

Visualization of Object-Oriented Variability Implementations as Cities

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

▶ **To cite this version:**

Johann Mortara, Philippe Collet, Anne-Marie Dery-Pinna. Visualization of Object-Oriented Variability Implementations as Cities. 9th IEEE Working Conference on Software Visualization (VISSOFT 2021), Sep 2021, Luxembourg (virtual), Luxembourg. 10.1109/VISSOFT52517.2021.00017 . hal-03312487

HAL Id: hal-03312487
https://hal.science/hal-03312487

Submitted on 2 Aug 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Visualization of Object-Oriented Variability Implementations as Cities

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

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

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

***Abstract*—Many large software systems are variability-rich, object-oriented, and implemented in a single code base. They then rely on multiple traditional techniques (inheritance, patterns) to realize variability, making these implementations not explicit. This directly hampers the comprehension of variability implementations, especially for newcomers in a project that need, in a short time, to understand the most important parts. In this paper, we propose *VariCity*, a visualization using the city metaphor to exhibit zones of interest, being zones of high density of variability implementations. The different forms of variability implementations are first detected through the usage of symmetries in code (*e.g.*, inheritance defines a substitution symmetry between the immutable part of the superclass and the possible changes in its subclasses). *VariCity* then creates a 3D city representation with buildings being classes while the metrics on the number of symmetries (*e.g.*, the number of overloaded methods, influence the building size, and their color if they are heavily loaded in symmetries). Contrary to the usual package-based organization in code-related city representations, the city streets are arranged according to the usage relationships between classes. Inheritance is simply represented with hoverable aerial links. Variability-related design patterns are depicted as buildings with specific geometric forms, while some classes specified as entry points can help in shaping the whole city organization. We also report on the evaluation of *VariCity* on a set of large object-oriented systems, showing that several usage scenarios helping a newcomer to spot critical variability-related zones are covered.**

***Index Terms*—variability, software visualization, software cities**

I. Introduction

Recent software-intensive systems of all scales and domains are more and more variability-intensive [1]–[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]. Being a key element of most systems [1], variability management has been heavily studied, notably leading to the Software Product Line (SPL) paradigm [5], [6] Within an SPL, there is a clear separation between the domain variability, commonly documented and managed in terms of features (often organized in a feature model [7]), and the implemented variability, that is mapped from the domain variability, usually using a single implementation technique such as preprocessor directives [8] or a form of modules [6]. Sometimes the implemented variability is also managed as a feature model [9], [10], but the main benefits of an SPL is to reason on consistency at the domain level and from a configuration, to derive a consistent software product.

However, many variability-rich software systems are not 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 implemented using traditional techniques (*i.e.*, inheritance, parameters, overloading, and some design patterns such as strategy and factory) [4], [11], [12]. The implemented variability in code assets is neither explicit nor documented, which hinders its management, but more basically, hampers the simple comprehension of it. As features to be understood are not known in advance, and the code is not cloned and modified per product, none of the feature location techniques [13]–[15] can be applied in this context.

Facing this comprehension problem in identifying variability implementations, we advocate that it demands a visualization based solution. Software visualization is a research area that focuses on methods and techniques for graphically representing the many facets of software [16]–[18]. In more than two decades, many visualization approaches have been proposed to support a diversity of software engineering activities, such as maintenance, evolution, reverse engineering [19], and more generally software comprehension and analysis [20], [21]. In the variability management field, visual representations most entirely focus on the domain variability and feature models [14]. When a visualization related to implementation is provided, it is only dedicated to the validation of a detection approach and not really adapted [22], [23].

In this paper, we propose *VariCity*, a visualization using the city metaphor [24] to exhibit zones of interest, being zones of high density of variability implementations. As the city metaphor has been shown to scale on large projects for visualizing metrics related to software quality [25]–[27], we adapt it to our identification problem.

Reusing a recent approach [22], different forms of variability implementations are first detected through the usage of symmetries in code [28], [29]. For example, inheritance defines a substitution symmetry between the immutable part of the superclass and the possible changes in its subclasses. These symmetries also appear in mechanisms such as overloading of constructors and methods, and patterns such as factories and strategies. All the occurrences of these symmetries are thus

detected, gathered as metrics (*e.g.*, the number of overloaded methods in a class), and complemented with information on inheritance and usage relationships between classes.

VariCity then creates a 3D city representation with buildings being classes while the metrics on the number of symmetries influence the building size, and their color if they are heavily loaded in symmetries. Contrary to the usual package-based organization in code-related city representations, the city streets are arranged according to the usage relationships between classes. Inheritance is simply represented with hoverable aerial links. Variability-related design patterns are depicted as buildings with specific geometric forms, while some classes specified as entry points can help in shaping the whole city organization. As variability implementations to be identified are close to elements being searched for in onboarding activities [30]–[32], we organize the usage of *VariCity* around scenarios based on these activities. This consists of facilitating the comprehension for a skilled newcomer while providing configuration capabilities for the expert to create adapted views for newcomers. As a result, we show that *VariCity* can facilitate both an high-level discovery of the variability implementations within a codebase, and a deep understanding in some specific areas. These usage scenarios have been validated over a set of ten medium to large object-oriented systems.

The remainder of this paper is organized as follows. Section II introduces concepts of object-oriented variability implementations and defines the visualization requirements based on onboarding scenarios. Section III gives some background on symmetry-based detection of object-oriented variability implementations, as well as on the city metaphor in software visualization. We then introduce the main principles and configurable views of *VariCity* in section IV. In section V we rely on the defined usage scenarios to evaluate the capabilities of *VariCity*. Threats to validity and limitations are discussed in section VI, while section VII studies related work. Finally, section VIII concludes this paper and briefly discusses future work.

II. Motivations

Many object-oriented software systems are variability rich but do not follow a fully-fledged software product line approach [5], [6]. Consequently, their domain variability (*i.e.*, features) is not very well documented and is not made explicit within code assets. 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 [33], or conducting an onboarding process for newcomers [34].

A. Object-oriented variability implementations

When variability is present in object-oriented systems, code assets can be structured into three different parts: core, commonalities, and variations [35]–[37]. The core part corresponds to assets that are included in any of the final software products [35]. A commonality is the common part between the related variations of code assets, while variations indicate how and when should code assets vary [1]. Such commonalities and variations are usually abstracted in terms of variation points (*vp*-s) and variants, respectively [38]–[40]. A variation point thus identifies one or more locations at which the variation will occur, while the way that a variation point is going to vary is expressed by its variants [38]. They are both related to concrete elements in code assets [41].

Usually, in object-oriented variability-rich systems implemented within a single codebase, the implementation of these concrete elements rely on different traditional techniques, such as inheritance, parameters, constructor and method overloading, or some software design patterns [4], [11], [12], [42]. In this context, the code units that structure the systems are classes, and they do not align well with domain features [6], [43].

