Explaining the Collaboration between Software Components through Semantic Classification

Momchil Milushev[1], Satrio Adi Rukmono[2], Arif Nurwidyantoro[3], Filip Zamfirov[1] and Michel R.V. Chaudron[1]

[1]*Eindhoven University of Technology, De Zaale, Eindhoven, The Netherlands*
[2]*Institut Teknologi Bandung, Ganesa 10, Bandung, Jawa Barat, Indonesia*
[3]*Universitas Gadjah Mada, Bulaksumur, Kab. Sleman, Yogyakarta, Indonesia*

Abstract

Understanding collaborations between classes is important for understanding a software system and evaluating its design quality and integrity. These collaborations are realised as method calls in the implementation. However, a call-based dependency view, such as the call graph, does not show the meaning or intent behind the call, e.g., whether it initiates a process or responds to a request. While prior work recovers design intent for entities such as classes and methods (the 'boxes' in a diagram), in this paper, we seek to classify the 'arrows'. We propose a taxonomy of semantic classifications of method call sites: the behavioural stereotype of a specific interaction in its caller context. We contribute a corpus of manually labelled call sites from three Java projects and an automated method for inferring their semantic classifications. We created a machine learning component trained on a manually labelled ground truth. We experimented with interpretable rule-learning classifiers (JRip, PART, J48). The classifiers achieved over 90% accuracy, well above the 70.6% name-only (syntactic) classifier baseline. To help understand the results, we created a visual representation that can show both individual method calls and the combined characteristics of all methods between two classes. In moving from the structural stereotypes of entities to the behavioural stereotypes of interactions, this work allows for more expressive architectural diagnostics of class interactions. In summary, we contribute a taxonomy, a labelled corpus, an interpretable classifier, and a visualisation to bring semantic meaning to the 'arrows' in a call graph.

Keywords

static analysis, software comprehension, method call sites, behavioural stereotypes, software visualization, interpretable machine learning, software design recovery, semantic classification

1. Introduction

Collaboration between software classes is a key mechanism by which object-oriented systems realise their behaviour, implemented through method calls. Dependency views such as call graphs show *that* classes interact [1], but not *why*: two structurally identical edges may represent very different design roles, such as data retrieval, state mutation, input validation, or workflow coordination. Which role a call plays is determined not only by the callee's responsibility but also by the calling context, i.e., the code of and around the method call.

Making these call roles explicit enables a higher-level interpretation of class/component interactions. Consider dependencies from a user-interface class to a domain class. A portion of the calls retrieves data for display, but another portion changes the domain object's state directly. These two types of dependencies, while structurally looking the same, represent differing levels of coupling between the two classes [2].

Prior work on recovering design intent typically concerns *entities*: class and method stereotypes are structural stereotypes, assigned to the 'boxes' in a diagram. In this paper, we instead label the 'arrows', the behavioural stereotype of an individual locus of collaboration. The unit of analysis in this study is thus the *call site*, a specific method invocation in its caller context, rather than the method declaration.

BENEVOL'26: ...
m.i.milushev@student.tue.nl (M. Milushev); sar@itb.ac.id (S. A. Rukmono); arifn@ugm.ac.id (A. Nurwidyantoro); f.zamfirov@tue.nl (F. Zamfirov); m.r.v.chaudron@tue.nl (M. R.V. Chaudron)
0000-0001-9480-7216 (S. A. Rukmono); 0000-0001-8683-3078 (A. Nurwidyantoro); 0009-0009-3654-4935 (F. Zamfirov); 0000-0001-7517-6666 (M. R.V. Chaudron)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

To our knowledge, this is the first work to define and automatically classify design-level behavioural stereotypes at the granularity of individual method call sites.

To support developers inspecting a repository, we use interpretable rule learners whose predictions are traceable to static evidence; Section 5.3 motivates this choice.

The objective of this study is to evaluate the feasibility of automatically classifying method call sites against a working taxonomy of semantic roles grounded in the design literature. The taxonomy is a means rather than the object of validation; establishing its conceptual validity is left for future work. We refine this objective into three research questions:

RQ-1: To what extent can interpretable rule learners infer the semantic role of a method call site from statically extracted features, and how much do they improve over a name-only syntactic baseline?

RQ-2: To what extent do the learnt classification rules generalise to projects unseen during training?

