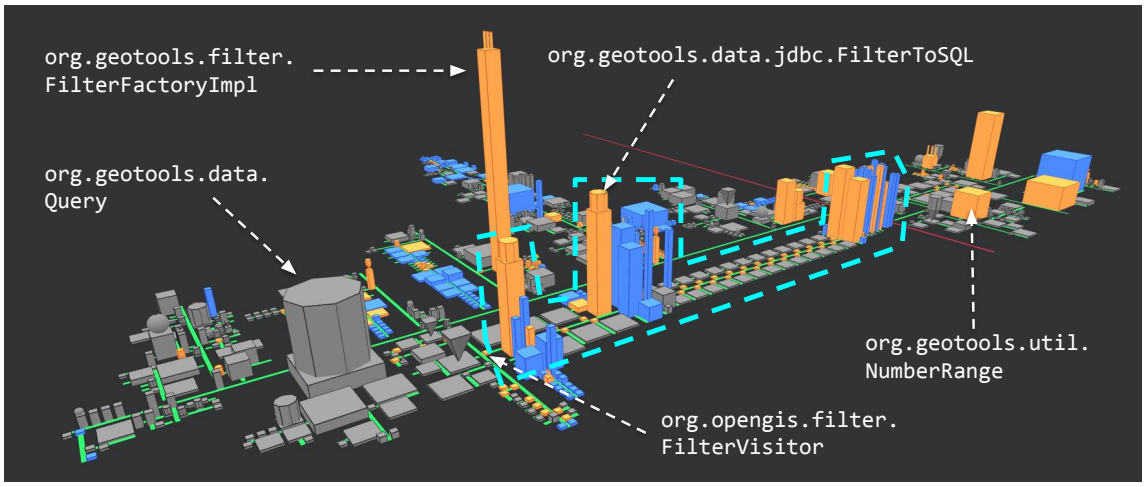


Figure 2: *VariCity* visualization of GeoTools.

3.1 Identification of OO variability implementations

The concept of symmetry has been studied in software [16, 18], and especially in mechanisms of object-orientation, such as inheritance, overloading, and design patterns, which can all be interpreted as forms of symmetry [19, 96]. Taking a codebase as a whole, Tërnava et al. [86] have shown that these implementation techniques can be seen as local symmetries, which allow a part of code to change while another part remains unchanged. Detecting seven techniques (*class as type*, *class subtyping*, *method and constructor overloading*, *strategy*, *template*, *decorator*, and *factory patterns*) in Java and C++ code with the *symfinder* toolchain [61, 62], the authors have also shown that the location where they are detected (mainly classes) represent accurate potential *vp*-s and variants [86].

The identification is facilitated in zones where variability implementation is dense because several techniques are used together, or a technique is heavily used (*e.g.*, many methods being overloaded) in a set of classes related by their usages (*e.g.*, one being attribute or method parameter of another) [64]. These zones have been defined as *hotspots* in the last version of the *symfinder* toolchain [64, 83] with direct relation to the computed variability metrics[7] (*e.g.*, number of overloaded methods, number of subclasses). A follow-up work has also shown that the detected OO variability implementations can be successfully mapped to domain features when they are available [63].

3.2 Visualization support in *VariCity*

3.2.1 Visualization dimensions. In *VariCity*, the city representing the system is organized to exhibit the classes concentrating variability implementations. A tall building shows an important number of method variants (*e.g.*, `FilterFactoryImpl` in fig. 2), whereas a large building shows an important number of constructor variants (*e.g.*, `Query`), exhibiting variability concentration at method level inside a class. Identified design patterns have a crown on their building (*e.g.*, `FilterFactoryImpl` is a Factory, whereas `Query` is a Strategy) [8]. The placement of the buildings by decreasing order of width on both sides of the street allows for exhibiting density between classes. Additional usage relationships are represented as underground streets, and inheritance relationships as aerial links, both displayed when hovering a building. Finally, classes being part of *hotspots* are displayed in color (*vp*-s in yellow and variants in blue) to make them easily noticeable.

3.2.2 Configuration capabilities of the visualization. Three parameters allow configuring the view. First, some classes selected by the user to represent points of interest of a system (*e.g.*, API endpoint, ...) can be defined as *entry point* classes to start its exploration. Then, the usage orientation determines whether buildings on a street are *using* the class initiating the street (orientation `IN`), or *used by* it (orientation `OUT`), or both (orientation `IN/OUT`). Finally, the usage level can be set to define the maximum number of hops to be traversed in the usage relationships from the studied classes (starting from entry points) to other classes to be displayed. With a usage level of n, all classes distant from an entry point by n usage relationships will be displayed.

The city is shaped by first aggregating the *entry point* classes on a red street. Then, starting from them, classes using (or being used by) them up to the usage level set are displayed. For example, a visualization set up with one entry point, usage orientation `IN`, and usage level of 2 will display the entry point, the classes using the entry point, and the classes using these classes. To generate fig. 2, *VariCity* has been configured to use `SimpleFeatureSource`[9] and `MapContent`[10] as *entry points*. The usage orientation has been set to `OUT`, and the usage level to 4.

[7] It must be noted that these metrics are used to identify variability, but are not quality metrics like code complexity or test coverage.

[8] Although design patterns often involve multiple classes, the crown is only present on the *vp* of the design pattern.

[9] `org.geotools.data.simple.SimpleFeatureSource`

[10] `org.geotools.map.MapContent`

4 *VARIMETRICS*: EXPLORING THE QUALITY OF VARIABILITY IMPLEMENTATIONS

As shown in section 2.2, although state-of-the-art approaches allow visualizing either the density of variability implementations (*e.g.*, with *VariCity* [59]) or quality metrics (*e.g.*, with CodeCity [90] or Evo-Streets [80]), no existing approach allows the simultaneous representation of both aspects of OO software systems. Therefore we adapt *VariCity* to display information about quality in the city.

4.1 Main principles

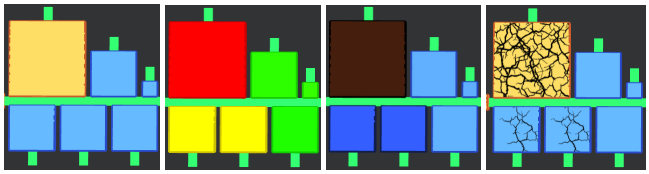
Although *VariCity*'s configuration capabilities detailed in section 3.2.2 allow to shape the city and show a desired subpart of the project, it is not possible to configure the displayed variability metrics (*i.e.*, the number of method overloads for the height, and the number of constructor overloads for the base). *VariMetrics*, however, aims to focus the expert on the quality-critical zones concentrating variability implementations. State-of-the-art proposes a plethora of quality metrics to measure several properties of a software system [15], ranging from the architecture [65] to the source code level [54, 81]. Since no metric is relevant for all software systems due to the elusive definition of quality [44], software practitioners need to pick and combine different metrics to obtain a quality measure relevant for their use case. *VariMetrics* extends the configuration of *VariCity* so that experts can choose the quality metrics they want to display, and how to combine them, to tailor the visualization according to their needs.

