Visualizing Variability Implemented with Object-Orientation and Code Clones: A Tale of Two Cities

Yann Brault, Philippe Collet, Anne-Marie Pinna-Dery

Visualizing Variability Implemented with Object-Orientation and Code Clones: A Tale of Two Cities

Yann Brault
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
yann.brault@etu.univ-cotedazur.fr

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

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

Abstract

Understanding variability in large software systems poses significant challenges for developers, especially when variability is implemented within a single codebase using diverse language mechanisms like in TypeScript. In this language, one can implement variability with traditional Object-Oriented (OO) techniques, but also with dynamic loading mechanisms that organize different forms of clones in sub-directories and files serving as internal variants. While certain approaches may facilitate partial identification or visualization, there exists no solution for handling all variability mechanisms simultaneously.

In this paper, we propose an approach by detecting all mechanisms and integrating two city-based representations to visualize these implemented variabilities. The first representation adapts the *VariCity* visualization and focuses on OO variability with classes as buildings and usage relationships as streets. The second representation leverages a codebase analysis combined with a code clone detection technique to visualize the directory hierarchy as streets and files as circular districts with shades and colors to highlight cloning. Some visual mechanisms enable to display relevant relationships between them, unveiling patterns of cross-usage and variability architecture. We also report on the application of the tooled approach on several large open-source systems.

CCS Concepts

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

Keywords

software variability, software visualization, reverse-engineering, code clones, clone detection

1 Introduction

Variability is more and more present in all forms and scales of modern software-intensive systems [15, 16, 19]. While software variability is usually defined as the ability of a software artifact to be efficiently extended, changed, customized, or configured towards a specific context [8], its management is the subject of many research advances, that led to the Software Product Line (SPL) [3, 32] paradigm. Still, numerous variability-rich systems do not follow a comprehensive SPL approach as they gradually manage variability using the host language mechanisms in a single codebase without any other additional information or annotation regarding variability [44]. Depending on the language, many different techniques and mechanisms can be used together [8, 14, 40], from inheritance, overloading, and design patterns in object-oriented settings, to more general-purpose parameterization or loading of source code files in dynamic languages. As all these mechanisms are also used to realize the business logic, the implemented variability is concealed within the whole codebase. This is directly hindering its identification and understanding as there is no traceability with domain information [41, 42].

This multiple usage of variability implementation mechanisms can be observed in recent general-purpose programming languages such as TypeScript, which is the focus of this work. As for variability implementation, TypeScript exhibits several mechanisms that can be employed. Some classical OO techniques [40] such as inheritance and design patterns are easily accessible or can be simulated. In addition, the dynamic capabilities of the language can be exploited to load files organized in directories and subdirectories to handle variant code. This can lead to files with the same name being duplicated, sometimes with several OO classes or functions, with the same or different code in their implementations.

Some recent approaches have been proposed to identify OO variability implementations [41, 42] and visualize them using a 3D city metaphor [28, 29] that have been shown to facilitate variability understanding [27]. Some 3D visualizations have also been proposed to represent code duplication metrics [18, 23, 38] while some other proposals [11, 36] attempt to provide a forward-engineering approach for building a SPL in JavaScript. To the best of our knowledge, there is no comprehensive solution available to handle all mechanisms simultaneously in codebases that do not contain any other information except the implementation mechanisms.

In this paper, we propose an approach that facilitates the identification and visualization of these different types of variability implementation mechanisms (*cf.* section 2) within a TypeScript codebase by integrating two city-based representations [47] (*cf.* Section 3) built from static analysis of the codebases. First, we adapt the *symfinder* extraction toolchain [30, 41, 42] to obtain metrics over OO mechanisms in TypeScript. We then rely on the *VariCity* visualization principles [27–29] to create the 3D representation of the first city, called *ObjectCity*, in which classes are represented as buildings whose dimensions correspond to variability metrics, while streets depict usage relationships. For the second city, another static analysis gathers information from the detection of files with the same names within all directories, and from a code clone detection algorithm applied to all files. The second city, *CloneCity*, visualizes the directory hierarchy as streets and files as circular districts with different colors to point out duplication and cylindrical shades to highlight clones obtained from a code clone detection phase. In addition, files sharing duplicated code are linked by aerial bridges, as well as classes visualized in both cities simultaneously. We apply our approach to several large TypeScript codebases, showing that different variability implementations can be detected and

displayed (*cf.* Section 4). Threats to validity are discussed in Section 5, while Section 6 concludes this paper and discusses future work.