To comprehend implementation variability, SPL migration techniques could be used in a reverse or forward engineering way. Their approaches are characterized as feature location, feature identification [13], [44], [45], feature delimitation (with annotations) [8], or feature modularization [6]. In these approaches, features commonly tend to describe the domain variability of an SPL or variability-rich system, but are required to be known in advance [10], [40]. However, domain variability is hardly documented in variability-rich systems [46], and with a single codebase, reengineering of features from clones of a system cannot be used [47]. As a result, migrating requires substantial manual effort and implies a complete paradigm shift.

While many studies relate on how to address variability with traditional object-oriented techniques [11], [12], [48], [49], identifying *vp*-s with variants [50] implemented with these techniques in a single codebase is known to be hard, by the diversity of the implementations [42], [50] and the lack of adapted visualization [22]. Moreover, according to a recent mapping study by [14], while many visual representations of variability management approaches are proposed in the context of SPL, they most often target domain variability (*e.g.*, features in a feature model).

As a result, while identifying the variability implementations directly in code assets, that is, variation points and their variants, is the first activity to comprehend variability, this activity has no dedicated support.

B. Requirements

As program comprehension is seen as a process of both information seeking [51] and feature location [13], 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 [34], and tools are essential to illustrate software reuse concepts [52]. We then first advocate that this

context naturally calls for visualization-based solutions [19]–[21]. As the essence of software visualization consists of creating an image of software by 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. Furthermore, the difficulty of discovering a codebase increases with its complexity, we believe that such visualization should be able to meet the constraints of an onboarding process. Onboarding is a case of program comprehension in which a new developer joins a project or a company [30], [31]. Contrary to the usual information seeking in program comprehension (*i.e.*, information pull), onboarding is more based on information push [53] and is harder when little is known about the system [54]. In onboarding, it has also been shown that newcomers look for major patterns [53], such as the ones used in variability implementations. Finally, to avoid frustration by newcomers being onboarded [55], the capability to configure and make up adapted visualization for an expert is also crucial.

In this context, we structure our requirement analysis around software comprehension scenarios for visualization within an onboarding process. We are then supposed to target two types of users:

- **newcomers** in the project, skilled but with no real knowledge about the code (this role can be generalized to anyone attempting to comprehend some software with little or no prior knowledge);
- **experts** in the project, with knowledge of the code and its architecture, but with no explicit vision of the variability implementations. With experts, once they have gained knowledge on the variability, they are likely to be more interested in its evolution [25], [56]. We consider that all evolution scenarios are out of the scope of this paper, as we first need to provide a visualization for a single snapshot of a project. Consequently, we focus on scenarios that engage the expert to comprehend the implemented variability while building a preconfigured visualization for newcomers.

We then propose two scenarios:

- *Scenario 1*: **The expert wants to facilitate the exploration of the codebase by giving a pre-configured visualization to the newcomer.** Through this scenario, the newcomer onboards on a large codebase of which he needs to have a global comprehension of the implemented variability (*e.g.*, understand a library or API that is going to be reused).
- *Scenario 2*: **The expert wants the newcomer to comprehend a subpart of the codebase for the newcomer to be able to reuse it.** Through this scenario, the newcomer onboards on a codebase in which she will be asked to add a new feature. She, therefore, has to understand in more detail the interactions between the classes implemented variability in this subpart.

Finally, Yates et al. [53] analyze the different types of information transmitted from an expert to a newcomer during onboarding sessions. It results that newcomers find helpful when experts give coarse-grained information about complex zones (ranging from a group of classes to design patterns) of the codebase to them, so they can dig into them by themselves. According to these findings, we can say that a visualization for a newcomer should: (*i*) display the main elements allowing her to understand the codebase (design patterns, zones with complex variability implementations), (*ii*) be configurable by the expert to tailor it for newcomers, (*iii*) provide navigation and interaction capabilities to be adapted by a newcomer (filtering, zooming), (*iv*) scale on large codebases.

III. Background

A. Object-oriented symmetries and variability

While symmetry in nature is often defined as the immunity to a possible change, this concept has also been studied in software, and especially in mechanisms of object-orientation, such as inheritance, overloading, and design patterns, which can also be interpreted in terms of symmetry [28], [29]. Taking a codebase as a whole, these implementation techniques can be seen as local symmetries [22], which allow a part of code to change while another part remains unchanged.

As an illustration, let us take an example from JFreeChart[1], an object-oriented library that provides a variability-rich family of charts (*e.g.*, meters, pies). In the left part of Fig. 1 is depicted a partial UML diagram of some important classes in this library, namely the abstract class `Plot` with its two subclasses `PiePlot` and `MeterPlot`. As inheritance defines a *substitution symmetry* for its subtypes [28], the *possibility of a change* in `Plot` corresponds to its different subtypes, such as `PiePlot` and `MeterPlot` (which vary in how they draw a chart). On their side, these classes *preserve* and conform to the common behavior of their superclass. Furthermore, the main object-oriented (OO) techniques implementing variability can be characterized by local symmetries (*e.g.*, constructor and method overloading, factory or strategy pattern), and at an abstract level, a *vp* represents the *unchanged* part while its variants are the *changed* parts in code assets of a system [22].

B. Automatic identification of variability implementations

Thanks to the available symmetries in object-oriented variability implementation techniques, its has been shown that seven different techniques can be detected (*class as type*, *class subtyping*, *method and constructor overloading*, *strategy*, *template*, *decorator*, and *factory patterns*) inside Java or C++ code [22], [57], [58]. It has also been observed that they represent good potential *vp*-s and variants [22], and that they can even be successfully mapped to domain features if a list of these features is provided [23]. In our example, the *vp*-s are present at both class and method levels, as shown on the UML diagram in Fig. 1. At the class level, the abstract class `Plot` is a superclass of `PiePlot` and `XYPlot`, making it a *vp* with two variants. At the method level, every overloaded constructor or method represents a *vp*, with the number of

