Visualization of Object-Oriented Software in a City Metaphor: Comprehending the Implemented Variability and its Technical Debt

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

Visualization of Object-Oriented Software in a City Metaphor: Comprehending the Implemented Variability and its Technical Debt

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

Université Côte d'Azur, CNRS, I3S, France

Abstract

While many large-scale software systems intensively implement variability to reuse software and speed up development, they often do not document it, hampering its comprehension. This is especially the case for variability-rich object-oriented (OO) systems that heavily rely on existing OO mechanisms (*i.e.*, inheritance, overloading and some patterns) to implement it in a single codebase. With no traceability information, the variability is buried in the codebase, hampering its identification, analysis, and understanding. While variability management becomes increasingly difficult over the system evolution, the implementation mechanisms also bring additional complexity to the codebase, which eventually leads to technical debt, threatening even more the software quality.

In this article, we report on the design and evaluation of an extensible visualization, *VariCity*, that exhibits zones of high density of OO variability implementations. It relies on the city metaphor to represent the classes of the system as buildings whose dimensions are used to show variability metrics inherent to the implementation classes. They are linked together through streets depicting usage relationships and grouping in neighborhoods classes using each other. The extensibility of *VariCity* is demonstrated with *VariMetrics*, which highlights OO quality metrics on the buildings, revealing quality-critical classes concentrating variability implementations. We evaluate the visualization capacity to reveal zones concentrating variability implementations and being quality-critical by applying it to multiple variability-intensive open source software systems. We also report on a controlled experiment comparing the gain brought by the visualization with to the use of an IDE.

Keywords: software variability, reverse-engineering, software visualization, software cities, program comprehension, software quality

1. Introduction

Whatever their scale and their domain, recent software-intensive systems and applications are more and more variability intensive [1, 2, 3]. Software variability is usually defined as the ability of a software artifact (*i.e.*, system or element that enables to develop it) to be efficiently extended, changed, customized, or configured towards a specific context [4]. Variability can be seen as an anticipated change that evolves over time [5], calling for appropriate management techniques for software engineers of variability-intensive systems.

Being a key element in most systems [1], variability management has been heavily studied, notably leading to the Software Product Line (SPL) [6, 7] and product family [4] paradigms. In the SPL paradigm, the domain variability, commonly documented and managed in terms of features in a feature model [8], is clearly separated from the implemented variability, which is mapped from the domain variability, using most of the times a single implementation technique. In this context, implementation techniques can be diverse, reusing existing mechanisms such as preprocessor directives [9], or being completely specific with a specialized form of modules [7]. The implemented variability may also be managed through an implemented-related feature model [10, 11], but the main benefits of an SPL are to reason on consistency at the domain level and from a configuration, to derive a consistent software product [12].

However, many variability-rich software systems introduce variability progressively or manage diversity in their functionalities without following a complete SPL approach. Many of them are object-oriented and implemented in a single codebase in which variability among the obtainable software products is realized using traditional techniques (*i.e.*, inheritance, parameters, overloading, and some design patterns such as strategy and factory) [13, 14, 4]. These mechanisms being also used to structure the implementation, the variability is buried in the codebase, hampering its identification, analysis, and understanding as there is no traceability with domain information [15, 16]. In this work, we target OO codebases that are not associated in any way with additional information regard-

Email addresses: johann.mortara@univ-cotedazur.fr (Johann Mortara), philippe.collet@univ-cotedazur.fr (Philippe Collet), anne-marie.pinna@univ-cotedazur.fr (Anne-Marie Dery-Pinna)

ing variability (*e.g.*, UML-based variability description, source code annotations). As a consequence, managing this variability becomes increasingly difficult throughout the evolution of the system [4, 17, 18], eventually leading to technical debt [19, 20]. The technical debt represents short-term applications of design and implementation constructs that make future modifications more costly or impossible, thus impacting system's maintainability and evolution [20]. Applied to variability mechanisms, it led to the term of variability debt [21], which is especially characterized by the lack of knowledge on implemented variability in the source code, with the same kind of negative impacts. Therefore there is a need for a solution to facilitate the comprehension of OO variability implementations and their quality in single code bases that do not contain any other information except the implementation mechanisms. The identification of such implementations can then be exploited in different ways, from the maintenance of the variable system to some variability reengineering [22] or a migration to a full software product lines [23]. For example, the result of the underlying detection used in our solution (*i.e.*, *symfinder*) has been successfully mapped to a domain feature model in two different variable systems [24, 25]. However, we only focus in this article on a visualization adapted to the identification of the OO implemented variabilities to facilitate variability comprehension of large OO systems.

Comprehending properties of software systems is often assisted by visualizations [26, 27, 28] which, relying on metaphors to represent them [29, 30], help their understanding [31]. On one side, while visual approaches have been proposed to understand variable systems [32], they focus on domain variability [33, 34] that is often not documented in the case of OO systems [35]. The *symfinder* approach [36, 37] and its extension *symfinder-2* [38] proposed to identify OO variability implementations rely on a graph to represent them. However, tackling large codebases is known to be a weak point of graph visualizations [39]. On the other side, metaphors scaling on large systems have been proposed such as the metaphor of the city [40]. This metaphor has been adapted to represent multiple properties of OO systems such as quality metrics [41] and their evolution [42] with a city shape based on packages. However, the proposed organizations are not adapted to represent OO variability implementation mechanisms as their identification relies on dense zones of classes with high variability metrics and the usage relationships between these classes.

In this paper, we propose a visualization, *VariCity*, that relies on the metaphor of the city to reveal zones of high density of variability implementations in an OO system. Classes with metrics about the OO mechanisms (*e.g.*, number of overloaded methods in a class) are obtained by reusing the *symfinder* toolchain [36]. In the visualization, buildings represent classes linked together through streets representing usage relationships (*i.e.*, composition, aggregation). While the buildings dimensions are used to represent computed variability metrics inherent to the classes (*i.e.*, overloads of methods and constructors), their color and texture allow representing quality metrics. Configuration options are provided to tailor a view on a subpart of the studied system and display only the variability of interest for the user. We also demonstrate the extensible nature of *VariCity* by introducing the *VariMetrics* extension which exploits software quality metrics to reveal critical zones concentrating technical debt. One can choose the quality metrics to be displayed, some displaying strategies (red-to-green sequence, saturation, crackled texture) and how to combine them, to tailor the visualization according to their needs. The capacity of the visualizations to reveal zones concentrating variability implementations and being quality-critical has been evaluated by applying them to multiple variability-intensive open source software systems implemented in Java. We also report on a controlled experiment comparing the gain brought by the visualization with the use of an IDE on a mid-size variable OO system.

An earlier version of the presented approach appeared at the VISSOFT'2021 conference [43] in which we introduced *VariCity*. The *VariMetrics* extension of *VariCity* adding support for quality metrics on the view has been presented at the SPLC'2022 conference [44]. This article substantially extends these publications by

- providing a more complete validation on the *VariCity* visualization, with quantitative measurements on the identified zones of variability;
- providing a similar validation on the *VariMetrics* visualization, which was only validated through usage scenarios [44];
- providing a controlled experiment evaluating the gain of using *VariCity* compared to an Integrated Development Environment (IDE) with 2 separated groups of 24 users.

The remainder of the paper is organized as follows. Section 2 introduces OO variability implementations and how they can be identified. Section 3 defines the requirements for a view to comprehend OO variability implementations and introduces the city metaphor. Section 4 presents *VariCity*, detailing the view's organization and the visual axes used to represent variability, and its extension *VariMetrics*. In Section 5, we apply the visualizations on multiple open source Java systems and show their capacity to reveal zones concentrating variability implementations (Sections 5.1 and 5.2). We also evaluate how the quality metrics visualization, when available, reveals indebted classes concentrating variability implementations (Section 5.3). We report on a controlled experiment conducted to evaluate the comprehensibility of the visualization by real users in Section 6. Threats to the validity of our approach and its limitations are discussed in Section 7, while Section 8 presents related work. Finally, Section 9 concludes the paper and discusses future work.

2. Background

Many object-oriented software systems become progressively variability rich and do not follow the full software product line paradigm [6, 7]. Their domain variability (*i.e.*, features) is then not very well documented and is not made explicit within code assets, potentially causing technical debt [21]. In this context, comprehending the variability at code level is crucial for its management. Activities related to comprehension can be as diverse as maintaining or evolving the code, mapping the implemented variability to domain features [17], or conducting an onboarding process for newcomers [45].

2.1. Implemented variabilities in object-oriented systems

Variability in object-oriented systems can be distinguished in code assets through three different parts: core, commonalities, and variations [46, 47, 48]. The core part corresponds to assets included in any of the final software products [46]. A commonality is a common part between the related variations of code assets, while the variations indicate how and when should code assets vary [1]. Commonalities and variations are respectively abstracted in terms of variation points (*vp*-s) and variants [49, 50, 51], which are both related to concrete elements in code assets [52]. A variation point identifies one or more locations where the variation will occur, while the way that a variation point is going to vary is expressed by its variants [49].

In object-oriented variability-rich systems implemented within a single codebase, the implementation of these variability elements reuses different existing mechanisms and techniques, such as inheritance, parameters, constructor and method overloading, or some software design patterns [13, 14, 4, 53]. Figure 1 shows an example of variability implemented using inheritance (the `Plot` superclass represents the *vp* while its subclasses represent the variants), constructor overloading (the `PiePlot` constructor is a *vp* with two variants being its two overloads) and method overloading (the `addDomainMarker` method is a *vp* with two variants being its two overloads). As the code units that structure the systems are classes, it has been shown that they do not align well with the implemented variability or the domain features [54, 7]. This hampers the comprehension of variability as classes cannot be directly used to understand the variability implementations. Variability implementation techniques have been analyzed according to different taxonomies [14, 55, 13, 56, 7], and it has been acknowledged that there is currently no approach to represent *vp*-s with variants in all forms of code assets [32]. In our proposition, we do not consider UML-based product lines, such as the ones that could be built with dedicated support like SMarty [57], nor codebases with variability-related annotations. In both cases, such additional information could be used and related to what is analyzed within the VariCity toolchain, while in this paper we focus on codebases with variability implemented with object-oriented techniques and with no other associated models.

Besides, if a complete representation of the implemented variability is not possible, it may be possible to comprehend it through SPL migration techniques. The techniques rely on diverse approaches, such as feature location, feature identification [58, 59, 22], feature delimitation (with annotations) [9], or feature modularization [7]. However, in all these techniques features tend to describe the domain variability of an SPL or a variability-rich system, but are required to be known in advance [11, 51]. As domain variability is hardly documented in variability-rich systems [35], and with a single codebase, reengineering of features from clones of a system cannot be used [60]. In our context, migrating thus requires substantial manual effort and implies a complete paradigm shift.

2.2. Metrics to identify OO variability implementations

Identifying *vp*-s with variants [61] implemented with object-oriented techniques in a single codebase is thus a hard problem by the diversity of the implementations, each one requiring its own way to be identified [61, 53]. Recently, Tërnava et al. [36] proposed an approach to identify OO variability implementation mechanisms and, consequently, *vp*-s with variants implemented by these techniques in Java systems. Figure 1 depicts the identification mechanism as implemented in the *symfinder* toolchain [37, 38] and the different kinds of information computed and obtained from the static analysis made by the toolchain through a graph database. Each class is represented as node in a graph with usage (references to other classes, *cf.* Definition 1) and inheritance relationships being edges. The static analysis adds labels on them together with metrics on the usage of variability related mechanisms. A *vp* with variants at class level is then labeled `VP`, and its variants have a `VARIANT` label (Definition 2). Classes with method level *vp*-s are labeled `METHOD_LEVEL_VP`. If a class is the *vp* of a design pattern, it is labeled with its name (*e.g.*, `STRATEGY`). *symfinder* is capable of detecting several implementations techniques, some related to inheritance (*i.e.*, class subtyping, method and constructor overloading,), the others being design patterns (*i.e.*, strategy, template, decorator, and factory). The different metrics on the number of *vp*-s and variants per class are also computed. For example, these metrics show that `XYPlot` exposes much more method *vp*-s and variants than `PiePlot`.

It has also been shown that locations in the code concentrating such mechanisms denote zones of interest in terms of variability [36]. Classes being part of such dense zones, called *hotspots*, exhibit two properties [38]: an *individual density* (Definition 3) representing a concentration of variability implementation mechanisms inside the class (overloads of methods and constructors), and a *collective density* (Definition 4) representing the proximity in terms of usage relationship to another class exhibiting variability implementations (*i.e.*, *vp* or variant). They can be

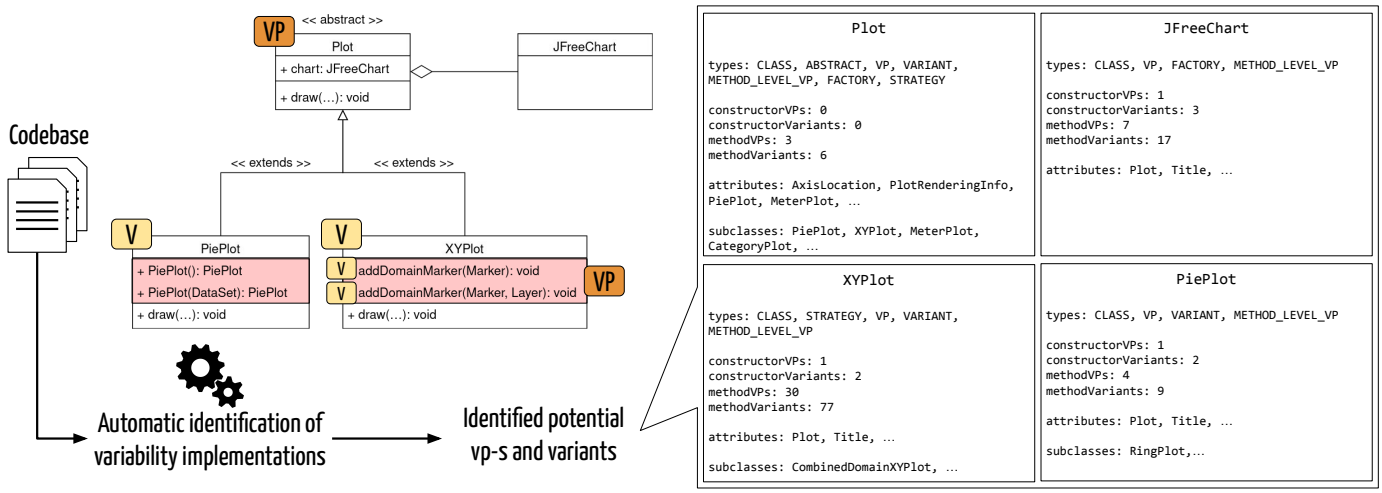


Figure 1: Depiction of *symfinder*'s variability identification mechanism

automatically determined thanks to the previously introduced metrics using thresholds on the minimum number of variants of a *vp* and its maximum distance in usage relationships with another *vp* or variant (Definition 5). Consequently, *symfinder* detects what could be defined as *implementation variability patterns*, either by the presence of specific design patterns or by the concentration of implementation techniques that all together denotes potential variability implementation locations. Both the design patterns and the density principle were validated on large software and for a single system with its software architect [16]. Besides, it must be noted that cross-cutting relations or dependencies between implemented variability locations cannot be determined by the *symfinder* technique. While a feature model could be generated from the *symfinder* output, we believe it would not be as useful as a visualization related to the codebase itself. However, some additional work has demonstrated that the detected variability can be mapped to a domain feature model with an average precision[1] and a good recall [24, 25]. The graph representation was also extended so that the mapped feature can be read while hovering on the different nodes representing the implementation classes. This shows the potential usages of the identification of this implemented variability, which could range from facilitating maintenance to a migration activity towards an SPL.

While an inheritance graph representation and the computed metrics were good enough to validate the relevance of *symfinder*'s identification technique in [36], its extension to represent the needed additional usage mechanisms considered in *symfinder-2* [38] has demonstrated that the resulting graph visualization becomes cumbersome to understand when the number of classes to display increases. While techniques have been proposed to tackle scalability issues of graph visualizations [62, 63], the challenge of tackling large codebases requires a scalable visualization, which is known to be a weak point of graph visualizations [39]. Moreover, according to a recent mapping study Lopez-Herrejon et al. [32], while many visual representations of variability management approaches are proposed in the context of SPLs, they most often target domain variability (*e.g.*, features in a feature model). These visual representations cannot then be reused directly.

Consequently, while identifying the variability implementations (*i.e.*, variation points and their variants) directly in code assets is the first activity to comprehend variability, it does not find satisfactory support at medium to large scales.

3. Motivations

In this section, we determine the requirements a visualization to assist the comprehension of OO variability implementation mechanisms needs to fulfill (Section 3.1). We then introduce the city metaphor and examine to what extent it can respect such requirements (Section 3.2).

3.1. Requirements

As program comprehension is seen as a process of both information seeking [64] and feature location [58], it is obvious that even if our problem is not related to *domain features* in a classic SPL terminology, identifying *vp*-s with variants is indeed a comprehension problem. Moreover, SPLs and variable software in general are known to be complex and difficult to apprehend [45], and tools are essential to illustrate software reuse concepts [65]. We then first advocate that this context naturally calls for visualization-based solutions as they are often used as supports to assist the comprehension of large software systems [66, 67, 68, 28] and aspects related to their implemented variability [69, 34, 33]. As the essence of software visualization consists of creating an image of software by

[1] the rather low precision is mainly due to the use of a partial feature model and the absence of mapping to mandatory features in the two considered systems.

means of visual objects that represent structure and/or behavior, we believe it is well suited to enable perception of variability implementations with a closer fit to the user mental model.