2 Motivations

Differentiating variability within code assets can be organized across three distinct parts: core, commonalities, and variations [4, 46]. The core corresponds to assets present in all final software products. Commonalities represent shared elements among related variations of code assets, while variations indicate the way and the time code assets actually vary. Commonalities and variations are abstracted as variation points (*vp*-s) and variants, respectively [20, 33]. A variation point identifies one or more locations where variation may occur, while its variants articulate the specific ways in which the variation point will diverge [20].

2.1 Implemented variabilities in TypeScript

The TypeScript language is built as an extension of JavaScript, being prototype-based with a mix of imperative and functional paradigms, with static typing over OO mechanisms. It is used extensively in front-end and back-end development, but also to develop utilities [1]. The language notably gains popularity [2] by a good combination of type safety, capacity to handle large projects, and availability of strong tooling support to enhance productivity. To explore the potential mechanisms in TypeScript to implement variability, we have manually studied the language definition [26] and several large open-source systems to seek typical implementations as defined in previous taxonomies [14, 40, 43]. The studied systems are the ones presented in Section 4.

Object-orientation. TypeScript OO mechanisms are the first candidates to implement variability, with inheritance between classes and interfaces, and *vp*-s being superclasses while subclasses are variants. While these mechanisms are similar to other OO languages such as Java, method overloading does not directly exist in TypeScript [3]. Still, it is simulated by defining several method signatures and a method that groups each of the available types and parameters that enables the corresponding method call. With these mechanisms, several design patterns that are typical of variability implementations (*i.e.*, factory, strategy, template, decorator [14, 40, 43]) can also be found in TypeScript.

Modules. Modules are an important part of the TypeScript language as they foster good structuring and reusable components. Every source file with an import or export directive is considered a module, making the code encapsulation explicit. Based on the library pattern from JavaScript, the module concept can then be used in projects of all sizes to organize features or plugins with dynamic loading. Usually, one can find a main folder named after the feature and inside sub-folders for each variant. Those sub-folders also share common structures, with more sub-folders or files that have the same name. As in many plugin architectures, naming conventions and interfaces are accompanied by precise guidelines to define an interaction protocol between the core of the system and a plugin. This naturally forms a *vp* on the top folder for plugins, where each plugin is then a variant.

We show on Figure 1 an extract of a typical organization of modules to handle some variability extracted from the source code of Echarts, a charting library for TypeScript. Both folders `bar` and `boxPlot` on Figure 1 are examples of modules. They have a duplicated file, *e.g.*, `install.ts` serving a standardization purpose, and non-duplicated files that exhibit some code duplication. In addition, object-oriented mechanisms can be nested inside these variant modules. Finally, we have also observed that this dynamic loading of a module can be used more specifically for parameterization, most often with different files of the same names containing a similar list of defined constants.

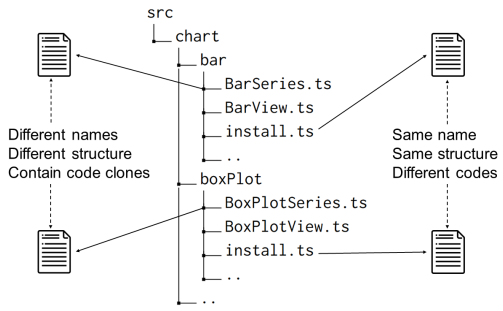


Figure 1: File duplication in Echarts file tree (Chart folder)

The analysis of the possible variability implementations in TypeScript shows that several different mechanisms can be used to realize variability with this language. Without any other additional information or annotation regarding variability [44] and all variants being organized within the same single codebase, these implementation spots are difficult to find and comprehend for an average developer [27]. As these mechanisms also serve business logic and project architecture, this further obscures variability implementations. In addition, the context of the single codebase implies that the techniques that re-engineer features from clones of a whole system cannot be applied [25].