RQ-3: Which learner offers the best balance between classification accuracy and model interpretability for use in an explainable visualisation?

To answer these questions, we train interpretable rule learners on statically extracted features from manually labelled Java call sites, evaluate them using cross-validation and cross-project validation, and visualise their predictions with proof traces.

In addition to introducing a new problem formulation that moves stereotype classification from declarations to interactions, this paper contributes: (1) a design-grounded taxonomy of call site roles; (2) a corpus of 1,000 manually labelled call sites from three open-source Java projects; (3) an interpretable, rule-learnt classifier with proof traces for every prediction; and (4) a visualisation that integrates these labels into a class-dependency view.

This feasibility study is a first step towards *interaction-aware software analysis*, in which role-typed dependencies support design-level diagnostics; Section 8.2 outlines this agenda.

2. Related Work

Software comprehension research has a long objective of capturing higher-level behavioural abstractions of source code. For example, Dragan et al. [3, 4] developed a rule-based tool called Stereocode to identify method and class stereotypes in C++ programs. This work was later extended as JStereocode by Moreno et al. for Java systems [5]. This line of work was recently consolidated in the Stereocode tool, which supports multiple languages and aggregates at the declaration level [6]. Another study developed a machine-learning-based tool to classify stereotypes of Java code [7], following Wirfs-Brock's class characterisation [8] of *information holder*, *structurer*, *service provider*, *controller*, *coordinator*, and *interfacer*. This stereotype classification has subsequently been used to study software architecture evolution [9]. As black-box classifiers, however, such approaches lack deterministic, inspectable decision logic, which motivates our use of interpretable rule learning.

Software visualisation serves as an effective means of presenting behavioural abstractions to support practitioners in comprehending large software systems. Building on role-stereotype abstractions [7], Ho et al. developed the RoleViz visualisation tool to promote the understanding of architectural roles within a software system [10]. Recently, Rukmono et al. introduced ClassViz [11] as a graph-based visualisation tool that represents source code as knowledge graphs [12], providing richer structural and knowledge support for software architecture analysis.

Despite these advances, existing studies on abstraction and visualisation mainly model individual software components (e.g., classes and methods), assigning each entity a single declaration-level stereotype, regardless of its use at specific call sites. We argue that understanding software behaviour also requires analysing how these entities interact, for example, when one class retrieves information from another, coordinates its behaviour, or commands it to act. The semantics of such relationships, as expressed by method invocations, remain largely unexplored. To address this, we propose a taxonomy of class interactions based on method call sites, with classification rules learnt from labelled data rather than hand-written, and we extend ClassViz to visualise these interactions within software systems.

3. Working Taxonomy

The taxonomy captures how a method is used in its caller context, including the enclosing statement and result usage. A repository lookup, for instance, is a *retrieval* when its result is returned to the requester, but may serve as a *delegation* step within a request-handling workflow. The same callee can therefore receive different roles at different call sites.

We stress that this is a *working* taxonomy: it suffices for studying the feasibility of automated classification, but its conceptual validity (including the completeness of the roles, their orthogonality, and their alignment with practitioners' understanding of call-site behaviour) has not been independently established and is the subject of future work.

1. **Query/Retrieve** obtains information without observable side effects. Grounded in Meyer's *Command–Query Separation* principle [13], reinforced in Fowler's *Separate Query from Modifier* refactoring principle [14] and Martin's view on side effects [15]. Example: `owner.getPets()`.
2. **Command/Mutate** changes the state of an object, a database, a user-interface element, or another component. The command side of CQS [13, 14, 15]; typically returns no meaningful data. Example: `owner.setLastName(name)`.
3. **Control/Orchestrate** coordinates work, delegates a step, or connects components in a workflow rather than performing the main computation itself. Grounded in the GRASP Controller responsibility pattern [16]. Example: a controller invoking `ownerRepository.findById(id)` as one step of request handling—the callee is query-like, but the dominant role at this site is orchestration.
4. **Compute/Transform** produces a derived value: parsing input, mapping one representation to another, or performing a local calculation. Grounded in functional cohesion from structured design [2]. Example: `Integer.parseInt(value)`.
5. **Validate/Guard** checks a condition, validates input, or guards a control-flow path. Often returns Boolean values and appears in conditionals, assertions, null checks, or guard clauses. Grounded in Design by Contract [13] and guard-clause practice [15]. Example: `bindingResult.hasErrors()` deciding whether request handling continues.
6. **Notify/Publish** signals an event, publishes a message, or dispatches information to listeners, corresponding to communication and event propagation rather than data retrieval or mutation. Grounded in the Observer pattern and publish–subscribe interaction style [17]. Example: `eventPublisher.publishEvent(event)`.
7. **Creational** creates, constructs, builds, or obtains a new object, including via factory methods, builders, and static construction helpers. Grounded in the creational design patterns [17]. Example: `BoardFactory.createBoard()`.
8. **Unknown/Mixed**: the fallback label when no single role is defensible. Forcing ambiguous cases into a main label would introduce noise into the corpus; the fallback keeps them from polluting the labelled data.

