Class Archetypes: Principles, Detection, Evolution

Mattia Giannaccari
REVEAL @ Software Institute — USI, Lugano, Switzerland
mattia.giannaccari@usi.ch

Marco Raglianti
REVEAL @ Software Institute — USI, Lugano, Switzerland
marco.raglianti@usi.ch

Michele Lanza
REVEAL @ Software Institute — USI, Lugano, Switzerland
michele.lanza@usi.ch

Abstract

Classes are the basic building blocks of object-oriented software systems. To understand key parts of the system we need to understand the roles that classes play by looking at their structure and their behavior. However, source code alone is too opaque to convey such roles, especially in large and complex systems. Moreover, as software evolves, classes originally designed for a single purpose often get new responsibilities, changing, expanding, and complicating their roles over time.

We present a heuristic-based approach for fully automatic class stereotypes identification. We use CodeQL queries to capture structural properties (*e.g.,* attribute composition, inheritance patterns) and to infer behavioral properties (*e.g.,* method interactions, attribute accesses) of classes. We leverage this information to determine intrinsic and system-level class roles. Based on these properties, we define class archetypes that simplify system understanding by using a common pattern language to highlight single and co-occurring roles. We applied our heuristics to a dataset of over 650 Java GitHub repositories. Besides the scalability of our approach, preliminary observations reveal underlying temporal trends in how certain class roles emerge, endure, or evolve, suggesting that automated, property-based class role inference can support future research on software design and system evolution analyses.

Replication Package: **https://doi.org/10.6084/m9.figshare.30610502**

CCS Concepts

• **Software and its engineering → Object oriented architectures**; **Abstraction, modeling and modularity**; *Software reverse engineering*; **Software evolution**; *Maintaining software*; Software maintenance tools.

Keywords

class archetypes, static analysis, object oriented systems, system comprehension, software evolution analysis

ACM Reference Format:
Mattia Giannaccari, Marco Raglianti, and Michele Lanza. 2026. Class Archetypes: Principles, Detection, Evolution. In *34th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (FSE Companion '26), July 05–09, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3803437.3805584

FSE Companion '26, Montreal, QC, Canada

ACM ISBN 979-8-4007-2636-1/2026/07
https://doi.org/10.1145/3803437.3805584

1 Introduction

Classes are the main abstraction unit of object-oriented software systems. They encapsulate data and behavior, define relationships, and organize responsibilities within a system [14].

Understanding class roles is key to program comprehension, maintenance, and evolution [3, 4]. However, inferring the intended role of a class just by reading source code is challenging, particularly because a class role is rarely "pure" and often depends on relationships that are not explicitly visible in the code [11, 16, 20, 26]. Moreover, as software evolves, classes frequently acquire new responsibilities, drifting away from their original design intent [25, 28, 29]. A class initially created to store data may later implement logic to validate data or coordinate data management of other dependent classes. This evolution complicates identifying class roles and increases the difficulty of reasoning about overall system design.

Class stereotypes provide exactly such a high-level abstraction of the design intent, hiding the nitty-gritty implementation details, and serving as a bridge between the two. Stereotypes describe common recurring roles that classes play in a system (*e.g.,* data holders, utility providers, creational helpers) [15, 29]. By using stereotypes for system modeling, developers can decide class responsibilities even before implementing them, and explicitly communicate design choices, supporting both consistency and understanding across development teams [15, 29]. Class stereotypes can also be reverse engineered from existing implementation artifacts to understand system evolution [5, 7, 9, 24], to generate automatic documentation [2, 19, 21], and to improve program comprehension [8, 10, 17].

Among the proposed approaches for automatic class stereotypes extraction, some use machine learning to classify stereotypes based on structural class features [22, 27]. Others analyze the frequency distribution of method-level patterns [1, 18], trading off abstraction level for low granularity discriminative power. Some approaches leverage visual patterns to infer stereotypes, offering rule-of-thumb visual cues for quick identification [12, 17], but fail to systematically capture stereotypes at scale without human interpretation.

Yet Another Classification? Current approaches to class stereotyping generally follow two directions. One defines semantic stereotypes a priori to declare design intent *before implementation*. The other reverse engineers stereotypes from existing code by mapping *both* implementation patterns and code metrics to a *single stereotype* per class. These methods face limitations: Taxonomies can be large and hard to remember, and labels may not clearly reflect class semantics, reducing their practical usefulness. We argue that these limitations hinder the effective adoption of class stereotypes.