2.2 Related work

Variability and city-based visualization. A recent mapping study revealed that visualizations in the SPL domain primarily concentrate on feature models, employing tree or graph representations [24]. These visualizations primarily aim to streamline the configuration process concerning features. To represent variability at the code level, certain approaches utilize colors or bar diagrams, while others emphasize feature traces or interactions between features and code [2, 5, 12, 17, 21]. Visualization-based tools are commonly employed to aid in the understanding of large software systems [6, 22, 39, 45], including aspects related to their variability implementations [2, 5, 13]. Given that software visualization involves creating visual representations of software using visual objects to depict structure and/or behavior, some recent approaches have been proposed to identify OO variability implementations [41, 42]

[1] https://blog.jetbrains.com/webstorm/2024/02/js-and-ts-trends-2024/
[2] ranked 2nd in march 2023 at https://www.libhunt.com/index
[3] https://www.typescriptlang.org/docs/handbook/2/functions.html#function-overloads

and visualize them using a 3D city metaphor [28, 29]. This solution, named *VariCity* [27] has been shown to facilitate variability understanding [27], but only when it is implemented with OO mechanisms. The reader will find more details on it in Section 3.1 as we reuse and adapt *VariCity* to build our proposed visualization.

Code clone detection and visualization. As code duplication is a simple way to reuse in software development that has many negative impacts (*e.g.*, bug introduction and propagation, higher maintenance cost), code clone detection has been studied for many years and different detection techniques have been proposed [1, 31, 34]. The output of detection is usually a *clone pair*, which is defined as "a pair of code portions/fragments which are identical or similar to each other". Code clones are classified into four main types [34], with the first three considered as textually similar, and the fourth one as functionally similar. We focus on Type 3 clones as they accept text additions and deletions as well as various minor changes [34], and we believe it can cover the many different changes that one could use to realize a variant within the same codebase. Various tools exist [9, 10, 35] but MSCCD [37, 48] is the most adapted to our needs as it focuses on Type 3 clones and works with a language-independent technique that can fit many programming languages. Besides, many techniques and tools for clone visualization have also been proposed [18]. Interestingly EvoStreet [38], which was one of the first OO metric visualization using a city metaphor, has also been enhanced to display code clones, but not for understanding variability and not with other techniques such as OO mechanisms like in our context. A similar 3D visualization was also proposed, still only for code clones [23].

3 The 2CITIES Visualization

Rather than extending *VariCity* [27, 30], which has been successfully applied to visualize OO variability implementation mechanisms, we decided to create two cities, one for each kind of variability mechanism, and to avoid visual clutter.

3.1 *VariCity* for TypeScript, *a.k.a.*, *ObjectCity*

First, we adapt the *symfinder* extraction toolchain [30, 41, 42] to obtain metrics over inheritance, function overloading, and variability-related design patterns (*i.e.*, strategy, factory, template, decorator) in TypeScript. We then rely on the *VariCity* visualization principles [27–29] to create the 3D representation of the first city, called *ObjectCity*. Classes, whatever source files they are in, are represented as buildings whose dimensions correspond to variability metrics (*e.g.*, height for method overloading), while streets depict usage relationships between these classes, as illustrated on Figure 2. Additionally, identified design patterns are decorated with special crowns, except for pyramidal crowns that designate entrypoint classes, the points of interest specified by the user to scope the search for OO variability and that are aggregated on the red street (*cf.* left of Figure 2). Finally, additional relationships such as inheritance are visualized by aerial blue links between the super-class and its sub-classes, as shown on the left of Figure 2.