[1] http://www.jfree.org/jfreechart/

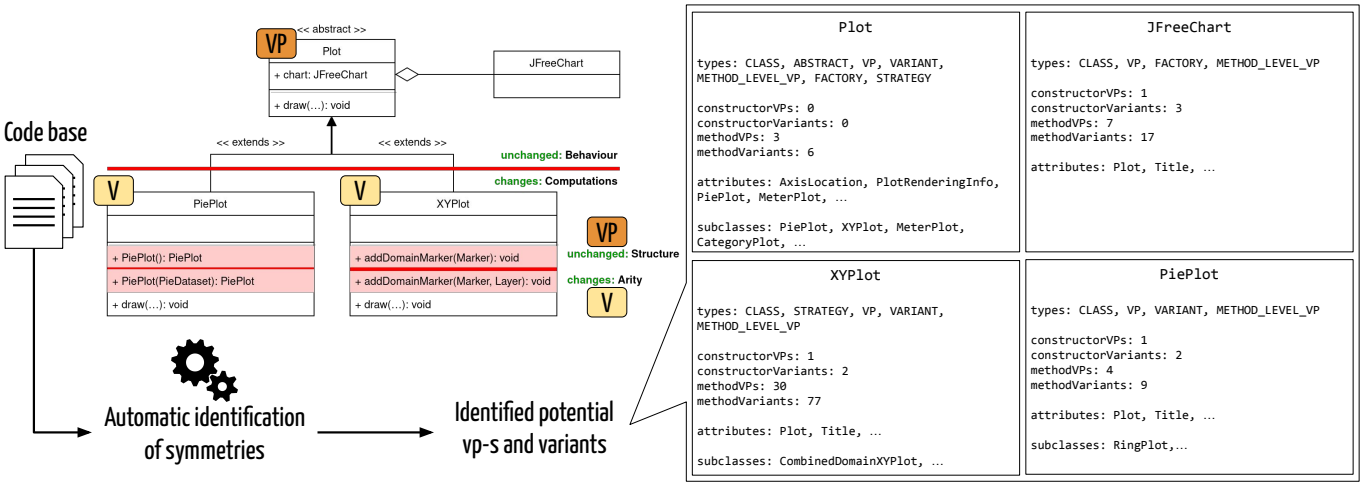


Fig. 1: Symmetries in object-oriented code and metrics that can be extracted

overloads being the number of variants of the *vp*. The two constructors in `PiePlot` and the two `addDomainMarker` methods in `XYPlot` both represent a *vp* with two variants. Information about the presence of a *vp* appear in the extracted information (on the right of Fig. 1): class level *vp*-s are labeled as `VP`, variants as `VARIANT`, and classes possessing method level *vp*-s are labeled `METHOD_LEVEL_VP`. Finally, if the class is the *vp* of a design pattern, a label with the pattern name is associated with the class information.

Besides, it has also been shown that the density of the variability implementations in a location denotes zones of interest in terms of variability [22]. In other words, zones with several techniques used, and/or with heavy usage of a technique (*e.g.*, a lot of methods being overloaded, a lot of subclasses in a hierarchy, or inside a strategy pattern) are very likely to be a zone of variability management.

Furthermore, if a class contains many usages of variability implementations, it is also naturally related by inheritance to some others, especially as the inheritance mechanism is the backbone of many variability implementation techniques [22]. More recently, the importance of usage relationships between classes, defined through typing of attributes and method parameters, has also been demonstrated to determine a zone of variability interest [59]. As a result, with information on symmetries, metrics on their occurrences, as well as relations of inheritance and usage between classes, zones of interest in terms of variability can be predetermined. For example, the metrics on the right part of Fig. 1 indicate that `XYPlot` possesses 30 overloaded methods (`methodVPs`), totalizing 77 overloads (`methodVariants`), making it a highly-variable class[2].

Taking all the information and metrics on potential *vp*-s and variants that can be easily extracted from an object-oriented code base, our aim with *VariCity* is thus to propose a visualization that meets the usage scenarios devised in section II-B.

[2]Since constructors all have the name of their class, multiple constructors represent only one *vp* (`constructorVPs`), with the number of constructors being the number of variants (`constructorVariants`).

C. On the city metaphor in software visualization

Metaphors are often used when designing visualizations as they bring an understandable graphical representation to concepts [17], such as the metaphor of the city [60], which has been applied to multiple types of metrics on software systems: dynamic behavior (such as concurrency between classes [61], memory consumption of heaps [62]), and static properties such as dependency and communication links between components [63].

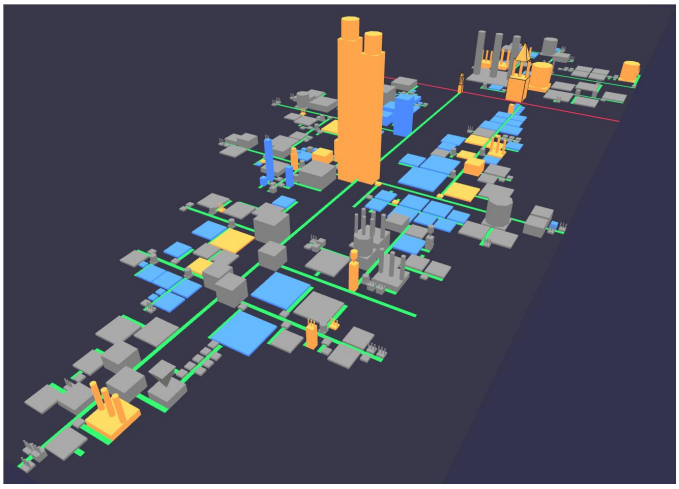
At a finer-grain, software cities to understand object-oriented software systems have been proposed, the first of them being CodeCity [24], [27] which uses buildings to represent classes, grouping them in districts representing packages. These principles were enhanced by adding a temporal dimension in the analysis to visualize the evolution of the metrics through multiple versions of the system, first in CodeCity [56] and also in a more recent approach called M3TRICITY [25]. The Evo-Streets [64] approach also aims at visualizing the evolution of the software but uses streets to represent the package decomposition (instead of nested boxes in CodeCity). Multiple approaches also reuse the city metaphor and adapt it to more immersive techniques, as virtual reality in VRCity [65] or Minecraft in CodeMetropolis [66].

IV. VARICITY

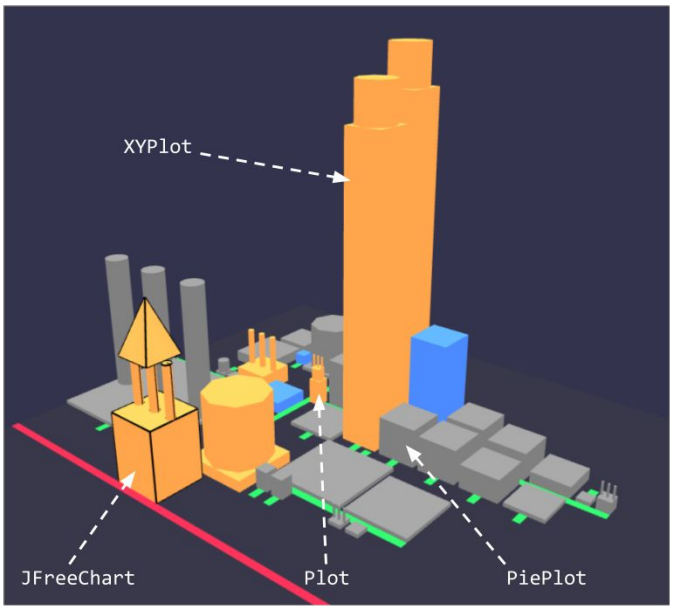
As shown in section III-C, the city metaphor is a recognized way to visualize different properties of software systems. We hence adapt this visualization to the data we want to visualize and the defined scenarios.

A. Main principles

Buildings. In CodeCity, classes are buildings and their size evolves 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). For example,



(a) Visualization of JFreeChart 1.5.0



(b) Example from Fig. 1 in *VariCity*

Fig. 2: Sample views of *VariCity*

an important number of methods will lead to a tall building, catching the attention of the user on it. In *VariCity*, we aim to focus the user on classes making heavy use of variability implementations. Therefore, the dimensions of every building represent the class-based metrics related to variability (*i.e.*, the number of variants at method level – a tall building shows an important number of method variants, whereas a large building shows an important number of constructor variants). Moreover, buildings in color on the visualization (by default yellow for *vp*-s and blue for non *vp*-s) represent classes defined as *hotspots*. 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[3] (*cf.* table I).