By default, *VariCity* displays in yellow *vp*-s being hotspots, in blue variants being hotspots, and in grey classes not being hotspots (fig. 3a). On their side CodeCity and Evo-Streets color the buildings to expose properties inherent to the classes [79, 92]. We thus propose two coloring strategies for quality metrics: a coloration following a red-to-green sequence (fig. 3b), and a saturation keeping the original colors of the buildings and lightening or darkening them (fig. 3c). While *VariMetrics* should enable some combination of metrics, combining both coloring strategies leads to bivariate chromatic maps, which are known to be difficult to read [88]. On the opposite, applying textures on colors has shown to be an efficient way to display multiple software quality metrics [36]. We hence provide a crackled texture (fig. 3d) variably covering the building, thus enabling views simultaneously exhibiting two quality metrics.

These three visual properties are configurable to be adapted to the metric they represent, as some quality metrics are symptoms of lower quality if they have a high value (*e.g.*, complexity) but other metrics with such values may instead indicate good quality (*e.g.*, test coverage). Analogously, not all projects have similar ranges of values for the same metric, and proposing a fixed range of values may not allow revealing a difference of quality in some projects, thus *VariMetrics* allows to specify these ranges.

4.2 Determining relevant quality metrics for OO variability debt

OO variability debt identification does not only require an appropriate visualization but also adequate metrics to be exploited in the visualization. Wolfart et al. [95] introduced a catalog of ten forms of variability debt, detailing for each of them its cause(s), consequence(s), and concerned type(s) of artifacts. In the following analysis, these forms are written in italics.



(a) *VariCity* **(b) Red-green** **(c) Saturation** **(d) Cracks**

Figure 3: Visual properties used to display quality metrics compared to the original *VariCity* visualization.

As OO variability implementations rely solely on standard OO mechanisms, the availability of the source code is the only requirement to identify them. Finding *Code duplication* is therefore possible, as well as *System-level structure quality issues* in the implementation. Most often, tests sources are provided along with the source code, enabling identification of *Lack of tests*.

However, other information is not always available, especially in the case of open-source systems, such as the documentation, leaving aside *Out-of-date or incomplete documentation* and *Duplicate documentation*. Identifying *Architectural anti-patterns* needs information on the domain and the associated design choices (*e.g.*, we cannot say if a Strategy pattern has the desired behavior solely by analyzing its structure). Covering *Poor test of feature interactions* would require a list of features and their mapping with their implementations, which are often not available in our case, while covering *Old technology in use* and *Multi-version support* implies having information about the versions of the supporting language and used libraries. Finally, identifying *Expensive tests* implies determining whether test cases have been formally defined or not [76], thus requiring test cases definitions.

It results that relying on the source code and its tests, we can cover *Code duplication*, *Lack of tests*, and *System-level structure quality issues* in the implementation. Hence, we need to determine quality metrics to identify these types of variability debt. A common metric to identify a lack of tests is the code coverage, which can be measured at different granularities (line, condition, ...). For our evaluation, we opted for a coverage metric that aggregates measures for different granularities. Similarly, code duplications are commonly identified at two levels of granularity: line or block. We advocate that blocks are more likely to represent duplicated code related to variability than a single line of code. Finally, structure quality issues in the codebase impact maintainability and evolution of the system. Even though code duplication and lack of tests impact maintainability and evolution of the system, the understanding of the implementation by the maintainers of the project is also an important aspect, and cognitive complexity [11] appears to be relevant for this purpose [67]. We thus choose as relevant metrics for our evaluations duplicated blocks, test coverage, and cognitive complexity.

Most often, standard tools for measuring software quality metrics also determine technical debt measures giving an estimation of the effort, as a duration, to fix the identified code smells [6]. We did not use such measures in our evaluation for multiple reasons. First, by providing an aggregated duration, this measure is more

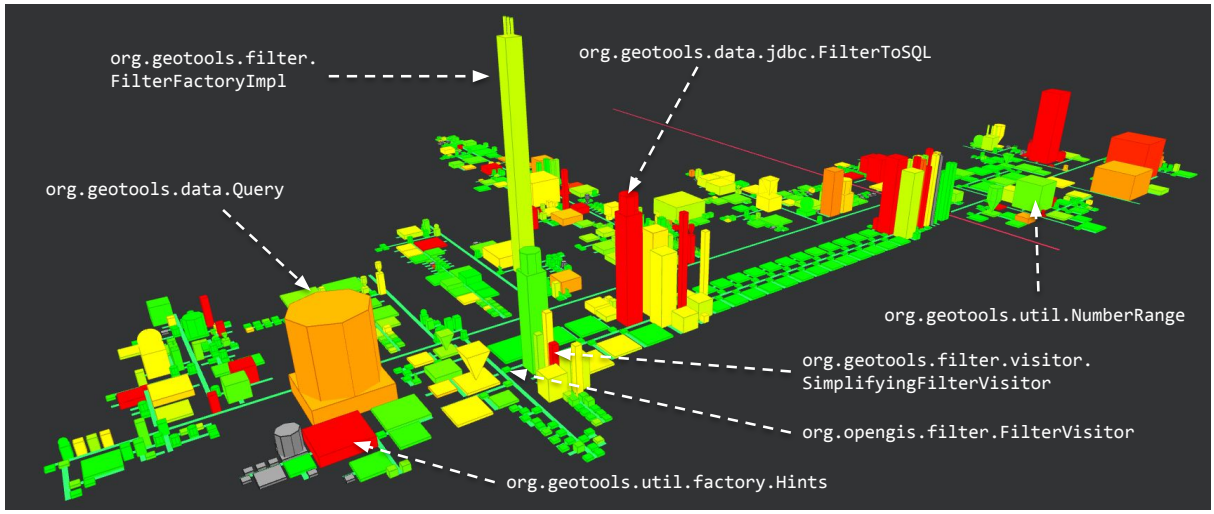


Figure 4: Figure 2 in *VariMetrics*. The view is configured to display the cognitive complexity using the red-to-green color scale.

helpful in estimating effort at the management level, but it does not describe the real causes of the debt. Then, some first empirical results seem to indicate a possible inaccuracy in the given values [8], and exploiting such metrics may therefore require some knowledge of the system and its implementation, which we do not have for our subject systems. Nevertheless, *VariMetrics* allows visualizing this metric if the experts find it relevant.

