An Exploratory Study of Decorators in the TypeScript Programming Language

Luiz Martins[1], **Phyllipe Francisco**[1], **Paulo Meirelles**[2]

[1]Instituto de Matemática e Computação – Universidade Federal de Itajubá (UNIFEI)
37.500–903 – Itajubá – MG – Brasil

[2]Instituto de Matemática e Estatística (IME) – Universidade de São Paulo (USP)
Rua do Matão, 1010, Cidade Universitária, 05508-090 – São Paulo – SP

{luizebmartins,phyllipe}@unifei.edu.br, paulormm@ime.usp.br

Abstract. *Decorators are a feature of the TypeScript language that allows the configuration of additional metadata on code elements, such as members, methods, and the class definition itself. It has been officially supported only since version 5.0. This feature also exists in other languages, such as Java annotations and C# attributes. To understand and further study how developers use decorators, we conducted an exploratory study on 30 real-world, open-source, and popular projects hosted on GitHub. To extract the characteristics of decorators from the analyzed projects, the DSniffer (Decorators Sniffer) was developed. It can collect source code metrics' values dedicated to metadata configuration. Among the results obtained, it was observed that 46% of the projects use decorators, among which we highlight some popular ones such as Angular and Visual Studio Code. We also noted that decorators use at most four parameters, but these can be quite complex objects, occupying up to 84 lines of source code. Thus, the study reveals high numbers of decorators in the projects and many repeated metadata configurations. This can negatively impact the readability of the source code and compromise maintenance. The study carried out in this work can serve as a basis for defining reference values or implementing a software visualization approach.*

1. Introduction

Decorators were officially introduced in the TypeScript language in version 5.0. They first appeared in version 1.5 and remained experimental until version 4.9. They are a feature of the TypeScript programming language that allows custom metadata to be configured into code elements such as classes, methods, members, and properties. This feature is also present in other programming languages, such as annotations in Java and *attributes* in C#. In the context of the Java language, the work of [Lima et al. 2018] showed that annotations are widely popular and used by several enterprise frameworks. It was observed that at least 76% of ".java" files have one annotation, and there are projects where all files use this feature. Additionally, the work of [Lima et al. 2023] showed, through software visualization, that there is a relationship between the presence of an annotation and the responsibility of a class or Java package. These numbers indicate that such a feature is quite popular. To investigate *decorators* in the TypeScript language, this work aims to conduct an initial study to understand the usage of this feature. The goal is to explore and empirically determine how *decorators* are used in TypeScript projects.

To achieve this goal, this work uses source code metrics, as these can extract and measure characteristics of code elements. Within software engineering and development, metrics allow us to extract elements' size, complexity, and cohesion. We can also define reference values with these numbers and build software visualization approaches [Lanza and Marinescu 2006]. For this work, we used the metrics defined by [Lima et al. 2018], but we generalized them to include the TypeScript language, as they were originally defined for Java annotations. With the metrics defined, we developed the *Decorator Sniffer* (DSniffer), an open-source software tool capable of reading ".ts" files and generating a report with the values of the metrics and other characteristics of TypeScript source code. For the exploratory study, we selected 30 open-source projects hosted on GitHub, using their popularity as one of the selection criteria.

From the sample, we obtained a total of 84,191 TypeScript files. As a result, it was observed that 46% of the projects use *decorators*. The Angular project has the highest number, totaling 10,070 *decorators*, and the TypeORM project has the highest average value per file, with an approximate value of 2.54. We also identified projects with a high number of repeated *decorators* and arguments occupying 84 lines of code.

2. Configuring Metadata with TypeScript

Decorators are an advanced feature that allows the enhancement of classes and their components by applying extra layers of behavior and adding new characteristics. According to the TypeScript documentation[1], *decorators* are a way to add extra behaviors to class elements, such as methods, properties, accessors, or parameters. They are identified by the "@" symbol, and the expression after it must be a function called at runtime to pass information about the decorated element. Figure 1 presents an example of a *decorator*:

Listing 1. ES6 (ECMAScript-2015) Listing

```
1 type Constructor = { new (...args: any[]): any };
2 function decorator(constructor: Constructor) {
3    // Script
4 }
5 @decorator
6 class User {
7
8 }
```

Figure 1. Example with Decorator

A *decorator* in TypeScript is a function that receives specific parameters according to the component it decorates. For example, the *decorator* of a class receives the constructor as a parameter. On the other hand, the *decorator* of a method receives the class constructor, the method name, and the method *descriptor*. It is important to note that the parameters passed to the *decorator* functions are immutable.