The goal of such a visualization being to assist the comprehension of variability implemented using OO mechanisms, it must therefore (*i*) reveal zones of interest as described in Section 2.2 (*i.e.*, zones exhibiting individual and collective density). As the variability metrics and the involved mechanisms to be displayed are diverse and of heterogeneous nature (classes, links between them, design patterns), displaying a view with all classes (and their usage / inheritance relationships) altogether would overload the visualization. Therefore, it should also (*ii*) provide configuration options allowing to display only the classes (and related relationships /metrics) of interest for the user as described by the Shneiderman information seeking mantra [70]: *overview first, zoom and filter, then details on demand*. Finally, understanding the quality of OO variability implementation mechanisms is essential as it can potentially harm their comprehensibility. While we could imagine navigating between a specific visualization and a tool specific to quality metrics, this would be cumbersome as it would require manually finding and mapping information having heterogeneous representations. The view should therefore be able to (*iii*) display quality metrics. State-of-the-art proposes a plethora of quality metrics to measure several properties of a software system [71], ranging from the architecture [72] to the source code level [73, 74]. Since no metric is relevant for all software systems due to the elusive definition of quality [75], software practitioners need to pick and combine different metrics to obtain a quality measure relevant for their use case. Being able to configure the quality metrics to display is therefore essential. We synthesize the determined requirements for our visualization as follows:

Requirement 1. Reveal zones of interest (*i.e.*, zones exhibiting individual and collective density).

Requirement 2. Provide configuration options allowing to display only the classes (and related relationships /metrics) of interest for the user.

Requirement 3. Display quality metrics and provide options to configure them.

Basing ourselves on this list of requirements, there is a need to find an appropriate visualization allowing to fulfill it.

3.2. On the city metaphor

A first approach could be to evolve the *symfinder*'s graph visualization to improve its scalability [62, 63], but as multiple visualizations rely on metaphors to get an understandable graphical representation [30], we sought for an appropriate metaphor. The city metaphor [29] has been applied to multiple types of metrics on software systems: dynamic behavior (such as concurrency between classes [76] or memory consumption of heaps [77]), and static properties such as dependency and communication links between components [78].

Visualization cities have also been proposed to understand OO software systems, with first CodeCity [41, 40] that uses buildings to represent classes, grouping them in districts representing packages. A temporal dimension was also added to visualize the evolution of the metrics through multiple versions of the system, first in CodeCity [79] and also in a more recent visualization called M3TRICITY [80, 81]. The Evo-Streets [42] approach also uses the city metaphor, but uses streets to represent the package decomposition instead of nested boxed areas in CodeCity. Multiple approaches also reuse the city metaphor by adding other visual dimensions such as arcs between buildings [82, 83] or by adapting it to more immersive techniques, such as virtual reality for CodeCity [84] and VRCity [85], or Minecraft for CodeMetropolis [86].

The popularity of this metaphor and the fact that CodeCity showed to help complete program comprehension tasks [87, 88] led us to the hypothesis that visualization based on the city metaphor could help the comprehension of OO variability implementations and their quality. However, cities such as CodeCity and Evo-Streets are organized relying on the package decomposition. While this arrangement is adapted to identify packages containing quality-critical classes, it does not allow to represent the inheritance and usage relationships between the classes. These mechanisms being involved in variability implementations, displaying them is essential to fulfill Requirement 1. We thus propose to adapt the city metaphor to display OO variability implementations.

4. VariCity: a configurable and extensible visualization for variability comprehension

We hereafter detail how *VariCity* adapts the city metaphor to the problem of OO variability comprehension by answering Requirements 1, 2 and 3 in Sections 4.1, 4.2 and 4.3 respectively.

4.1. Main principles for revealing zones of interest

Buildings. In CodeCity [79], classes are buildings and their dimensions evolve according to metrics related to code quality which are inherent to the represented class, such as the cyclomatic complexity or the number of lines of code (LoC). A large number of methods leads to the creation of a tall and eye-catching building. *VariCity* aims to focus the user on classes making heavy use of variability implementations (*cf.* Requirement 1). Therefore, we choose to shape the dimensions of every building so to represent the class-based metrics related to variability (*i.e.*, the number of variants at method level). A tall building then shows a large number of method variants, whereas a large building shows a large number of constructor variants. In addition the color of buildings (by default yellow

Table 1: Visual properties and their default color

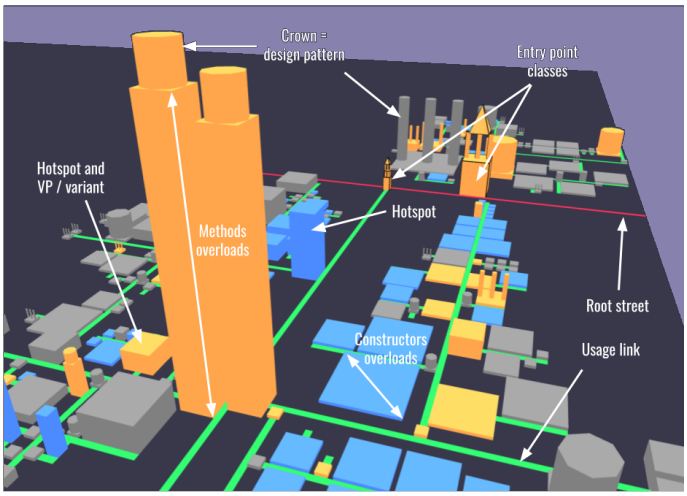
Representation in *VariCity*	Signification
Buildings	
Yellow color	Variation point that is part of a hotspot
Blue color	Non *vp* class that is part of a hotspot
Gray color	Class that is not part of a hotspot
Pyramide crown	Entry point class
Dome crown	Strategy pattern
Chimneys crown	Factory pattern
Inverted Pyramide crown	Template pattern
Sphere crown	Decorator pattern
Streets	
Plan (red)	Street aggregating entry point classes
Plan / Underground (green)	Usage relationship
Aerial (blue)	Inheritance relationship

for *vp*-s and blue for non *vp*-s) distinguishes classes defined as *hotspots* (*cf.* Section 2.2). Such classes are part of dense zones of variability and are *vp*-s identified by matching one of the two following requirements: (*i*) they have a minimum number of variants, 5 in our experiments, or (*ii*) they are close in usage to another *vp* (*i.e.*, they are situated at less than 3 transitive hops in the usage relationships graph). The shape of the building is altered according to the design pattern(s) exhibited by the class (*cf.* Table 1). A design pattern often involves multiple classes, however only the *vp* of the design pattern has a special crown on it, not to overload the visualization.

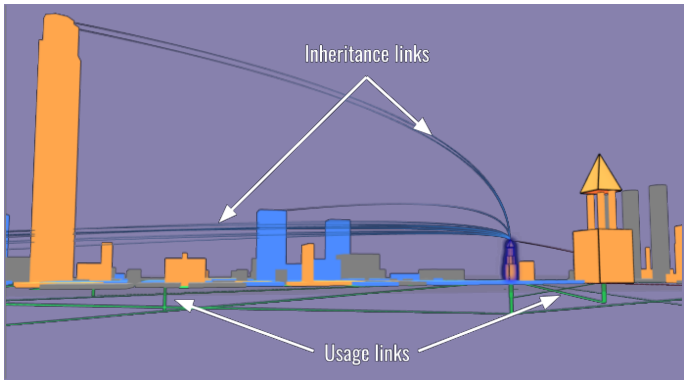
Displaying differently classes being *hotspots* and/or exhibiting design patterns brings to the user insights on highly variable zones of the project, which she can then explore in more detail by using the different interactions provided by the visualization (spanning, zooming).

Streets. Analogously, as the representation proposed by CodeCity groups classes belonging to the same package in a district, our objective is to group in the same neighborhood classes that concentrate a high density of variability implementations.

As stated in Section 3.2, although the nested districts allow to efficiently represent the decomposition hierarchy of classes belonging to nested packages, it is not adapted to our notion of density of variability implementations which derives from usage relationships between classes (as a class can use and/or be used by multiple other classes). We thus rely on the visualization proposed by Evo-Streets [42], which uses streets to decompose a hierarchy instead of boxes. In the original Evo-Streets layout, streets represent subsystems, with orthogonal branching streets representing their subsystems. The buildings on a street represent the modules belonging to this system. We adapt the visualization with buildings on streets being classes, and streets departing from a building (instead of another street) to represent a usage relationship between this class and every other class whose building is on the street. As we consider inheritance links as less important for variability, they are represented as aerial links between buildings, being only displayed when hovering over a building. This enables the user to see the inheritance information if needed, while the hotspot coloring and streets for usage bring the most important information first. A summary of the visual properties is presented in Table 1 and illustrated in Figure 2.



(a) Elements displayed by default



(b) Inheritance links and underground usage links appear when hovering a building

Figure 2: Visual properties of *VariCity*

4.2. Configurable cities

As stated by Requirement 2, configuration options must be provided to focus the visualization around known points of interest of the system. The idea is therefore to allow the users to create a city in line with the most important elements for them and to give a first simplified vision of the city which does not show all the relationships between classes.

Input parameters. The visualization algorithm thus relies on three inputs that focus the view.

Entry point classes. Entry point classes represent important points of interest for the comprehension of the system (*e.g.*, endpoint of an API that could be automatically inferred, or complex classes of the system) from which we start the exploration of the system. Adapting the entry

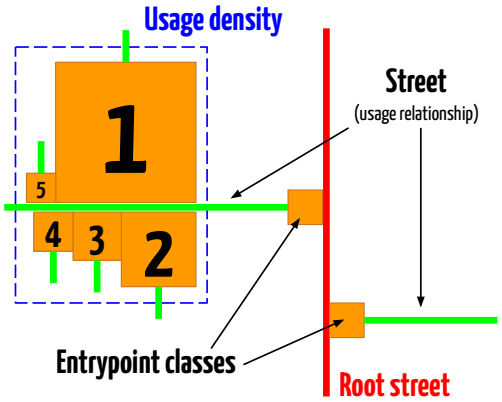


Figure 3: Placement algorithm

point classes allows to delimit the subpart of the system to explore.

Usage orientation. The usage orientation can be set to `IN` and/or `OUT` depending on the objective of the exploration and what the user aims to understand.

- An orientation `IN` means that the classes displayed will be the classes *using* the defined entry points (*i.e.*, having it as an attribute or method parameter). This fits cases where one wants to reuse a part of the implementation as it will show which classes already use the entry point so that a newcomer can see how the class is already used.
- On the opposite, an orientation `OUT` means that the classes displayed will be the classes *being used* by the defined entry points (*i.e.*, being an attribute or method parameter of the entry point). This is particularly adapted in the case one wants to add a new feature as it enables to see which classes are used by the entry points to know which classes one may need to reuse.
- Finally, choosing `IN/OUT` gives an overview of both aspects.

Usage level. The usage level is an integer value. With a usage level of n, all classes distant from an entry point by n usage relationships will be displayed. For example, a visualization set up with an entry point, usage orientation `OUT` and usage level of 2 will display the entry point, the classes being used by the entry point, and the classes used by these classes. Being able to adapt this value is important as depending on the complexity or the layered architecture of a system, a given level of usage might be adapted to it but shows too many classes on another one. Determining the usage level can only be done empirically. A level too low might hide important information for the comprehension of the variability, and a level too high might display too much information.

Shaping the city. The three input parameters are used to shape the city. Figure 3 illustrates the city organization. The root street appears in red and aggregates all the entry points. Then, starting from them, classes using (or being used by) them up to the usage level set are displayed. A street is initiated from an entry point, and for each class related to it, a building is placed on the border on the street. In order to exhibit density between classes, we need to place as close as possible buildings linked by a usage relationship to the same class. Following this principle, we place the buildings by decreasing order of width on both sides of the street, minimizing the total length of the street to keep the buildings as close as possible.

Our placing algorithm can lead to long straight streets if a class uses many others. Work presenting techniques to prevent this behaviour and keep cities compact (such as folding) exist [89]. However, this information is valuable in the case of *VariCity* as it allows to quickly visualize classes concentrating many usage relationships. It is also likely to happen that a class is linked through a usage relationship to multiple visualized classes. In that case, these additional usage relationships are represented as green underground streets and appear only when hovering the class, as well as the inheritance relationships not to overload the visualization[2]. An example of visualization after generation is presented in Figure 2a. Additional links appearing on hover are presented in Figure 2b.

(Re)configuring the view. Since determining values for the three parameters is dependent on every codebase, they can then be adapted to gradually explore the system's variability by modifying the nature and number of displayed relationships and classes. We will illustrate in Section 5 how different values for these inputs impact the structure of the visualization.

Additional options are provided to adapt the visualization, such as visual settings (colors of the visual elements, padding between the buildings) that may improve the readability of the visualization. Metrics for the height and width of the buildings can also be adapted. This parameter may be useful for the expert that has a particularly deep understanding of the system. For example, if the method level variability of classes is due to constructor overloads, it may be useful to use this metric for the height instead of the width of the buildings. Finally, a *blocklist* enables filtering out individual classes or packages considered as irrelevant.

4.3. Extending VariCity *to reveal indebted variability implementations*

As stated by Requirement 3, the visualization must also be extensible, and as first validation of this extensibility, we should be able to create an extended visualization

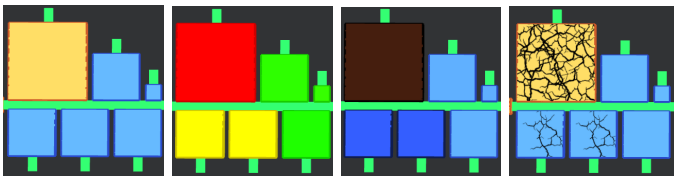
[2]When hovering over, class names are also displayed in a sidebar for the same reason.

to display quality metrics and find indebted implementations of variability.

Wolfart et al. [21] defined variability debt as *"Technical debt caused by defects and sub-optimal solutions in the implementation of variability management in software systems"*. Software quality metrics have been recognized as useful for determining *technical debt* at the code level, and in the domain of OO systems, multiple works focus on determining software quality metrics [90, 91, 92, 73, 93, 94], measuring the system evolution [95, 96], and validating the relevance of these metrics [97, 98]. Multiple tools and approaches also exist to compute metrics on an OO codebase, analyze its quality [99, 100], and determine technical debt [101]. Such metrics are often exploited in visualizations [102, 100], such as CodeCity [41] and Evo-Streets [42] that are now bundled in reference code analysis tools such as SonarQube[3]. 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 one by one.

The *VariMetrics* extension thus increases the capabilities 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. To do so, the *symfinder* toolchain has been extended to support fetching of the quality metrics from the SonarCloud API ([4].) or from a local SonarQube instance. While by default *VariCity* displays in yellow *vp*-s being hotspots, in blue variants being hotspots, and in grey classes not being hotspots (Figure 4a), other city visualizations such as CodeCity and Evo-Streets color the buildings to expose properties inherent to the classes [103, 104]. As we study variability implemented using OO mechanisms, we advocate that OO quality metrics at the class level are relevant to identify OO variability debt. We thus propose to use the walls of the buildings to display quality information in *VariMetrics*. Two coloring strategies are proposed: a coloration following a red-to-green sequence (Figure 4b), and a saturation keeping the original colors of the buildings and lightening or darkening them (Figure 4c). 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 [105]. On the opposite, applying textures on colors has shown to be an efficient way to display multiple software quality metrics [106]. We thus provide a crackled texture (Figure 4d) variably covering the building, 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. It must be noted that we rely in this article on the quality metrics allowing to identify OO variability debt, but the users can configure *VariMetrics* for the metric they want, as long as it is inherent to the class.



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

Figure 4: Visual properties used to display quality metrics compared to the original *VariCity* visualization.

5. Validating the VariCity visualization approach

To evaluate whether the proposed *VariCity* approach is adapted to visualize OO variability implementations, we aim to answer the following research questions:

RQ_1**: Does the *VariCity* view reveal zones concentrating variability implementations?** This *RQ* has for purpose to validate Requirement 1. To answer it, we design *VariCity* views for 10 variable open source systems. We then enumerate revealed zones concentrating variability implementations and manually evaluate their relevance (Section 5.1).

RQ_2**: Are the configuration capacities useful to reveal the zones concentrating variability implementations?** This *RQ* has for purpose to validate Requirement 2. To answer it, we illustrate how we designed the views presented in RQ_1 and show how modifying the parameters allow to shape different cities (Section 5.2).

RQ_3**: Does the *VariMetrics* extension allow revealing indebted zones of variability implementations?** This *RQ* has for purpose to validate Requirement 3. To answer it, we design *VariMetrics* views for 7 variable open source systems. We then enumerate zones concentrating both variability implementations and critical quality (Section 5.3).

5.1. Answering RQ_1

The *symfinder* toolchain, which detects potential *vp*-s with variants, has been applied on ten popular open-source and variability-rich Java systems [38], being applications, frameworks, or libraries, with different characteristics (size, variation points, explicit API provided). We chose to select the same systems to test the results

[3] `https://www.sonarqube.org/`
[4] `https://sonarcloud.io/web_api`

Table 2: Subject systems, LoCs obtained with the `cloc` [107] tool

System	Analysed LoC	# *vp*-s/variants	# zones		View configuration		
			revealed	relevant	# entry point(s)	Usage level	Usage orientation
Java AWT	67,229	2,501	7	7	1	3	IN/OUT
Apache CXF	656,472	11,028	5	4	1	6	OUT
JUnit	7,717	354	4	4	2	6	IN/OUT
Apache Maven	80,839	1,759	5	5	3	7	OUT
JFreeChart	94,384	2,849	10	7	2	4	OUT
ArgoUML	134,359	2,735	6	6	3	2	IN/OUT
Cucumber	42,662	520	3	1	1	9	IN/OUT
Logbook	225,125	258	6	5	2	4	OUT
Riptide	12,626	320	4	2	1	6	IN/OUT
NetBeans	1,284,416	10,357	8	7	1	5	IN/OUT

of *VariCity*. Table 2 lists the systems and the determined *VariCity* configuration to facilitate the exploration or deepening of a particular area. For each generated visualization, we manually identify the revealed zones concentrating variability implementations. As detailed in Section 4.1, such zones are characterized by buildings of large dimensions (representing density of variability implementations in a class), specific buildings for variability-related design patterns, long streets (representing usage between such classes), and hotspot classes appearing in color. Then, we manually examine the involved classes and, relying on comments and documentation, determine whether they represent actual variability. We hereafter illustrate this procedure on the largest studied systems, NetBeans, from which more than 1.2 MLoC were analyzed.