We propose *Class Archetypes*, an approach distinct from class stereotypes, derived from a class's semantic role rather than metrics, which are used only for secondary classification. Our method also considers system-level usage and allows multiple archetypes per class, acknowledging that classes are rarely semantically pure.

Table 1: Class archetypes: Definition and comparison with the ones in the literature.

Archetype	Description	[15]	[17]	[29]	[10]
Constants Definer	Stores constant data (*e.g.,* settings, default values) to be available to other classes. A class is a Constants Definer if it defines at least one class-side constant attribute (*i.e.,* a `final static` attribute in Java) that is accessed from a non-local method.	»	✗	»	✗
Creational	Responsible for creating instances of other classes (*e.g.,* factory or builder design patterns). A class is Creational if it defines at least one *Creational Method*. A method qualifies as such, if all of the following conditions hold: It creates an instance of a class other than the one in which it is defined; It returns the created instance or a collection of such instances; The created instance is not assigned to any field of the defining class; The method is not invoked from within the class.	»	✗	✓	✓
Data Holder	Stores data rather than implementing complex logic. A class qualifies as a Data Holder if at least 80% of its attributes are Data Holder Attributes, that is, attributes where every accessing method is public, local to the class, and functions exclusively as a getter, setter, constructor, or lazy initializer.	»	✓	»	«
Interface Definer	Specifies methods that its subtypes implement to ensure a shared usage interface. A class is an Interface Definer if all of the following conditions hold: It defines no attributes; It defines at least one method; None of the defined methods have an implementation; All of the defined methods are overridden.	✗	✓	✓	✗
Marker	Names a type without defining any logic (*e.g.,* for type-checking purposes). A class is a Marker if all of the following conditions hold: It defines no attributes or methods; It defines at most one constructor; The only defined constructor only calls the superclass constructor.	✗	✓	✗	✓
Singleton	Ensures that at most one instance of the class exists and provides a global access point to it. A class is a Singleton if all of the following conditions hold: It declares a class-side attribute of the same type as itself; It declares a class-side method that returns this instance (*i.e.,* a getter); All of its constructors are private.	»	✗	»	✗
Utility	Defines reusable pieces of logic that do not alter the state of the program. Defines at least one *Utility Method*, which is the case if it is class-side; it is not empty; no class-side method in its call tree modifies the state of any field (*i.e.,* it is a *pure function*).	✗	✗	✗	✗
Wrapper	Encapsulates another class to modify or control how it is accessed (*e.g.,* by adapting the interface of the wrapped class). At least 80% of its methods are *Wrapper Methods*. A method qualifies as such, if all of the following conditions hold: It invokes a method of the wrapped entity; It contains fewer than four statements (*i.e.,* no complex logic); It does not call any method nor modifies the state of its defining class.	»	»	«	«

2 Related Work

Class stereotypes were introduced as a high-level means to extend the Unified Modeling Language (UML) and to explicitly communicate a design intent [15, 28, 29]. Subsequent research shifted to reverse engineer intent from implementation. Different approaches to stereotype reverse engineering have been explored. Some rely on predefined lists of stereotypes derived from the observation of implementation artifacts [10], from syntactic properties without predefined categories [23], or from visual patterns that reveal class roles through custom visualization metaphors [13, 17]. Dragan *et al.* [8, 10] analyzed the frequency distribution of method stereotypes to infer class stereotypes, framing them as semantic abstractions of software behavior. Clarke *et al.* explored structural clustering of classes based solely on syntactic characteristics, demonstrating that design categories can arise even without a predefined taxonomy [23]. There are various strategies for the automatic detection of class stereotypes (*e.g.,* rule-based pattern recognition [1, 18], including machine learning-based stereotype classification [22, 27]).

Previous research on method and class stereotypes evolution focused on detecting how changes in the code trigger a stereotype change [7], on characterizing commits with stereotypes [9], and on automatically generating commit messages via stereotypes [5, 24].

Our work defines a new set of stereotypes (*i.e., Class Archetypes*) derived by applying heuristics to a large number of object-oriented Java repositories. Our work (in progress) focuses on the interactions between classes, methods, and attributes, an aspect that has been under-explored in the previous literature. For example, Lanza and Ducasse examined class relationships in detail (detecting 13 inheritance patterns that we do not detect), but their work was geared toward visual pattern recognition rather than automatic detection [17]. Moreover, previous research on stereotype and code co-evolution has focused on analyzing method stereotypes in single commits [5, 7, 9, 24]. We analyze how the archetypes of a class change throughout the entire history of a repository, including the acquisition of new archetypes and changes or loss of existing ones.