Figure 4 shows the *VariCity* view of fig. 2 in *VariMetrics* showing the cognitive complexity using the red-to-green color scale. Where the classes concentrating variability implementations revealed by *VariCity* (cf section 3.2) remain visible independently of their quality (*e.g.*, `FilterFactoryImpl` or `NumberRange`), *VariMetrics* also exposes quality-critical classes, being variable (*e.g.*, `Query` or `FilterToSQL`) or not (*e.g.*, `Hints` or `SimplifyingFilterVisitor`).

4.3 Implementation

The *symfinder* toolchain, used by *VariCity* to identify the variability implementations and compute the related variability-related metrics, has been extended to support fetching of the quality metrics and their mapping with the identified variability information. If a SonarCloud account exists for the system, metrics are fetched by using the SonarCloud Web API[11]. Otherwise, a SonarQube server is executed locally to extract the metrics while running the *symfinder* analysis. The *symfinder* configuration has been extended to specify wherever running a SonarQube instance is needed or not.

5 EVALUATION

The evaluation of *VariCity* presented by Mortara et al. [59] validates its capacity to exhibit zones in the code concentrating mechanisms used in OO variability implementations (*cf.* section 3.1). *VariMetrics* should therefore be able to reveal the subset of these classes having quality issues. To evaluate *VariMetrics* against the requirements expressed by section 2.1 (*i.e.*, identifying variability implementations for which quality metrics are problematic), we apply our approach to multiple open-source systems. We select views with metrics combinations revealing the variability implementations that are shown by *VariCity* while being the most quality-critical (section 5.1). We then validate the relevance of such classes by applying maintenance actions on these classes within one project, JFreeChart (section 5.2), and show the impact on the view of the project.

[11] https://sonarcloud.io/web_api

Table 1: Subject systems and their available metrics.

Project	Version	Java LoCs	# *vp*-s / variants	Available metrics DB	COMP	COV
Azureus	5.7.6.0	633,248	10,105	A	S	✗
GeoTools	23.5	1,312,727	22,534	A	S	✗
JDK	17-10	2,434,983	71,489	S	S	✗
JFreeChart	1.5.0	94,203	2,849	S	S	S
JKube	1.7.0	40,952	795	A	S	S
OpenAPI Generator	5.4.0	88,172	768	S	S	S
Spring framework	5.2.13	662,579	12,622	A	S	✗

DB – duplicated blocks, COMP – cognitive complexity, COV – coverage
✗ – unavailable metric, A – available metric, S – significant metric (available and showing differences between classes)

5.1 Quantitative evaluation

Subject systems. We used for this evaluation 7 variability-rich open-source Java systems of various sizes, depicted in table 1. Five of them were chosen as their documentation clearly states they implement variability: **Azureus** (Vuze) is a BitTorrent client which supports multiple network communication protocols, **GeoTools** a library for geospatial data management providing multiple tools and filtering capabilities to manipulate maps, **JKube**, a Maven plugin to generate different types of container images, **OpenAPI Generator**, a library to create APIs for a plethora of programming languages, and the **Spring framework**, providing a Java-based support for components and services with many different plugins, on persistence management, validation, security, etc. We also picked the **Java Development Kit (JDK)** for its large size of ~2.5M LoC to evaluate the scalability of our approach. Finally, we also used

Table 2: Number of noticeable classes due to their variability concentration, criticality, and both aspects for the given views on all subject systems.

Project	View configuration				Noticeable classes *w.r.t.*		
	Entry point classes	Usage orientation	Usage level	Metrics (visual property)	variability	criticality	both
Azureus	`com.aelitis.azureus.core.AzureusCoreComponent`	OUT	4	COMP (red-green)	74	32	12
GeoTools	`org.geotools.data.simple.SimpleFeatureSource` `org.geotools.map.MapContent`	OUT	4	COMP (red-green)	104	27	18
JDK	`java.net.URI` `java.net.URL`	IN	1	COMP (red-green) DB (cracks)	84	17	13
JFreeChart	`org.jfree.chart.JFreeChart` `org.jfree.chart.plot.Plot`	OUT	4	COV (red-green) DB (cracks)	35	31	10

JKube	`org.eclipse.jkube.generator.api.support.BaseGenerator`
	`org.eclipse.jkube.generator.javaexec.JavaExecGenerator`
	`org.eclipse.jkube.generator.api.Generator`
		IN/OUT	7	COV (red-green)
				COMP (cracks)
					28	115	14

OpenAPI Generator	`org.openapitools.codegen.languages.OpenAPIGenerator`	IN/OUT	6	COV (red-green) COMP (cracks)	77	51	21
Spring framework	`org.springframework.beans.factory.parsing.BeanComponentDefinition` `org.springframework.beans.factory.support.AbstractBeanFactory`	IN	8	COMP (red-green)	57	13	6

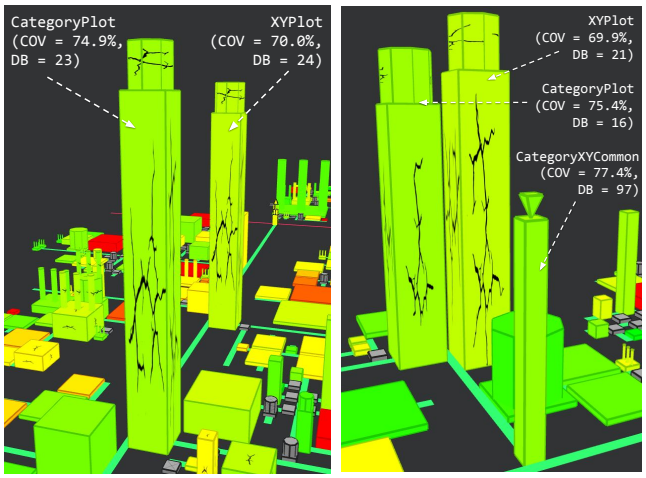
JFreeChart, a charting library used as a subject system in the evaluation of *VariCity* by Mortara et al. [59], as its size enables us to master the implemented variability at a fine granularity. Five projects are forks from their original repositories in the Corpus-2021 GitHub organization[12], designed by Irrazábal et al. [38] to serve as a catalog of software projects to analyze their metrics. They provide a SonarCloud instance for these projects[13], allowing us to reuse these metrics for our study. Two others have also a SonarCloud instance and JFreeChart is the only one for which we had to use our prototyped setup with Sonarqube to obtain the quality metrics. Besides, the JFreeChart's build configuration was also adapted to be analyzed by a local SonarQube instance [60].