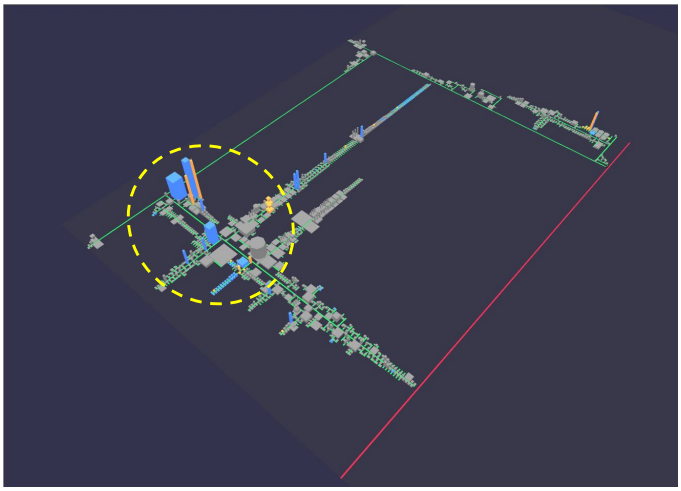
We configure the visualization to use the endpoint of the API, namely `JavaPlatform`[5], as the entry point of the visualization. To have a first overview of the classes being closely related to the endpoint of the API, both classes using and being used by `JavaPlatform` on 5 levels (usage level 5, orientation `IN` and `OUT`) are configured to be shown. The obtained visualization is shown in Figure 5a.

A neighborhood of tall and colored buildings (circled in yellow) detaches from the other buildings in the city, revealing zones with classes heavily using variability implementation techniques. They thus represent a revealed zone of the visualization, which one can consider being variability patterns. By zooming and spanning the visualization, we can focus on this precise part of the city (Figure 5b). The different implemented design patterns are distinguishable thanks to the special shape of their buildings (*e.g.*, `JavaFix`[6] is a Strategy, `testng.AbstractTestGenerator`[7] and `junit.AbstractTestGenerator`[8] are Templates). The two last classes are not only design patterns but also hotspots, giving a strong intuition about the relevance of the potential identified *vp*. In fact, these classes allow to generate test code for two different unit test libraries, JUnit[9] and TestNG[10] and are variants of the `CancellableTask` interface[11]. This zone is thus considered as relevant as it represents actual variability.

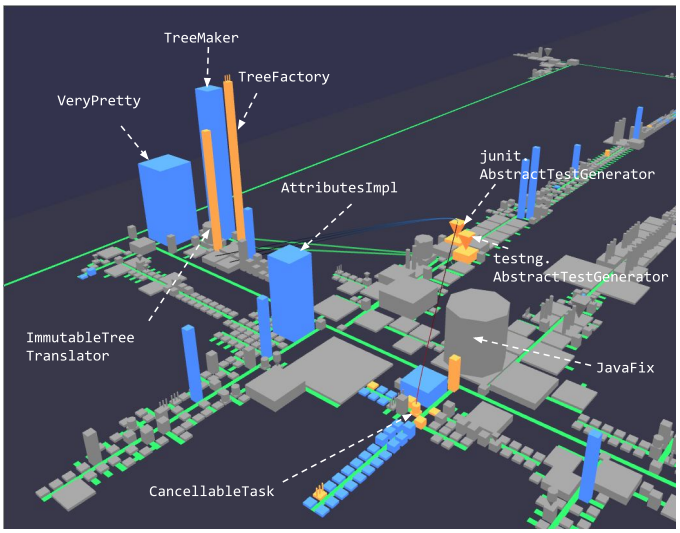
We notice that, as expected, the number of revealed zones fluctuates between the systems. This can be explained not only by the fact that being implemented using OO mechanisms, the structure of the variability is strictly related to the OO structure of the system. Consequently, as we used for all systems the same threshold values to identify hotspot classes (*i.e.*, ≥ 20 variants per *vp*, ≤ 5 usage relationships of distance, *cf.* Definition 5), smaller systems like Logbook or Riptide show little to no hotspot classes. There is thus a need to configure these thresholds for each project [38]. Nevertheless, the obtained results show that revealed zones dense in variability implementations actually correspond to implemented variability. This is however not the case for Cucumber, a BDD testing library, for which 3/4 zones do not represent actual variability (Figure 6). The ① class is `io.cucumber.core.runner.PickleStepTestStep` and is a variant providing the default implementation of the `io.cucumber.plugin.event.PickleStepTestStep` interface. The ② class is `InvalidMethodSignatureException` and is identified as a *vp* as it has 3 subclasses. However, it does not implement variability and is only used as a way to aggregate behaviour between the variants. Finally, the long street initiating the ③ group has for origin the `CachingGlue` class that uses multiple *vp*-s to aggregate information in a single representation allowing to cache them and speed up testing, and therefore does not represent variability. It is important to notice that as illustrated with this example, zones are not only revealed by being identified as hotspots by the underlying identification technique, but also by the organization of the city grouping together classes using each other.

Answer to RQ_1. As a result, for all the considered systems, which have also been studied with the extrac-

[5] `org.netbeans.api.java.platform.JavaPlatform`
[6] `org.netbeans.spi.java.hints.JavaFix`
[7] `org.netbeans.modules.testng.AbstractTestGenerator`
[8] `org.netbeans.modules.junit.AbstractTestGenerator`
[9] `https://junit.org/junit5/`
[10] `https://testng.org/doc/`
[11] See here and here.



(a) `java` package of NetBeans 12.2, usage level 5, orientation `IN/OUT`, `JavaPlatform` as entry point.



(b) Zoom on a hotspot zone

Figure 5: Visual identification of revealed zones dense in variablity implementations

tion tool *symfinder* [38], the *VariCity* visualization reveals the same zones concentrating variability implementations. Such zones mainly represent actual variability for a majority of the systems, thus providing a positive answer to RQ_1. The reader can find in the reproduction package (after the conclusion), annotated views excerpts that detail, for each system, the revealed and relevant zones.

5.2. Answering RQ_2

While the previous section illustrates that the visualization can exhibit relevant dense zones of variability implementations, it requires as a first step to determine adequate values for the parameters shaping the view (*i.e.*, entry point classes, usage orientation and usage level). Hereafter, we illustrate how adapting these parameters allow shaping the view by detailing how we obtained the view for one of the projects listed in Table 2, JFreeChart.

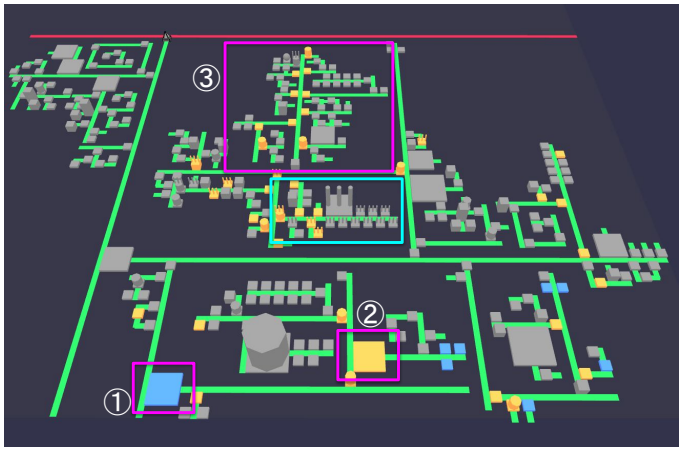


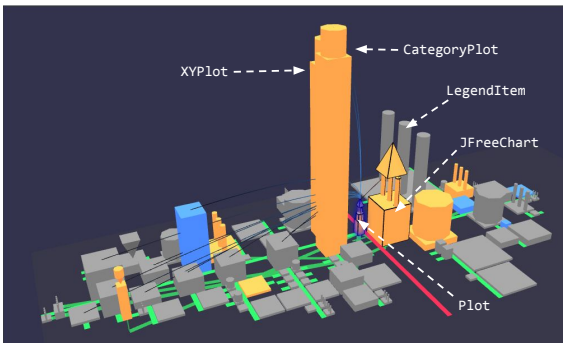
Figure 6: Revealed zones in the generated visualization for Cucumber. Irrelevant zones are delimited by violet boxes and the relevant zone by a blue box.

JFreeChart is a library written in Java allowing to draw different types of charts. As this system is a library, the endpoint it provides enabling its reuse represents a relevant entry point to start its exploration. We therefore set as entry point `JFreeChart`[12], being the endpoint of the library used by the users to create plots. We also set as entry point `Plot`[13], the superclass of all classes implementing a different type of chart. To visualize the classes of the system starting by the library's endpoint, the usage orientation is set to `OUT`. Finally, the usage level is set to 2 to evaluate a first visualization with a small set of classes, shown in Figure 7a.

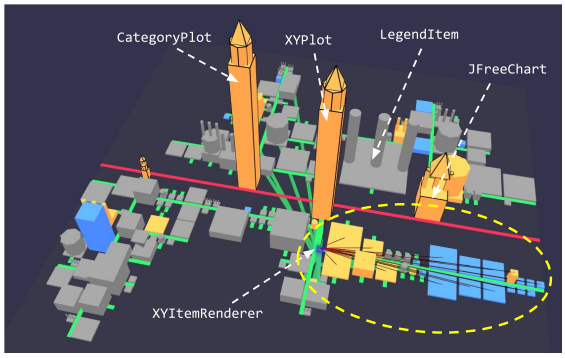
By hovering over `Plot`, we can see the different displayed subclasses of the class (*i.e.*, the variants of the *vp* `Plot`). Two classes, `XYPlot`[14] and `CategoryPlot`[15], are noticeable due to their important height showing an important number of method overloads. Besides, they are both design patterns. As the visualization shows few zones concentrating variability implementations, we add `XYPlot` and `CategoryPlot` as entry points (Figure 7b). The shape of the city changes to display the usages related to each entry point in separated neighborhoods, allowing to better visualize if (*i*) a particular entry point is the starting point of a dense zone of variability implementations, and (*ii*) a class is related, to a certain degree, to two entry points with underground streets. On Figure 7b, an important number of classes making heavy use of variability implementations is visible (circled in yellow), and are directly used by `XYItemRenderer`[16], itself related to both `XYPlot` and classes related to `CategoryPlot`.

While the addition of entry points allows displaying more classes related to these new entry points, increasing the usage level can be used to broaden the view by

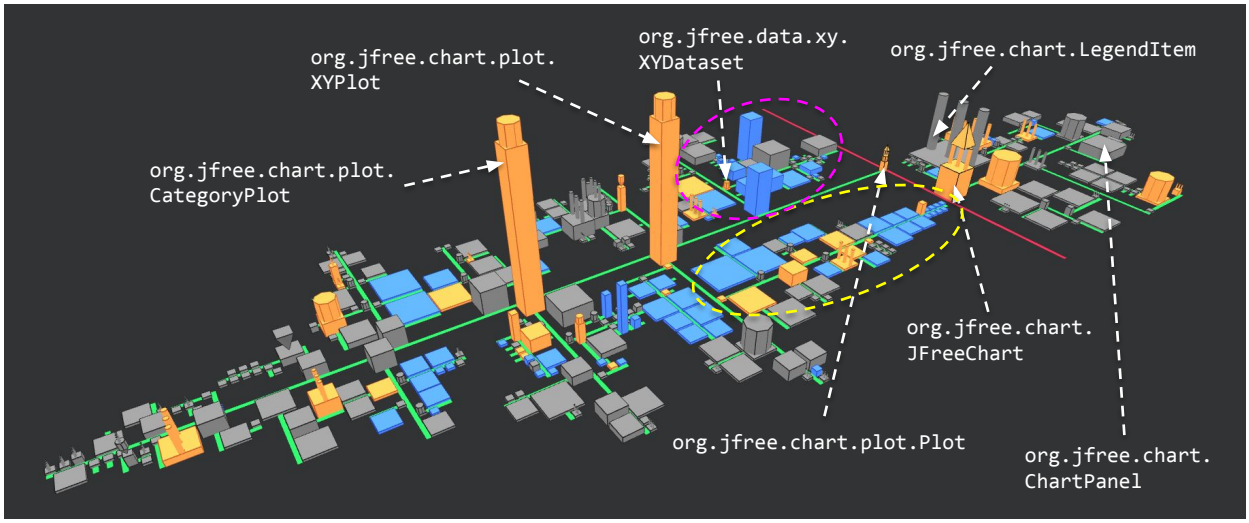
[12] `org.jfree.chart.JFreeChart`
[13] `org.jfree.chart.plot.Plot`
[14] `org.jfree.chart.plot.XYPlot`
[15] `org.jfree.chart.plot.CategoryPlot`
[16] `org.jfree.chart.renderer.xy.XYItemRenderer`



(a) JFreeChart, usage level 2, orientation OUT, `JFreeChart` and `Plot` as entry points. Displaying links of `Plot` reveals that `XYPlot` and `CategoryPlot` are subclasses.



(b) Figure 7a after adding `XYPlot` and `CategoryPlot` as entry points



(c) Figure 7a after increasing the usage level to 4

Figure 7: Designing JFreeChart's visualization

visualizing classes being used and/or using each of the already displayed classes, as shown on Figure 7c. We notice that although the visualization is more furnished, the noticeable zone issuing from `XYItemRenderer` is still visible. Additionally, increasing the usage level led to displaying `XYDataset`[17] and the multiple classes it uses (circled in violet), creating a zone of interest. Therefore, while adding entry points can be used to focus on some particular classes, increasing the usage level is particularly useful to explore the system without any particular focus.

We notice in Table 2 that the values allowing to design the views differ between the projects. For example, while a usage level of 4 for JFreeChart leads to a visualization revealing 10 zones (Figure 7c), a usage level of 11 for Cucumber reveals 3 zones (Figure 6). This difference can be explained not only by the fact that JFreeChart is larger than Cucumber, but also by the fact that their organizations are different. Although they are both libraries, JFreeChart provides a main endpoint for reuse, `JFreeChart`, while Cucumber exposes a much larger number of classes to reuse, being a BDD testing library.

Answer to RQ_2. It results that adapting the view's parameters allow to tailor it to display different sets of classes. Such configuration parameters are therefore essential to design views focusing on a restrained number of classes regardless of a system's architecture, thus providing a positive answer to RQ_2.

5.3. Answering RQ_3

To answer RQ_3 and evaluate whether the extended *VariMetrics* visualization enables visualizing indebted zones of variability implementations, 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.

5.3.1. Relevant quality metrics

In the work of Wolfart et al. [21] on the concept of *variability debt*, the authors introduce a catalog of ten forms

[17] `org.jfree.data.xy.XYDataset`

of variability debt resulting from an analysis of 52 industrial case studies reporting technical debt issues on variable software systems. We therefore base ourselves on this work to determine relevant occurrences of variability debt in our context.

As OO variability implementations rely solely on the standard object mechanisms, the availability of the source code is the only requirement to identify them. From the possible causes of variability debt, it is then possible to find *code duplication* and *system-level structure quality issues*. Most often, tests sources are provided along with the source code, enabling also the 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*, *duplicate documentation*, *old technology in use* and *multi-version support*), test cases definitions (leaving aside *expensive tests*), design choices (leaving aside *architectural anti-patterns*) or a list of features and their mapping with their implementations (leaving aside *poor test of feature interactions*).

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. There is therefore a need, for each of these types of variability debt, to determine quality metrics allowing their identification. We choose code block duplication to measure *Code duplication* as we believe that line duplications could lead to multiple duplications not related to variability. We choose the test coverage to measure *Lack of tests*. As test coverage can be measured at multiple granularities (line, condition, ...), we selected a metric aggregating measures for different granularities. Finally, structure quality issues in the codebase impact maintainability and evolution of the system and hamper the system's comprehension. Therefore, we advocate that assets suffering from this type of variability debt are hardly understandable, and cognitive complexity [93] appears to be relevant for this purpose [108]. We therefore define test coverage, duplicated blocks and cognitive complexity as relevant metrics for this identification.

5.3.2. Subject systems

Appropriate subject systems for this evaluation must (*i*) be variability-rich and (*ii*) provide OO quality metrics relevant to identify variability debt (Section 5.3.1). Although all the subject systems listed in Table 2 match the first criterion, none of them provides quality metrics. While we could adapt JFreeChart's build configuration to be analyzed by a local SonarQube instance [109], we could not achieve to do so for the other systems. We therefore chose 6 other systems for which the quality metrics are available on SonarCloud[18], allowing us to reuse these metrics for our study. Five of them (**Azureus**, **GeoTools**, **JKube**, **OpenAPI Generator** and **Spring framework**) were chosen as their documentation clearly states they implement variability. We also picked the **Java Development Kit (JDK)** for its large size of circa 2.5M LoC to evaluate the scalability of the approach. Metrics for Azureus, GeoTools, Spring framework and Java Development Kit (JDK) have been extracted from a catalog of software projects designed by Irrazábal et al. [110] to analyze their metrics, forking popular open source systems from their original repositories in the Corpus-2021 GitHub organization[19]. All seven systems are depicted in Table 3.

Table 3: Subject systems and their available metrics.

Project	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)

5.3.3. Evaluation process

For each project we first generated a *VariCity* visualization with the same process as described in Section 5.2. We then identified manually on each view the classes that are the most visible for us as described in Section 5.1 to obtain a set of "noticeable classes *w.r.t.* variability". For example, for JFreeChart (Figure 7c), classes such as `JFreeChart`, `Plot`, `CategoryPlot`, and `XYPlot` draw attention due to their size and/or the fact that they are hotspots, as opposed to `ChartPanel`.