3. Research Methodology

For this study, we used source code metrics to extract the characteristics of *decorators*. We then selected projects to extract the metric values from them. With the available

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

values, an exploratory study was conducted. To collect the metrics, the DSniffer tool was developed. In summary, the steps are listed as follows:

1. Redefine annotation metrics for *Decorators*
2. Develop the DSniffer tool to automate the process of collecting metrics from selected projects.
3. Select TypeScript software projects.
4. Conduct an exploratory study of the data obtained from the metric collection.

3.1. Redefinition of Metrics

The work of [Lima et al. 2018] proposed a suite of source code metrics dedicated to Java language annotations. The first step in this work was to analyze these metrics and define which ones would be adapted to *decorators* for TypeScript. Section 4 presents the original metrics for Java and the redefinition process for TypeScript *decorators*. The goal is to move towards the most generic suite of metrics possible.

3.2. DSniffer Tool

To collect the metrics, we developed the DSniffer[2]. This free software tool can read a ".ts" file and extract *decorator* metric values. The tool was developed in TypeScript.

The tool's input is a ".ts" file. Using the TypeScript compiler itself, the Abstract Syntax Tree (AST) is generated for this file. With the compiler's API, we can traverse the AST and search for *decorator* and its code elements. After collecting data from a project, a JSON report is generated with the metric values. For the exploratory study, these reports constitute the data to be analyzed and provide an initial understanding and empirical identification of how *decorators* are present in TypeScript projects.

3.3. Project Selection and Criteria

We selected 30 popular and free software projects hosted on GitHub to generate the data and perform the collection. It is understood that GitHub stars measure the popularity of a project and can be used as a valid selection criterion [Borges and Tulio Valente 2018]. Following is the list of criteria.

- Use the TypeScript language;
- More than 10 thousand stars;
- READ.ME is predominantly written in English;
- Exclude repositories that are collections for didactic purposes, such as tutorials, book companion material, etc.
- No syntax errors in ".ts" files.

The search string used on GitHub was: "`stars:"> 10,000" language:TypeScript`". After selecting, we obtained the list of projects shown in Table 1. The list contains widely used and established projects like Angular (web development framework), Visual Studio Code (text editor), and other frameworks, libraries, tools, and applications. This provides good representativeness and diversity for the selected project list. This diversity is important when conducting an empirical study on source code [Nagappan et al. 2013].

[2] `https://github.com/metaisbeta/DSniffer`

Table 1. List of Selected Projects

Name	Description	Link	Stars (k)
vscode	Code Editor	https://github.com/microsoft/vscode	147
Typescript	Typescript language	https://github.com/microsoft/TypeScript	91.7
Angular	Web development	https://github.com/angular/angular	88.4
Ant Design	UI library	https://github.com/ant-design/ant-design	86.1
Storybook	Tool to create components in web pages	https://github.com/storybookjs/storybook	78.9
Nest	Server-side framework	https://github.com/nestjs/nest	57.1
Ionic	Mobile development framework	https://github.com/ionic-team/ionic-framework	49.1
DefinitelyTyped	Typescript type definition	https://github.com/DefinitelyTyped/DefinitelyTyped	44.1
Vuetify	Library for Vue JS component	https://github.com/vuetifyjs/vuetify	37.3
Lerna	Manage and publish JS/TS packages	https://github.com/lerna/lerna	34.6
react-hook-form	Form validation	https://github.com/react-hook-form/react-hook-form	35.1
Typeorm	ORM	https://github.com/typeorm/typeorm	31.4
Prisma	ORM	https://github.com/prisma/prisma	31.7
n8n	Workflow automation	https://github.com/n8n-io/n8n	30.9
mobx	Library for state management	https://github.com/mobxjs/mobx	26.5
angular-cli	CLI tool	https://github.com/angular/angular-cli	26.1
appsmith	Low-code development tool	https://github.com/appsmithorg/appsmith	27.5
trpc	HTTP requests API	https://github.com/trpc/trpc	26
vue	Front-end framework	https://github.com/vuejs/vue.git	204
xstate	State Machine	https://github.com/statelyai/xstate	23.9
NativeScript	Mobile development framework	https://github.com/NativeScript/NativeScript	22.6
editor.js	Text editor	https://github.com/codex-team/editor.js	23.1
autocomplete	Autocomplete text in terminal	https://github.com/withfig/autocomplete	22
lenster	Social media app	https://github.com/lensterxyz/lenster	21.7
TanStack/table	Interface to create tables and grids	https://github.com/TanStack/table	21.6
directus	SQL Database management	https://github.com/directus/directus	21.7
zod	Create and validate schemas	https://github.com/colinhacks/zod	22.5
recharts	Library to generate charts	https://github.com/recharts/recharts	20.4
novu	Notifications API	https://github.com/novuhq/novu	20.8
expo	Mobile development framework	https://github.com/expo/expo	20.7