Evaluation process. We first generated for each project a visualization with *VariCity* following the same stages as in the *VariCity*'s evaluation [59]. After determining entry points by selecting important classes after exploring codebases and documentations, we experimented empirically with different combinations of usage level and usage orientation to obtain a visualization we consider relevant (*i.e.*, exhibiting classes detaching from others because they concentrate variability implementations). We finally identified manually on each view the classes that are the most visible for us (by being a hotspot or a design pattern, or due to their dimensions) to obtain a set of "noticeable classes *w.r.t.* variability". For example, for GeoTools (fig. 2), classes such as `FilterFactoryImpl`, `FilterToSQL`, `Query`, and `NumberRange` draw attention due to their size and/or the fact that they are hotspots, as opposed to `FilterVisitor`.

To determine a relevant *VariMetrics* view, we systematically applied all available metrics on each project and selected the ones being relevant to identifying OO variability debt (*cf.* section 4.2). During this step, it happened that no building stood out for a metric (*i.e.*, no class exhibits variability debt), suggesting that the overall quality is decent *w.r.t.* this metric. On the opposite, if all classes appear as quality-critical, it may indicate that this metric has been neglected in quality requirements for the project as a whole. We thus restrained in this evaluation the set of *significant* metrics relevant to identify OO variability debt to those showing some differences in quality between classes. Table 1 summarizes for each system the relevant metrics being available and significant. We then manually identified on the views the classes appearing to be quality-critical, regardless of their variability, by enumerating the classes that appeared to be the most cracked and/or red to obtain a set of "noticeable classes *w.r.t.* criticality". For example, for GeoTools (fig. 4), `Hints`, `Query`, `SimplifyingFilterVisitor`, and `FilterToSQL` are easily discernible. The quality-critical and variability intense classes of the project thus correspond to the intersection between the two sets of classes (*i.e.*, in this example, `FilterToSQL` and `Query`).

In all observed systems, it appears that although fewer classes are noticeable *w.r.t.* criticality than *w.r.t.* variability, there is no direct relation between variability and quality, as it can already be seen in fig. 4. Whereas some *vp*-s have an important number of variants, they can be reliable, such as `FilterFactoryImpl` in GeoTools, and thus do not need particular attention. On the opposite, some critical classes may not concentrate variability implementations, such as `Hints` in GeoTools, and they are therefore less important for maintaining the functional code. This shows that, in the studied systems, visualizing both variability and quality is useful to determine quality-critical variability implementations. To evaluate to which extent, we calculated for each project the number of noticeable classes *w.r.t.* variability, *w.r.t.* criticality, and *w.r.t.* both aspects. The results with the configuration for each view are reported in table 2. This shows that representing on a single view variability and quality information allows reducing the number of classes appearing as relevant on the visualization between 50% (JKube) and 91% (Spring framework) compared to the *VariCity* visualization. We believe the mildly encouraging results obtained on JKube come from its size, so that less variability intense zones have been identified by *VariCity* compared to larger projects. An important number of classes are also noticeable in this project as it has globally a low code coverage. Besides, by adapting the thresholds on which the hotspot detection relies, we could obtain fewer zones and better results, but we consider these experiments as out

[12] https://github.com/Corpus-2021
[13] https://sonarcloud.io/organizations/corpus-2021/projects



(a) Before refactor (b) After refactor

Figure 5: View of `XYPlot` and `CategoryPlot` before and after the refactor. Block duplications are displayed on the red-green scale (range: 0 → 50 blocks) and test coverage using the crackled texture (range: 0% → 100%).

of the scope of this paper. The definition of a hotspot is elusive [64] and determining whether a class is a hotspot or not depends on user-defined thresholds, a limitation already evoked in the work on *VariCity* [59]. Nevertheless, we consider these results as satisfying, because without *VariMetrics*, finding OO variability debt would have needed to manually map relevant classes on the *VariCity* view to their metrics, which, already on the smallest project being JKube, represents 28 classes.

Summary. By representing OO variability implementations and quality metrics in a unified representation, *VariMetrics* not only allows to visualize both classes concentrating variability implementations and critical classes, but also to focus on specific zones of OO variability debt.

5.2 Qualitative evaluation

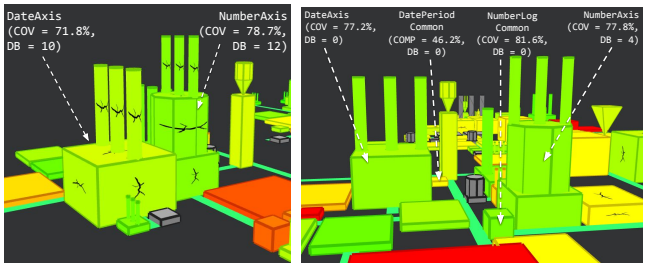
Identifying technical debt helps to understand where to apply maintenance actions aiming to improve software quality. Therefore, if zones of variability debt identified by *VariMetrics* are relevant, correcting identified weaknesses should improve the project quality, and the effects should be visible in the visualization. To validate the relevance of these zones, we conduct an experiment in which we apply modifications to the identified classes in one project, JFreeChart.

Subject system. We chose JFreeChart as a subject system not only for its intermediate size allowing an easy discovery of the codebase, but also because this system has been extensively studied in previous work from *VariCity*'s authors [59, 83, 86], where they provide details on the implemented variability.

Evaluation process. We first selected in the set of 10 critical variability intense classes determined in the quantitative evaluation (*cf.* table 2) the ones maximizing their number of duplicated blocks or minimizing their test coverage. Six classes remained, of

Table 3: Measures of the refactored and added classes, before and after the refactor.

Class name (in the `org.jfree.chart` package)		Coverage	# duplicated blocks	Cognitive complexity
Identified relevant classes				
`plot.CategoryPlot`	*before*	74.9%	23	503
	after	75.4%	16	392
`plot.XYPlot`	*before*	70.0%	24	666
	after	69.9%	21	603
`axis.DateAxis`	*before*	71.8%	10	201
	after	77.2%	0	139
`axis.NumberAxis`	*before*	78.7%	12	163
	after	77.8%	4	127
`entity.ChartEntity`	*before*	30.7%	0	26
	after	90.5%	0	26
`ChartPanel`	*before*	25.7%	7	322
	after	52.2%	2	295
Other already present classes				
`axis.LogarithmicAxis`	*before*	16.0%	4	315
	after	17.1%	0	281
`axis.LogAxis`	*before*	45.3%	10	92
	after	47.0%	7	87
`axis.PeriodAxis`	*before*	29.3%	2	112
	after	30.6%	1	104
Added classes				
`plot.CategoryXYCommon`	*before*	–	–	–
	after	77.4%	6	97
`axis.DatePeriodCommon`	*before*	–	–	–
	after	46.2%	0	8
`axis.NumberLogCommon`	*before*	–	–	–
	after	81.6%	0	5
Overall project	*before*	54.5%	604	15,858
	after	55.3%	565	15,622

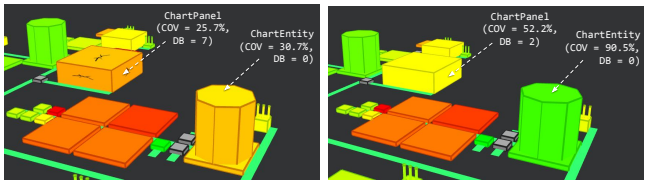


(a) Before refactor (b) After refactor

Figure 6: View of `DateAxis` and `NumberAxis` before and after the refactor. Visualization settings are identical to fig. 5.