We then determine a relevant *VariMetrics* view on each project by systematically applying all available metrics that are related to variability debt (Section 5.3.1). During this step, it happened that no building stood out for a given 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 3 summarizes for each system the relevant metrics being available and significant. While this step was necessary for us to determine which quality metrics are significant, an expert will likely already know which metrics are significant for their system. We then manually identified on the

[18] https://sonarcloud.io

[19] `https://github.com/Corpus-2021`

Table 4: 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

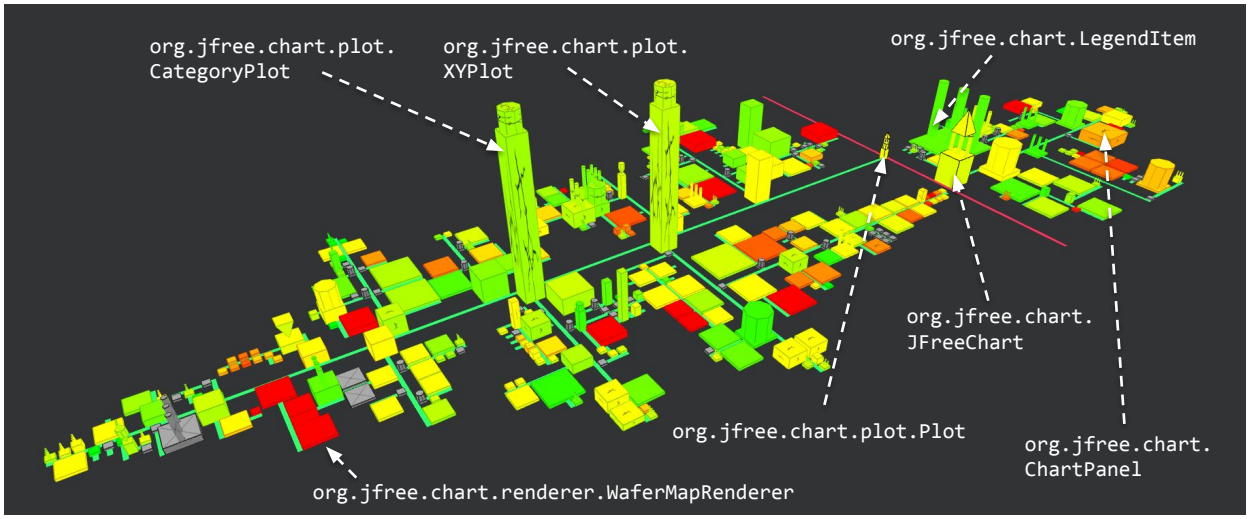


Figure 8: Figure 7c in *VariMetrics*. The view is configured to display the test coverage using the red-to-green color scale and the duplicated blocks using cracks.

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 JFreeChart (Figure 8), `XYPlot`, `CategoryPlot`, `ChartPanel`, and `WaferMapRenderer` 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, `CategoryPlot` and `XYPlot`).

We also observed that, in all systems, while fewer classes are noticeable *w.r.t.* criticality than *w.r.t.* variability, there is no direct relation between variability and quality, as also shown in Figure 8. Consequently, some *vp*-s have an important number of variants and are at the same time reliable, such as `LegendItem` in JFreeChart. On the contray, one can observe some critical classes that do not contain variability implementations, such as `WaferMapRenderer` in JFreeChart. 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 4. 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 is 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 of the scope of this article. The definition of a hotspot is parameterized,

and determining whether a class is a hotspot depends on user-defined thresholds, as stated in [38] 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.

Answer to RQ_3. While the *VariCity* view highlights classes concentrating variability, *VariMetrics* is able, by displaying quality metrics on the buildings, to reveal such classes being quality-critical. Additional configuration parameters allow to configure the metrics, making the view applicable to multiple systems. These results thus provide a positive answer to RQ_3.

5.3.4. Summary

By adapting the city metaphor to organize the city relying on OO variability metrics, the *VariCity* visualization reveals zones concentrating variability implementations. Such zones have shown to be largely relevant on the studied subjects systems. Its configuration capabilities not only allow to design views for multiple systems, but also to explore a system by focusing the view on a subset of classes the user finds relevant. Finally, 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.

6. Evaluating the comprehensibility of the *VariCity* visualization

In Section 5.1, we demonstrated the capacity of *VariCity* to reveal dense zones of variability implementations. However, as the authors of the approach designed the showcased views, the comprehensibility of the visualization by actual users has not been evaluated. To fully assess whether *VariCity* actually helps the comprehension of the implemented OO variability, there is a need to complete this first assessment by evaluating *VariCity* in a real variability comprehension scenario. We thus design a controlled experiment with external users to observe how using *VariCity* impacts the time needed to complete variability comprehension tasks and their difficulty.

6.1. Experimental design

Wettel et al. [88, 87] designed an empirical evaluation of CodeCity aiming to evaluate whether the view helped the identification of quality-critical zones in an OO codebase. They extracted from the literature a wish list of requirements for their experiment. As we conduct a similar evaluation, we therefore rely on this list to design our experiment and detail its design in the remainder of this section. Table 5 summarizes, for each of these requirements, how our design fulfills them or not.

6.1.1. Research Questions

With this experiment, we aim to evaluate the comprehensibility of the *VariCity* visualization. Chen et al. [111] extracted multiple definitions of software visualization from the state-of-the-art and it results that they are mainly described as (i) making the information they represent easier to understand, having for goal to (ii) save time analyzing it. These two dimensions therefore appear as useful to evaluate the comprehensibility of our approach. Additionally, since *VariCity* is presented as a tool to be used by developers in their workflow, we argue that the ease of use is also of great significance. Consequently, we aim with this experiment to answer the following question:

RQ_4: Does *VariCity* help the comprehension of OO variability implementations?

We decompose this research question as follows:

$RQ_{4.1}$: Does the use of *VariCity* increase the correctness of the solutions to variability identification tasks, compared to state-of-the-practice tools?

$RQ_{4.2}$: Does the use of *VariCity* reduce the time needed to solve variability identification tasks, compared to state-of-the-practice tools?

$RQ_{4.3}$: Is *VariCity* regarded as easy to use to solve variability identification tasks compared to state-of-the-practice tools?

The null and alternative hypotheses derived from these research questions are described in Table 6.

6.1.2. Subjects

This experiment was realized as part of a reverse engineering graduate course at the Polytech Nice Sophia engineering school. The population is made of 49 students in the last year of Master's in Computer Science, specialized in Software Architecture. As a preliminary part of the experiment, they were asked to fill an anonymous survey to better know their level of experience on program comprehension and variability. Some gathered information is detailed in Figure 9.

While it is known that having students as subjects for controlled experiments does not always give reliable results as they might not be representative of the target population [112], we think that they are representative of a subset of developers for the two following reasons. First, 38/49 subjects have more than 6 months of professional experience, mainly thanks to internships and apprenticeships in industry. Moreover, 43/49 students already had to discover a system by exploring its codebase. Additionally, being in the last year of Master's in Computer Science the remaining subjects will integrate an industrial company in the next few months and need to onboard on an unknown codebase. They thus exactly match the usage scenario of *VariCity*. Second, as a part of their curriculum, they

Table 5: Elements from the experimental design responding to requirements extracted from Wettel et al. [87]'s wish list.

Requirement	Experimental design element
Fulfilled requirements	
Avoid comparing using a technique against not using it. Provide the same data to all participants.	As we validate the visualization approach, we give a CSV document opened with a spreadsheet containing structured information on the classes of the system visualized with the given settings using *VariCity* by the other group (Sections 6.1.3 and 6.1.5).
Provide a not-so-short tutorial of the experimental tool to the participants. Use the tutorial to cover both the research behind the approach and the implementation.	To complement the 1h30 lecture given two weeks prior to the experiment, a short tutorial introducing definitions and demonstrating the tool has been given before the experiment (Section 6.1.7).
Find a set of relevant tasks. Include tasks on which the expected result is not always to the advantage of the tool being evaluated.	The tasks are inspired by the onboarding scenarios in the first evaluation of *VariCity*, and some of them are expected to be more easily completed using the IDE and the CSV document (Section 6.1.6)
Choose real object systems that are relevant for the tasks.	JFreeChart has been chosen for its medium size and because the domain and the implementation are accessible to the subject students (Section 6.1.3)
Provide all the details needed to make the experiment replicable.	The questionnaires, slides and answers given by the students are available online at https://deathstar3.github.io/varicity-demo/
Report results on individual tasks.	Additionally to the answers given, subjects were asked for each task to provide the start and end time, an estimation of their perceived difficulty and the list of the actions they accomplished to solve the task (Section 6.1.6).
Non-fulfilled requirements	
Include more than one subject system in the experimental design.	Having a second subject system would have led to four treatments of 12 people each and would have prevented drawing any relevant conclusion.
Involve participants from industry. Take into account the possible wide range of experience level of the participants.	This requirement could not fulfilled due to organizational constraints. Although apprentice students have more professional experience, the difference of professional experience with the other students is not important enough to conclude on whether the performance of subjects differs *w.r.t.* this parameter (Section 6.1.2).
Avoid, whenever possible, to give the tutorial right before the test.	Although we gave a long lecture introducing variability concepts about one month before the experiment, we could not give the more detailed tutorial before the day of the experiment for organizational constraints (Section 6.1.7)
Limit the amount of time allowed for solving each task.	While the overall set of tasks should be completed in 1h10, we did not limit the time for each task to prevent fast but less qualitative answers (Section 6.1.6).

Table 6: Null and alternative hypotheses

Null hypotheses	Alternative hypotheses
H_{1_0}: *VariCity* does not impact the correctness of the tasks' solutions.	H_{1_a}: *VariCity* impacts the correctness of the tasks' solutions.
H_{2_0}: *VariCity* does not impact the time spent to solve the tasks.	H_{2_a}: *VariCity* impacts the time spent to solve the tasks.
H_{3_0}: *VariCity* does not impact the tasks' difficulty.	H_{3_a}: *VariCity* impacts the tasks' difficulty.

also followed multiple courses prior to the experiment related to the comprehension of complex code architectures, thus preventing a bias on their knowledge of these aspects. This Master's trains them to be advanced developers in Java, thus mastering object-oriented programming concepts. They also followed multiple courses prior to the experiment related to the comprehension of complex code architectures, thus preventing a bias on their knowledge of these aspects.

6.1.3. Purpose and variables

Through the three defined research questions, the goal of this experiment has been set towards evaluating whether *VariCity* allows subjects to better identify patterns involved in complex zones of variability implementations (*i.e.*, the *effectiveness* of the approach). Additionally, we aim to assess whether *VariCity* reduces the time needed for subjects to answer the tasks and their perceived difficulty compared to state-of-the-practice tools, *i.e.*, the *efficiency* of the approach. Such goals being identical in the empirical evaluation of CodeCity by Wettel et al. [87], we therefore share identical dependent and independent variables. We detail them hereafter.

Independent variables. Our first independent variable concerns **the tool** used to solve the task. In order to mitigate the effect of this variable, we must compare our approach with a state-of-the-practice approach used to achieve an identical goal, that is, understanding the variability implemented in OO software systems. While comparing *VariCity* to *symfinder-2* would allow evaluating the potential gain brought by the city metaphor, we cannot consider it a state-of-the-art approach as it is not used regularly by the subjects. Therefore, the comparison would be irrelevant as between two approaches that subjects do not master. Since, to the extent of our knowledge, no similar and commonly used approach exists, we build a baseline ourselves relying on tools that developers would actually use to navigate and understand the code artifacts. IDEs are widely used tools for program comprehension [113]. While a majority of our subject students use the IntelliJ

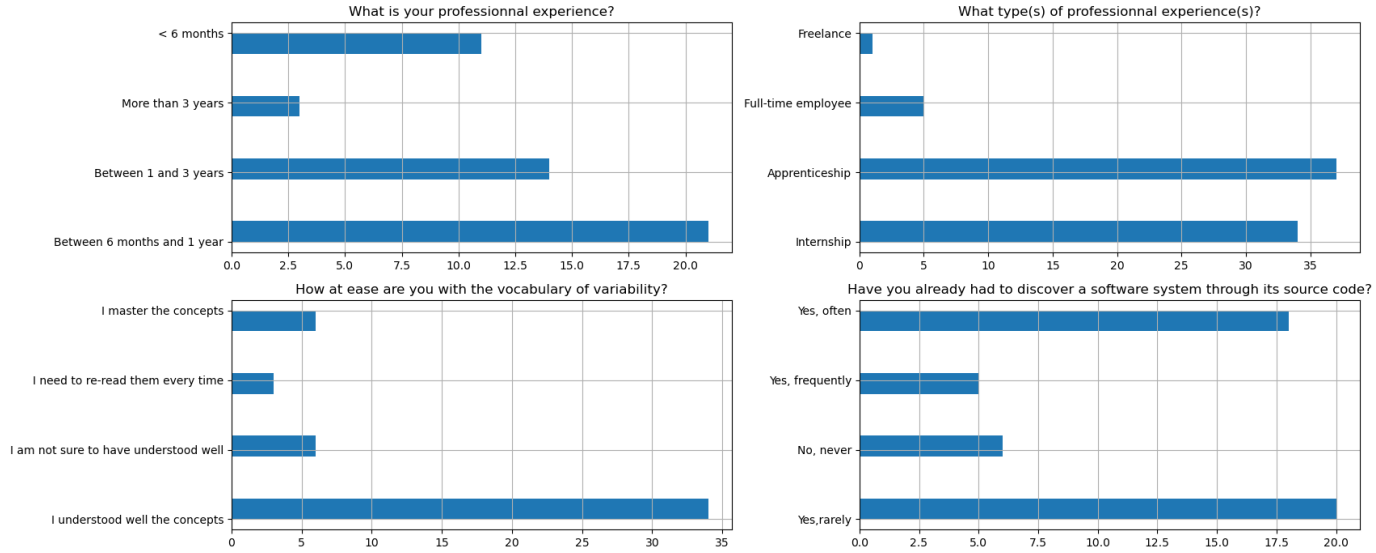


Figure 9: Information on the experience of the controlled experiment's participants

IDEA IDE, we did not impose any particular IDE as (*i*) the given tasks (listed in Section 6.1.6) can be answered using only basic features supported by a large majority of IDEs (as finding usages, code navigation), thus we do not expect them to use advanced features that would be specific to a specific IDE (*e.g.*, tracing, dynamic analysis) and (*ii*) we limit the bias regarding the mastering of the IDE as every subject can use the one they master the most.

VariCity however uses data and metrics that are previously computed by the identification backend (*cf.* Figure 1). Since our goal is to compare the gain of *VariCity* compared to the use of an IDE, we should provide the subjects with all information given by *VariCity* that cannot be determined using the IDE's features. The inheritance and usage relationships between classes being standard navigation features, it is thus possible to infer the variants at class level, and to determine hotspot classes and design patterns, whose definitions are given to the subjects. It results that the only missing information is the number of overloads of methods and constructors. Although such metrics could be provided by static analysis tools, Nachtigall et al. [114] recently studied how 46 such tools fulfill 36 usability criteria from the literature, revealing that they are majoritarily limited regarding their usability and capacity to limit false positives. We therefore collected this information in a CSV file to complete the baseline, thus ensuring their correctness and mitigating the risk that subjects will lose time manipulating additional tools. The structure of the file is given in Table 7. As for the IDE, no restriction has been imposed on a particular spreadsheet to manipulate the CSV file for similar reasons. Finally, we do not provide any documentation on the implemented features as (*i*) they are little documented in practice [35] and (*ii*) they are not used by *VariCity*.

Our second independent variable regards **the studied object system** and its architecture. While a large system or with multiple layers of abstraction would require too much time to be understood in such an experiment, a too small system would on the opposite not require an approach as *VariCity* to help its understanding and therefore not allow evaluating its potential gain. For these reasons, we selected JFreeChart 1.5.0 as an object system, whose characteristics are presented in Table 8. Not only does its 95k LoC make it a system of medium size, being a charting library that we studied to evaluate *symfinder* and *symfinder-2*, we know that both the domain and the implementation are accessible to the subject students. For similar reasons, we selected ArgoUML as a test project on which the subjects can familiarize themselves before the actual experiment on JFreeChart (*cf.* Section 6.1.7). Given the size of the population, we decided not to experiment on a second object system as the groups for each treatment would have been too small (around 12 subjects) to draw any conclusion (*cf.* Section 6.1.5).

Table 7: Structure of the given CSV containing data on the classes

Class name	**Method variants**	**Constructor variants**
`org.jfree.chart.ChartPanel`	6	5
`org.jfree.chart.ChartRenderingInfo`	0	2
`org.jfree.chart.JFreeChart`	17	3
`org.jfree.chart.LegendItem`	0	10
...	...	...

Table 8: Detailed statistics on the object system used for the experiment, JFreeChart

#LoCs	**#classes**	**#*vp*-s**			**#variants**		
		class level	method level	total	class level	method level	total
94,384	990	259	667	926	275	1,648	1,923

Dependent variables. Our dependent variables regard the **correctness** of the solution given for a task and the **time** to complete the task, which respectively allow measuring the *effectiveness* and the *efficiency* of our approach.

6.1.4. Controlled variables

In their experiment, Wettel et al. [87] benefited from a large panel of subjects from academia (ranging from bachelor students to professors) and industry. Therefore, subjects in this panel exhibited large differences in terms of background of experience, potentially having an influence on their capacity to complete the tasks. In our case, all our subjects are students having studied similar topics. We thus consider as negligible the impact that their different professional experiences could have on their capacity to solve the tasks and therefore do not consider these variables for our experiment.

6.1.5. Treatments

Due to the homogeneity of profiles constituting our population (*cf.* Sections 6.1.2 and 6.1.4), we split the subjects in two groups by relying on a *completely randomized design*[20]:

***VariCity* (24 subjects)**. The first group is given a link to a GitHub repository containing:

- the result of JFreeChart's and ArgoUML's analysis by *symfinder-2*, used as input by *VariCity*;
- a *VariCity* configuration file to display the views.

The *VariCity* image is distributed as a Docker image hosted on the Docker Hub, thus requiring no installation on the students' computers.

IDE + CSV (25 subjects). The second group is given a link to a ZIP file containing:

- the source code of JFreeChart and ArgoUML;
- the CSV file containing the variability metrics for the classes[21].

6.1.6. Tasks

The 11 tasks are listed in Table 9 and derive from our requirements (Section 3.1). The expected answers for each task are listed in Table A.12. The subjects have 1h10 to complete all the tasks.