3.4. Data Collection and Exploratory Study

With the list of projects prepared and the DSniffer developed, we proceed with data collection, i.e., for each project, we run the tool to extract the metric values for *decorators*.

After extracting the metrics, the next step is to conduct an exploratory study on the obtained values to identify the usage of *decorators*. We aim to answer questions such as, but not limited to: (i) Are *decorators* used in free software projects? (ii) Are there files that concentrate more *decorators*? (iii) Are the *decorator* values notably large? (iv) Do the *decorators* present have many arguments?

4. Source Code Metrics for Decorators

Source code metrics are valuable in software engineering as they allow developers to maintain control over the unchecked growth of specific system characteristics, such as coupling, complexity, and lack of cohesion [Lanza and Marinescu 2006, Valente 2020]. There are several widely known metrics, such as LOC (*Lines of Code*), CBO (*Coupling Between Objects*), LCOM (*Lack of Cohesion of Methods*), NOM (*Number of Methods*), and others [Chidamber and Kemerer 1991].

In the context of the Java language and annotations, the work of [Lima et al. 2018] defined a suite of seven metrics that measure characteristics such as complexity, size, and coupling of Java annotations. More generally, this suite measures characteristics of metadata configurations declared in a class (or in a file), but within the context of Java.

For this current work, we selected four metrics and validated them in the context of *decorators* with TypeScript. To facilitate understanding, these four metrics will be presented with their original names (Java context) and with their redefined names for generalization.

- AC - *Annotations in Class* (Annotations in Class) =¿ This metric counts the number of annotations declared in all elements of the class, or more generally, in a ".java" file. It emits a value per class. For TypeScript, it will measure the number of *decorators* present in a ".ts" file. To generalize, its name will be redefined as **MF**: *Metadata in File*.
- UAC - *Unique Annotations in Class* (Unique Annotations in Class): This metric counts the number of annotations that are unique in the class. For TypeScript, it will measure the number of distinct *decorators* in a ".ts" file. A *decorator* is considered unique under the same conditions as annotations in Java. To generalize, its name will be redefined as **UMF**: *Unique Metadata in File*.
- AA - *Attribute in Annotations* (Attribute in Annotation): Annotations in Java can receive parameters, including other annotations. The AA metric measures the number of parameters for each annotation. In the context of TypeScript, *decorators* can also receive parameters, and this metric will count the number of parameters/arguments per *decorator* for each ".ts" file. To generalize, its name will be redefined as **AM**: *Arguments in Metadata*.
- LOCAD - *Lines of Code in Annotation Declaration* (Lines of Code for Annotation Declaration): The LOCAD metric counts the number of lines used to declare the annotation, or *decorator*. For generalization, its name will be redefined as **LOCMD**: *Loc in Metadata Declaration*.

5. Exploratory Data Analysis and Results

In this section, we present the results and discussion of the exploratory study on the metric values obtained from the collection of a sample of 30 real-world, open-source projects written in TypeScript.

5.1. Total Number of *Decorators* per Project

To calculate the total number of *decorators* used in a project, we use the MF metric (*Metadata in File*), which represents the number of *decorators* in each ".ts" file. To obtain the total value for the project, we added the MF values of all files. Figure 2 presents each project's number of *decorators* found.

Observing Figure 2, we have 16 projects, or 53%, that use at least one *decorator*. Comparing this with annotations in Java, the work by [Lima et al. 2018] shows that 76% of projects have at least one annotation. The lower usage in TypeScript might be associated with the fact that *decorators* became official only in version 5.0 of the language [Rosenwasser 2023]. On the other hand, 14 projects, or 47%, do not use *decorators*, opting for other approaches to achieve the same goals. This may also suggest that their use is tied to a particular coding style, and not all developers may find it suitable.