3.2 *CloneCity*

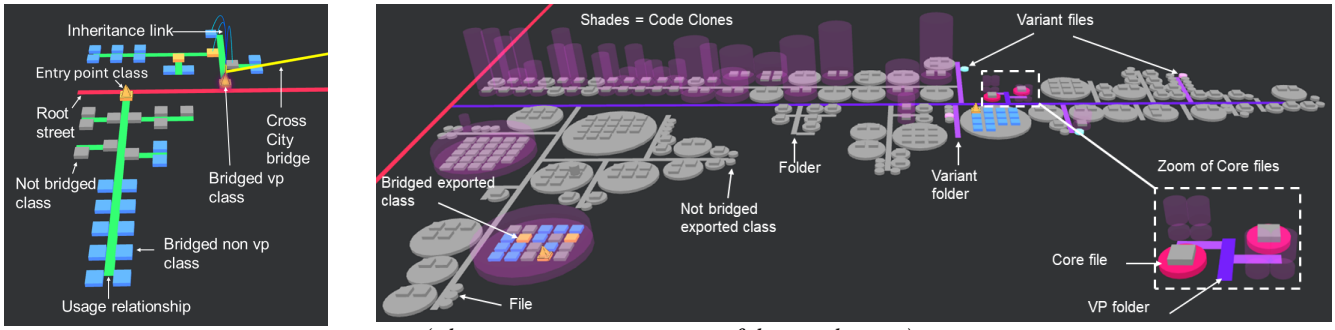
The objective of *CloneCity* is to visualize variability implemented with modules and files duplication, and code clone duplication (*cf.* Section 2.1). We first extend the *symfinder* toolchain to perform a second detection over the whole project tree to find file duplications based on the following rules. We define as a *variant file* a file duplicated with the same name in at least two folders on the same hierarchy level. If the file is duplicated in all folders at the same level, it is referred to as a *core file*. Additionally, we define as a *variant folder* a folder containing a *variant file*, and a *vp folder* is a folder being parent to at least two *variant folders*. Finally, the toolchain detects code clone duplication with the tool MSCCD [37, 48], a language-independent tool focusing on Type 3 clones (*cf.* Section 2.2). Files containing duplicated code are labeled as cloned files and create a clone pair (*cf.* Section 2.2).

CloneCity aims at visualizing the different forms of clones that have been detected while helping to comprehend their organization and the potential relationships with its *ObjectCity* counterpart. The city is visualized with folders as roads and files as circular districts along the roads as illustrated on the right part of Figure 2. Files are represented with their exported classes on top to obtain information about potential OO variability implementations as mechanisms can be nested (*cf.* Section 2.1). By default, all colored elements visualize variability implementations. Colored files and folders represent the use of modules and file duplication, with variations in tone for the level of duplication, and a change in color for the different duplicated files. The zoomed part on the right of Figure 2 shows the highest level of duplication with the structure of a *vp* folder and various core files from the Vim editor plugin for VsCode. Code clone duplications are visualized with purple shades located on top of files, visible on the top right part of Figure 2, which scale according to the number of clones the file counts. Finally, both files of a clone pair are linked by an aerial purple bridge when one is hovered or selected to help the user pinpoint where code clones are concentrated. This should reveal variant files that are actually sharing some functionality by code being copied from one folder to another.

3.3 Bridging the two cities

Each separate city offers a dedicated visualization of the different variability implementations to focus the comprehension activity, but the two cities are related in several ways. First, the process to configure the *2Cities* visualization actually starts by selecting a source directory for *CloneCity*. Then the revealed zones in *CloneCity* show some classes of interest, they can be added as entry points in the *ObjectCity* view. Quite often, this is complemented by other entry points selected by looking at the documentation and the source code. For example Figure 3 depicts the *2Cities* visualization for Vim, in which several classes have been selected as entry points and create the long streets showing their usages.