For each task, the subjects are also asked to:

- **input the start and end time** of the task. With this information, we aim to evaluate whether the time spent completing a task differs when using *VariCity* or the IDE.
- **rate the difficulty of the task** on a scale from 1 to 4. Likert scales are commonly represented with an odd number of choices, using the median value as a midpoint (*i.e.*, a neutral level of opinion). However, in practice, subjects might choose the midpoint for other reasons [115], for example because they are unfamiliar with the topic to be evaluated [116] or because they prefer to avoid providing a negative opinion [117], especially on a five-points scale [118]. As we believed our subjects could be in such cases, we preferred to omit the midpoint and defined a scale from 1 to 4. With this information, we aim to evaluate whether the perceived difficulty for a task differs when using *VariCity* or the IDE.
- **list the actions they accomplished** (*e.g.*, navigating the inheritance in the IDE or zooming on the visualization). We plan to use this data to better understand how the tools were used to solve the task and better understand the causes of the results obtained with the two previous pieces of information.

6.1.7. Operation protocol

Pilot experiment. A pilot experiment with four subjects of various levels of experience (a Master's student in computer science, two graduate students in computer science working on research topics related to variability and an associate professor in computer science having experience with controlled experiments) was held to evaluate the whole experimental setup. This dry run allowed us to refine some aspects of the experiment such as the time limit. As defining a time limit for each task can cause subjects to go through the tasks faster [119], we did not set any in our pilot experiment. However, we realized that since variability comprehension is complex by nature [35], our test subjects tended to get stuck as they were unsure about their answers. We thus decided to impose a global time limit to prevent this behaviour, while leaving the subjects manage the time they spend on each task.

Before the experiment. A lecture of one hour and a half introducing the main concepts related to variability was given to the students from both groups on January 12, 2022. As a result, a majority of students felt confident about their knowledge of variability concepts and terminologies at the moment of the experiment (*cf.* Figure 9).

The day of the experiment (February 23, 2022). On the day of the experiment, a short lecture was given to all the students, presenting definitions and examples of the various terminologies used in the tasks (about 45 mins). Then, after splitting, each group benefited from a short session to fill the preliminary questionnaire aiming to gather personal information (*cf.* Section 6.1.2), setup their environment and familiarize with the tools they will manipulate (about 40 mins):

- subjects in the *VariCity* group were introduced to the visualization's features, had to open the visualization for the test project (ArgoUML), and had to

[20] https://www.itl.nist.gov/div898/handbook/pri/section3/pri331.htm

[21] Although configuring the view might add or remove classes on the visualization, the given tasks do not require this action. Therefore, the given data is strictly identical between both groups.

Table 9: The 11 variability comprehension tasks given to the subjects.

ID	Task	Goal
1	When discovering the visualization / the code in the IDE and the data in the spreadsheet, according to you, what classes seem to be important to explore in priority? For each one of them, explain why.	With this task, we want to evaluate whether the used tools provide information that guide the user towards a starting point of exploration.
2	Identify 2 variants at class level for each of the following variation points (*cf.* Definition 2): • `org.jfree.chart.plot.Plot` • `org.jfree.chart.title.TextTitle`	Inheritance is a heavily-used mechanism to implement OO variability (Section 2.1), therefore their identification and exploration is crucial to identify variability in this context. With this task, we want to evaluate whether the used tools allow the exploration of such mechanisms.
3	How many classes are linked with a usage relationship to each of the following classes? Give 3 examples. • `org.jfree.chart.plot.CategoryPlot` • `org.jfree.chart.title.CompositeTitle`	The collective density of variability implementations is characterized by a cluster of *vp*-s linked through usage relationships (Definition 4). With this task, we want to evaluate whether the used tools allow an overview of these mechanisms by distinguishing the classes linked by usage relationships to a given one.
4	Complete the following sentences: • Classes (1) and (2) have an important number (≥ 5) of subclasses (*i.e.*, are variation points with an important number of variants at class level). • Classes (3) and (4) have an important number (≥ 10) of overloaded methods and constructors (*i.e.*, are variation points with an important number of variants at method level).	While usage relationships induce collective density, the individual density of variability implementations is characterized by the presence of a *vp* with an important number of variants at class or method level (Definition 3). With this task, we want to evaluate whether the used tools allow an overview of where such mechanisms are concentrated in the codebase.
5	Identify the 3 classes with the highest individual density higher to the threshold $v = 20$ (*cf.* Definition 3).	For a given density threshold, the number of classes characterized as dense can remain important depending on the dimensions and architecture of the studied system. It is therefore important to be able to focus on the most dense classes. With this task, we want to evaluate whether the used tools allow this.
6	Give 2 examples of each of the following design patterns (*cf.* Appendix B.2): • Strategy pattern; • Factory pattern.	Those two design patterns being used to implement OO variability (Section 2.1), we want to evaluate with this task whether the used tools allow their identification.
7	What is the distance between the `org.jfree.chart.JFreeChart` and `org.jfree.chart.title.DateTitle` classes?	As the density of OO variability implementations relies on usage relationships between classes, we aim to evaluate whether state-of-the-practice tools allow a user to compute the distance between two given classes.
8	Identify 3 hotspots for an individual density threshold of $v = 20$ and a collective density threshold of $d = 5$ (*cf.* Definition 5).	The most dense zones concentrating variability implementations are characterized as *vp*-s being simultaneously individually and collectively dense (*cf.* Definition 5) and it is therefore important to identify them. With this task, we want to evaluate whether the used tools allow to identify them.
9	Identify the classes that according to you implement each of the following features, and specify if they are hotspots for $v = 20$ and $d = 5$ (*cf.* Definition 5): • "draw a chart" feature; • "title of the chart" feature.	Variability identification activities being often conducted to help the comprehension of this variability [35], we aim with this task to evaluate whether the exploration of the system with the given tools allowed the subjects to determine the classes involved in the implementation of a feature.
10	What are according to you the classes to add/reuse/modify to implement a new type of chart (*i.e.*, to implement a new variant of the `org.jfree.chart.plot.Plot` *vp*) ?	This task aims at evaluating whether completing given the variability identification tasks actually allowed the subjects to gather enough knowledge on the implemented domain to determine how to implement a new feature.
11	Following this observation of JFreeChart, what are according to you the main abstractions used in this charting library?	With this task, we aim at evaluating whether the exploration of the system with the given tools allowed the subjects to better understand the domain implemented in the system.

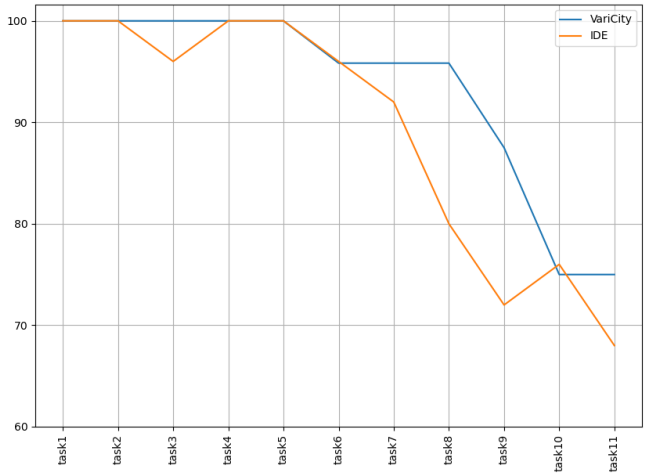


Figure 10: Percentage of subjects having given at least a partial answer for each task

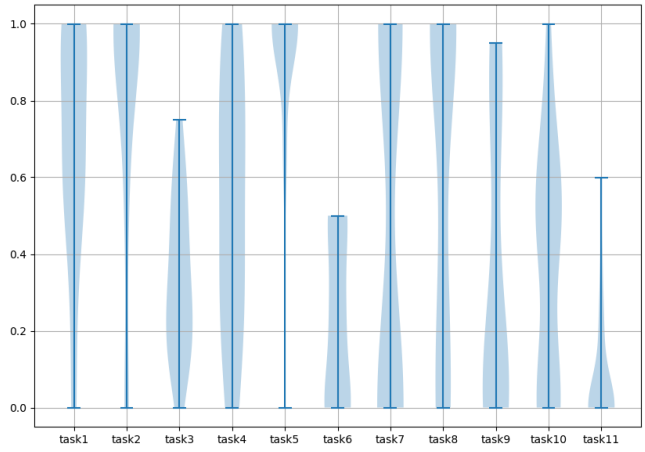
answer a few questions ensuring that they can interpret the visualization accurately;

- subjects in the IDE + CSV group were introduced to the data they are given, had to open the test project's codebase in their IDE, and had to answer a few questions ensuring that they can interpret the data accurately.

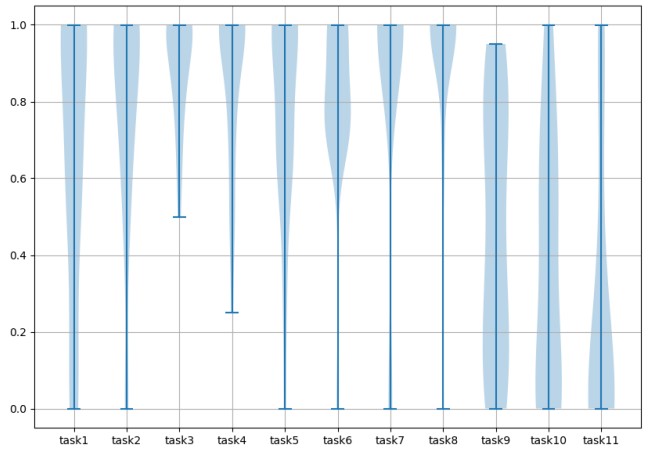
After this session, the subjects had 1h10 to answer the tasks, having for only help a cheat sheet of the definitions detailed in the lecture and detailed in Appendix B. Finally, the students filled out a questionnaire aiming to gather their feedback on the experiment.

6.2. Results

We detail hereafter the results obtained from our experiment. Due to the limited time allocated for the experiment, not all subjects could finish all the tasks. 17/24 subjects in the *VariCity* group and 14/25 subjects in the IDE group gave at least a partial answer (*i.e.*, filled at least one of the answer fields of the task) to all the tasks. Figure 10 presents, for each task, the percentage of subjects having given at least a partial answer. We observe a drop in the percentage of answers at task 7 and 8 for the IDE and *VariCity* groups respectively for two reasons. First, we ordered the tasks in increasing order of difficulty as we perceived it. Then, we made the choice not to impose a time limit for each task as we were unsure about the average time that the subjects would take depending on the tool(s) they use and feared a too low answers rate. Therefore some subjects took more time than they should have on the first tasks and happened to be late at the end of the experiment. In the following results, we considered answers that are at least partial. We also performed for each task a statistical test on the obtained answers, durations and perceived difficulty for both groups to determine



(a) IDE + CSV group



(b) *VariCity* group

Figure 11: Correctness of the answers given by the two groups.

the significance of the difference in the obtained results is and thus validate our hypotheses from Table 6. Obtained p-values for each task are listed in Table 10.

6.2.1. Answer to $\mathbf{RQ_{4.1}}$

Figure 11 presents the correctness of the answers given by the two groups on each task. For tasks 2 to 9 included, we compute the correctness of an answer to a task by calculating the percentage of correct answer elements. For example, when 2 names of classes are expected, the correctness is 0% if no class is in the expected set of classes, 50% for 1 class, and 100% for both classes in the set. Tasks 1, 10 and 11 being open questions, the correctness of the given answers has been evaluated manually based on the presence of expected elements as listed in Table A.12.

We notice that subjects using *VariCity* globally gave more accurate answers to the tasks than subjects using the IDE + CSV combination. For a number of tasks, we believe this is due to the layout of the visualization and the choice of visual axes. For example, using the dimensions of the buildings to represent their methods and constructors

Table 10: p-values obtained by comparing the sets of answers, durations and perceived difficulty of both *VariCity* and IDE + CSV groups on each task. Values in **orange** are in the $]0.05, 0.06]$ range while values in **red** show statistically better results for the IDE + CSV group.

Task	1	2	**3**	**4**	5	**6**	**7**	**8**	9	10	**11**
Correctness	0.96453	0.84286	$\mathbf{1.64023 \times 10^{-14}}$	**0.00024**	0.07749	$\mathbf{8.56822 \times 10^{-13}}$	**0.00259**	**0.00481**	0.24788	0.89565	**0.03911**
Task duration	0.16519	**0.00484**	**0.01191**	0.61467	0.29706	$\mathbf{4.72027 \times 10^{-8}}$	**0.00334**	**0.00122**	0.71231	0.80312	0.90128
Perceived difficulty	0.76113	0.18915	0.10044	0.66913	0.25888	$\mathbf{1.26422 \times 10^{-7}}$	**0.00551**	$\mathbf{2.22208 \times 10^{-5}}$	$\mathbf{3.43713 \times 10^{-5}}$	0.33002	0.64472

overloads, and aerial links for subclasses exhibiting individual density helped subjects to complete task 4 (finding *vp*-s with an important number of class or method level variants, $p = 0.00024$). Similarly, choosing to use streets to represent usage relationships helps their understanding, exhibiting collective density, and helped subjects to complete tasks 3 (finding classes linked through usage relationships to a given one, $p = 1.64023 \times 10^{-14}$) and 7 (finding the distance between two given classes, $p = 0.00259$). Answering tasks 6 (finding design patterns, $p = 8.56822 \times 10^{-13}$) and 8 (identifying hotspots, $p = 0.00481$) was facilitated by the automatic computation of dense zones and design patterns provided by the *symfinder-2* approach used by *VariCity* and their visualization using crowns on buildings and colors respectively.

We notice however that using *VariCity* does not improve the correctness of the answers to all the tasks compared to the IDE + CSV combination. This is the case for task 9 for which the obtained correctness values are too similar between both groups to determine an impact from *VariCity* ($p = 0.24788$). This task was divided in two parts, (i) the identification of classes implementing a given feature, and (ii) indicating whether they were hotspots or not. To answer the first part, subjects from both groups mainly relied on the names of the classes that are given by both *VariCity* and the IDE, leading to similarly accurate answers.

Comparable correctness results are also obtained on task 2 ($p = 0.84286$) where the goal was to give two variants at class level (*i.e.*, two subclasses) for two *vp*-s. Both *VariCity* and the IDE allow searching a class by its name and easy access to the subclasses of a given class (by hovering its building in *VariCity*, and a button in the sidebar of the IDE to navigate the inheritance hierarchy). Analyzing the actions achieved by the subjects in both groups confirms that they heavily relied on these features, and a majority of the subjects in the IDE + CSV group did not use the CSV file. Comparable results are also obtained on task 1 ($p = 0.96453$), which aimed at listing classes that appeared as important to observe when discovering the visualization or the codebase. This can be explained by the fact that a majority of subjects from both groups sought for classes maximizing their number of method level variants, either by identifying tall and large buildings on the visualization or by sorting the CSV file's data in the spreadsheet using the columns for the number of method and constructor variants, optionally summing them. Task 10 also shows comparable results ($p = 0.89565$). Subjects were asked what classes would need to be added or reused to implement another type of chart. Multiple subjects using *VariCity* pointed out that according to them, their answers to this task were limited by not having the source code (*e.g.*, *"Not knowing the library, it is hard to find all the classes linked to a feature."*, *"The tool allows me to have an idea, but is maybe not precise enough regarding the implementation of the classes."*), suggesting that this limitation prevented *VariCity* to assist in solving this task. However, this limitation did not prevent them to grasp the main abstractions used in the library as the correctness is in average better for this group on task 11 ($p = 0.03911$).

Finally, subjects from the IDE + CSV group perform slightly better on task 5 ($p = 0.07749$), which consisted of the three most individually dense classes. For the IDE + CSV group, obtaining the answer to this task consisted in finding the classes maximizing the sum of their method and constructor overloads, and was achieved by most students. On the opposite, *VariCity* displays those two pieces of information using the height and width of the buildings, making it less intuitive to identify the buildings maximizing both aspects. As a result, some subjects from this group not only indicated tall but also wide buildings that were maximizing their constructors' overloads but not the total method variants.

It results that subjects using *VariCity* gave statistically more correct answers on 6/11 tasks compared to subjects using the IDE + CSV combination. Such tasks focus on identifying complex variability implementation patterns which is eased by to the organization of the information provided by the city metaphor as well as its computation of other information such as the presence of a design pattern or a hotspot. Given these encouraging results and the little rate of wrong answers given, we choose not to exclude them from the analyses answering RQ_2 and RQ_3.

6.2.2. Answer to $\mathbf{RQ_{4.2}}$

Figure 12 presents the distribution of the time spent on each task for both groups. It results from this figure and from the obtained p-values (Table 10) that no tool performs better on all the tasks.

VariCity performs better on tasks 3 ($p = 0.01191$), 6 (4.72027×10^{-8}), 7 ($p = 0.00334$) and 8 ($p = 0.00122$) as it directly exhibits on the visualization the presence of hotspots and design patterns, while IDE users need to identify them manually. The structure of the city based on usage relationships also helped the subjects to identify the distance and usage relationships between two classes while

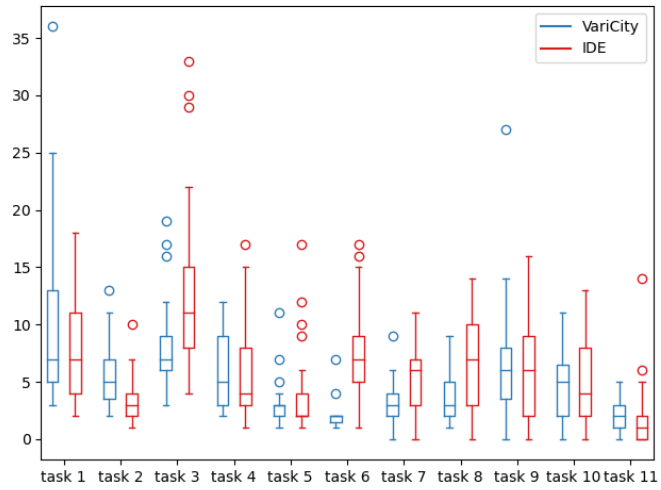


Figure 12: Distribution of the completion time (in minutes) for each task when using *VariCity* or the IDE

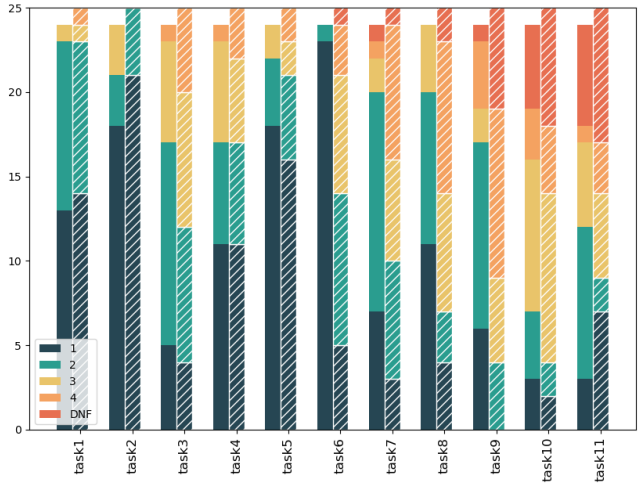


Figure 13: Distribution of the difficulty values for each task when using *VariCity* (plain colors) or the IDE (striped pattern). The students which did not finish are represented in the DNF category.