From the results, the Expo[3] and Vue[4] projects each use a single *decorator*. In the Vue project, a generic *decorator* is used in a class, but it is unclear what it does.

[3] `https://docs.expo.dev/guides/typescript/`

[4] `https://vuejs.org/`

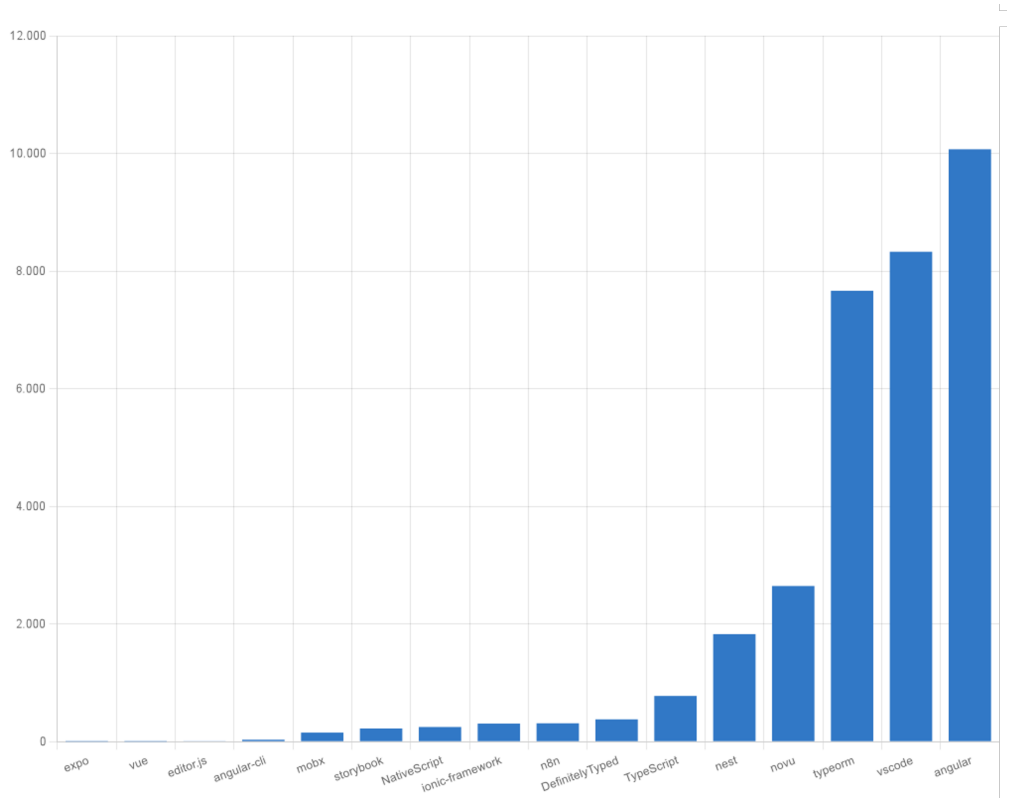

Figure 2. Total decorators per project

Examining the code suggests that a developer was testing the reception of *decorators* in this class and did not remove the code. In the Expo project, the use of *decorators* seems even more generic and is unrelated to the project itself.

Looking at the projects that use the most *decorators*, we have Angular (10,070), Visual Studio Code (8,328), and TypeORM (7,665). These projects use significantly more *decorators* than the others. Furthermore, both Angular and Visual Studio Code are among the three most popular TypeScript projects hosted on GitHub. This observation suggests that popular and complex projects make extensive use of *decorators*.

5.2. Average *Decorators* per File

To calculate this average value, we use the arithmetic mean. We extract the values from the MF metric and divide by the total number of files in the project. Figure 3 presents the results.

Observing Figure 3, we notice that the top three projects that use *decorators* remain the same as in Figure 2, suggesting that these projects do not concentrate *decorators* in just a few files, but rather distribute them throughout the entire project. However, there has been a change, and the TypeORM project now holds the top position in Figure 3, with an average of 2.5 *decorators* per file.

