Identification of Variability Implementations in TypeScript: the 2Cities Visualization

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

Identification of Variability Implementations in TypeScript: the 2Cities Visualization

Yann Brault
Université Côte d'Azur, CNRS, I3S
Sophia Antipolis, France
yann.brault@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

When variability is directly implemented in a single codebase with languages supporting many different mechanisms, its identification and comprehension are impeded by the absence of documentation, their scattered locations, and obviously, their diversity. This is typically the case with TypeScript in which variability can be implemented with Object-Oriented (OO) mechanisms, and design patterns, but also with dynamic loading of files. This latter mechanism allows for organizing internal variants of part of code in directories and files, usually containing code clones of different forms. In this paper, we demonstrate *2Cities*, a dedicated visualization based on the city metaphor to highlight variability implementations in a single TypeScript codebase. We introduce the detection toolchain that gathers all the necessary information and variability metrics. Then we detail the visualization mechanisms that use two dedicated cities whose relations also highlight the architecture of the implementation. The first one, *ObjectCity*, adapts the *VariCity* visualization with classes as buildings shaped by their variability and usage relationships as streets. The second one, *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.

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

Not all variability-intensive systems are implemented like a full Software Product Line (SPL). Many of them only realize variability in a single codebase through the mechanisms provided by the host programming language. As these mechanisms are also used to realize the business logic and that no additional information or annotation is available concerning variability [13], there is no traceability with the domain knowledge [11]. Worse still, the very identification of the implemented variability is hampered by its burying in the code base and by the diversity of the used mechanisms [5, 10]. This is the case with recent languages, which typically integrate many different mechanisms, just like TypeScript. This language is an extension with static OO typing of the JavaScript language, which combines imperative and functional programming with a prototype-based semantic.

In a companion paper, we manually studied the variability implementation mechanisms in TypeScript on several large open-source systems such as Angular, Grafana, Echarts, or Nest [1]. Several different kinds of implementation can then be related to the concepts of variation point (*vp*), a location where variation may occur, and variants, the specific ways the *vp* differ [3]. First, similarly to other OO languages, such as Java [12], mechanisms like inheritance between classes, and design patterns (*i.e.*, factory, strategy, template, decorator), can be spotted in TypeScript and are the first candidates to implement variability.

As shown in the work on *symfinder* and *VariCity*, these mechanisms can be directly related to *vp* and variants, *e.g.*, with inheritance the superclass is the *vp* while subclasses are variants. Interestingly, method overloading does not exist in TypeScript but can be simulated by several method signatures and one method declaration grouping all previous signatures.

In addition, variability in TypeScript can be implemented by reusing the JavaScript dynamic code loading capabilities through the concept of modules. It enables one to create reusable components that can be loaded from files through import/export instructions. They are structural patterns organizing folders on various hierarchical levels, sometimes with a main folder standing for a feature and sub-folders being the different implementations, frequently leading to file duplication, notably for files with the same goal, or having the same name. Figure 1 illustrates how the project Echarts, a charting library for TypeScript, handles the variability of its charts, with a `chart` folder as *vp* with its sub-folders containing the variants. These folders all duplicate the same file, `install.ts`, unifying the loading of charts in the system. Duplicated files may only contain functions (*cf.* Figure 1), or constants with different values, but they may also use classes and OO mechanisms, mixing the different implementation mechanisms.

To comprehend the variability implemented with OO mechanisms and design patterns, some recent approaches could be used to identify it [10, 11], and visualize it through a 3D city visualization [5–7]. However, the detection and visualization mechanisms were only applied to the Java language. Moreover, these approaches do not cover the identification of the internal variants that are characterized by file duplication and code clones in specific directories. While some 3D visualizations have also been proposed to represent code duplication metrics [2, 4] no overarching approach can currently detect and visualize the different variability implementation mechanisms of TypeScript.