3 Class Archetypes

Class Archetypes capture two complementary aspects of a class: Its internal structure and behavior (*e.g.,* attribute composition, method interactions), and its role in the system. The former reflects the design intent behind the class. The latter reveals how effectively this intent aligns with the class' actual role within the codebase.

Table 1 defines our growing collection of class archetypes, providing a synthetic comparison to existing categorizations. ✓ means that the previous work defined the same concept, possibly with an alternative name. ✗ means that the previous work does not feature such a concept. « means that a concept similar to our archetype is present but split into a finer-grained classification. For example, the work of Dragan *et al.* [10] features a finer-grained classification, splitting the data holder archetype into three subcategories according to the distribution of method stereotypes. » is the opposite: The related work provides coarser-grained concepts compared to our approach (*e.g.,* the broad categories defined by Jacobson *et al.* [15]).

Our catalogue of archetypes is not complete, as we will show in the next section. In all approaches, including ours, there are cases which cannot be mapped (yet) to a concept, sometimes collected in broad "fallback" categories, for example, Entity in the work of Dragan *et al.* [10]. With respect to their work, we also merged *Data Class* and *Minimal Entry* into a single *Data Holder* archetype, yet encompassing more than the *Data Storage* stereotype of Lanza and Ducasse [17]. In our classification, a minimal entry is a data class with additional constraints regarding its size. Furthermore, unlike Dragan *et al.*, we do not allow data classes to include complex logic (*e.g., Command* methods [10]); such classes are instead categorized as regular classes. The only logic we allow within a data class is the special case of lazy initialization implemented in a getter method. We also merged, for consistency, *Controller* and *Pure Controller* from Dragan *et al.* (and *Coordinator* and *Controller* [29]) into a single *Wrapper* archetype. In our classification, a class that exhibits only the *Wrapper* archetype is equivalent to a pure controller; otherwise, if it has additional archetypes, it corresponds to a controller.

4 Analysis

CodeQL[1] is a semantic code analysis engine that allows querying code as data in a database. We implement our heuristics as CodeQL queries and execute them on 652 Java repositories from GitHub. Using `seart-ghs` [6], we selected Java projects with at least 100 stars, 100 watchers, and 35,000 lines of code. From the 726 retrieved repositories, we filtered out 74 for which CodeQL could not build a database. An example of CodeQL database query to detect the *Utility* archetype is shown in Figure 1. All the queries and their results are included in the replication package.

```
class UtilityMethod extends ClassMethod {
  UtilityMethod() {
    this.isPublic() and this.getBody().getNumStmt() > 0
    and not exists(FieldAccess a | this = a.getEnclosingCallable())
    and not exists(Callable callee, FieldWrite write |
       this.calls*(callee) and not callee instanceof Constructor
       and callee = write.getEnclosingCallable()
)}}
```

Figure 1: Example of CodeQL sub-query to identify utility methods in utility classes.

To efficiently collect data at scale, we developed a parallelized miner with the following workflow: We perform a shallow clone (*i.e.,* clone with a depth of 0, no history) of each repository to minimize storage and network overhead; we create the CodeQL database for the repository and perform all our custom queries on it; finally, we decode the binary output from each query as JSON (JavaScript Object Notation) and merge all the outputs into a single TSV (tab-separated values) file for the subsequent analyses.

For the temporal analysis (Section 4.3), we leverage a similar process for every commit. We clone each repository (this time with a deep clone for the complete history) and copy it n times locally, allowing n parallel processes to analyze a distinct batch of commits. Each process repeats checkout, database build, query execution, and output decoding, independently for each commit in the batch.

Overall, our queries collect 47 structural and behavioral properties for a total of 1,967,993 classes. Eight boolean labels mark the presence or absence of each of our class archetypes.

Figure 2 shows the distribution of the percentage of classes covered by at least one archetype across all analyzed repositories via a violin plot. The median of classified archetypes is about 45%, with most repositories having between 30% and 60% of their classes automatically assigned to at least one archetype. A small number of repositories shows higher classification coverage (up to 90%), while very few have extremely low coverage. This indicates that our heuristics can automatically identify at least one archetype for about 45% of the codebase on average, with just eight archetypes.