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 to exhibit the packages containing the most complex classes, our goal is to group in the same neighborhood classes concentrating a high density of variability implementations.

However, 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 [64], 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.

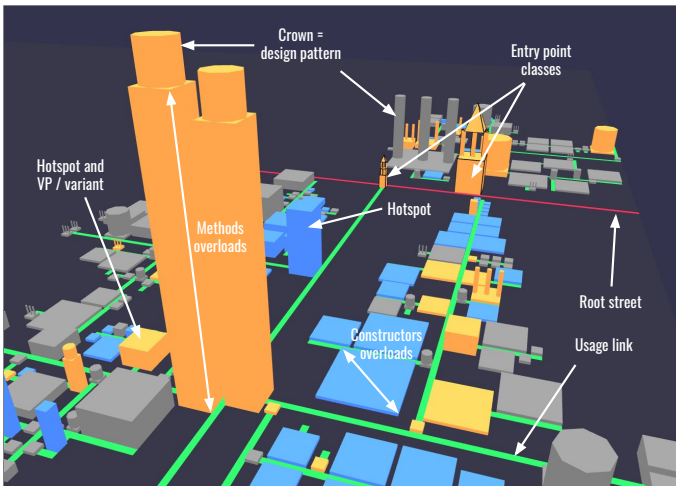
A summary of the visual properties is presented in table I and illustrated in Fig. 3.

TABLE I: Visual properties and their default color

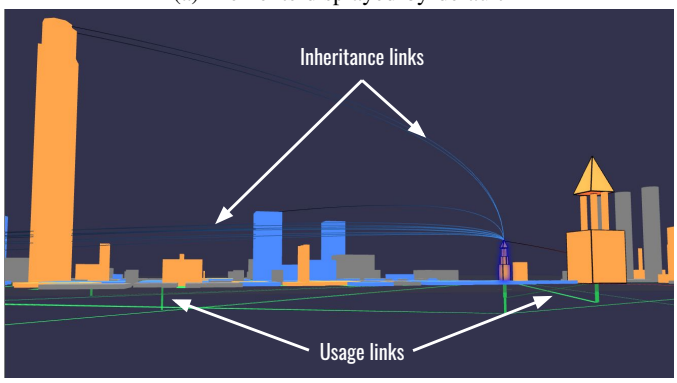
Representation in *VariCity*	**Signification**
Buildings	
Yellow color	Variation point that is part of a hotspot
Blue color	Non *vp* class that is part of a hotspot
Gray color	Class that is not part of a hotspot
Pyramide crown	Entry point class
Dome crown	Strategy pattern
Chimneys crown	Factory pattern
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

Adaptable cities. While the view should allow to quickly spot dense zones of variability implementations, a lot of

[3]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.



(a) Elements displayed by default



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

Fig. 3: Visual properties of *VariCity*

information of different nature needs to be displayed: classes, links between them, design patterns. However, on a large project, providing a first view with all classes (and their usage / inheritance relationships) displayed would bring too much information. There is thus a need to focus the visualization around known points of interest of the system. The idea is therefore to allow the expert to create a city in line with the most important elements for her and to give a first simplified vision of the city which does not show all the relationships between classes. The visualization algorithm thus relies on a certain number of inputs that focus the view (*cf.* section IV-B). From this first visualization, it will also be possible to gradually adapt the city, among other things, by adding or removing relationships and classes (*cf.* section IV-C).

B. From buildings and streets to a city

The goal of *VariCity* is to display the main elements allowing one to understand the variability implementations related to a given point of interest in the system (*cf.* section II-B). To do so, *VariCity* relies on three mandatory inputs. The first one defines *entry point* classes, which represent important points of interest for the comprehension of the system (*e.g.*, endpoint of an API that could be automatically inferred, or

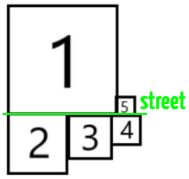


Fig. 4: Result example of the placement algorithm

complex classes of the system given by the expert). The second input is the *usage orientation*, which can be `IN` and/or `OUT`. 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). 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). Depending on the objectives of the onboarding scenario envisaged by the expert, she might show either how the entry point uses or is used by other classes. More detailed examples are given in section IV-C. Finally, setting the orientation to `IN/OUT` displays classes using or being used by the entry points. The third input is the *usage level*, which 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.

The root (first) street, in red on Fig. 3a, 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 (Fig. 4).

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 [67]. 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 [4].

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

An example of visualization after generation is presented in Fig. 3a. Additional links appearing on hover are presented in Fig. 3b.

C. Configuring the view to adapt the city

The configuration of *VariCity* is done in two steps. The first step concerns the adaptation of the mandatory inputs required to build the visualization (*i.e.*, entry point classes, usage level, and usage orientation), which are preconfigured by the expert. Based on her knowledge, the expert determines which classes are relevant enough to be entry point classes. The orientation will be set depending on what she expects the newcomer to understand from the system: if she wants the newcomer to reuse a part of the implementation, she will likely choose the `IN` orientation as it will show which classes already use the entry point so that the newcomer can see how the class is already used. On the opposite, if she wants the newcomer to add a new feature, she will more likely choose `OUT` so that the newcomer sees which classes are used by the entry points to know which classes she may need to reuse. Finally, choosing `IN/OUT` gives an overview of both aspects. 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. Such characteristics are dependent on every codebase. For example, the visualization of JFreeChart presented in Fig. 2a has `JFreeChart` and `Plot` as entry points, a usage level of 4, and a usage orientation `OUT`. The expert can also choose not to display classes that she considers irrelevant by putting them in a *blacklist*.

The second step represents options allowing 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 might be useful to use this metric for the height instead of the width of the buildings.

Although all these parameters for both steps have default values set by the expert, they can also be adapted by the newcomer while exploring the visualization in a sidebar to maximize her autonomy. We will illustrate in section V how different values for the inputs in the first step impact the structure of the visualization by detailing the two scenarios presented in section II-B.

D. Implementation

VariCity implementation relies on an existing toolchain complemented by its dedicated visualization.

The automatic identification of the symmetries and, relying on them, of the potential *vp*-s and variants, depicted in Fig. 1 is done by reusing *symfinder* [57]. After cloning the Git repository of a single Java or C++ codebase, *symfinder* identifies the symmetries present in the code and stores their representation in a graph database to perform the identification of the potential *vp*-s and variants. Information is structured by class and used by *VariCity* to build the visualization (*cf.* section IV-B), relying on the settings provided in a configuration file. The web-based visualization is standalone, and developed in TypeScript with the Babylon.js[5] 3D library. The whole application is deployed with Webpack and requires only a web browser to be viewed. Both *VariCity* and *symfinder* are deployed using Docker to ease their reuse and reproducibility of the visualizations presented in this paper. The source code of *VariCity* is available online [68].