5.3. Distinct *Decorators* per Project

The use of *decorators* in projects is characterized as an additional metadata configuration on code elements. If several *decorators* are identical, i.e., they configure the same

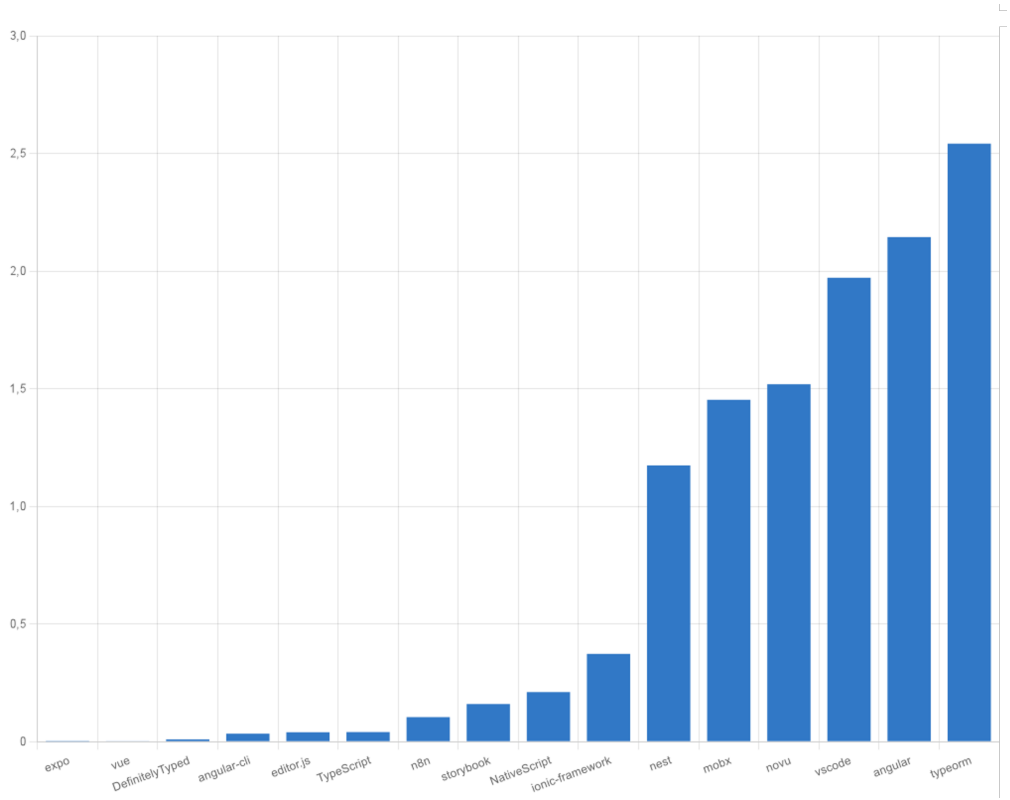

Figure 3. Average Value of Decorators per File

metadata, it could indicate that there is much repetition of *decorators* across files. If this is detected, it may be possible to combine the use of *decorators* with code conventions [Teixeira et al. 2018].

To measure the number of distinct *decorators*, we used the metric UMF (*Unique Metadata in File*), which counts, for each file, the number of distinct *decorators*. The metric checks for distinction by file, and to ensure that the *decorator* is unique across the project, we performed a second processing step. Figure 4 presents the results of extracting the number of distinct *decorators* per project.

Comparing Figures 2 and 4, we notice some interesting characteristics. Most projects show a direct relationship between the total number of *decorators* per project, suggesting that the higher the total number of *decorators*, the higher the number of distinct *decorators*. However, this is not always true. For example, the "novu" project has a higher total number of *decorators* than the "nest" project, as shown in Figure 2. But when we analyze the number of distinct *decorators*, the "nest" project has a higher count than the "novu" project, as shown in Figure 4. Thus, we notice that some projects repeatedly use the same metadata configuration strategy. Therefore, a hybrid approach, combining code conventions and *decorators*, can be considered.