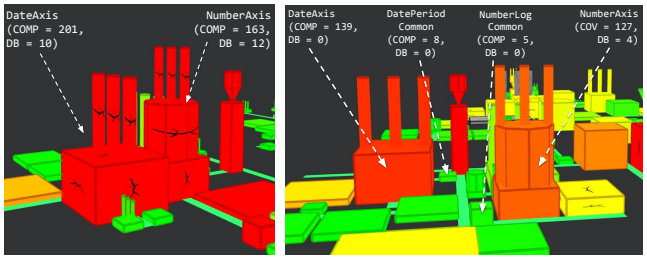
which four suffer from code duplication (`CategoryPlot`, `XYPlot`, `DateAxis`, and `NumberAxis`, visible due to their extensively cracked texture on fig. 5a for the first two and on fig. 6a for the last two) and two others from a lack of tests (`ChartPanel` and `ChartEntity`, visible due to their orange and yellow colors on fig. 7a).

We then defined and applied maintenance actions for these classes. Regarding classes suffering from code duplication, duplicated blocks were factorized in new methods. It happened that block duplications were present in different classes (*e.g.*, behavior



(a) Before refactor **(b) After refactor**

Figure 7: Classes lacking tests before and after the refactor. Visualization settings are identical to fig. 5.



(a) Before refactor **(b) After refactor**

Figure 8: Figure 6 displaying cognitive complexity on the red-green scale instead of the coverage (range: 0 → 150).

from `CategoryPlot` is duplicated in `XYPlot`). In this case, the factorization was placed in another class, created for that purpose (here, `CategoryXYCommon`). Regarding classes lacking tests, new test cases for several methods that were little to not tested have been added to the existing test classes. To ensure as much as possible that our modifications did not hamper the system stability, we did not change the logic of existing tests and made sure that the project could build with all tests passing.

A first observation we made concerns the nature of the duplicated blocks. Whereas some duplications are pure technical debt in classes concentrating variability implementations, others clearly correspond to improperly managed variability implementations. For example, in `DateAxis`, multiple lines of the `refreshTicksHorizontal`[14] method are duplicated in `refreshTicksVertical`[15]. They correspond to the common part creating the time tick, whereas the variable part concerns the orientation of the text on the plot. Therefore, such zones exhibited by *VariMetrics* actually spot improper variability management. We reapplied *VariMetrics* on the new codebase [60] and observed the differences shown in figs. 5b, 6b and 7b. We also computed the test coverage, cognitive complexity, and number of duplicated blocks for all the classes impacted by our maintenance actions before and after their modification, and summarized these results in table 3.

Regarding the classes suffering from code duplications, evolutions can be observed in figs. 5 and 6. The disappearance of the cracks on `NumberAxis` and `DateAxis` suggests that very little to no duplication remains, while the reduced amount of cracks on `CategoryPlot` denotes a decrease of duplications while some are still present. Finally, `XYPlot` appears equally cracked, propounding that duplications are still present. These observations are confirmed by the values from table 3: duplications in `NumberAxis` and `DateAxis` have been reduced by 75% and 100%, leaving respectively 4 and 0 duplications. Although the number of duplications in `CategoryPlot` diminished by 29%, 16 duplicated blocks remain, representing a non-negligible amount. Finally, 3 duplications have been removed in `XYPlot`, representing 13% of reduction, that is not significative enough to be shown on the visualization.

Similarly, improvements can also be seen in the classes that were lacking tests (fig. 7b). The transition from 31% to 91% of coverage for `ChartEntity` is translated on the visualization by a bright green color for its building, where the more contained improvement on `ChartPanel`'s coverage leads to its building color changing from orange to yellow.

Another effect induced by these maintenance actions can be seen in the visualization. The crack on `ChartPanel`'s building visible in fig. 7a disappeared in fig. 7b, although removing duplications was not a maintenance action for this class. This is because testing some methods required splitting them, leading to smaller blocks that could be reorganized. In this case, three duplicated blocks were extracted in a single testable method.

Finally, it appears that the maintenance actions on these classes improved their quality *w.r.t.* the considered metrics (*i.e.*, coverage and duplicated blocks). These changes however did not only impact the six considered classes, but also three other existing classes having duplications and led to the creation of three new classes to host some duplications. It is therefore important to consider these classes and ensure that they do not express the variability debt that has been treated. Modifications applied to the already existing classes solely concern the removal of duplications, therefore their quality has also been improved. Regarding the newly created classes, they are now visible (*cf.* figs. 5b and 6b). `DatePeriodCommon`'s yellow color presents a relatively low test coverage of 46%, which can be explained by the low initial test coverage of `PeriodAxis` of 29.3%. Adding tests would help to solve the issue. The other two classes have high coverages above 70%, and none of the three classes has a cracked texture, showing that no variability debt related to these metrics has been created.

By presenting the coverage and the number of duplicated blocks, the visualizations exhibited in figs. 5 to 7 can only demonstrate variability debt related to those two metrics. However, as explained in section 4.2, cognitive complexity is also a factor of variability debt. As this metric is significant for JFreeChart (*cf.* table 1), it is thus important to evaluate its evolution. It appears in table 3 that the cognitive complexity globally decreased for all relevant classes and the other already present ones. This can be explained by the fact that removing code duplications and adding tests often implies splitting methods into smaller ones, thus reducing cognitive complexity. This decrease can also be observed using *VariMetrics*, as its configuration capabilities easily allow to adapt the view to display this metric (*cf.* fig. 8 with an intensity decrease on `DateAxis` and `NumberAxis`). Concerning the newly created classes, `CategoryXYCommon`'s important cognitive complexity of 97 is because `CategoryPlot` and `XYPlot` have major cognitive complexities of 503 and 666 respectively. Therefore, the factorized blocks are themselves complex,

[14] https://github.com/jfree/jfreechart/tree/v1.5.0/src/main/java/org/jfree/chart/axis/DateAxis.java#L1558-L1609
[15] https://github.com/jfree/jfreechart/tree/v1.5.0/src/main/java/org/jfree/chart/axis/DateAxis.java#L1676-L1726

and would need further refactoring (*e.g.*, splitting into separate methods) to reduce this complexity and remove its 6 duplicated blocks.