V. Evaluation

In [59], the *symfinder* toolchain, which detects potential *vp*-s with variants, was applied on ten popular open-source and variability-rich Java systems, being applications, framework, or libraries, with different characteristics (size, variation points, explicit API provided). We chose to select the same systems to test the results of *VariCity*. In table II are listed the systems and their *VariCity* configuration to facilitate the exploration or deepening of a particular area, as shown by our scenarios. The entry points have been determined by exploring the codebases and documentations, and selecting important classes accordingly. The values for usage level and orientation were determined empirically to provide a visualization showing interesting zones. By tailoring the inputs for these systems, we show that our approach is applicable to systems of various sizes and structures. The generated cities for all systems are available in the reproduction package [68]. Entry point classes being preconfigured, the user just needs to adapt the values for the usage level and orientation.

In this section, we evaluate whether *VariCity* answers to the needs expressed in section II-B, relying on the scenarios presented in section IV-C. We chose the Apache NetBeans IDE with its 5 MLoC[6] for Scenario 1 to illustrate the exploration of a large codebase. We chose the JFreeChart charting library for Scenario 2 to illustrate comprehension for reuse, as this scenario requires a finer-grained knowledge of the codebase, and we already detailed parts of its variability implementations in previous work [22]. A video walkthrough of the scenarios is available on *VariCity*'s website at https://deathstar3.github.io/varicity-demo/.

A. Scenario 1: exploration of the codebase

Objectives: With this scenario, we want to evaluate how *VariCity* and its configuration capabilities can help to distinguish zones of high density of variability in a codebase, which are manifested by buildings of particular height or width (*i.e.*, important number of method level variability implementations), in color (*i.e.*, part of dense zones of variability implementations), or with a crown (*i.e.*, presence of a design pattern).

[5]https://www.babylonjs.com/

[6]https://netbeans.apache.org/

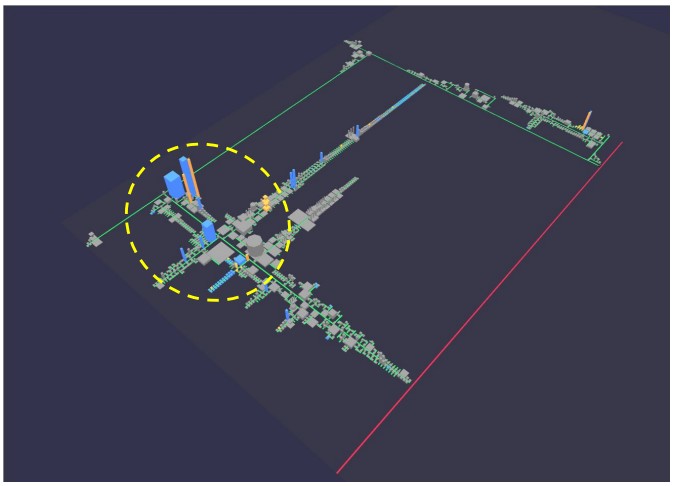
TABLE II: Subject systems

Entry point(s)	Usage level	Usage orientation
Java AWT		
`java.awt.Shape`	3	IN/OUT
Apache CXF		
`org.apache.cxf.endpoint.Endpoint`	6	OUT
JUnit		
`org.junit.Assert` `org.junit.rules.TestRule`	3	IN/OUT
Maven		
`org.apache.maven.Maven` `org.apache.maven.execution.MavenSession`	7	OUT
JFreeChart		
`org.jfree.chart.JFreeChart` `org.jfree.chart.plot.Plot`	2	OUT
ArgoUML		

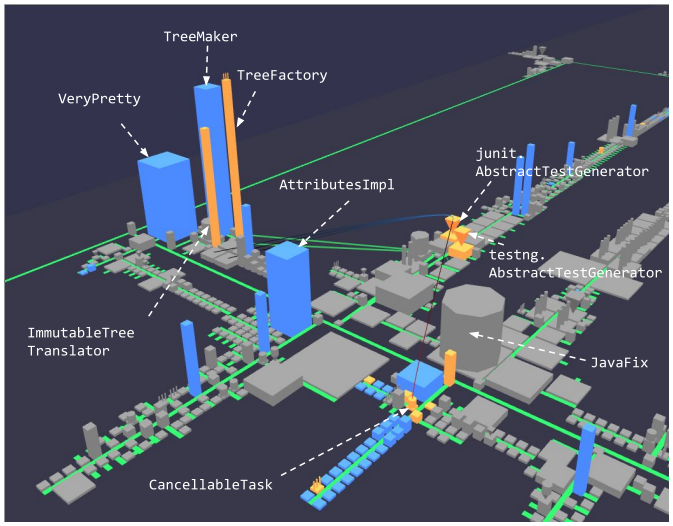
`org.argouml.cognitive.Designer`
`org.argouml.uml.ui.UMLModelElementListModel2`
`org.argouml.uml.diagram.ui.FigNodeModelElement`
	2	IN/OUT

Cucumber		
`io.cucumber.plugin.event.Event` `io.cucumber.java.StepDefinitionAnnotation`	11	IN/OUT
Logbook		
`org.zalando.logbook.Logbook` `org.zalando.logbook.Sink`	4	OUT
Riptide		
`org.zalando.riptide.Http`	6	IN/OUT
NetBeans		
`org.netbeans.api.java.platform.JavaPlatform`	5	IN/OUT

Unfolding the scenario: The newcomer onboards on the NetBeans IDE code base and needs to use the JavaPlatform API, which configures the version and location of Java to be used when building and running a project[7]. To better understand the operation of the API, the newcomer thus needs to have a global vision of the structure of the usage and inheritance relationships between the classes. To this effect, the expert configures the visualization to use the endpoint of the API, namely `JavaPlatform`[8], as the entry point of the visualization. Both classes using and being used by `JavaPlatform` on 5 levels (usage level 5, orientation `IN` and `OUT`) are shown to have a first overview of the classes being closely related to the endpoint of the API. The obtained visualization is shown in Fig. 5a. A neighborhood of tall and colored buildings (circled in yellow) detaches from the other buildings in the city, showing to the user zones with classes heavily using variability implementation techniques. By zooming and spanning the visualization, the user can focus on this precise part of the city (Fig. 5b)[9]. The different implemented design patterns are distinguishable due to the special shape of their buildings (*e.g.*, `JavaFix` is a Strategy, `ngtest.AbstractTestGenerator` and `junit.AbstractTestGenerator` 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[10] and TestNG[11] and are variants of the `CancellableTask` interface[12].



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



(b) Zoom on a hotspot zone

Fig. 5: Visualization of the package `java` of NetBeans 12.2

B. Scenario 2: comprehension of a subpart of the codebase for reuse

Objectives: With this scenario, we want to evaluate how the customization of the view by the newcomer can allow her to tailor the visualization to obtain fine-grained details about the codebase.

[7]https://bits.netbeans.org/12.2/javadoc/org-netbeans-modules-java-platform/overview-summary.html

[8]`org.netbeans.api.java.platform.JavaPlatform`

[9]The names and arrows have been manually added on the figure. The name of the class corresponding to a building appears in a sidebar of the visualization when hovering over the building. Packages, when unnecessary, have been omitted for readability.