We deliberately chose to use broad labels, as finer-grained ones introduce more annotation ambiguity and require more data. Furthermore, annotation is single-label: when a call site signifies several interpretations, the dominant role in its local context is assigned, and Unknown/Mixed is reserved for cases where no such dominance is defensible.

4. Study Design

Following Stol and Fitzgerald's guidelines [18], we conducted a knowledge-seeking laboratory experiment. This experiment is descriptive and is meant to characterise the performance of our trained classifier against the working taxonomy.

Table 1
Study subjects. Extracted and Annotated columns show the number of call sites.

Project (GitHub)	Files analysed	LOC	Extracted	Annotated	Caller classes
thunderbird/thunderbird-android	206	27,691	9,134	351	117
spring-projects/spring-petclinic	47	2,089	1,170	325	30
SERG-Delft/jpacman	59	2,699	801	324	47
Total	**312**	**32,479**	**11,105**	**1,000**	**194**

The unit of analysis in this experiment is a directed call-site instance: a specific method invocation in a specific caller context. We examine the code surrounding each such invocation and measure how well different machine-learning methods predict its semantic role from that context.

In selecting the repositories for training and testing, we employed purposive sampling [19], considering domain variety, size range, and source-code availability. We therefore weighted availability over randomness, which means the sample is not representative of any particular category of software repository.

Within each repository, we sampled call-site instances while manually labelling them with semantic roles. We stratified the sample to keep label counts roughly equal, with two exceptions: Notify/Publish, whose feature shape is so consistent that adding more examples would not help a rule-based learner, was capped rather than matched to other labels; Unknown/Mixed remained under-represented because it consists of outliers.

To validate the manual annotation, we selected 10% of the labelled dataset (100 call sites), stratified across both projects and labels, to be annotated by a second annotator. The second annotator is an expert software engineer with more than 30 years of industry experience and did not see the first annotator's label before completing their own.

The treatments in this experiment are three rule-learning algorithms: JRip [20] (Weka's implementation of RIPPER [21]), PART [22], and J48 (an implementation of C4.5 [23]). All three produce interpretable models: ordered rule lists for JRip and PART, and a decision tree for J48. We motivate the choice of rule learning in Section 5.3.

For performance comparison, we include a name-only baseline: a hand-written mapping that predicts a label solely from the CalleeNameAction feature, which summarises the naming signal of the invoked method. This baseline involves no learning and lets us judge whether the richer feature set contributes information beyond what the callee's name (a syntactic property) already reveals.

Three Java projects were used in the experiment: Spring PetClinic, a Spring-based web application; JPacman, a game with self-contained domain logic; and Thunderbird for Android, a large Android application. Table 1 summarises these projects and the distribution of the 1,000 labelled call sites. Sampling and labelling yielded 161–162 instances per major label, except Notify/Publish (25 instances) and Unknown/Mixed (7 instances).

We evaluate the learnt models in two setups: (1) 10-fold stratified cross-validation on the 1,000 call sites to estimate overall performance, and (2) leave-one-project-out (LOPO) validation for JRip and PART, where models are trained on two projects and tested on the third to assess generalisation to unseen projects. The name-only baseline has no learnt parameters and is project-independent, so neither setup applies; we score it once on all 1,000 labelled call sites.

We report accuracy, per-class precision, recall, and F-1, as well as weighted F-1 across all classes and Cohen's kappa [24], which corrects for the agreement expected by chance.