Summary. By implementing maintenance actions on the identified quality-critical variability intense classes, we improved their quality regarding the considered metrics without introducing new debt factors, leading to a positive impact at the project level. These changes are also clearly observable in the visualization. Moreover, part of the identified variability debt directly concerned roughly managed variability that could be refactored.

6 THREATS TO VALIDITY AND LIMITATIONS

As we did not conduct an empirical evaluation, the major threat of our work is related to the design and realization of the evaluations done by ourselves, including the configuration of the views and choice of the metrics. Nevertheless, the scenarios demonstrating *VariCity* [59] gave us insights into the criteria to design views exhibiting relevant variability implementations. The metrics choice was driven by recent work on the factors causing variability debt [95], giving us confidence in their relevance in our context. Moreover, as the views we obtained allowed us to obtain positive results, we expect real experts to obtain good outcomes on their systems by applying their settings.

We evaluated our approach on 7 systems. Although this dataset is small, the studied systems have various sizes (40k → 2.5M LoC) and architectures (API, standalone library…), and represent different domains (charting, programming language, geospatial data management…). We are thus confident in the applicability of our results to other Java-based systems.

Regarding the visualization, we chose to offer as many configuration capabilities as possible to the expert so that they can tailor it freely and reach the view that helps them most. Combining multiple metrics on different axes can yet induce cognitive load and hamper the view's understanding. While measuring this load is of prime importance when designing visualizations [37] including city-based ones [13, 21] to ensure readability and usability, it would require in our case to empirically validate our approach with real experts to exchange on their needs[16]. This is part of our future work.

As for scalability, the analysis part is directly related to *symfinder* capabilities, which can handle projects with several millions of LoC but takes hours to do so (more than 120h for the JDK as shown in table 4). As the analysis can be synchronized with main releases, this is still reasonable for such very large projects. On the rendering side, our extension of *VariCity* only configures coloring and adds some textures, which are negligible for the rendering time. We are thus dependent on the main bottleneck of *VariCity* rendering, which lies in the computation of the city shape and streets (more than 5 five minutes for the JDK). For very large projects, this is hampering the configuration of the view as recomputation must be done when usage orientation and levels are changed. Nevertheless, from our analysis of the algorithm used in *VariCity*, we believe that some significant improvements could be made to make (re-)rendering practicable.

[16] Such a validation would also exhibit potential accessibility issues that can be tackled by extending the existing configuration capabilities.

Table 4: Subject systems and their execution times.

System	*symfinder* execution	*VariMetrics* city rendering
Azureus	1 h 25 min	4 s
GeoTools	24 h	1 min 10 s
JDK	123 h 22 min	5 min 40 s
JFreeChart	5 min 13s	1 s
JKube	2 min 8 s	< 1 s
OpenAPI Generator	3 min 45 s	< 1 s
Spring framework	1 h 5 min	6 s

System for *symfinder*: Ubuntu 18.04.2 LTS with Intel Xeon CPU E5-2637v2 @ 3.5GHz and 128Go memory.
System for *VariMetrics*: Google Chrome 99.0.4844.84 on Arch Linux 5.16.16-arch1-1 with Intel i7-9850H (12 cores) @ 4.6GHz and 32Go memory.

7 CONCLUSION

When object-oriented variability-rich software systems implement variability in a single codebase, they rely on mechanisms from the supporting language to realize it (*i.e.*, inheritance, overloading, design patterns), making them prone to induce variability debt. Identifying its causes is essential for software maintenance and quality. In this paper, we proposed *VariMetrics*, an extension of the city-based *VariCity* visualization to support the organized display of OO quality metrics as additional visual properties in a city in which dense zones of highly visible buildings already show zones of potential variability implementations. Multiple additional options allow the user to configure the view by choosing and combining the desired metrics to match their definition of quality. We conducted a quantitative evaluation on several open-source systems and a deeper qualitative evaluation on one of them, showing how *VariMetrics* can help to distinguish quality-critical zones of variability implementations.

We expect *VariMetrics* to be a first step towards a better understanding of variability debt in the context of variable OO systems. As future work, we plan to first explore how some code smells such as code duplication can be related to a form of badly implemented variability implementations, to improve quality. We will also conduct an empirical evaluation with experts to better understand how *VariMetrics* could be extended to match the industry's needs and expectations.

OPEN SCIENCE

A reproducible artifact is available online as an archive [60] containing the source code of *VariMetrics*, the Excel file used to obtain the data presented in table 2, additional views for all projects presented in table 2, the codebases of JFreeChart before and after the refactor presented in section 5.2 (with the corresponding diff file, excerpts of the SonarQube analysis of both codebases showing the information presented in table 3. These information can also be found on a companion webpage[17].

ACKNOWLEDGMENTS

We thank Patrick Anagonou, Guillaume Savornin and Anton van der Tuijn for their contribution in the development of *VariMetrics*.

[17] https://deathstar3.github.io/varimetrics-demo/

REFERENCES

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the linux kernel: a qualitative analysis. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 421–432.

[2] Wasim Alsaqaf, Maya Daneva, and Roel Wieringa. 2017. Quality requirements in large-scale distributed agile projects–a systematic literature review. In *International working conference on requirements engineering: foundation for software quality*. Springer, 219–234.

[3] Berima Andam, Andreas Burger, Thorsten Berger, and Michel RV Chaudron. 2017. Florida: Feature location dashboard for extracting and visualizing feature traces. In *Proceedings of the Eleventh International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 100–107.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[5] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. 2016. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, Vol. 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[6] Paris C Avgeriou, Davide Taibi, Apostolos Ampatzoglou, Francesca Arcelli Fontana, Terese Besker, Alexander Chatzigeorgiou, Valentina Lenarduzzi, Antonio Martini, Athanasia Moschou, Ilaria Pigazzini, et al. 2020. An overview and comparison of technical debt measurement tools. *IEEE Software* 38, 3 (2020), 61–71.

[7] Felix Bachmann and Paul Clements. 2005. *Variability in Software Product Lines*. Technical Report CMU/SEI-2005-TR-012. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675

[8] Maria Teresa Baldassarre, Valentina Lenarduzzi, Simone Romano, and Nyyti Saarimäki. 2020. On the diffuseness of technical debt items and accuracy of remediation time when using SonarQube. *Information and Software Technology* 128 (2020), 106377.