As a result, the classes displayed in *CloneCity* are a subset of the ones visible in *ObjectCity*. When they appear in both cities, they take the visual properties defined in *ObjectCity*, *i.e.*, color, size variations, design pattern crowns (*cf.* both parts of Figure 2). On the contrary, the classes sitting on top of file districts that are not visible in *ObjectCity* are represented as low gray buildings. To complement the identical visual properties of the classes represented in both cities, a yellow bridge (*cf.* left part of Figure 2) connecting the two instance roofs of a class building can be displayed across the



(white annotations are not part of the visualization)

Figure 2: partial *ObjectCity* (left) and *CloneCity* (right) for the Vim system

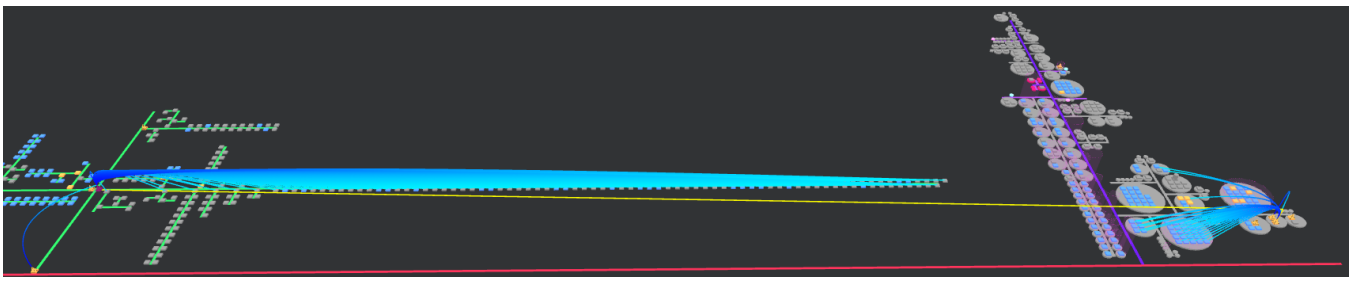


Figure 3: A complete *2Cities* visualization of Vim

two cities when either building is hovered or selected. In addition to this cross-city bridge, if the class has inheritance relationships with other classes, they are displayed simultaneously in both in *ObjectCity* and *CloneCity*, only towards bridged classes. Consequently, we expect the bridging of the two cities to help in visualizing and comprehending integration and interaction between folders, files, and classes within the whole codebase.

4 Preliminary Validation

To validate our approach, we applied it to 7 TypeScript (TS) open-source projects, from Web applications to libraries and frameworks, with a large range in size, either in Lines of Code (LoCs) or the number of files. Systems were selected on Github considering their popularity and their potential for variability. All results are available online in a dedicated Zenodo archive [7].

Quantitative evaluation. We configured views revealing some variability implementations on both cities and calculated metrics to evaluate to which extent variability implementations are highlighted *w.r.t.* the whole codebase. For each project, we made some automatic and manual measurements over the two cities that made the *2Cities* visualization, as well as the ratio of bridged classes.

In the different subject systems (*cf.* Table 1), we observed different zones of implemented variability in *ObjectCity*. Some systems like Angular reveal a large number of zones (30) with a small number of variants in total (170), showing many focused usage of OO variability for different elements in the framework. On the contrary, for the Vim plugin for VsCode, out of around 570 potential *vp*-s and variants, the visualization has helped identify 4 zones, visualizing around 400 class variants. We also observed different organizations from the *CloneCity* metrics. Some like Vim have a small structure of clones, but with almost 25% of cloned files inside, representing 40% of bridged OO classes, meaning that some OO variability is also managed with variant files. Other systems have a non-negligible ratio of variant files, like Echarts with a ratio of 14.3%, and 25% of cloned files as well. Interestingly, Angular and Material-ui are quite opposed in their use of object orientation and code duplication.