Projects like Angular have a large number of *decorators* (1,070) and also a large number of distinct *decorators* (5,653). This suggests that it is a complex project that requires unique metadata configurations throughout the project. On the other hand, projects like Visual Studio Code experienced a significant drop when considering only the distinct

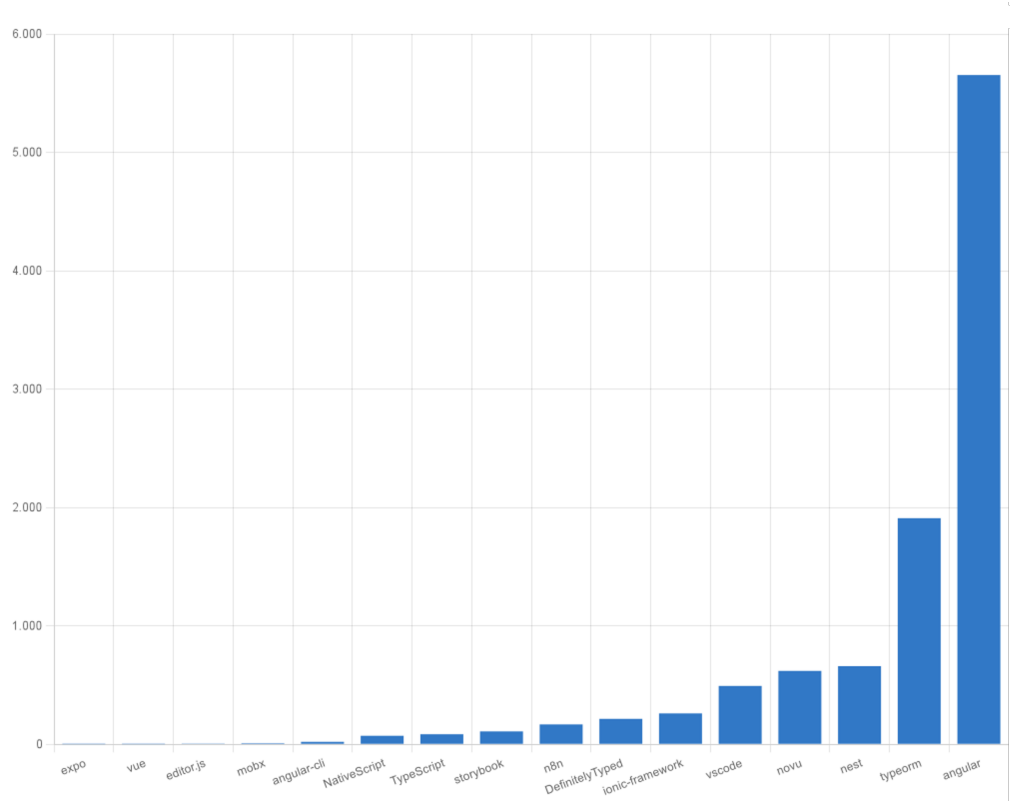

Figure 4. Unique Decorators per Project

decorators, falling from 8,328 to 493. Therefore, this is evidence of repeated metadata configurations.

5.4. Argument Values in *Decorators*

As presented in Section 2, *decorators* can receive parameters (or arguments). To measure this characteristic, we used the AM (*Arguments in Metadata*) metric, as introduced in Section 4. The goal of this analysis is to check if there are *decorators* with high numbers of parameters, suggesting highly complex configurations. Table 2 presents the results of the *decorators*, by project, with the most parameters.

Observing Table 2, the *decorator* with the most arguments has 4, belonging to the NativeScript project. This shows that the values are low. Comparing this result with the Java language, the work by [Lima et al. 2018] showed that the highest value found was 9, which is twice as high as the case with *decorators*. This initially suggests that, in the TypeScript context, the metadata configuration does not have significant complexity, and the *decorators* receive only the minimal parameters needed to add additional information to the code elements.

5.5. Lines of Code Used by *Decorators*

Looking solely at the number of arguments used may not reflect the true complexity of these *decorators*, as they can receive objects that, in themselves, can be naturally complex. Therefore, it is not the number of arguments but rather their complexity that matters. To address this, a second analysis involves measuring the number of lines used to declare

Table 2. Decorators with the most arguments

Project	Decorator	Arguments
angular-cli	Component	1
mobx	action	1
angular	ViewChild	2
ionic-framework	HostListener	2
novu	Matches	2
storybook	ViewChild	2
TypeScript	y	2
DefinitelyTyped	filter	3
n8n	Length	3
nest	EventPattern	3
typeorm	ManyToOne	3
vscode	throttle	3
NativeScript	PseudoClassHandler	4

the *decorators*. The LOCMD (*LOC in Metadata Declaration*) metric serves this purpose, as presented in Section 4. Table 3 presents the values obtained after analyzing, for each project, which *decorator* used the most lines in its declaration.