[9] Woubshet Behutiye, Pertti Karhapää, Lidia López, Xavier Burgués, Silverio Martínez-Fernández, Anna Maria Vollmer, Pilar Rodríguez, Xavier Franch, and Markku Oivo. 2020. Management of quality requirements in agile and rapid software development: A systematic mapping study. *Information and software technology* 123 (2020), 106225.

[10] Alexandre Bergel, Razan Ghzouli, Thorsten Berger, and Michel R. V. Chaudron. 2021. FeatureVista: Interactive Feature Visualization. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A*. Association for Computing Machinery, New York, NY, USA, 196–201. https://doi.org/10.1145/3461001.3471154

[11] G Ann Campbell. 2018. Cognitive complexity: An overview and evaluation. In *Proceedings of the 2018 international conference on technical debt*. 57–58.

[12] Rafael Capilla, Jan Bosch, Kyo-Chul Kang, et al. 2013. Systems and software variability management. *Concepts Tools and Experiences* (2013).

[13] Pierre Caserta, Olivier Zendra, and Damien Bodénes. 2011. 3D hierarchical edge bundles to visualize relations in a software city metaphor. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE, 1–8.

[14] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. 2020. A systematic literature review of modern software visualization. *Journal of Visualization* 23, 4 (2020), 539–558.

[15] Fatima Nur Colakoglu, Ali Yazici, and Alok Mishra. 2021. Software product quality metrics: A systematic mapping study. *IEEE Access* (2021).

[16] James Coplien, Daniel Hoffman, and David Weiss. 1998. Commonality and Variability in Software Engineering. *IEEE Software* 15, 6 (1998), 37–45. https://doi.org/10.1109/52.730836

[17] James O. Coplien. 1999. *Multi-Paradigm Design for C++*. Addison-Wesley Longman Publishing Co., Inc.

[18] 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.

[19] James O Coplien and Liping Zhao. 2005. Toward a General Formal Foundation of Design-Symmetry and Broken Symmetry.

[20] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wasowski. 2012. Cool features and tough decisions: a comparison of variability modeling approaches. In *Proceedings of the sixth international workshop on variability modeling of software-intensive systems (VaMoS'12)*. 173–182. https://doi.org/10.1145/2110147.2110167

[21] Veronika Dashuber, Michael Philippsen, and Johannes Weigend. 2021. A Layered Software City for Dependency Visualization.. In *VISIGRAPP (3: IVAPP)*. 15–26.

[22] Christof Ebert and Michel Smouts. 2003. Tricks and traps of initiating a product line concept in existing products. In *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 520–525.

[23] Jorge Echeverría, Francisca Pérez, José Ignacio Panach, and Carlos Cetina. 2021. An empirical study of performance using Clone & Own and Software Product Lines in an industrial context. *Information and Software Technology* 130 (2021), 106444.

[24] Wolfram Fenske and Sandro Schulze. 2015. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*. 3–10.

[25] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. 2017. Software landscape and application visualization for system comprehension with ExplorViz. *Information and software technology* 87 (2017), 259–277.

[26] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[27] Critina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR '01)*. ACM, 109–117. https://doi.org/10.1145/375212.375269

[28] 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. https://doi.org/10.1145/3302333.3302336

[29] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2013. Variability in Software Systems — A Systematic Literature Review. *IEEE Transactions on Software Engineering* 40, 3 (2013), 282–306. https://doi.org/10.1109/TSE.2013.56

[30] Matthias Galster, Uwe Zdun, Danny Weyns, Rick Rabiser, Bo Zhang, Michael Goedicke, and Gilles Perrouin. 2017. Variability and complexity in software design: Towards a research agenda. *ACM SIGSOFT Software Engineering Notes* 41, 6 (2017), 27–30.

[31] Eddy Ghabach. 2018. *Supporting Clone-and-Own in software product line*. Ph. D. Dissertation. COMUE Université Côte d'Azur (2015-2019).

[32] Eddy Ghabach, Mireille Blay-Fornarino, Franjieh El Khoury, and Badih Baz. 2018. Guiding Clone-and-Own When Creating Unplanned Products from a Software Product Line. In *International Conference on Software Reuse*. Springer, 139–147.

[33] Orla Greevy, Michele Lanza, and Christoph Wysseier. 2005. Visualizing feature interaction in 3-D. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 1–6.

[34] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. 2015. Tracking the software quality of android applications along their evolution (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 236–247.

[35] Rich Hilliard. 2010. On Representing Variation. In *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume (ECSA '10)*. ACM, 312–315. https://doi.org/10.1145/1842752.1842810

[36] Danny Holten, Roel Vliegen, and Jarke J Van Wijk. 2005. Visual realism for the visualization of software metrics. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 1–6.

[37] Weidong Huang, Peter Eades, and Seok-Hee Hong. 2009. Measuring effectiveness of graph visualizations: A cognitive load perspective. *Information Visualization* 8, 3 (2009), 139–152.

[38] Emanuel Irrazábal, Juan Andrés Carruthers, and Juan Alberto Pinto Oppido. 2021. Modelo para curaduría de proyectos software de fuente abierta para estudios empíricos en ingeniería de software. In *XXIII Workshop de Investigadores en Ciencias de la Computación (WICC 2021, Chilecito, La Rioja)*.

[39] Ivar Jacobson, Martin Griss, and Patrik Jonsson. 1997. *Software reuse: architecture process and organization for business success*. Vol. 285. acm Press New York.

[40] Isabel John, Jaejoon Lee, and Dirk Muthig. 2007. Separation of Variability Dimension and Development Dimension. In *Proocedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '07)*. 45–49.

[41] Dennis Kafura and Sallie Henry. 1981. Software quality metrics based on interconnectivity. *Journal of Systems and Software* 2, 2 (1981), 121–131.

[42] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *SPLC (2)*. 303–312.

[43] Raees Ahmad Khan, Khurram Mustafa, and Syed I Ahson. 2007. An empirical validation of object oriented design quality metrics. *Journal of King Saud University-Computer and Information Sciences* 19 (2007), 1–16.

[44] Barbara Kitchenham and Shari Lawrence Pfleeger. 1996. Software quality: the elusive target [special issues section]. *IEEE software* 13, 1 (1996), 12–21.

[45] Claire Knight and Malcolm Munro. 2000. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*. IEEE, 198–205.

[46] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.

[47] Duc Le, Eric Walkingshaw, and Martin Erwig. 2011. # ifdef confirmed harmful: Promoting understandable software variation. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 143–150.

[48] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are sonarqube rules inducing bugs?. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 501–511.