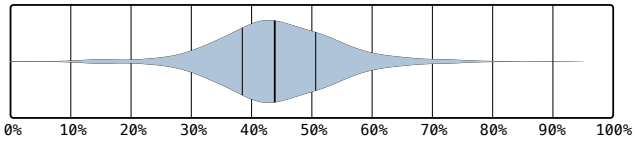

Figure 2: Archetypes distribution across repositories.

[1] https://codeql.github.com/

4.1 Coverage Analysis

The number of occurrences of class archetypes across approximately two million classes gives a clue about how developers structure large object-oriented systems (with an average of more than three thousand classes per system). Table 2 shows that the most prevalent single-role archetypes are *Creational*, *Data Holder*, and *Interface Definer*, together spanning roughly 25% of all classes.

Table 2: Count of single occurrences, co-occurrences and total number of archetypes.

Single Archetype	Count (#)	Count (%)
Creational	206,012	10.47%
Data Holder	150,455	7.65%
Interface Definer	145,238	7.38%
Utility	99,140	5.04%
Constants Definer	79,912	4.06%
Marker	42,363	2.15%
Wrapper	27,113	1.38%
Singleton	3,449	0.18%
	753,682	38.30%

Co-Occurrence	Count (#)	Count (%)
2 Archetypes	124,286	6.32%
3 Archetypes	15,683	0.80%
4 Archetypes	410	0.02%
5 Archetypes	13	0.00%
	140,392	7.13%

	Count (#)	Count (%)
1+ Archetypes	894,074	45.43%
No Archetypes	1,073,919	54.57%

One notable example is the repository `LWJGL/lwjgl3`,[2] a wrapper for various low-level graphics APIs, which shows one of the highest classification rates, with 54% of classes being constant definers, 58% creational classes, and 67% utility classes. This characterization aligns with the nature of the project: As a wrapper for low-level APIs, it contains many constants and utility classes that model and allow interaction with the underlying APIs. There are few stateful classes, since most of its functionality involves delegating calls to the underlying native libraries and constructing higher-level objects from their results.

Conversely, the relatively low occurrence of the other four archetypes, may indicate a different style of object-oriented design. Rather than emphasizing traditional object-oriented abstractions and inheritance hierarchies, modern Java libraries, especially those focused on performance and interoperability, tend to favor static utility methods and immutable data structures. This shift suggests a more pragmatic and functional use of Java, where object orientation is employed primarily for organization and encapsulation rather than as the dominant architectural paradigm.

4.2 Co-Occurrence Analysis

Table 3 shows the occurrences of each archetype (diagonal) and the co-occurrences for each pair of archetypes. The diagonal values are much larger than off-diagonal values, showing that most classes are primarily of a single archetype, with little overlap (less than 1% given from the ones with more than two archetypes). *Singleton* most frequently co-occurs with other archetypes (89.2%), suggesting it is more a property than a standalone role. *Constants Definer* (37.9%) and *Utility* (36.8%) follow, highlighting how the class-side can be as a standalone entity or a support to other archetypes. In contrast, *Interface Definer* and *Marker* (both mutually exclusive with all the other archetypes, by definition) show no co-occurrences, indicating purely standalone roles. Overall, archetype hybridization is low, with most classes assigned to a single type.

[2] https://github.com/LWJGL/lwjgl3

Table 3: Co-occurrence of class archetypes.

	Creational	Utility	Constants Definer	Data Holder	Singleton	Wrapper	Interface Definer	Marker
Creational	206,012	24,205	17,315	8,494	25,987	1,592	0	0
Utility	24,205	99,140	19,323	13,698	230	182	0	0
Constants Definer	17,315	19,323	79,912	10,255	1,657	283	0	0
Data Holder	8,494	13,698	10,255	150,455	753	299	0	0
Singleton	25,987	230	1,657	753	3,449	13	0	0
Wrapper	1,592	182	283	299	13	27,113	0	0
Interface Definer	0	0	0	0	0	0	145,238	0
Marker	0	0	0	0	0	0	0	42,363
Total	283,605	156,778	128,745	183,954	32,089	29,482	145,238	42,363
Co-Occurrence	**77,593** *(27.36%)*	**57,638** *(36.76%)*	**48,833** *(37.93%)*	**33,499** *(18.21%)*	**28,640** *(89.25%)*	**2,369** *(8.04%)*	**0** *(0.00%)*	**0** *(0.00%)*

4.3 Temporal Analysis

To conclude our analysis, we extend it to the domain of time with a notable example from the `ArgoUML` GitHub repository.[3] Figure 3 shows the full evolution of the `ToDoList` class. Since its creation, the class contained three constant values, leading it to be initially classified as a *Constants Definer*. In commit `#d8e44de`, although the number of constants remains unchanged, they are no longer referenced by other classes, causing the class to lose its archetype.