IDE users needed to explore the code. Regarding task 5, while Figure 12 shows a more reduced range of durations suggesting a positive impact of *VariCity* for this task, the obtained p-value of 0.29706 does not allow us to conclude.

Concerning task 2 ($p = 0.00484$), subjects with the IDE performed better. As the answers given by the two groups on task 1 were of equivalent correctness, we suppose that students with *VariCity* needed some more time to discover the view and get familiar with it. Completing task 2 is equivalent to finding the subclasses of a given class, an action that the subjects are used to accomplish regularly with their IDE. This is confirmed by the fact that 23/25 subjects in the IDE group used the IDE only to complete the task. Therefore, subjects in this group were on average faster than subjects in the *VariCity* group that had to find the information in a visualization they were less familiar with. While Figure 12 shows a more reduced range of durations suggesting a positive impact of the IDE for task 1, the obtained p-value of 0.16519 does not allow us to conclude.

Finally, on tasks 4 ($p = 0.61467$), 9 ($p = 0.71231$), 10 ($p = 0.80312$) and 11 ($p = 0.90128$), the performances of both approaches appear as equivalent. To complete task 4, finding the classes with an important number of variants at method level consisted in looking at the corresponding column in the CSV for the IDE group, and in looking at high and/or wide buildings in the visualization, suggesting that all subjects had a clear idea of where to find this information. Regarding the number of variants at class level, while subjects in the IDE group heavily relied on class diagrams reverse-engineered with the IDE, subjects in the *VariCity* group mainly hovered over random classes until finding the information, suggesting that finding class level variants in *VariCity* is not intuitive. We would expect task 9 to take less time using *VariCity*. In practice, it results that the average time spent is equivalent as multiple subjects from the IDE group did not give the hotspot information. However, correlating this result with the fact that 50% of subjects indicated the maximum perceived difficulty (*cf.* Figure 13) is a strong indication that this part of the task was too difficult for the IDE group. Finally, tasks 10 and 11 (finding classes needed to implement a new type of chart, finding the main abstractions) require few interactions with the tools apart from simple class searching, explaining the similar average time spent.

It results that the subjects in the IDE group performed better on single tasks they are familiar with (*e.g.*, finding a class, navigating inheritance). On the opposite, identifying complex zones concentrating variability implementations is more efficient in *VariCity* (*e.g.*, design patterns, hotspots). While determining the individual density of variability appears to be equally time consuming thank to the CSV file, determining the collective density of variability and hotspots is more efficient by the organization of the *VariCity* view.

6.2.3. Answer to $\mathbf{RQ_{4.3}}$

Figure 13 presents the distribution of the difficulty as perceived by the subjects on each task for both groups. We notice a difference in perceived difficulty in favor of *VariCity* on tasks 6 to 9 which are focused on identifying design patterns, hotspots and distance between classes. This result is confirmed by the obtained p-values for these tasks, listed in Table 10. This is also coherent with the results obtained regarding the time spent completing these tasks. Tasks 6, 7 and 8 took considerably more time for subjects using the IDE compared to subjects using *VariCity*. This is not the case for task 9 as, as explained in the previous section, an important number of subjects did not give information about the presence of a hotspot and only partially completed the task.

Regarding the other tasks, while subjects using *VariCity* globally tended to give lesser difficulty values than subjects using the IDE, they are not enough to exhibit a significant difference and thus prevent us from concluding on any impact *VariCity* could have on these tasks. Similarly, although difficulties 1 and 2 are more represented in the IDE + CSV group on task 2, this difference is not significant enough to demonstrate the impact of the use of the IDE on this task ($p = 0.18915$).

It results that subjects in the *VariCity* group find it easier to complete 4/11 tasks compared to subjects in the IDE group, representing tasks related to the identification of zones concentrating variability implementations using complex structures, such as design patterns or hotspots that implement variability at both class and method levels. While this represents a minority of tasks, it also appears that no task seemed easier to subjects using the IDE.

6.2.4. Summary

By providing a visual representation of the system's classes and exhibiting metrics on their variability, subjects using *VariCity* statistically gave more correct answers on tasks focusing on the identification of complex variability implementations, thus validating H_{1_0} (Table 6). Additionally, the completion time and the perceived difficulty of these tasks is statistically reduced (thus validating H_{2_0} and H_{3_0}), demonstrating the visualization's capacity to help the identification of complex variability implementations. A deeper analysis of the actions performed by subjects from both groups suggests some improvements for *VariCity*, especially regarding tasks where the results appear as equivalent between both groups. Some information is more easily accessible with the visualization, such as the distance in usage between classes or the density of variability implementations. Other actions were facilitated with the IDE, such as obtaining a reverse-engineered class diagram to navigate the inheritance relationships.

Feedback from some subjects in the *VariCity* group reveals that although the view allowed to quickly spot important zones of the system (*"The visualization allows us to easily notice the features present in the system. Classes that are less important can be ignored to focus on the ones that have more variability."*), their comprehension was limited by not having the actual source code (*"We stay really abstract by visualizing the code with VariCity, a Java IDE would allow us for example to have access to comments that can help comprehension."*, *"It is hard to understand how the system works only with the buildings."*). This feedback suggests that although *VariCity* helps in guiding the exploration of a system, having access to the source code remains of prime importance to have a deep understanding of it.

As a first step towards addressing this feedback, we proposed an extension of the *VariMetrics* approach providing a full integration of the visualization and its configuration in the JetBrains IntelliJ IDEA IDE [120]. The visualization is available as a panel in the IDE's interface and can be configured through the IDE's menus as for any other plugin. This integration also provides bidirectional navigation between the code opened in the editor and the visualization.

7. Threats to validity and limitations

Conclusion and construct validity threats are discussed in Table 11. In the following we analyze internal and external threats, while discussing limitations at the end of this section.

Internal threats. The main threat of our evaluation concerns the fact that both authors and developers of *VariCity* designed and conducted the evaluation of the approach in Section 5. The inputs (entry points, usage levels, and orientations) and quality metrics have been determined empirically based on their knowledge of the systems and of *VariCity*'s capabilities. Still, even by having a coarse-grain understanding compared to a real expert, the obtained visualizations already exhibit satisfying results by revealing zones of interest in the systems. Although the identification of these revealed zones concentrating variability implementations and/or being quality-critical has been achieved by the authors that know how to read the visualization, this enumeration has been done systematically relying on determined criteria (*e.g.*, important height/width, intense color). Finally, the validation of their relevance is based on documentations and comments in the concerned classes, giving us confidence in its soundness.

External threats. As *VariCity* relies on the *symfinder* toolchain to detect variability implementations, it is subject to the same threat on the considered implementation techniques and the Java-only focus. As for the techniques, in our work, six main object-oriented techniques are identified while other mechanisms, functional or at the statement level, could be used to implement variability [53]. Moreover, no cross-cutting variability can be detected by *symfinder* while the detection of the usage of the same parameters in different locations could be a first solution. However, the identifications made with *symfinder* were successful [36] and we reused the same systems for validation.

Due to organizational constraints, the controlled experiment has been conducted with a population whose subjects have similar experience and background. Consequently, the size of the population constrained us to conduct the experiment on a single system to prevent having too small groups for each treatment. While those biases could not be mitigated, we believe the obtained conclusions to be good enough to support the results obtained in the evaluation in Section 5 as the profile of the subjects match our target population (Section 6.1.2). In future work, we will aim at extending this controlled experiment by extending our panel of subjects and diversifying

Table 11: Actions taken to mitigate conclusion and construct validity threats [121]

Threat	**Action**
Conclusion validity threats	
Low statistical power	We mitigated by computing p-values to statistically measure the relevance of the difference in the answers (Table 10)
Violated assumptions of statistical tests	We used *t-test* as our samples are independent and randomly extracted from the same population.
Fishing and the error rate	We do not adapt the significance level and keep the standard threshold of 0.05 (Table 10).
Reliability of measures	Completion time is the only measure we rely on that is not an answer from the subjects. While students were asked to fill the start and end time of each task themselves, the little numbers of outliers visible on Figure 12 comforts us in the coherence of this data.
Reliability of treatment implementation	The treatment is strictly identical between all subjects of a same group. *VariCity* was provided as a Docker image, preventing possible side effects related to the installation of the tool (Section 6.1.5).
Random irrelevancies in experimental setting	The experiment was held in an room of the Polytech Nice Sophia engineering school with doors closed to prevent any distraction. The experiment has also not been interrupted.
Random heterogeneity of subjects	Threats related to the homogeneity of our population are discussed in the *External threats* paragraph in Section 7.
Construct validity threats	
Design threats	
Inadequate preoperational explication of constructs	Variability concepts were explained during the lecture given on January 12, 2022. A question in the pre-experimentation survey aimed at validating their understanding of these notions, which is confirmed by the answers obtained (Figure 9).
Mono-operation bias	Due to the size of our population, we could run the experiment on a single system only. This threat is detailed in the *Independent variables* paragraph in Section 6.
Mono-method bias	We defined oracles to evaluate the correctness of the tasks, listed in Table A.12.
Confounding constructs and levels of constructs	Subjects have similar knowledge levels of variability concepts and of the Java language. Their mastering of the tools is also equivalent.
Interaction of different treatments	Each subject does only one treatment.
Interaction of testing and treatment	Subjects do not evaluate the actions they performed or the code they produced.
Restricted generalizability across constructs	Although *VariCity* does not perform better than the IDE + CSV on all tasks, the IDE + CSV performs statistically better on only one task (*cf.* Section 6.2.2) while evaluating the completion, thus we consider that *VariCity* does not negatively impact either the correctness or the difficulty. Additionally, the only aspect that subjects using *VariCity* reported to lack is access to the source code, a limitation we started tackling with an integration of *VariCity* in an IDE [120] (*cf.* Section 6.2.4).
Social threats	
Hypothesis guessing Experimenter expectancies	Questions have been designed so that they do not give insights on what the answer could be. This pattern has been observed by our test subjects during the pilot experiment and corrected.
Evaluation apprehension	In such cases, subject apprehending evaluation would try to avoid answering, for example by selecting the midpoint in a Likert scale [117]. For this reason, we designed a 4-options scale (*cf.* Table 9). We also comforted them by stating that there is no wrong answer and that they are not evaluated in any way.

their background. We also aim at validating whether real experts are able to determine appropriate inputs in real settings, including quality metrics.

Limitations. Concerning the structure of the visualization, the placement of the buildings on a street only relies on the width of the buildings to compact them in the street. This implies that the variability represented by the height of the buildings is not taken into account. Even if this dimension is largely visible on the visualization, this might call for an adaptation of the placement algorithm to take into account both dimensions while placing the buildings.

Similarly, although we chose to provide as many configuration capabilities as possible so that a user can design a view matching her needs, 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 [122] including city-based ones [123, 82] to ensure readability and usability, it would require in our case to empirically validate our approach with real experts to exchange on their needs[22]. We leave this to future work.

By relying on the *symfinder* toolchain, *VariCity* inherits from its limitations. As mentioned in Section 2.2, the dependencies between variability implementations are not detected, and coupling this kind of information with the *symfinder* output could be really interesting. On the visualization side, this would call for adaptations, especially to not overload the city representation. Regarding hotspot classes, their identification is based on two thresholds that are to be determined manually, based on one's knowledge of the system [38]. As a result, inappropriate thresholds result in too few or too many colored buildings. Still, the dimensions of the buildings, the crowns for the design patterns and the organization of the view allow to reveal relevant zones, as it is the case for Cucumber (Figure 6).

8. Related work

In this section, we discuss work related to visual and tooled approaches to assist for variability management and program comprehension.

8.1. Visualization in the Software Product Line field

A recent mapping study has shown that visualizations in the SPL domain mainly target feature models, using tree or graph representations [32]. These visualizations are mainly used to facilitate the configuration process over features. To visualize variability at the code level, some approaches use colors [124] or bar diagrams [125], while some others focus on feature traces [33] or feature interactions between features and code [126, 34]. None of them focus on object-oriented techniques as variability implementations.

In *VariCity*, we reused the symmetry-based detection part of *symfinder* [36, 38], but this tool also provides a graph-based visualization in which each class level *vp* and variant is represented as a circle node that points out the used implementation technique, with size and shades of nodes indicating the presence of OO variability implementation mechanisms. These nodes are linked with both inheritance and usage relationships being different kinds of edges, forming a set of disconnected graphs. While this visualization allows showing some dense zones of variability and has filtering capabilities, it has only been used for the validation of the capabilities of *symfinder* in identifying potential *vp*-s and variant. It is not adapted for comprehending variability as in our considered scenarios, especially in large-scale systems in which the resulting visualization is not usable (approx. 4k nodes for NetBeans).

8.2. Visualization for software comprehension

Most of the time spent in software maintenance is dedicated to understanding the software system itself. Representing source code through adapted abstractions and metaphors helps in understanding and software visualization has been successfully experimented with and applied to do so [127, 128]. While visualizations centered on code lines or classes have been proposed, more attention has been put on visualizing the architecture of software, with treemaps [127, 128] or city metaphors, which we have discussed in Section 3.2. These visualizations usually complement more classical ones that aim at understanding relationships, with graphs or UML diagrams extracted through static analysis. Node-link graph visualizations can easily represent dependencies but become confusing on very large applications, while dependency structure matrices has been shown to help in identifying software dependencies [127, 128]. On its side, the usage of UML diagrams have been shown to be only helpful to experienced engineers when provided with a codebase [129]. Such diagrams can also help when no comments are provided in code [130], but need to be extracted from it (as opposed to diagrams from the requirement phases) to be useful.

Our visualization problem being metric-based, we have naturally turned to software metaphors that are adapted [127, 128]. Besides the city, other metaphors have also been proposed. The *Software Cartography* visualization [131] explores the map metaphor, displaying code assets as islands made of classes using common terminologies. *CodeSurveyor* [132] uses a cartographic metaphor representing high-level architectural components containing directories and source files as continents decomposed in countries and states. Metrics on the source files are used to determine the size of the nested regions. The island metaphor has also been proposed to represent OSGi[23] systems in virtual reality [133].

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

[23] https://www.osgi.org/

A system is an archipel made of islands representing OSGi bundles. An island is decomposed in regions containing buildings, depicting packages and the classes they contain. However, such approaches do not represent relationships between classes. Code Park [134] displays classes as rooms in a 3D environment, that can be explored in first-person view as in a game. Rooms are organized by their position in the directory structure, and their dimensions evolve according to the size of the class they represent. The content of the class is displayed as wallpapers on the internal walls of the room. Although this metaphor allows displaying the code and the high-level information about some metrics, it does not allow to represent variability metrics.

8.3. Tools for program comprehension

Multiple tooled approaches to assist the user in software comprehension activities have been proposed as plugins for popular IDEs, such as VRLifeTime [135], an IntelliJ plugin to "visualize lifetime for Rust programs and help programmers avoid lifetime-related mistakes", or ArCode [136], an IntelliJ plugin to suggest API misuse corrections. These plugins are centered on comprehending the structure of the codebase, where *VariCity* focuses on comprehending the implemented variability of the system.

Full environments have also been proposed to help software comprehension, such as SolidFX [137], which provides a set of visualizations and tools to help to understand a C/C++ codebase, or Hunter [138], a complete environment providing a graph visualization of JavaScript source code. Not only those two solutions are not applicable to Java codebases, but they also focus on quality metrics to support code comprehension. SolidFX combines such metrics with reverse-engineered UML diagrams of the project. Hunter's graph uses nodes to represent JS files, with their size varying with the number of LoCs, and edges for usage relationships. On the opposite, *VariCity* aims to facilitate the comprehension of the implemented variability by guiding the user towards the parts of the codebase that contain heavy use of variability implementations.

More specifically, IDE plugins to assist feature location activities have also been designed. FeatureDashboard [139] is an Eclipse plugin that allows visualizing known mappings between features and code assets as well feature annotations in a codebase. This approach has been extended with a notation for embedded feature annotation and the FAXE (Feature Annotation eXtraction Engine) [140] tool to process such annotations. HAnS [141] is an IntelliJ plugin to assist the management and edition of feature annotations in code assets. Using these plugins, however, requires having the list of the implemented features and their mapping with the code assets, and are therefore not adapted to our context. FLAT^3 [142] is an Eclipse plug-in performing feature location in an opened project. This location is performed statically based on names of classes and methods, and dynamically using tests execution traces. The obtained results can be used to annotate the corresponding artifacts. A visualization is provided, representing classes as boxes with rows of pixels corresponding to sections of code that are highlighted if they implement a given feature. Although it performs feature location, this plugin does not identify variability implemented with OO mechanisms.

9. Conclusion

Object-oriented software systems often reuse OO mechanisms to implement their variability in a single codebase. As these implementations are not explicit, their comprehension is very difficult, if not impossible. In this paper, we proposed *VariCity*, a configurable and extensible 3D visualization adapting the city metaphor to exhibit zones of high density of variability implementations. The provided configuration options allow the user to design views focusing on a subpart of the system, enabling comprehension of the variability at fine grain. The proposed solution has been extended to support additional metrics on properties of the studied system such as its quality, creating the *VariMetrics* visualization. The application of *VariCity* and *VariMetrics* on several open source systems written in Java showed the capacity of the visualization and its configuration capabilities to reveal relevant zones concentrating variability implementations and/or being quality-critical. The comprehensibility of the adapted metaphor has been evaluated with a controlled experiment showing the gain brought by *VariCity* to solve variability comprehension tasks on a medium-sized system compared to the use of an IDE.