Comparing Tables 3 and 2, some values stand out and may even seem contradictory. While the *decorator* with the most arguments has a value of four, the *decorator* that used the most lines has a value of 86. The latter is the "NgModule" from the Angular project, as seen in Table 3.

By analyzing the code of the Angular project, where the NgModule *decorator* was used, it can be observed that there is no direct relationship between the number of arguments and the number of lines it occupies. This *decorator* has only one parameter, but it occupies 86 lines because it is a very complex JSON (JavaScript Object Notation) object with several properties. In the Angular project, the *decorator* that uses the most parameters is actually ViewChild, which has two parameters, but does not occupy as many lines of code.

As a general observation, most of the *decorators* used in the analyzed projects are simple and straightforward, with a low number of arguments and few or only a single line of code being used.

6. Conclusion

Decorators were considered experimental until TypeScript version 4.9. During this period, they were refined to better meet developers' needs. However, due to their experimental status, empirical studies on the use of *decorators* in open-source software projects are still scarce, making it difficult to understand the advantages and disadvantages that this feature can bring to the code.

This work used and redefined source code metrics, designed initially for Java annotations, to better understand the use of *decorators* in TypeScript projects. It was nec-

Table 3. Decorators the most lines of code

Project	Decorator	Lines
editor.js	_	1
expo	d	1
mobx	observable	1
NativeScript	NativeClass	1
vue	decorate	1
TypeScript	(	5
vscode	throttle	5
n8n	JoinTable	11
angular-cli	NgModule	14
typeorm	JoinTable	27
nest	Module	32
ionic-framework	NgModule	37
novu	Component	45
storybook	Component	59
DefinitelyTyped	Component	78
angular	NgModule	86

essary to adapt them to be more generic and applicable to the *decorators* context. To extract the metric values, we developed the DSniffer tool. For the exploratory study, 30 real-world projects were selected. The total sample consisted of 84191 ".ts" files.

Our findings show that 46% of these projects have at least one *decorator*. The Angular project has the highest presence, with 10,070 *decorators* identified in this project. Additionally, we observed repeated configurations, with many identical *decorators* scattered across the projects. We also found no direct relationship between the number of arguments in a *decorator* and how many lines it requires to configure the element.

Among the threats to this project's validity is the development of the DSniffer tool. Since we couldn't find a similar tool, its accuracy was validated manually. Regarding the exploratory study, the criteria defined for project selection may affect the values found, even though we minimized these effects by following related works. Finally, as seen, the mean value is not representative, and therefore, another statistical approach is needed to try to obtain reference values (*threshold values*).

For future work, we want to expand the metric suite to extract more information and characteristics of *decorators* present in the source code. With the available metric values, a natural next step is to use or define some software visualization approach to extract more insights from the metrics and bring the values closer to the developer. It is also possible to use the tool as part of the continuous integration pipeline in the project, where the metric values would be analyzed. Another relevant point is to study, through interviews or surveys, how the use of *decorators* by a team can impact their daily development work, whether in terms of maintainability, code readability, or team productivity.

References

Borges, H. and Tulio Valente, M. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object-oriented design. In *Proceedings...*, pages 197–211, New York. ANNUAL ACM SIGPLAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING SYSTEM, LANGUAGES, AND APPLICATIONS, ACM.

Lanza, M. and Marinescu, R. (2006). *Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer.

Lima, P., Guerra, E., Meirelles, P., Kanashiro, L., Silva, H., and Silveira, F. (2018). A metrics suite for code annotation assessment. *Journal of Systems and Software*, 137:163 – 183.

Lima, P., Melegati, J., Gomes, E., Pereira, N. S., Guerra, E., and Meirelles, P. (2023). Cadv: A software visualization approach for code annotations distribution. *Information and Software Technology*, 154:107089.

Nagappan, M., Zimmermann, T., and Bird, C. (2013). Diversity in software engineering research. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 466–476, New York, NY, USA. ACM.

Rosenwasser, D. (2023). Announcing typescript 5.0 rc. `https://devblogs.microsoft.com/typescript/announcing-typescript-5-0-rc/`.

Teixeira, R., Guerra, E., Lima, P., Meirelles, P., and Kon, F. (2018). Does it make sense to have application-specific code conventions as a complementary approach to code annotations? In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection*, pages 15–22.

Valente, M. T. (2020). Engenharia de software moderna. *Princípios e Práticas para Desenvolvimento de Software com Produtividade*, 1.