In commit `#fb9ed46`, the methods `getDecisions` and `getGoals` are converted to `static` and remain so until the latest commit, causing the class to be identified as a *Utility* archetype. In this commit, the implementation of the two methods remains the same, only the signature changes. We could argue that they should have been `static` from the beginning, thus highlighting how this temporal analysis can shed light on errors and wrong patterns of the past.

The methods `getOffenders` and `getPosters` change their return type in commit `#20ef946` from `VectorSet` to `ListSet`. Since `ListSet` is a defined within the same system, `ToDoList` is subsequently categorized as *Creational*. Later, in commit `#35ba393`, these methods store the created instances to internal variables before returning them, which causes the class to lose its *Creational* archetype. `ToDoList` represents an atypical implementation of the *Singleton* pattern, as its constructor is not declared `private`. Commit `#fa5fc18` specifically addresses this issue. The implications of these changes of archetype are further discussed in the next section.

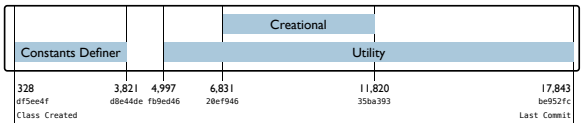

Figure 3: Archetypes evolution of `ToDoList`

5 Discussion

Our archetypes partially match those found in the literature, especially in terms of granularity. While archetypes can classify any kind of class, we initially focus on those that identify classes with a particular interface and structure, implying a specific semantic and behavior. The heuristics defining the *Instance Definer* archetype, for instance, capture Java interfaces and abstract classes, but also regular classes that define reusable interfaces without relying on specific language constructs (*e.g.,* through methods with an empty body instead of abstract methods).

[3] https://github.com/argouml-tigris-org/argouml/

Granularity: Archetypes capture the high-level structural aspects of a class. More fine-grained distinctions are captured by the metrics we consider in the heuristics, which provide an additional perspective to that of archetype classification. For example, a Data Holder may be considered small or large depending on specific quantitative measures, such as the number of lines of code or the number of defined fields. A Data Holder that defines internal classes for its types might appear larger in terms of lines of code, even though it declares fewer attributes than another class. These considerations apply similarly across all archetypes.

Variations: Our heuristics are malleable. For example, the heuristics defining the *Singleton* archetype can be relaxed by removing the constraint on the *getter* method, thereby allowing the "instance" attribute to be `public` and directly accessible by other classes. Such a relaxation enables the identification of non-standard implementations of the Singleton design pattern. Similarly, for the *Utility* archetype, instead of requiring constants to have the `final` modifier, the heuristic can verify whether a field is written by any method. These variations allow the detection of atypical implementations of the archetype and can be fine-tuned for specific needs. For instance, we restrict the methods in the *Wrapper* archetype to have a simple implementation, whereas some developers, in practice, may allow a wrapper to include more complex adaptation logic.

6 Limitations and Future Work

The primary limitation of our current work is its coverage, influenced by the analyzed repositories and the evolving set of archetypes. In this initial analysis, we defined the detection rules to minimize false positives, at the cost of reduced coverage. The first author manually inspected a random sample of 400 classes and did not find any false positives. We plan to extend our study to a larger and more diverse dataset and compare their metrics.

Iterating on interesting patterns emerging from these metrics, we aim to improve the coverage with new archetypes at similar granularity levels. We will further explore hybrid archetypes and the degree to which a class implementation belongs to one or more of them. The coverage of our approach is also limited by the absence of a "fallback" archetype, as *Entity* in the work of Dragan *et al.* [10]. Given the partial nature of our catalog, we left unclassified instances for further refinement in the future, towards a more comprehensive list of archetypes. To accomplish this we also need to extend and improve our heuristics.