We observed that both cities are able to focus on their kind of variability while offering filtering capabilities so that specific zones can be easily identified in each city. Cross-city bridging of classes seems to help identify variability patterns, with classes sometimes being used together with cloned files to implement variability.

Detailed case study. We aimed at evaluating whether the zones of variability implementations highlighted by *2Cities* are relevant, *i.e.*, valuable for understanding variability. We studied Vim (*cf.* Table 1) by manually going back to the code and the documentation from the visualization to systematically relate the highlighted zones to some potential high-level features and their implementations. We illustrate this case with the right part of Figure 2 and especially the zoom on core files. With complementary information from the source code, we grasp that this structure is used to implement a feature related to the communication between the file system and Vim, which is variable *w.r.t.* to the platform on which the engine runs. An example of cross-city variability is shown on Figure 3. It displays a dense inheritance pattern responsible for all commands of Vim, with the abstract class `BaseCommand` on the far left, and all variant implementations, more than 150, displayed on the long

					ObjectCity				*CloneCity*					**Bridges**
Subject	Main purpose	# TS LoCs	# TS files	# TS files exc. test	# potential vps + variants	# zones revealed	# exposed variants in zones	# zones relevant	# vp+ variant folders	# variant files	ratio variant files/ files	# cloned files	ratio cloned files/ files	ratio bridged classes in zones
Angular	Frontend framework	554k	5.8K	4.5K	2,843	30	~170	30	212	244	5.4%	457	10%	~88%
Echarts	Charting library	85.7K	589	553	973	8	~110	8	60	79	14.3%	139	25%	~17%
Grafana	Data dashboard	645K	5.9K	2.1K	2,246	12	~150	12	158	286	13.6%	395	18.8%	~27%
Material-ui	Frontend components	270K	4K	1K	1,231	1	7	1	25	20	1.9%	496	47.7%	100%
Nest	Backend framework	75.5K	1.4K	1.2K	783	16	~90	16	30	45	3.8%	216	18%	~97%
TypeScript	Language compiler	666K	18.8K	317	1,662	9	27	9	11	11	3.5%	62	19.6%	59%
Vim	VsCode editor plugin	47.3K	245	156	572	4	~410	4	8	10	6.4%	36	23%	~40%

Table 1: Subject systems and metrics

street. However, it is noticeable by comparing the number of inheritance links displayed in *CloneCity* that only a fraction of the variants are actually exported to be reused, indicating there is some internal variability in the implementation of this feature.

Obviously, we cannot generalize from these findings, but the current studies on the other systems also seem to show that the most visible zones in both cities allow for unveiling the variability related to some important features of the considered systems.

5 Threats to validity

Our preliminary evaluation is obviously limited as it does not rely on an empirical assessment with for example an independent user study. The main threat to its validity comes from the selection of subjects and the observations. The subject systems were empirically determined by the authors, which may inadvertently introduce biases based on our perspectives or preferences. While we have designed the evaluation scenarios ourselves, they are derived from the validation of *VariCity* [27]. While this provides a foundation for comparison and continuity, it also introduces the risk of inheriting biases or assumptions. This is partly the case with our starting process to guide the developer while building the *2Cities* view. We always started with the *CloneCity* visualization, and then found object-oriented elements in this view or in the project (*i.e.*, source code, documentation) to populate the entry points necessary to bootstrap the *ObjectCity* visualization.

As *2Cities* is relying on external tools, it is also subject to their induced threats. First, we reimplemented a *symfinder* analyzer for TypeScript that is prone to the same limitations, *i.e.*, the sensitivity to different thresholds for hotspot determination and the lack of fine-grained detection within code blocks. Still, the addition of code clone detection mitigates this latter threat. We rely on the *VariCity* visualization engine [27], which is configurable, but does not automatically support the adaptation for visual impairments (*e.g.*, color blindness). Our toolchain is also directly reusing an automated code clone detection tool, whereas studies on clones mainly validate true and false positives with human expertise [1, 31, 34].