[10]https://junit.org/junit5/

[11]https://testng.org/doc/

[12]See here and here.

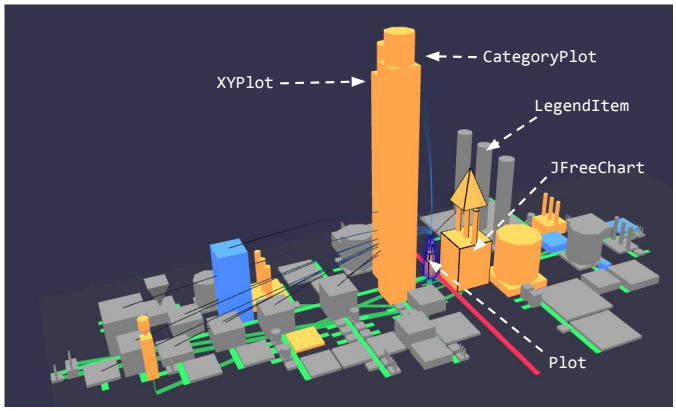
Unfolding the scenario: The newcomer onboards on JFreeChart, a Java library allowing to draw different types of charts, and is asked to implement a new type of chart in the library. Contrary to the first scenario, the newcomer aims at adding a new feature to the codebase, thus she needs a more fine-grained understanding of it, as, for example, the classes used by the other charts that she might also need to use. The expert thus configures the visualization to use as entry points `JFreeChart`[13], being the endpoint of the library used by the users to create plots and `Plot`[14], the superclass of all classes implementing a different type of chart. As the goal is to display which classes are used by these two entry points, the usage orientation is set to `OUT` and the usage level to 2. The obtained visualization is shown in Fig. 6a.

We notice that the colored buildings, which represent classes being part of dense zones of variability implementations, do not align with the most variable classes. For example, `LegendItem` is a factory and, due to its large base, exhibits an important number of constructor variants. However, although this class is internally dense in variability, it is a utility object which not related to any other *vp*, and for this reason, is not characterized as a hotspot.

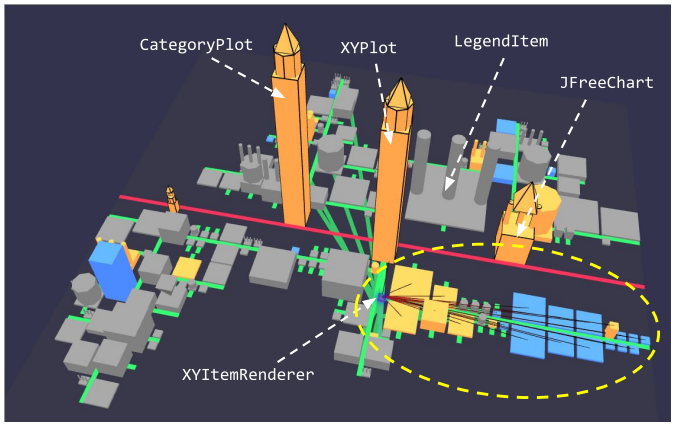
By hovering over `Plot`, the newcomer can see the different displayed subclasses of the class (*i.e.*, the variants of the *vp* `Plot`). To add another type of chart in the library, she will need to implement a new variant of this *vp* and needs thus to have an overview of the classes used by these subclasses. To do so, the user adds the two most variable ones (`XYPlot`[15] and `CategoryPlot`[16]) as entry points (Fig. 6b). 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 Fig. 6b, an important number of classes making heavy use of variability implementations is visible, and are directly used by `XYItemRenderer`[17], itself related to both `XYPlot` and classes related to `CategoryPlot`. Given these characteristics, the newcomer may need to reuse it to implement his feature and thus can add it as another entry point to visualize its usage if needed.

To visualize the classes used by `XYPlot` and `CategoryPlot`, the newcomer could also have chosen to increase the usage level on the visualization given by the expert, as shown on Fig. 6c. However, an important number of classes and relationships not related to the newcomer's interest would appear, hampering the comprehension. The newcomer could also have chosen other class variants of `Plot` to add as entry points. However, most likely the classes that will be added to the visualization are not dense in variability, thus less interesting for the scenario.

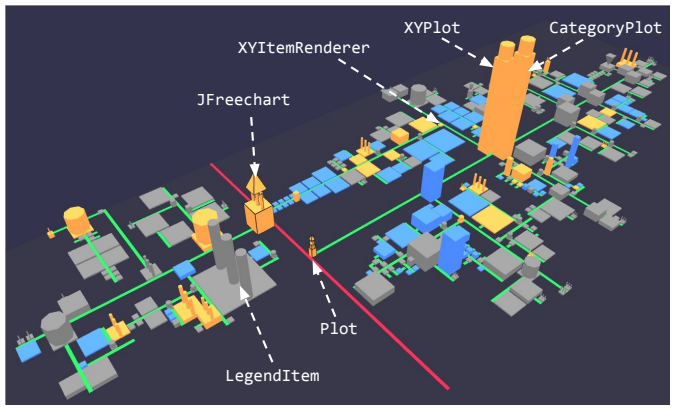
[13] `org.jfree.chart.JFreeChart`
[14] `org.jfree.chart.plot.Plot`
[15] `org.jfree.chart.plot.XYPlot`
[16] `org.jfree.chart.plot.CategoryPlot`
[17] `org.jfree.chart.renderer.xy.XYItemRenderer`



(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) Fig. 6a after adding `XYPlot` and `CategoryPlot` as entry points



(c) Fig. 6a after increasing the usage level to 4

Fig. 6: Scenario 2

C. Summary

Through these two scenarios, the newcomer is able, using *VariCity*, to see variability implementations in an unknown codebase from a high-level perspective, and also to dig into them to have a more precise understanding. Providing a more complete experimental evaluation based on feedback from both experts and newcomers on the use of *VariCity* is part of our future work.

VI. Threats to validity and Limitations

Without an empirical assessment, the main threat of our evaluation concerns the scenarios that we designed by ourselves. Nevertheless, we relied on both empirical work on onboarding with real experts and newcomers [53] and challenges related to the comprehension of variability concepts [52], giving us good confidence in the relevance of the scenarios.

Another threat arises by the fact that both authors and developers of *VariCity* determined empirically the inputs (entry points, usage level and orientation) for each scenario, 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. We expect real experts to be able to determine appropriate inputs in real settings.

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 calls for an adaptation of the placement algorithm to take into account both dimensions while placing the buildings.

VII. Related Work

Works on the city metaphor in software visualizations were studied in section IV-A. In this section, we discuss work related to visualization for variability management and to assist onboarding activities.

A. 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 [14]. These visualizations are mainly used to facilitate the configuration process over features. To visualize variability at the code level, some approaches use colors [69] or bar diagrams [70], while some others focus on feature traces [71] or feature interactions between features and code [72], [73]. None of them focus on object-oriented techniques as variability implementations.