7 Conclusion

We presented the concept of class archetypes. We automatically identify archetypes with heuristics in CodeQL and show how they communicate the role of a class in a repository. Roles from archetypes help developers in forming their mental models during maintenance and evolution, reducing the needs for implementation details. Our work extends the previous literature on stereotypes with a different granularity level and complete automation of detection strategies at scale. Moreover, providing automated high-level class semantic is increasingly important given the increase in code written by LLMs. The co-occurrence analysis we performed only scratches the surface of a complex phenomenon of identity and evolution towards a better understanding of the role of classes.

Acknowledgments

This work is supported by the Swiss National Science Foundation (SNSF) through the project "FORCE" (SNF Project No. 232141).

References

[1] Ali F. Al-Ramadan, Joshua A. C. Behler, Michael J. Decker, Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. 2024. Stereocode: A Tool for Automatic Identification of Method and Class Stereotypes for Software Systems. In *Proceedings of ICSME 2024 (International Conference on Software Maintenance and Evolution)*. IEEE, 898–902. https://doi.org/10.1109/ICSME58944.2024.00099

[2] Olena Andriyevska, Natalia Dragan, Bonita Simoes, and Jonathan I. Maletic. 2005. Evaluating UML Class Diagram Layout Based on Architectural Importance. In *Proceedings of VISSOFT 2003 (International Workshop on Visualizing Software for Understanding and Analysis)*. IEEE, 1–6. https://doi.org/10.1109/VISSOF.2005.1684296

[3] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. 2004. *Object-Oriented Analysis and Design with Applications* (3rd ed.). Addison–Wesley.

[4] Elliot J. Chikofsky and James H. Cross. 1990. Reverse Engineering and Design Recovery: A taxonomy. *IEEE Software* 7, 1 (1990), 13–17. https://doi.org/10.1109/52.43044

[5] Luis F. Cortés-Coy, Mario Linares-Vásquez, Jairo Aponte, and Denys Poshyvanyk. 2014. On Automatically Generating Commit Messages via Summarization of Source Code Changes. In *Proceedings of SCAM 2014 (International Working Conference on Source Code Analysis and Manipulation)*. IEEE, 275–284. https://doi.org/10.1109/SCAM.2014.14

[6] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *Proceedings of MSR 2021 (International Conference on Mining Software Repositories)*. IEEE, 560–564. https://doi.org/10.1109/MSR52588.2021.00074

[7] Michael J. Decker, Christian D. Newman, Natalia Dragan, Michael L. Collard, Jonathan I. Maletic, and Nicholas A. Kraft. 2018. Which Method-Stereotype Changes are Indicators of Code Smells?. In *Proceedings of SCAM 2018 (International Working Conference on Source Code Analysis and Manipulation)*. IEEE, 82–91. https://doi.org/10.1109/SCAM.2018.00017

[8] Natalia Dragan. 2011. Emergent Laws of Method and Class Stereotypes in Object Oriented Software. In *Proceedings of ICSME 2011 (International Conference on Software Maintenance and Evolution)*. IEEE, 550–555. https://doi.org/10.1109/ICSM.2011.6080829

[9] Natalia Dragan, Michael L. Collard, Maen Hammad, and Jonathan I. Maletic. 2011. Using Stereotypes to Help Characterize Commits. In *Proceedings of ICSME 2011 (International Conference on Software Maintenance and Evolution)*. IEEE, 520–523. https://doi.org/10.1109/ICSM.2011.6080822

[10] Natalia Dragan, Michael L. Collard, and Jonathan I. Maletic. 2010. Automatic Identification of Class Stereotypes. In *Proceedings of ICSM 2010 (International Conference on Software Maintenance)*. IEEE, 1–10. https://doi.org/10.1109/ICSM.2010.5609703

[11] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S. Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan I. Maletic. 2021. gazel: Supporting Source Code Edits in Eye-Tracking Studies. In *Proceedings of ICSE 2021 (International Conference on Software Engineering)*. IEEE, 69–72. https://doi.org/10.1109/ICSE-Companion52605.2021.00038

[12] Mattia Giannaccari, Marco Raglianti, and Michele Lanza. 2025. Code Refactoring in Virtual Reality. In *Proceedings of IDE 2025 (Workshop on Integrated Development Environments)*. IEEE, 7–12. https://doi.org/10.1109/IDE66625.2025.00006

[13] Mattia Giannaccari, Marco Raglianti, and Michele Lanza. 2025. Skylines: Visualizing Object-Oriented Software Systems Through Class Contours. In *Proceedings of VISSOFT 2025 (Working Conference on Software Visualization)*. IEEE, 64–68. https://doi.org/10.1109/VISSOFT67405.2025.00015

[14] John Hunt. 1997. *Smalltalk and Object Orientation: An Introduction* (1st ed.). Springer.