In this paper, we demonstrate *2Cities*, a 3D visualization based on the city metaphor [14] to highlight variability implementations in a single TypeScript project from static analyses of its codebase. The visualization aggregates two city representations with links between them that can complement each separate representation. The

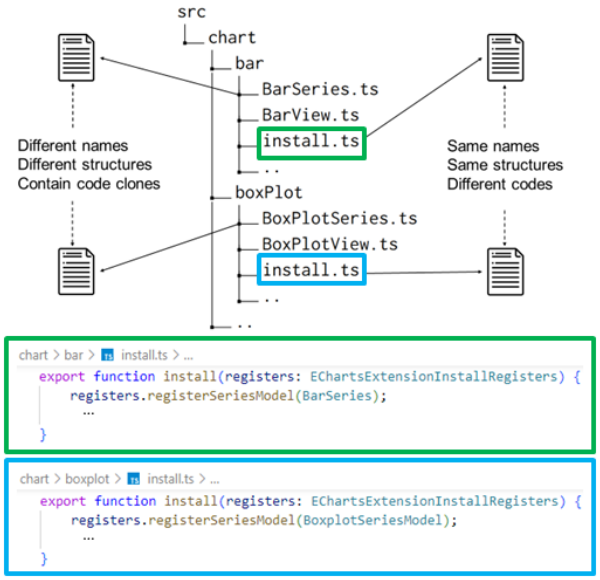


Figure 1: Duplication in TypeScript modules (Echarts)

first one, *ObjectCity*, focuses on displaying together classes that are heavily loaded in variability implementation mechanisms. It adapts the *VariCity* [5–7] visualization with classes as buildings shaped by their variability metrics and usage relationships as streets. Design patterns present in TypeScript are also displayed through specific crowns over the concerned building. The second one, *CloneCity*, is completely new. It gives a directory-centric overview of the codebase and focuses on internal code clones with information on the detection of files with the same names within all directories, and results of a code clone detection algorithm applied to all files. *CloneCity* represents the directory hierarchy as streets, with specific colors when similarities are present, and files as circular districts whose colors change according to the level of duplication found. In addition, different links (inheritance, bridge between the same class represented in each city, clone pairs) appear on hovering. All together the two representations and their relations facilitate the understanding of the implemented variability[1].

In the following, we first present in section 2 the detection toolchain that gathers all the necessary information and variability metrics. Then we detail how the visualization is built and used in section 3.

2 Toolchain

Figure 2 depicts the complete toolchain organization, which is split into three main parts. First, a source fetcher gets the source code of the subject system from a *git* repository given in parameter. Then a multi-stage detection searches for the OO variability implementations and the clone-related ones. Finally, the web-based visualization is generated so that it can be explored with a web browser.

[1]The source code of *2Cities* is available at https://github.com/DeathStar3/Varicity-TS, while a demonstration video can be found at https://youtu.be/tMxa1_q5fq4.