In *VariCity*, we reused the symmetry-based detection part of *symfinder* [22], [59], 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 some occurrences of symmetries. 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).

B. Visual tools to assist onboarding

Some visualizations have been especially proposed for onboarding activities. Isopleth [74] represents call relationships in front-end JavaScript implementations in the form of a call graph, which is interactive and can be edited to see the impact in real-time on the page. Other tools integrate information from the organization to help information seeking during development activities, such as Tesseract [75] which visualizes the relationships between technical information from a codebase and related social data (*e.g.*, developers, communication, code, and bugs). Finally, recent studies on onboarding in SPLs [34] explore concept maps [76] to structure information about the SPL. However, this approach, as many others evoked in section VII-A relies on a feature model and documentation, which does not apply in our case.

VIII. Conclusion

Variability-rich object-oriented software systems often implement variability in a single code base using mechanisms from the supporting language, such as inheritance, overloading, and design patterns. These implementations are thus not explicit and difficult to comprehend, especially for a newcomer onboarding on a project and trying to comprehend variability in it. In this paper, we proposed *VariCity*, a 3D visualization based on the city metaphor to propose adapted and configurable views that exhibit zones of high density of variability implementations. The density relies on previous work on automated detection of symmetries in the variability implementation mechanisms. Metrics on their occurrences together with information on inheritance and usage relationships are exploited to build a city with, notably, classes as buildings and streets as usage relationships. We also detailed two onboarding scenarios showing how the different capabilities of the visualization can help to spot critical variability-related zones of a codebase and obtain fine-grained information about them.

The visualization toolchain is publicly available and has been applied to several large object-oriented systems. We expect it can help different kinds of users, from experts to newcomers, to understand variability in this kind of system. As future work, we first plan to conduct an evaluation with real experts of a codebase building scenarios for newcomers onboarding on their project. We also plan to integrate the visualization with a development environment that would automate the interactions between the source code and the city. At another level, we also want to explore the coupling of the metrics used in this visualization with other software metrics [77], such as complexity or test coverage. With these experiments and improvements, we expect to gain new insights on how to better facilitate the identification of variability implementations.

Acknowledgment

We thank Paul-Marie Djekinnou, Florian Focas and François Rigaut for their contribution in the development of *VariCity*.

REFERENCES

[1] R. Hilliard, "On representing variation," in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA '10. ACM, 2010, p. 312–315. [Online]. Available: https://doi.org/10.1145/1842752.1842810

[2] M. Galster, D. Weyns, D. Tofan, B. Michalik, and P. Avgeriou, "Variability in software systems — a systematic literature review," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 282–306, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2013.56

[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*, ser. VaMoS '19. ACM, 2019, pp. 1–1. [Online]. Available: https://doi.org/10.1145/3302333.3302336

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

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

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

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, Tech. Rep., 1990.

[8] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and 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.

[9] K. Schmid and I. John, "A customizable approach to full lifecycle variability management," *Science of Computer Programming*, vol. 53, no. 3, pp. 259–284, 2004. [Online]. Available: https://doi.org/10.1016/j.scico.2003.04.002

[10] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and 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*, ser. RE '07. IEEE, 2007, pp. 243–253. [Online]. Available: https://doi.org/10.1109/RE.2007.61

[11] C. Gacek and M. Anastasopoules, "Implementing product line variabilities," in *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, ser. SSR '01. ACM, 2001, pp. 109–117. [Online]. Available: https://doi.org/10.1145/375212.375269

[12] M. Svahnberg, J. Van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software: Practice and experience*, vol. 35, no. 8, pp. 705–754, 2005.

[13] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013. [Online]. Available: https://doi.org/10.1002/smr.567

[14] R. E. Lopez-Herrejon, S. Illescas, and A. Egyed, "A systematic mapping study of information visualization for software product line engineering," *Journal of software: evolution and process*, vol. 30, no. 2, p. e1912, 2018.

[15] G. K. Michelon, L. Linsbauer, W. K. Assunção, S. Fischer, and A. Egyed, "A hybrid feature location technique for re-engineeringsingle systems into software product lines," in *15th International Working Conference on Variability Modelling of Software-Intensive Systems*, 2021, pp. 1–9.

[16] J. A. Domingue, *Software visualization: Programming as a multimedia experience*. MIT press, 1998.

[17] C. Knight and 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.

[18] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[19] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, 2003.

[20] M.-A. D. Storey, D. Čubranić, and 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.

[21] A. R. Teyseyre and M. R. Campo, "An overview of 3D software visualization," *IEEE transactions on visualization and computer graphics*, vol. 15, no. 1, pp. 87–105, 2008.

[22] Xh. Tërnava, J. Mortara, and P. Collet, "Identifying and visualizing variability in object-oriented variability-rich systems," in *the 23rd International Systems and Software Product Line Conference*. Paris, France: ACM Press, Sep. 2019, pp. 231–243. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02339296

[23] J. Mortara, Xh. Tërnava, and P. Collet, "Mapping Features to Automatically Identified Object-Oriented Variability Implementations - The case of ArgoUML-SPL," in *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20)*, Magdeburg, Germany, Feb. 2020. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02421353

[24] R. Wettel and 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.

[25] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza, "Visualizing Evolving Software Cities," in *2020 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2020, pp. 22–26.

[26] F. Steinbrückner and C. Lewerentz, "Representing development history in software cities," in *Proceedings of the 5th international symposium on Software visualization*, 2010, pp. 193–202.

[27] R. Wettel and M. Lanza, "CodeCity: 3D visualization of large-scale software," in *Companion of the 30th international conference on Software engineering*, 2008, pp. 921–922.

[28] L. Zhao and J. Coplien, "Understanding symmetry in object-oriented languages," *Journal of Object Technology*, vol. 2, no. 5, pp. 123–134, 2003.

[29] J. O. Coplien and L. Zhao, "Toward a general formal foundation of design. symmetry and broken symmetry," *(Forthcoming publication)*, 2020.

[30] L. M. Berlin, "Beyond program understanding: A look at programming expertise in industry," *ESP*, vol. 93, no. 744, pp. 6–25, 1993.

[31] S. E. Sim and R. C. Holt, "The ramp-up problem in software projects: A case study of how software immigrants naturalize," in *Proceedings of the 20th international conference on Software engineering*. IEEE, 1998, pp. 361–370.

[32] Y. Park and C. Jensen, "Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers," in *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009, pp. 3–10.

[33] A. Metzger and K. Pohl, "Software product line engineering and variability management: achievements and challenges," in *Future of Software Engineering Proceedings*, 2014, pp. 70–84.

[34] M. Azanza, A. Irastorza, R. Medeiros, and 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.

[35] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," *Journal of Systems and Software*, vol. 49, no. 1, pp. 3–15, 1999. [Online]. Available: https://doi.org/10.1016/S0164-1212(99)00062-X

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

[37] F. Bachmann and P. Clements, "Variability in software product lines," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2005-TR-012, 2005. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=7675

[38] I. Jacobson, M. Griss, and P. Jonsson, *Software reuse: architecture process and organization for business success*. acm Press New York, 1997, vol. 285.