As a future work, we intend to design a controlled experiment with real developers to comfort or complement results of the experimentation conducted in Section 6. We also plan to extend *VariCity* to take into account additional information on the implemented variability such as annotations [143, 144, 145] which, when available, can further assist variability comprehension. Finally, we aim to extend both *VariCity* and the variability identification approach [36, 38] to support other languages and help the comprehension of multi-language systems.

Reproduction package

A reproduction package [146] is publicly available containing:

- the source code of *VariCity* and its *VariMetrics* extension;
- preconfigured views for all systems presented in Tables 2 and 4;
- annotated screenshots for each view displaying the revealed and relevant zones detailed in Table 2;
- the Excel file used to obtain the numbers of noticeable classes presented in Table 4;

- the source code of the *symfinder-2* toolchain enabling reproduction of these views;
- the subjects' answers to the comprehension tasks as well as a Jupyter notebook processing them to obtain the diagrams showcased in Section 6.

Acknowledgments

We thank Patrick Anagonou, Paul-Marie Djekinnou, Florian Focas, François Rigaut, Guillaume Savornin and Anton van der Tuijn for their contribution in the development of *VariCity* and *VariMetrics*. We also thank the subjects of our pilot experiment (Prof. Philippe Renevier–Gonin, Nassim Bounouas, Yassine El Amraoui and Yann Brault) for their feedback on its design, as well as Antonia Ettorre for helping us analyzing the controlled experiment's data. Finally, we thank all the Master's students which participated as subjects of our experiment.

Appendix A. Expected answers

Appendix B. Cheat sheet

This appendix presents the definitions contained in the cheat sheet handed out to the students before the experiment.

In the following definitions:

- *class* will indifferently refer to a class or an interface.
- *subclass* will indifferently refer to a subclass of a class or a class implementing an interface (use of `extends` or `implements` in Java). For this reason, although the studied system is written in Java, the *subclass* term may be used in the plural.

Appendix B.1. Varability concepts

Definition 1 (Usage relationship). There is a usage relationship between two classes `A` and `B` if `A` uses `B`. `A` uses `B` means that `B` is used as the type of an attribute of `A` or of a parameter of a method of `A`.

Definition 2 (Variation point (*vp*) and variant). A **variation point** can represent:

- a class having at least 2 subclasses. The **variants** then represent the subclasses, and are qualified as *class level variants*.
- a class which has at least one method or an overloaded constructor. The **variants** then represent the overloads, and are qualified as *method level variants*.
- a design pattern (*cf.* Appendix B.2).

Note: A class can have both class-level and method-level variants.

Definition 3 (Individual density). A *vp* is individually dense for a threshold v if it has at least v variants (at least v subclasses or at least v method and constructor overloads).

Example: A class is individually dense for $v = 20$ if it has at least 20 subclasses and/or at least 20 method and constructor overloads.

Definition 4 (Collective density). A *vp* is collectively dense for a threshold d if it is at most d usage relationships away from another *vp* (*cf.* Definition 2).

Example: A class is collectively dense for $d = 2$ if it is linked through a usage relationship with a class that is itself linked through a usage relationship with another *vp*.

Definition 5 (Hotspot). A hotspot is a class being dense both individually for a threshold v and collectively for a threshold d.

Appendix B.2. Design patterns

Definition 6 (Strategy). We define as a *Strategy* a class for which at least one of the following two statements is correct:

- its name ends with `Strategy`;
- it has at least 2 subclasses and is used as an field in another class.

Example:

```
class Algorithm {  }

class Algorithm1 extends Algorithm {  }

class Algorithm2 extends Algorithm {  }

class App {
    Algorithm algo;
}
```

`Algorithm` has 2 subclasses (`Algorithm1` and `Algorithm2`) and is used as a field in another class (`App`), it is thus a *Strategy*.

Definition 7 (Factory). We define as a *Factory* a class for which at least one of the following two statements is correct:

- its name ends with `Factory`;
- it has a method returning an object whose type is a subclass of the method's return type.

Example:

```
class Animal {  }

class Dog extends Animal {  }

class Cat extends Animal {  }

public class AnimalCreator {

  Animal create(String animalType) throws
      AnimalCreationException {
```

Table A.12: The expected answers for the 11 variability comprehension tasks given to the subjects.

ID	Task	Expected answers
1	When discovering the visualization / the code in the IDE and the data in the spreadsheet, according to you, what classes seem to be important to explore in priority? For each one of them, explain why.	For this open question, we expected in the answer the presence of classes such as `CategoryPlot` and `XYPlot` for which OO variability measures are higher than for other classes, as well as their subclasses.
2	Identify 2 variants at class level for each of the following variation points (*cf.* Definition 2): • `org.jfree.chart.plot.Plot` • `org.jfree.chart.title.TextTitle`	• For `Plot`: `org.jfree.chart.plot.{PolarPlot, XYPlot, CompassPlot, MultiplePiePlot, CategoryPlot, ThermometerPlot, PiePlot, WaferMapPlot, SpiderWebPlot, FastScatterPlot, MeterPlot, dial.DialPlot}` • For `TextTitle`: `org.jfree.chart.title.{ShortTextTitle, DateTitle}`
3	How many classes are linked with a usage relationship to each of the following classes? Give 3 examples. • `org.jfree.chart.plot.CategoryPlot` • `org.jfree.chart.title.CompositeTitle`	• For `CategoryPlot` (20 classes): `org.jfree.data.category.CategoryDataset`, `org.jfree.chart.annotations.CategoryAnnotation`, `org.jfree.chart.util.ShadowGenerator`, `org.jfree.chart.renderer.category.CategoryItemRenderer`, `org.jfree.chart.plot.{CombinedDomainCategoryPlot, CombinedRangeCategoryPlot, CategoryMarker, DatasetRenderingOrder, CategoryCrosshairState, PlotRenderingInfo, Marker}`, `org.jfree.chart.annotations.CategoryAnnotation`, `org.jfree.chart.renderer.category.CategoryItemRenderer`, `org.jfree.chart.LegendItemCollection`, `org.jfree.chart.axis.{CategoryAnchor, ValueAxis, AxisLocation, AxisSpace, CategoryAxis}`, `org.jfree.chart.ui.RectangleInsets` • For `CompositeTitle` (2 classes): `org.jfree.chart.block.{BlockContainer, RectangleConstraint}`
4	Complete the following sentences: • Classes (1) and (2) have an important number (≥ 5) of subclasses (*i.e.*, are variation points with an important number of variants at class level). • Classes (3) and (4) have an important number (≥ 10) of overloaded methods and constructors (*i.e.*, are variation points with an important number of variants at method level).	• **(1) and (2)**: `org.jfree.data.xy.XYDataset`, `org.jfree.data.category.CategoryDataset`, `org.jfree.chart.renderer.xy.XYItemRenderer`, `org.jfree.chart.needle.MeterNeedle`, `org.jfree.chart.title.Title`, `org.jfree.chart.block.Arrangement`, `org.jfree.chart.plot.{Plot, dial.DialLayer}` • **(3) and (4)**: `org.jfree.chart.plot.{CategoryPlot, XYPlot, PolarPlot, SpiderWebPlot, PiePlot, ThermometerPlot, dial.DialPlot}`, `org.jfree.chart.axis.ValueAxis`, `org.jfree.chart.JFreeChart`, `org.jfree.chart.LegendItem`
5	Identify the 3 classes with the highest individual density higher to the threshold $v = 20$ (*cf.* Definition 3).	`org.jfree.chart.plot.{CategoryPlot, XYPlot, PolarPlot}`
6	Give 2 examples of each of the following design patterns (*cf.* Appendix B.2): • Strategy pattern; • Factory pattern.	• **Strategy**: `org.jfree.chart.plot.{CategoryPlot, XYPlot, Plot, Marker}`, `org.jfree.chart.title.{TextTitle, Title}`, `org.jfree.chart.block.Arrangement`, `org.jfree.chart.axis.TickUnit` • **Factory**: `org.jfree.chart.JFreeChart`, `org.jfree.chart.plot.Plot`, `org.jfree.chart.item.LegendItem`, `org.jfree.chart.block.BlockContainer`, `org.jfree.chart.ChartRenderingInfo`
7	What is the distance between the `org.jfree.chart.JFreeChart` and `org.jfree.chart.title.DateTitle` classes?	2
8	Identify 3 hotspots for an individual density threshold of $v = 20$ and a collective density threshold of $d = 5$ (*cf.* Definition 5).	`org.jfree.chart.plot.{CategoryPlot, XYPlot, Plot, PolarPlot, DefaultDrawingSupplier, dial.DialPointer}`, `org.jfree.chart.axis.{ValueAxis, AxisSpace}`, `org.jfree.chart.title.{TextTitle, LegendTitle, PaintScaleLegend}`, `org.jfree.chart.JFreeChart`, `org.jfree.chart.block.BlockContainer`, `org.jfree.data.xy.XYDataset`, `org.jfree.chart.annotations.XYAnnotation`, `org.jfree.chart.renderer.xy.XYItemRenderer`, `org.jfree.chart.ui.Size2D`
9	Identify the classes that according to you implement each of the following features, and specify if they are hotspots for $v = 20$ and $d = 5$ (*cf.* Definition 5): • "draw a chart" feature; • "title of the chart" feature.	• **"draw a chart"**: `org.jfree.chart.plot.{PiePlot3D, PiePlot, RingPlot, DialPlot, MeterPlot, ThermometerPlot, CompassPlot, WaferMapPlot, Plot, SpiderWebPlot, CategoryPlot, MultiplePiePlot, FastScatterPlot, PolarPlot, XYPlot, PlotRenderingInfo, CombinedDomainCategoryPlot, CombinedRangeXYPlot, CombinedDomainXYPlot, CombinedRangeCategoryPlot, PiePlotState}` • **"title of the chart"**: `org.jfree.chart.title.{ShortTextTitle, TextTitle, LegendTitle, Title, CompositeTitle, ImageTitle, DateTitle}`
10	What are according to you the classes to add/reuse/modify to implement a new type of chart (*i.e.*, to implement a new variant of the `org.jfree.chart.plot.Plot` *vp*) ?	For this open question, we expected as part of the answer: • **classes to add**: a subclass of `Plot`, possibly a new class for a dataset such as `XYDataset` or `CategoryDataset` • **classes to reuse**: classes being already used by other variants of `Plot` (*e.g.*, `ValueAxis`) • **classes to modify**: classes already using other variants of `Plot` that might be extended to support this new variant
11	Following this observation of JFreeChart, what are according to you the main abstractions used in this charting library?	For this open question, we expected to find in the answer terms related to the main implemented features such as *plot*, *title*, *axis* or *legend*.

```
    if(animalType.equals("cat")) {
      return new Cat();
    }
    else if(animalType.equals("dog")) {
      return new Dog();
    }
    throw new AnimalCreationException("Unable to
        create a " + animalType);
  }
}
```

The method `create(String)` in `AnimalCreator` returns a `Cat` or a `Dog`, being subtypes of the method's return type (*i.e.*, `Animal`), it is thus a *Factory*.

Definition 8 (Decorator). We define as a *Decorator* a class for which at least one of the following two statements is correct:

- its name ends with `Decorator`;
- it has at least a subclass and has a field whose type is one of its superclasses having at least 2 subclasses.

Example:

```
interface Window {  }

class SimpleWindow implements Window {  }

abstract class AbstractDecorator implements
    Window {
    Window window;
}

class ConcreteDecorator extends WindowDecorator {
      }
```

`AbstractDecorator` has a subclass (`ConcreteDecorator`), uses the interface it implements as a field (`Window`), that itself has two subclasses (`SimpleWindow` and `WindowDecorator`), `AbstractDecorator` is thus a *Decorator*. `ConcreteDecorator` will also be identified as a *Decorator* as its name ends with `Decorator`.

Definition 9 (Template). We define as a *Template* a class for which at least one of the following two statements is correct:

- its name ends with `Template`;
- it is abstract, has at least a subclass and has an abstract method that is called in a concrete method of the same class.

Example:

```
abstract class Algorithm {

    abstract void abstractStep();

    void run() {
        ...
        abstractStep();
        ...
    }
}
```

`Algorithm` has an abstract method `abstractStep` that is called in the concrete method `run`, it is thus a *Template*.

References

[1] R. Hilliard, On representing variation, in: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ECSA '10, ACM, 2010, p. 312–315.

[2] M. Galster, D. Weyns, D. Tofan, B. Michalik, P. Avgeriou, Variability in software systems — a systematic literature review, IEEE Transactions on Software Engineering 40 (2013) 282–306.

[3] M. Galster, 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, 2019, pp. 1–1.

[4] R. Capilla, J. Bosch, K.-C. Kang, et al., Systems and software variability management, Concepts Tools and Experiences (2013).

[5] P. Paskevicius, R. Damasevicius, V. Štuikys, Change impact analysis of feature models, in: International Conference on Information and Software Technologies, ICIST '12, CCIS 319, Springer, 2012, pp. 108–122.

[6] K. Pohl, G. Böckle, F. J. van Der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer Science & Business Media, 2005.

[7] S. Apel, D. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines, Springer, 2013.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.

[9] J. Liebig, S. Apel, C. Lengauer, C. Kästner, M. Schulze, 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, 2010, pp. 105–114.

[10] K. Schmid, I. John, A customizable approach to full lifecycle variability management, Science of Computer Programming 53 (2004) 259–284.

[11] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, G. Saval, Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis, in: 15th IEEE International Requirements Engineering Conference, RE '07, IEEE, 2007, pp. 243–253.

[12] D. Benavides, P. Trinidad, A. Ruiz-Cortés, Automated reasoning on feature models, in: International Conference on Advanced Information Systems Engineering, Springer, 2005, pp. 491–503.

[13] C. Gacek, M. Anastasopoules, Implementing product line variabilities, in: Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context, SSR '01, ACM, 2001, pp. 109–117.

[14] M. Svahnberg, J. Van Gurp, J. Bosch, A taxonomy of variability realization techniques, Software: Practice and experience 35 (2005) 705–754.

[15] Xh. Tërnava, P. Collet, Tracing imperfectly modular variability in software product line implementation, in: International Conference on Software Reuse, ICSR '17, Springer, 2017, pp. 112–120.

[16] X. Tërnava, J. Mortara, P. Collet, D. Le Berre, Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach, Journal of Automated Software Engineering 29 (2022) 1–51.

[17] A. Metzger, K. Pohl, Software product line engineering and variability management: achievements and challenges, in: Future of Software Engineering Proceedings, Association for Computing Machinery, New York, NY, USA, 2014, pp. 70–84.

[18] M. Acher, L. Lesoil, G. A. Randrianaina, X. Tërnava, O. Zendra, A call for removing variability, in: 17th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS 2023), 2023.

[19] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, Ieee software 29 (2012) 18–21.

[20] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), in: Dagstuhl Reports, volume 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
[21] D. Wolfart, W. K. G. Assunção, J. Martinez, Variability Debt: Characterization, Causes and Consequences, in: XX Brazilian Symposium on Software Quality, 2021, pp. 1–10.
[22] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empirical Software Engineering 22 (2017) 2972–3016.
[23] M. A. Laguna, Y. Crespo, A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring, Science of Computer Programming 78 (2013) 1010–1034.
[24] J. Mortara, X. Tërnava, P. Collet, 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, VaMoS '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 1–9.
[25] J. Mortara, Mastering variability in the wild: on object-oriented variability implementations and variability-aware build systems. (Maîtriser la variabilité enfouie dans les systèmes orientés objet et les systèmes de construction logicielle), Ph.D. thesis, Côte d'Azur University, Nice, France, 2022.
[26] S. Bassil, R. K. Keller, Software visualization tools: Survey and analysis, in: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, IEEE, 2001, pp. 7–17.
[27] S. Diehl, Software visualization: visualizing the structure, behaviour, and evolution of software, Springer Science & Business Media, 2007.
[28] A. R. Teyseyre, M. R. Campo, An overview of 3D software visualization, IEEE transactions on visualization and computer graphics 15 (2008) 87–105.
[29] C. Knight, M. Munro, Comprehension with [in] virtual environment visualisations, in: Proceedings Seventh International Workshop on Program Comprehension, IEEE, 1999, pp. 4–11.
[30] C. Knight, M. Munro, Virtual but visible software, in: 2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics, IEEE, 2000, pp. 198–205.
[31] H. A. Duru, M. P. Çakır, V. İşler, How does software visualization contribute to software comprehension? a grounded theory approach, International Journal of Human-Computer Interaction 29 (2013) 743–763.
[32] R. E. Lopez-Herrejon, S. Illescas, A. Egyed, A systematic mapping study of information visualization for software product line engineering, Journal of software: evolution and process 30 (2018) e1912.
[33] B. Andam, A. Burger, T. Berger, M. R. Chaudron, 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, 2017, pp. 100–107.
[34] A. Bergel, R. Ghzouli, T. Berger, M. R. V. Chaudron, 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, 2021, p. 196–201.
[35] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, T. Berger, Where is My Feature and What is it About? A Case Study on Recovering Feature Facets, Journal of Systems and Software 152 (2019) 239–253.
[36] X. Tërnava, J. Mortara, P. Collet, Identifying and visualizing variability in object-oriented variability-rich systems, in: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A, SPLC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 231–243.
[37] J. Mortara, X. Tërnava, P. Collet, symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations, in: Proceedings of the 23rd International Systems and Software Product Line Conference - Volume B, SPLC '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 5–8.
[38] J. Mortara, X. Tërnava, P. Collet, A.-M. Dery-Pinna, Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships, in: Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume B, SPLC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 91–98.
[39] R. Pienta, J. Abello, M. Kahng, D. H. Chau, Scalable graph exploration and visualization: Sensemaking challenges and opportunities, in: 2015 International conference on Big Data and smart computing (BIGCOMP), IEEE, 2015, pp. 271–278.
[40] R. Wettel, M. Lanza, Visualizing software systems as cities, in: 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, 2007, pp. 92–99.
[41] R. Wettel, M. Lanza, CodeCity: 3D visualization of large-scale software, in: Companion of the 30th international conference on Software engineering, 2008, pp. 921–922.
[42] F. Steinbrückner, C. Lewerentz, Understanding software evolution with software cities, Information Visualization 12 (2013) 200–216.
[43] J. Mortara, P. Collet, A.-M. Dery-Pinna, Visualization of Object-Oriented Variability Implementations as Cities, in: 2021 Working Conference on Software Visualization (VISSOFT), Luxembourg (virtual), Luxembourg, 2021, pp. 76–87.
[44] J. Mortara, P. Collet, A.-M. Dery-Pinna, Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A, SPLC '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 43–54.
[45] M. Azanza, A. Irastorza, R. Medeiros, O. Díaz, Onboarding in Software Product Lines: Concept Maps as Welcome Guides, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET), IEEE, 2021, pp. 122–133.
[46] C. R. Turner, A. Fuggetta, L. Lavazza, A. L. Wolf, A conceptual basis for feature engineering, Journal of Systems and Software 49 (1999) 3–15.
[47] J. O. Coplien, Multi-Paradigm Design for C++, Addison-Wesley Longman Publishing Co., Inc., 1999.
[48] F. Bachmann, P. Clements, Variability in Software Product Lines, Technical Report CMU/SEI-2005-TR-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005.
[49] I. Jacobson, M. Griss, P. Jonsson, Software reuse: architecture process and organization for business success, volume 285, acm Press New York, 1997.
[50] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, A. Wasowski, 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, 2012, pp. 173–182.
[51] R. Rabiser, 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, 2019, pp. 134–136.
[52] I. John, J. Lee, D. Muthig, Separation of variability dimension and development dimension, in: Proocedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems, VaMoS '07, 2007, pp. 45–49.
[53] Xh. Tërnava, P. Collet, 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, 2017, pp. 81–88.
[54] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, A. Sousa, A model-driven traceability framework