[49] Valentina Lenarduzzi, Alberto Sillitti, and Davide Taibi. 2018. A survey on code analysis tools for software maintenance prediction. In *International Conference in Software Engineering for Defence Applications*. Springer, 165–175.

[50] 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.

[51] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 105–114.

[52] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2018. A systematic mapping study of information visualization for software product line engineering. *Journal of software: evolution and process* 30, 2 (2018), e1912.

[53] Antonio Martini and Jan Bosch. 2015. The danger of architectural technical debt: Contagious debt and vicious circles. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 1–10.

[54] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320.

[55] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. 2017. Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering* 44, 5 (2017), 453–469.

[56] Mira Mezini and Klaus Ostermann. 2004. Variability management with feature-oriented programming and aspects. *ACM SIGSOFT Software Engineering Notes* 29, 6 (2004), 127–136.

[57] Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. 2018. A suite of object oriented cognitive complexity metrics. *IEEE Access* 6 (2018), 8782–8796.

[58] Austin Mordahl, Jeho Oh, Ugur Koc, Shiyi Wei, and Paul Gazzillo. 2019. An empirical study of real-world variability bugs detected by variability-oblivious tools. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 50–61.

[59] 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. https://doi.org/10.1109/VISSOFT52517.2021.00017

[60] Johann Mortara, Philippe Collet, Anne-Marie Pinna-Dery, Patrick Anagonou, Guillaume Savornin, and Anton van der Tuijn. 2022. *Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations - Artifact.* https://doi.org/10.5281/zenodo.6644634

[61] Johann Mortara, Philippe Collet, and Xhevahire Tërnava. 2020. Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder. In *24th ACM International Systems and Software Product Line Conference (SPLC '20)*, ACM, New York, NY, and USA (Eds.). MONTREAL, QC, Canada, 9–12. https://doi.org/10.1145/3382025.3414987 Virtual Conference.

[62] 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. https://doi.org/10.1145/3307630.3342394

[63] Johann Mortara, Xhevahire Tërnava, and Philippe Collet. 2020. Mapping Features to Automatically Identified Object-Oriented Variability Implementations - The case of ArgoUML-SPL. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems*. Magdeburg, Germany, 1–9. https://doi.org/10.1145/3377024.3377037

[64] 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

[65] Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source code metrics: A systematic mapping study. *Journal of Systems and Software* 128 (2017), 164–197.

[66] Jevgenija Pantiuchina, Michele Lanza, and Gabriele Bavota. 2018. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 80–91.

[67] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program Comprehension and Code Complexity Metrics: An fMRI Study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 524–536.

[68] Luca Pellegrini, Andrea Alexander Janes, and Davide Taibi. 2018. On the Fault Proneness of SonarQube Technical Debt Violations. An empirical study. *Ph. D. dissertation* (2018).

[69] Federico Pfahler, Roberto Minelli, Csaba Nagy, and Michele Lanza. 2020. Visualizing Evolving Software Cities. In *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 22–26.

[70] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Science & Business Media.

[71] Rick Rabiser. 2019. Feature Modeling vs. Decision Modeling: History, Comparison and Perspectives. In *Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B (SPLC '19)*. ACM, 134–136. https://doi.org/10.1145/3307630.3342399

[72] Ghulam Rasool and Zeeshan Arshad. 2015. A review of code smell mining techniques. *Journal of Software: Evolution and Process* 27, 11 (2015), 867–895.

[73] Linda H Rosenberg and Lawrence E Hyatt. 1997. Software quality metrics for object-oriented environments. *Crosstalk journal* 10, 4 (1997), 1–6.

[74] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing cloned variants: a framework and experience. In *Proceedings of the 17th International Software Product Line Conference*. 101–110.

[75] Danilo Sato, Alfredo Goldman, and Fabio Kon. 2007. Tracking the evolution of object-oriented quality metrics on agile projects. In *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 84–92.

[76] Syed Muhammad Ali Shah, Marco Torchiano, VETRO' ANTONIO, and Maurizio Morisio. 2013. Exploratory testing as a source of testing technical debt. *IT Professional IEEE Computer Society Digital Library* (2013), 25.

[77] Sandra A Slaughter, Donald E Harter, and Mayuram S Krishnan. 1998. Evaluating the cost of software quality. *Commun. ACM* 41, 8 (1998), 67–73.

[78] Iuri Santos Souza, Ivan Machado, Carolyn Seaman, Gecynalda Gomes, Christina Chavez, Eduardo Santana de Almeida, and Paulo Masiero. 2019. Investigating Variability-aware Smells in SPLs: An Exploratory Study. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*. 367–376.

[79] Frank Steinbrückner and Claus Lewerentz. 2010. Representing development history in software cities. In *Proceedings of the 5th international symposium on Software visualization*. 193–202.

[80] Frank Steinbrückner and Claus Lewerentz. 2013. Understanding software evolution with software cities. *Information Visualization* 12, 2 (2013), 200–216.

[81] Srdjan Stevanetic and Uwe Zdun. 2015. Software metrics for measuring the understandability of architectural structures: a systematic mapping study. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. 1–14.

[82] Mikael Svahnberg, Jilles Van Gurp, and Jan Bosch. 2005. A taxonomy of variability realization techniques. *Software: Practice and experience* 35, 8 (2005), 705–754.

[83] 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. https://doi.org/10.1007/s10515-022-00329-x

[84] Xhevahire Tërnava and Philippe Collet. 2017. On the Diversity of Capturing Variability at the Implementation Level. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. ACM, 81–88. https://doi.org/10.1145/3109729.3109733

[85] 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. https://doi.org/10.1007/978-3-319-56856-0_8

[86] 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

[87] C. Reid Turner, Alfonso Fuggetta, Luigi Lavazza, and Alexander L. Wolf. 1999. A Conceptual Basis for Feature Engineering. *Journal of Systems and Software* 49, 1 (1999), 3–15. https://doi.org/10.1016/S0164-1212(99)00062-X

[88] Howard Wainer and Carl M Francolini. 1980. An empirical inquiry concerning human understanding of two-variable color maps. *The American Statistician* 34, 2 (1980), 81–93.

[89] 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.

[90] 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.

[91] Richard Wettel and Michele Lanza. 2008. Visual exploration of large-scale system evolution. In *2008 15th Working Conference on Reverse Engineering*. IEEE, 219–228.

[92] Richard Wettel and Michele Lanza. 2008. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM Symposium on Software Visualization*. 155–164.

[93] 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*. 551–560.

[94] Karl Wiegers and Joy Beatty. 2013. *Software requirements.* Pearson Education.

[95] 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.

[96] Liping Zhao and James Coplien. 2003. Understanding symmetry in object-oriented languages. *Journal of Object Technology* 2, 5 (2003), 123–134.