[39] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and 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*, ser. VaMoS'12, 2012, pp. 173–182. [Online]. Available: https://doi.org/10.1145/2110147.2110167

[40] 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*, ser.

SPLC '19. ACM, 2019, pp. 134–136. [Online]. Available: https://doi.org/10.1145/3307630.3342399

[41] I. John, J. Lee, and D. Muthig, "Separation of variability dimension and development dimension," in *Proocedings of the 1st International Workshop on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '07, 2007, pp. 45–49.

[42] Xh. Tërnava and 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*, ser. SPLC '17. ACM, 2017, pp. 81–88. [Online]. Available: https://doi.org/10.1145/3109729.3109733

[43] N. Anquetil, U. Kulesza, R. Mitschke, A. Moreira, J.-C. Royer, A. Rummler, and A. Sousa, "A model-driven traceability framework for software product lines," *Software & Systems Modeling*, vol. 9, no. 4, pp. 427–451, 2010.

[44] J. Rubin and M. Chechik, "A survey of feature location techniques," in *Domain Engineering*. Springer, 2013, pp. 29–58.

[45] W. K. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed, "Reengineering legacy applications into software product lines: a systematic mapping," *Empirical Software Engineering*, vol. 22, no. 6, pp. 2972–3016, 2017.

[46] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is my feature and what is it about? a case study on recovering feature facets," *Journal of Systems and Software*, vol. 152, pp. 239–253, 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.01.057

[47] J. Martinez, Xh. Tërnava, and 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*, ser. SPLC '18. ACM, 2018, pp. 132–142. [Online]. Available: https://doi.org/10.1145/3233027.3233038

[48] J. Coplien, D. Hoffman, and D. Weiss, "Commonality and variability in software engineering," *IEEE Software*, vol. 15, no. 6, pp. 37–45, 1998. [Online]. Available: https://doi.org/10.1109/52.730836

[49] T. Patzke and D. Muthig, "Product line implementation technologies. programming language view," Fraunhofer IESE, Tech. Rep., 2002. [Online]. Available: http://publica.fraunhofer.de/dokumente/N-14684.html

[50] A. Lozano, "An overview of techniques for detecting software variability concepts in source code," in *International Conference on Conceptual Modeling*, ser. ER '11. Springer, 2011, pp. 141–150. [Online]. Available: https://doi.org/10.1007/978-3-642-24574-9

[51] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[52] M. Acher, R. E. Lopez-Herrejon, and R. Rabiser, "Teaching software product lines: A snapshot of current practices and challenges," *ACM Transactions on Computing Education (TOCE)*, vol. 18, no. 1, pp. 1–31, 2017.

[53] R. Yates, N. Power, and J. Buckley, "Characterizing the transfer of program comprehension in onboarding: an information-push perspective," *Empirical Software Engineering*, vol. 25, no. 1, pp. 940–995, 2020.

[54] I. Steinmacher, M. A. G. Silva, and M. A. Gerosa, "Barriers faced by newcomers to open source projects: a systematic review," in *IFIP International Conference on Open Source Systems*. Springer, 2014, pp. 153–163.

[55] A. Begel and B. Simon, "Struggles of new college graduates in their first software development job," in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, 2008, pp. 226–230.

[56] R. Wettel and M. Lanza, "Visual exploration of large-scale system evolution," in *2008 15th Working Conference on Reverse Engineering*. IEEE, 2008, pp. 219–228.

[57] J. Mortara, Xh. Tërnava, and P. Collet, "symfinder: A Toolchain for the Identification and Visualization of Object-Oriented Variability Implementations," in *the 23rd International Systems and Software Product Line Conference*, vol. B. Paris, France: ACM Press, Sep. 2019, pp. 5–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02342730

[58] J. Mortara, P. Collet, and Xh. Tërnava, "Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder," in *24th ACM International Systems and Software Product Line Conference (SPLC '20)*, ACM, N. York, NY, and USA, Eds., MONTREAL, QC, Canada, Oct. 2020, virtual Conference. [Online]. Available: https://hal.archives-ouvertes.fr/hal-02908531

[59] J. Mortara, Xh. Tërnava, P. Collet, and A.-M. Dery-Pinna, "Extending the Identification of Object-Oriented Variability Implementations using Usage Relationships," in *SPLC 2021 - 25th ACM International Systems and Software Product Line Conference*, vol. Volume B. Leicester, United Kingdom: ACM, Sep. 2021, pp. 1–8. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03284626

[60] C. Knight and M. Munro, "Comprehension with [in] virtual environment visualisations," in *Proceedings Seventh International Workshop on Program Comprehension*. IEEE, 1999, pp. 4–11.

[61] J. Waller, C. Wulf, F. Fittkau, P. Döhring, and 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.

[62] M. Weninger, L. Makor, and 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.

[63] F. Fittkau, A. Krause, and W. Hasselbring, "Software landscape and application visualization for system comprehension with explorviz," *Information and software technology*, vol. 87, pp. 259–277, 2017.

[64] F. Steinbrückner and C. Lewerentz, "Understanding software evolution with software cities," *Information Visualization*, vol. 12, no. 2, pp. 200–216, 2013.

[65] J. Vincur, P. Navrat, and 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.

[66] G. Balogh and 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.

[67] J. Kratt, H. Strobelt, and O. Deussen, "Improving Stability and Compactness in Street Layout Visualizations," in *VMV*, 2011, pp. 285–292.

[68] J. Mortara, P. Collet, and A.-M. Dery-Pinna, "Visualization of Object-Oriented Variability Implementations as Cities — Reproduction package," Jun. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5034199

[69] C. Kästner, S. Trujillo, and S. Apel, "Visualizing software product line variabilities in source code," in *SPLC (2)*, 2008, pp. 303–312.

[70] S. Duszynski and 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.

[71] B. Andam, A. Burger, T. Berger, and 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.

[72] O. Greevy, M. Lanza, and 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.

[73] A. Bergel, R. Ghzouli, T. Berger, and M. R. V. Chaudron, "Featurevista: Interactive feature visualization," in *Proceedings of the 25th ACM International Systems and Software Product Line Conference*, ser. SPLC '21. ACM, 2021.

[74] J. Hibschman, D. Gergle, E. O'Rourke, and H. Zhang, "Isopleth: Supporting Sensemaking of Professional Web Applications to Create Readily Available Learning Experiences," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 26, no. 3, pp. 1–42, 2019.

[75] A. Sarma, L. Maccherone, P. Wagstrom, and J. Herbsleb, "Tesseract: Interactive visual exploration of socio-technical relationships in software development," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 23–33.

[76] J. D. Novak and A. J. Cañas, "The theory underlying concept maps and how to construct them," *Florida Institute for Human and Machine Cognition*, vol. 1, 2006.

[77] S. El-Sharkawy, N. Yamagishi-Eichler, and K. Schmid, "Metrics for analyzing variability and its implementation in software product lines: A systematic literature review," *Information and Software Technology*, vol. 106, pp. 1–30, 2019.