for software product lines, Software & Systems Modeling 9 (2010) 427–451.
[55] T. Patzke, D. Muthig, Product Line Implementation Technologies. Programming Language View, Technical Report, Fraunhofer IESE, 2002.
[56] C. Fritsch, A. Lehn, T. Strohm, R. Bosch, Evaluating variability implementation mechanisms, in: Proceedings of International Workshop on Product Line Engineering (PLEES), Citeseer, 2002, pp. 59–64.
[57] E. OliveiraJr (Ed.), UML-Based Software Product Line Engineering with SMarty, Springer, 2023.
[58] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: A taxonomy and survey, Journal of Software: Evolution and Process 25 (2013) 53–95.
[59] J. Rubin, M. Chechik, A survey of feature location techniques, in: Domain Engineering, Springer, 2013, pp. 29–58.
[60] J. Martinez, Xh. Tërnava, T. Ziadi, Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study, in: Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1, SPLC '18, ACM, 2018, pp. 132–142.
[61] A. Lozano, An overview of techniques for detecting software variability concepts in source code, in: International Conference on Conceptual Modeling, ER '11, Springer, 2011, pp. 141–150.
[62] M. Burch, C. Vehlow, F. Beck, S. Diehl, D. Weiskopf, Parallel edge splatting for scalable dynamic graph visualization, IEEE Transactions on Visualization and Computer Graphics 17 (2011) 2344–2353.
[63] F. Beck, M. Burch, S. Diehl, D. Weiskopf, A taxonomy and survey of dynamic graph visualization, in: Computer graphics forum, volume 36, Wiley Online Library, 2017, pp. 133–159.
[64] J. Sillito, G. C. Murphy, K. De Volder, Asking and answering questions during a programming change task, IEEE Transactions on Software Engineering 34 (2008) 434–451.
[65] M. Acher, R. E. Lopez-Herrejon, R. Rabiser, Teaching software product lines: A snapshot of current practices and challenges, ACM Transactions on Computing Education (TOCE) 18 (2017) 1–31.
[66] D. A. Boehm-Davis, J. E. Fox, B. H. Philips, Techniques for exploring program comprehension, in: Empirical Studies of Programmers, 1996, pp. 3–37.
[67] R. Koschke, Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, Journal of Software Maintenance and Evolution: Research and Practice 15 (2003) 87–109.
[68] M.-A. D. Storey, D. Čubranić, D. M. German, On the use of visualization to support awareness of human activities in software development: a survey and a framework, in: Proceedings of the 2005 ACM symposium on Software visualization, 2005, pp. 193–202.
[69] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, G. Saake, Do background colors improve program comprehension in the# ifdef hell?, Empirical Software Engineering 18 (2013) 699–745.
[70] B. Shneiderman, The eyes have it: A task by data type taxonomy for information visualizations, in: Proceedings 1996 IEEE Symposium on Visual Languages, IEEE, 1996, pp. 336–343.
[71] F. N. Colakoglu, A. Yazici, A. Mishra, Software product quality metrics: A systematic mapping study, IEEE Access (2021).
[72] A. S. Nuñez-Varela, H. G. Pérez-Gonzalez, F. E. Martínez-Perez, C. Soubervielle-Montalvo, Source code metrics: A systematic mapping study, Journal of Systems and Software 128 (2017) 164–197.
[73] T. J. McCabe, A complexity measure, IEEE Transactions on software Engineering (1976) 308–320.
[74] S. Stevanetic, U. Zdun, 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, 2015, pp. 1–14.
[75] B. Kitchenham, S. L. Pfleeger, Software quality: the elusive target [special issues section], IEEE software 13 (1996) 12–21.
[76] J. Waller, C. Wulf, F. Fittkau, P. Döhring, W. Hasselbring, Synchrovis: 3d visualization of monitoring traces in the city metaphor for analyzing concurrency, in: 2013 First IEEE Working Conference on Software Visualization (VISSOFT), IEEE, 2013, pp. 1–4.
[77] M. Weninger, L. Makor, H. Mössenböck, Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor, in: 2020 Working Conference on Software Visualization (VISSOFT), IEEE, 2020, pp. 110–121.
[78] F. Fittkau, A. Krause, W. Hasselbring, Software landscape and application visualization for system comprehension with ExplorViz, Information and software technology 87 (2017) 259–277.
[79] R. Wettel, M. Lanza, Visual exploration of large-scale system evolution, in: 2008 15th Working Conference on Reverse Engineering, IEEE, 2008, pp. 219–228.
[80] F. Pfahler, R. Minelli, C. Nagy, M. Lanza, Visualizing Evolving Software Cities, in: 2020 Working Conference on Software Visualization (VISSOFT), IEEE, 2020, pp. 22–26.
[81] S. Ardigò, C. Nagy, R. Minelli, M. Lanza, M3tricity: visualizing evolving software & data cities, in: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, 2022, pp. 130–133.
[82] V. Dashuber, M. Philippsen, J. Weigend, A layered software city for dependency visualization., in: VISIGRAPP (3: IVAPP), 2021, pp. 15–26.
[83] V. Dashuber, M. Philippsen, Static and dynamic dependency visualization in a layered software city, SN Computer Science 3 (2022) 1–18.
[84] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. Gonzalez-Barahona, M. Lanza, Codecity: A comparison of on-screen and virtual reality, Information and Software Technology 153 (2023) 107064.
[85] J. Vincur, P. Navrat, I. Polasek, VR City: Software Analysis in Virtual Reality Environment, in: 2017 IEEE international conference on software quality, reliability and security companion (QRS-C), IEEE, 2017, pp. 509–516.
[86] G. Balogh, A. Beszedes, CodeMetropolis-code visualisation in MineCraft, in: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2013, pp. 136–141.
[87] R. Wettel, M. Lanza, R. Robbes, Empirical validation of CodeCity: A controlled experiment, Technical Report, Università della Svizzera italiana, 2010.
[88] R. Wettel, M. Lanza, R. Robbes, Software systems as cities: A controlled experiment, in: Proceedings of the 33rd International Conference on Software Engineering, 2011, pp. 551–560.
[89] J. Kratt, H. Strobelt, O. Deussen, Improving Stability and Compactness in Street Layout Visualizations, in: VMV, 2011, pp. 285–292.
[90] D. Kafura, S. Henry, Software quality metrics based on interconnectivity, Journal of Systems and Software 2 (1981) 121–131.
[91] L. H. Rosenberg, L. E. Hyatt, Software quality metrics for object-oriented environments, Crosstalk journal 10 (1997) 1–6.
[92] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.
[93] G. A. Campbell, Cognitive complexity: An overview and evaluation, in: Proceedings of the 2018 international conference on technical debt, 2018, pp. 57–58.
[94] S. Misra, A. Adewumi, L. Fernandez-Sanz, R. Damasevicius, A suite of object oriented cognitive complexity metrics, IEEE Access 6 (2018) 8782–8796.
[95] D. Sato, A. Goldman, F. Kon, Tracking the evolution of object-oriented quality metrics on agile projects, in: International Conference on Extreme Programming and Agile Processes in Software Engineering, Springer, 2007, pp. 84–92.
[96] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, L. Duchien,

Tracking the software quality of android applications along their evolution (t), in: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2015, pp. 236–247.
[97] R. A. Khan, K. Mustafa, S. I. Ahson, An empirical validation of object oriented design quality metrics, Journal of King Saud University-Computer and Information Sciences 19 (2007) 1–16.
[98] J. Pantiuchina, M. Lanza, G. Bavota, Improving code: The (mis) perception of quality metrics, in: 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2018, pp. 80–91.
[99] G. Rasool, Z. Arshad, A review of code smell mining techniques, Journal of Software: Evolution and Process 27 (2015) 867–895.
[100] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, Journal of Systems and Software 101 (2015) 193–220.
[101] P. C. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, T. Besker, A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, A. Moschou, I. Pigazzini, et al., An overview and comparison of technical debt measurement tools, IEEE Software 38 (2020) 61–71.
[102] N. Chotisarn, L. Merino, X. Zheng, S. Lonapalawong, T. Zhang, M. Xu, W. Chen, A systematic literature review of modern software visualization, Journal of Visualization 23 (2020) 539–558.
[103] R. Wettel, M. Lanza, Visually localizing design problems with disharmony maps, in: Proceedings of the 4th ACM Symposium on Software Visualization, 2008, pp. 155–164.
[104] F. Steinbrückner, C. Lewerentz, Representing development history in software cities, in: Proceedings of the 5th international symposium on Software visualization, 2010, pp. 193–202.
[105] H. Wainer, C. M. Francolini, An empirical inquiry concerning human understanding of two-variable color maps, The American Statistician 34 (1980) 81–93.
[106] D. Holten, R. Vliegen, J. J. Van Wijk, Visual realism for the visualization of software metrics, in: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, 2005, pp. 1–6.
[107] AlDanial, cloc, `https://github.com/AlDanial/cloc`, 2021. Last access 02.12.2021.
[108] N. Peitek, S. Apel, C. Parnin, A. Brechmann, J. Siegmund, Program Comprehension and Code Complexity Metrics: An fMRI Study, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE, 2021, pp. 524–536.
[109] J. Mortara, P. Collet, A.-M. Pinna-Dery, P. Anagonou, G. Savornin, A. van der Tuijn, Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations - Artifact, 2022.
[110] E. Irrazábal, J. A. Carruthers, J. A. Pinto Oppido, 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), 2021.
[111] M. Chen, L. Floridi, R. Borgo, What is visualization really for?, The Philosophy of Information Quality (2014) 75–93.
[112] D. G. Feitelson, Using students as experimental subjects in software engineering research–a review and discussion of the evidence, arXiv preprint arXiv:1512.08409 (2015).
[113] R. Minelli, A. Mocci, M. Lanza, 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, 2015, pp. 25–35.
[114] M. Nachtigall, M. Schlichtig, E. Bodden, A large-scale study of usability criteria addressed by static analysis tools, in: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 532–543.
[115] S. Y. Chyung, K. Roberts, I. Swanson, A. Hankinson, Evidence-based survey design: The use of a midpoint on the likert scale, Performance Improvement 56 (2017) 15–23.
[116] J. T. Nadler, R. Weston, E. C. Voyles, Stuck in the middle: the use and interpretation of mid-points in items on questionnaires, The Journal of general psychology 142 (2015) 71–89.
[117] R. Garland, The mid-point on a rating scale: Is it desirable, Marketing bulletin 2 (1991) 66–70.
[118] M. S. Matell, J. Jacoby, Is there an optimal number of alternatives for likert scale items? study i: Reliability and validity, Educational and psychological measurement 31 (1971) 657–674.
[119] A. J. Ko, T. D. LaToza, M. M. Burnett, A practical guide to controlled experiments of software engineering tools with human participants, Empirical Software Engineering 20 (2015) 110–141.
[120] J. Mortara, P. Collet, A.-M. Dery-Pinna, IDE-assisted visualization of indebted OO variability implementations, in: Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume B, SPLC '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 74–77.
[121] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.
[122] W. Huang, P. Eades, S.-H. Hong, Measuring effectiveness of graph visualizations: A cognitive load perspective, Information Visualization 8 (2009) 139–152.
[123] P. Caserta, O. Zendra, D. Bodénes, 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, 2011, pp. 1–8.
[124] C. Kästner, S. Trujillo, S. Apel, Visualizing Software Product Line Variabilities in Source Code, in: SPLC (2), 2008, pp. 303–312.
[125] S. Duszynski, M. Becker, Recovering variability information from the source code of similar software products, in: 2012 Third International Workshop on Product LinE Approaches in Software Engineering (PLEASE), IEEE, 2012, pp. 37–40.
[126] O. Greevy, M. Lanza, C. Wysseier, Visualizing feature interaction in 3-d, in: 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, IEEE, 2005, pp. 1–6.
[127] K. Zhang, Software Visualization: From Theory to Practice, volume 734, Springer Science & Business Media, 2003.
[128] P. Caserta, O. Zendra, Visualization of the static aspects of software: A survey, IEEE Trans. Vis. Comput. Graph. 17 (2011) 913–933.
[129] C. Gravino, G. Scanniello, G. Tortora, Source-code comprehension tasks supported by UML design models: Results from a controlled experiment and a differentiated replication, J. Vis. Lang. Comput. 28 (2015) 23–38.
[130] G. Scanniello, C. Gravino, M. Genero, J. A. Cruz-Lemus, G. Tortora, M. Risi, G. Dodero, Do software models based on the UML aid in source-code comprehensibility? aggregating evidence from 12 controlled experiments, Empir. Softw. Eng. 23 (2018) 2695–2733.
[131] A. Kuhn, D. Erni, P. Loretan, O. Nierstrasz, Software cartography: Thematic software visualization with consistent layout, Journal of Software Maintenance and Evolution: Research and Practice 22 (2010) 191–210.
[132] N. Hawes, S. Marshall, C. Anslow, Codesurveyor: Mapping large-scale software to aid in code comprehension, in: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), IEEE, 2015, pp. 96–105.
[133] A. Schreiber, M. Misiak, Visualizing software architectures in virtual reality with an island metaphor, in: Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation: 10th International Conference, VAMR 2018, Held as Part of HCI International 2018, Las Vegas, NV, USA, July 15-20, 2018, Proceedings, Part I 10, Springer, 2018, pp. 168–182.
[134] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, J. Laviola,

Code Park: A new 3D code visualization tool, in: 2017 IEEE Working Conference on Software Visualization (VISSOFT), IEEE, 2017, pp. 43–53.

[135] Z. Zhang, B. Qin, Y. Chen, L. Song, Y. Zhang, VRLifeTime–An IDE Tool to Avoid Concurrency and Memory Bugs in Rust, in: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, 2020, pp. 2085–2087.

[136] A. Shokri, M. Mirakhorli, ArCode: A Tool for Supporting Comprehension and Implementation of Architectural Concerns, arXiv preprint arXiv:2103.06735 (2021).

[137] A. Telea, L. Voinea, An interactive reverse engineering environment for large-scale c++ code, in: Proceedings of the 4th ACM symposium on Software visualization, 2008, pp. 67–76.

[138] M. Dias, D. Orellana, S. Vidal, L. Merino, A. Bergel, Evaluating a Visual Approach for Understanding JavaScript Source Code, in: Proceedings of the 28th International Conference on Program Comprehension, 2020, pp. 128–138.

[139] S. Entekhabi, A. Solback, J.-P. Steghöfer, T. Berger, Visualization of Feature Locations with the Tool FeatureDashboard, in: Proceedings of the 23rd International Systems and Software Product Line Conference-Volume B, 2019, pp. 1–4.

[140] T. Schwarz, W. Mahmood, T. Berger, A common notation and tool support for embedded feature annotations, in: Proceedings of the 24th ACM International Systems and Software Product Line Conference-Volume B, 2020, pp. 5–8.

[141] J. Martinson, H. Jansson, M. Mukelabai, T. Berger, A. Bergel, T. Ho-Quang, HAnS: IDE-based editing support for embedded feature annotations, in: Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume B, 2021, pp. 28–31.

[142] T. Savage, M. Revelle, D. Poshyvanyk, Flat3: Feature location and textual tracing tool, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2, 2010, pp. 255–258.

[143] M. V. Couto, M. T. Valente, E. Figueiredo, Extracting software product lines: A case study using conditional compilation, in: 2011 15th European Conference on Software Maintenance and Reengineering, IEEE, 2011, pp. 191–200.

[144] W. Ji, T. Berger, M. Antkiewicz, K. Czarnecki, Maintaining feature traceability with embedded annotations, in: Proceedings of the 19th International Conference on Software Product Line, 2015, pp. 61–70.

[145] M. Seiler, B. Paech, Documenting and exploiting software feature knowledge through tags., in: SEKE, 2019, pp. 754–777.

[146] J. Mortara, P. Collet, A.-M. Dery-Pinna, P. Anagonou, P.-M. Djekinnou, F. Focas, F. Rigaut, G. Savornin, A. van der Tuijn, Visualisation of object-oriented software in a city metaphor: comprehending the implemented variability and its technical debt — Reproduction Package, 2023. URL: `https://doi.org/10.5281/zenodo.7687914`.