[15] Ivar Jacobson, Grady Booch, and James Rumbaugh. 1999. *The Unified Software Development Process* (1st ed.). Addison–Wesley.

[16] Ahmad Jbara and Dror G. Feitelson. 2017. How Programmers Read Regular Code: A Controlled Experiment Using Eye Tracking. *Empirical Software Engineering* 22, 3 (2017), 1440–1477. https://doi.org/10.1007/s10664-016-9477-x

[17] Michele Lanza and Stéphane Ducasse. 2001. A Categorization of Classes Based on the Visualization of Their Internal Structure: The Class Blueprint. In *Proceedings of OOPSLA 2001 (Conference on Object-Oriented Programming, Systems, Languages, and Applications)*. ACM, 300–311. https://doi.org/10.1145/504282.504304

[18] Laura Moreno and Andrian Marcus. 2012. JStereoCode: Automatically Identifying Method and Class Stereotypes in Java Code. In *Proceedings of ASE 2012 (International Conference on Automated Software Engineering)*. ACM, 358—-361. https://doi.org/10.1145/2351676.2351747

[19] Laura Moreno, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. JSummarizer: An Automatic Generator of Natural Language Summaries for Java Classes. In *Proceedings of ICPC 2013 (International Conference on Program Comprehension)*. IEEE, 230–232. https://doi.org/10.1109/ICPC.2013.6613855

[20] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *Proceedings of FSE 1995 (Symposium on Foundations of Software Engineering)*. ACM, 18–28. https://doi.org/10.1145/222124.222136

[21] Christian D. Newman, Reem S. AlSuhaibani, Michael L. Collard, and Jonathan I. Maletic. 2017. Lexical Categories for Source Code Identifiers. In *Proceedings of SANER 2017 (International Conference on Software Analysis, Evolution and Reengineering)*. IEEE, 228–239. https://doi.org/10.1109/SANER.2017.7884624

[22] Arif Nurwidyantoro, Truong Ho-Quang, and Michel R. V. Chaudron. 2019. Automated Classification of Class Role-Stereotypes via Machine Learning. In *Proceedings of EASE 2019 (International Conference on Evaluation and Assessment in Software Engineering)*. ACM, 79—-88. https://doi.org/10.1145/3319008.3319016

[23] Clarke J. Peter, Babich Djuradj, King M. Tariq, and Kibria B. M. Golam. 2008. Analyzing Clusters of Class Characteristics in OO Applications. *Journal of Systems and Software* 81, 12 (2008), 2269–2286. https://doi.org/10.1016/j.jss.2008.03.056

[24] Jinfeng Shen, Xiaobing Sun, Bin Li, Hui Yang, and Jiajun Hu. 2016. On Automatic Summarization of What and Why Information in Source Code Changes. In *Proceedings of COMPSAC 2016 (Annual Computer Software and Applications Conference)*. IEEE, 103–112. https://doi.org/10.1109/COMPSAC.2016.162

[25] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of FSE 2006 (Symposium on Foundations of Software Engineering)*. ACM, 23–34. https://doi.org/10.1145/1181775.1181779

[26] Margaret-Anne Storey. 2005. Theories, Methods and Tools in Program Comprehension: Past, Present and Future. In *Proceedings of IWPC 2005 (International Workshop on Program Comprehension)*. IEEE, 181–191. https://doi.org/10.1109/WPC.2005.38

[27] Ho-Quang Truong, Nurwidyantoro Arif, Adi R. Satrio, Chaudron R.V. Michel, Fröding Fabian, and Nguyen N. Duy. 2022. Role Stereotypes in Software Designs and Their Evolution. *Journal of Systems and Software* 189, Article 111296 (2022), 24 pages. Issue C. https://doi.org/10.1016/j.jss.2022.111296

[28] R.J. Wirfs-Brock. 2006. Characterizing Classes. *IEEE Software* 23, 2 (2006), 9–11. https://doi.org/10.1109/MS.2006.43

[29] Rebecca Wirfs-Brock and Alan McKean. 2002. *Object Design: Roles, Responsibilities, and Collaborations* (1st ed.). Addison–Wesley.