6 Conclusion

In this paper, we proposed a tooled approach to detect the different variability implementation mechanisms of TypeScript and visualize them in a two city-based representations. A first city takes the information from the codebase structure and filenames combined with a code clone detection to visualize the directory hierarchy as streets and files as circular districts with clones being highlighted as cylindrical shades on files or as colored districts. The second city adapts the *VariCity* visualization [27, 30] to exhibit OO variability zones with classes as buildings and usage relationships as streets. Classes appearing in both cities can be bridged to ease the understanding between the two representations. We also reported on the first validation steps on several open-source projects. It shows that the different implementation zones are highlighted with different means in the resulting visualizations. First qualitative observations on a project also demonstrated that the zones of interest actually correspond to valuable high-level features.

As future work, we first plan to integrate the visualization with a development environment to reproduce the type of controlled experiment with developers that have been conducted on *VariCity* [27]. In the longer term, our plan entails analyzing variability in API definitions to enhance the facilitation of identification and understanding of implemented variability.

Acknowledgments

We thank Martin Bruel for his contribution in the TypeScript extension of *symfinder*, as well as Alexandre Arcil, Gabriel Cogne, Chenzhou Liao, and Dan Nakache for their contribution in the development of the first prototype of *2Cities*.

References

[1] Danyah Alfageh, Hosam Alhakami, Abdullah Baz, Eisa Alanazi, and Tahani Alsubait. 2020. Clone Detection Techniques for JavaScript and Language Independence. *International Journal of Advanced Computer Science and Applications* 11, 4 (2020).

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

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

[4] Felix Bachmann and Paul Clements. 2005. *Variability in Software Product Lines*. Technical Report CMU/SEI-2005-TR-012. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

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

[6] Deborah A Boehm-Davis, Jean E Fox, and Brian H Philips. 1996. Techniques for exploring program comprehension. In *Empirical Studies of Programmers*. 3–37.

[7] Yann Brault, Philippe Collet, and Anne-Marie Pinna-Dery. 2024. *Visualizing Variability Implemented with Object-Orientation and Code Clones: A Tale of Two*

Cities - Companion Technical Report. Technical Report. https://doi.org/10.5281/zenodo.12527153

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

[9] Wai Ting Cheung, Sukyoung Ryu, and Sunghun Kim. 2016. Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering* 21 (2016), 517–564.

[10] James R. Cordy and Chanchal K. Roy. 2011. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension*. 219–220. https://doi.org/10.1109/ICPC.2011.26

[11] Alejandro Cortiñas, Miguel R Luaces, and Óscar Pedreira. 2022. spl-js-engine: a JavaScript tool to implement software product lines. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*. 66–69.

[12] Slawomir Duszynski and Martin Becker. 2012. 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, 37–40.

[13] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. 2013. Do background colors improve program comprehension in the# ifdef hell? *Empirical Software Engineering* 18, 4 (2013), 699–745.

[14] Cristina Gacek and Michalis Anastasopoules. 2001. Implementing Product Line Variabilities. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR '01)*. ACM, 109–117.

[15] Matthias Galster. 2019. Variability-Intensive Software Systems: Product Lines and Beyond. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '19)*. ACM, 1–1.

[16] Matthias Galster, Danny Weyns, Dan Tofan, Bartosz Michalik, and Paris Avgeriou. 2013. Variability in Software Systems — A Systematic Literature Review. *IEEE Transactions on Software Engineering* 40, 3 (2013), 282–306.

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

[18] Muhammad Hammad, Hamid Abdul Basit, Stan Jarzabek, and Rainer Koschke. 2020. A systematic mapping study of clone visualization. *Computer Science Review* 37 (2020), 100266. https://doi.org/10.1016/j.cosrev.2020.100266

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

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

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

[22] Rainer Koschke. 2003. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice* 15, 2 (2003), 87–109.