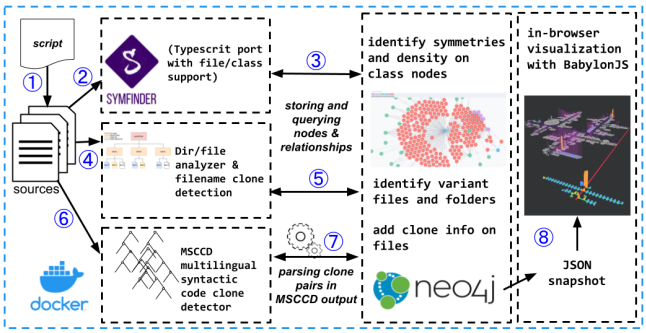


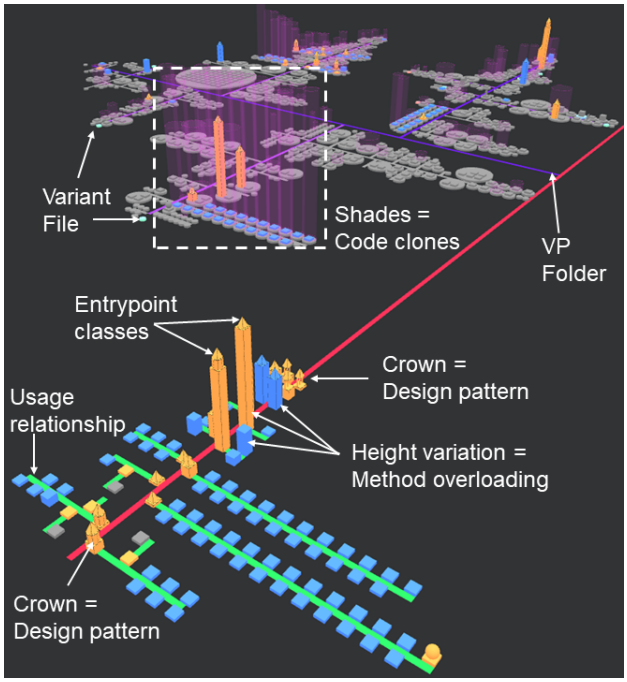
Figure 2: The *2Cities* toolchain

Detecting OO variability in TypeScript. To identify OO variability implementation mechanisms, the *symfinder* toolchain [10, 11] is reused and extended. We re-implemented the parser in TypeScript to analyze TypeScript code and detect inheritance, function overloading as defined in the language, and several design patterns (*strategy*, *template*, *decorator*, and *factory*). Following the *symfinder* principles, it identifies *vp*-s and variants and stores them in a Neo4J graph database [10]. The concept of *hotspot* is also reused to identify zones of interest, either containing many classes related by usage relationships or the use of several OO mechanisms, or a class heavily using one specific mechanism (*e.g.*, many methods being overloaded) [11]. This detection relates to steps 2 and 3 on fig. 2.

Detecting duplication at the file level. The second part of the detection process, steps 4 and 5 per Figure 2, performs the identification of duplication between files and folders. At this stage, the tool traverses files and folders to detect file duplication over names to identify potential variants based on the following rules: If a file is duplicated with the same name in at least two distinct folders at the same level, the file and its duplicate are defined as *Variant files*. In the case where the file is duplicated in all folders of the same hierarchical level, it is referred to as a *Core file*. We define a folder containing a *Variant file* as a *Variant folder*, and a folder containing at least two *Variant folders* as a *VP folder*.

Detecting duplication at the code level. Finally, a code clone duplication detection is conducted by an external tool, MSCCD [9, 15]. It is a language-independent code clone detection tool relying on ANTLR grammar to adapt its detection to several languages. It mainly detects Type 3 clones [2] as they are the only type of clones allowing for addition and deletion while keeping textual similarities [8]. We consider that this type of clone is the most appropriate to match a potential variability implementation mechanism. The detection of code clones, step 6 on Figure 2, outputs a list of clone pairs, which are pairs of identical or textually similar code fragments, ranging from a code instruction to a whole file [8]. This list is parsed and used to complement files data from the Neo4J graph database, step 7 on Figure 2, to label cloned files, and add code clone relationships between files of the same clone pair. We define a cloned file as a file that is part of a clone pair, and which

[2]A Type 3 clone corresponds to copied fragments including modification, addition, or deletion of statements, and changes in identifiers, literals, types, layout, and comments.



(white annotations are not part of the visualization)

Figure 3: The *2Cities* visualization for Nest

is consequently the source or the target of a code clone relationship. Finally, a snapshot of the database is extracted to a dedicated back-end server to be used for visualization (step 8 on fig. 2).

3 Visualization

While *VariCity* [5, 6] reuses the *symfinder* output on detected OO mechanisms to build a city-based visualization, it does not support intra-clone variability. To build a complete visualization, we decided to split it into two cities [14], rather than doing an extension of *VariCity*. As clone-related data bring a lot of information, *i.e.*, several hierarchical directory levels, file duplication, and code clones, this extension would be too overloaded. In addition, we consider that each type of variability implementation (*i.e.*, OO and clone-related) is worth visualizing by itself to comprehend their organizations, but also together to understand their interactions, their potential impacts on each other, and to make structural patterns appear. Figure 3 is used in the following to illustrate visual properties of the *2Cities* visualization.

ObjectCity. The first city is built following the *VariCity* principles [5, 6] so that classes concentrating variability implementations are highlighted. Buildings varying in size account for many internal variants, with method variants influencing the height and constructor variants the width. A variation in shape, specifically with crowns, identifies design patterns, for example, chimneys identify factories. Entrypoints buildings are the starting point of *ObjectCity*, representing classes of interest selected by the user. They are aggregated along a main red road and are identified with a pyramidal crown. All buildings representing classes related to them by usage relationships are organized around green roads starting from the concerned building. More links, such as additional usage or inheritance relationships, can be displayed as the building is hovered over. To make them noticeable, classes being part of *hotspots* are displayed in yellow for *vp*-s and blue for variants.