[23] Rainer Koschke and Marcel Steinbeck. 2021. SEE Your Clones With Your Teammates. In *2021 IEEE 15th International Workshop on Software Clones (IWSC)*. 15–21. https://doi.org/10.1109/IWSC53727.2021.00009

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

[25] Jabier Martinez, Xhevahire Tërnava, and Tewfik Ziadi. 2018. 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, 132–142.

[26] Microsoft. 2012. TypeScript language Website. https://www.typescriptlang.org/ Accessed on April 2024.

[27] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2024. Visualization of object-oriented software in a city metaphor: Comprehending the implemented variability and its technical debt. *J. Syst. Softw.* 208 (2024), 111876. https://doi.org/10.1016/J.JSS.2023.111876

[28] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2021. Visualization of Object-Oriented Variability Implementations as Cities. In *2021 Working Conference on Software Visualization (VISSOFT)*. Luxembourg (virtual), Luxembourg, 76–87.

[29] Johann Mortara, Philippe Collet, and Anne-Marie Dery-Pinna. 2022. Customizable Visualization of Quality Metrics for Object-Oriented Variability Implementations. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 43–54.

[30] Johann Mortara, Philippe Collet, and Xhevahire Tërnava. 2020. Identifying and Mapping Implemented Variabilities in Java and C++ Systems using symfinder. In *Proceedings of the 24th ACM International Systems and Software Product Line Conference - Volume B* (Montreal, QC, Canada) *(SPLC '20)*. Association for Computing Machinery, New York, NY, USA, 9–12.

[31] Morteza Zakeri Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. 2023. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.* 204 (2023), 111796. https://doi.org/10.1016/J.JSS.2023.111796

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

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

[34] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *Queen's School of Computing TR* 541, 115 (2007), 64–68.

[35] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

[36] Alcemir Rodrigues Santos, Ivan do Carmo Machado, and Eduardo Santana de Almeida. 2016. Riple-hc: javascript systems meets spl composition. In *Proceedings of the 20th International Systems and Software Product Line Conference*. 154–163.

[37] Yuichi Semura, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. 2018. Multilingual Detection of Code Clones Using ANTLR Grammar Definitions. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 673–677. https://doi.org/10.1109/APSEC.2018.00088

[38] Marcel Steinbeck, Rainer Koschke, and Marc O Rudel. 2019. Comparing the evostreets visualization technique in two-and three-dimensional environments a controlled experiment. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 231–242. https://doi.org/10.1109/ICPC.2019.00042

[39] Margaret-Anne D Storey, Davor Čubranić, and Daniel M German. 2005. 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*. 193–202.

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

[41] Xhevahire Tërnava, Johann Mortara, and Philippe Collet. 2019. Identifying and Visualizing Variability in Object-Oriented Variability-Rich Systems. In *Proceedings of the 23rd International Systems and Software Product Line Conference - Volume A* (Paris, France) *(SPLC '19)*. Association for Computing Machinery, New York, NY, USA, 231–243.

[42] Xhevahire Tërnava, Johann Mortara, Philippe Collet, and Daniel Le Berre. 2022. Identification and visualization of variability implementations in object-oriented variability-rich systems: a symmetry-based approach. *Journal of Automated Software Engineering* 29 (Feb. 2022), 1–51.

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

[44] Xhevahire Tërnava and Philippe Collet. 2017. Tracing Imperfectly Modular Variability in Software Product Line Implementation. In *International Conference on Software Reuse (ICSR '17)*. Springer, 112–120.

[45] Alfredo R Teyseyre and Marcelo R Campo. 2008. An overview of 3D software visualization. *IEEE transactions on visualization and computer graphics* 15, 1 (2008), 87–105.

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

[47] Richard Wettel and Michele Lanza. 2007. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 92–99.

[48] Wenqing Zhu, Norihiro Yoshida, Toshihiro Kamiya, Eunjong Choi, and Hiroaki Takada. 2022. MSCCD: grammar pluggable clone detection based on ANTLR parser generation. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 460–470. https://doi.org/10.1145/3524610.3529161