CloneCity. As the other part of the static analysis focuses on finding clones at the file level and within files throughout the whole codebase, the second city, *CloneCity*, is built to display the complete directory hierarchy given as input, usually a `src` folder. The city is thus structured by folders, displayed as streets. Files are aggregated along the streets and represented by circular districts. Each district has on its top the classes it exports, visualized similarly as in *ObjectCity*, which allows the distinction of files with many classes, but also the use of OO mechanisms within files, facilitating the identification of combined variability implementations. All elements are displayed in gray by default and are colored when detected as variability implementations. Duplicated folders are assigned a purple color, darkening according to their duplication level, while the more a file is duplicated, the more its color is vibrant. As file duplication is based on naming, a new color is assigned to the duplicated files for each different name, making it easier to differentiate each file. In addition, cloned files are identified by a cylindrical purple shade, whose height changes according to the number of code clones, allowing the detection of files with highly duplicated code. Finally, files of a clone pair are linked by an aerial purple bridge when one of them is hovered over or selected. All this helps reveal files that have duplicated code fragments from one another and help understand if it is a structural duplication or variability-related duplication.

Identification process. Each city can be analyzed by itself and used to comprehend the variability implementations it visualizes. However, the visualization is meant to start with *CloneCity* first to have a grasp of the file structure and then use the city to find classes of interest and input them into *ObjectCity*. This city is then extended with classes found from complementary information taken in the source code or the project documentation[3]. Classes displayed in both cities (*i.e.*, the ones exported by their file in *CloneCity*) have all the visual properties defined in *ObjectCity*, *i.e.*, color, size variations, design pattern crowns, and various links, and are connected by a cross-cities yellow bridge. However, if they are visualized in only one city, they keep their size variations and design pattern crowns if they have some, but are automatically grayed. This color distinction helps to understand which classes might only serve structural purposes from those serving project-wide features. If cities are worth visualizing by themselves, the bridging of both brings another dimension to help comprehend integration and interaction between folders, files, and classes within the whole codebase, as well as observing the structural organization of implemented variabilities.

For example, in the *ObjectCity* part on Figure 3 two towers detach from the city. The taller, located behind the highlighted clone shades, is a logger that provides multiple logging levels, all implemented through overloaded methods to accept various parameters,

[3]Contrary to *VariCity* [5], our current prototype is not integrated into an IDE and does not support code browsing from the visualization.

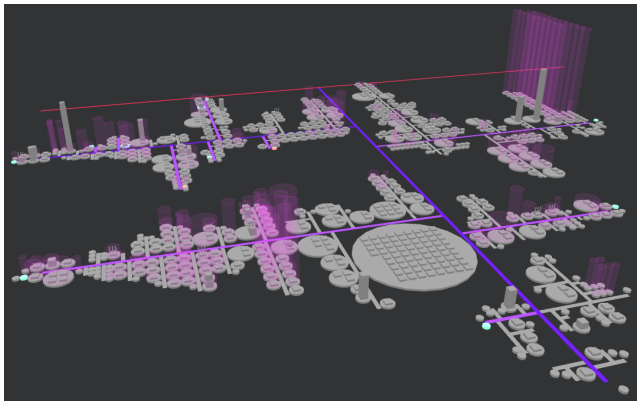


Figure 4: *CloneCity* for Nest (folder packages)

hence the size of the building. The second pillar, far in the background in *CloneCity*, is an adapter to communicate with different web servers, through duplicated methods and method overloading. Similarly, the long streets aggregating many classes are visible in both cities. All these classes are exceptions, with the longest street representing the hierarchy of runtime exceptions and the other web exceptions. In *CloneCity*, one can also observe that web exceptions are the only ones covered with clone shades. This shows that these exceptions combine multiple variability implementation mechanisms by heavily duplicating their codes. One could also interpret this as a bad smell.

4 Conclusion

2Cities is a visualization that highlights the different variability implementations that can be observed in variability-rich TypeScript projects that realize it in a single codebase. Based on the city metaphor applied twice, it displays a first city, *CloneCity*, that visualizes the directory hierarchy as streets and files as colored circular districts to show file duplication and code clones. The second one, *ObjectCity*, supports the visualization of Object-Oriented variability implementations with city displaying classes as buildings shaped by their variability metrics. Results on multiple large open-source projects are available in a companion paper [1].

As future work, we aim to enhance the whole toolchain to support JavaScript and cover more projects in a broader validation. In the longer term, we want to analyze other artifacts related to configuration and deployment to be able to identify variability across all of them.

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] Yann Brault, Philippe Collet, and Anne-Marie Dery-Pinna. 2024. Visualizing Variability Implemented with Object-Orientation and Code Clones: A Tale of Two Citie. In *Proceedings of the 28th International Systems and Software Product Line Conference - Volume A* (Luxemburg) *(SPLC '24)*.

[2] 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

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

[4] 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

[5] 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

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

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

[8] 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.

[9] 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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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