5. Pipeline

The pipeline of this study includes feature extraction, annotation, training & evaluation, and visualisation. This pipeline is depicted in Figure 1 and elaborated below.

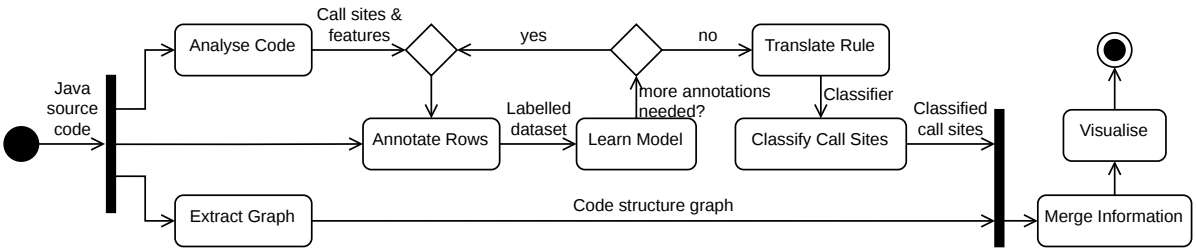


Figure 1: Study pipeline.

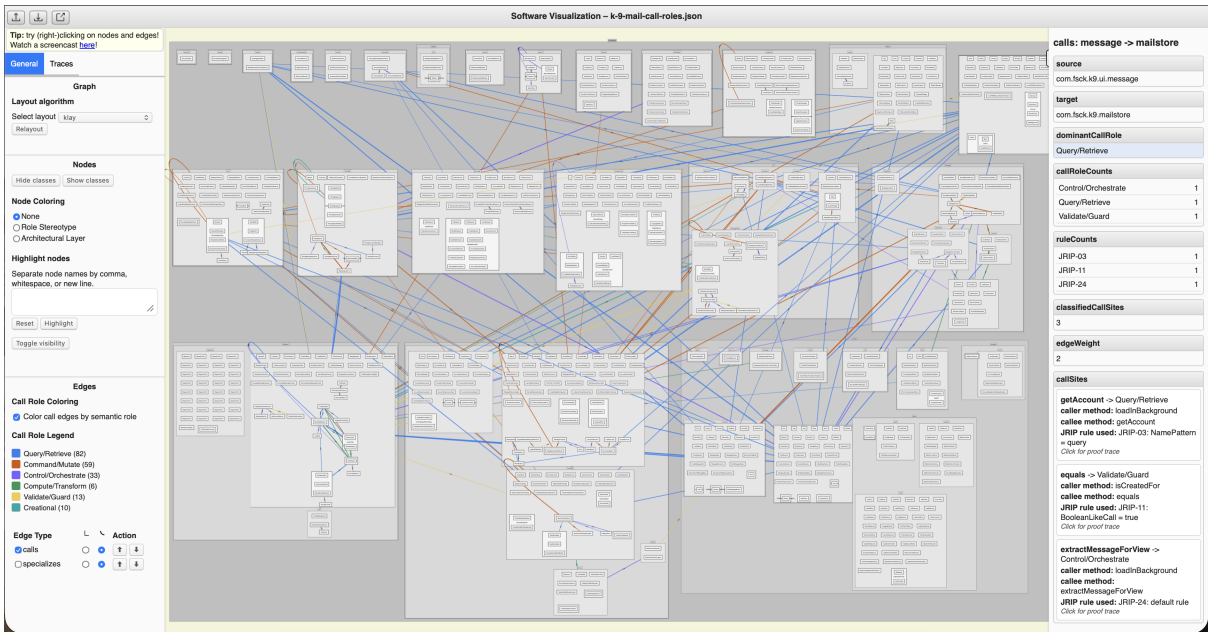


Figure 2: Classification results for Thunderbird visualised in an extended version of ClassViz.

5.1. Feature Extraction

A Spoon-based extractor analyses the Java abstract syntax tree without executing the program, producing one feature row per method invocation.

Each call site has 85 features in seven categories: (1) *Method and return features* describe the callee's name and return type, including MethodName and IsVoidCall. (2) *Result-usage features* capture how the caller uses or ignores the result, summarised in ResultUsageKind. (3) *Control-flow and guard features* capture statement context, including conditional usage and preceding guard clauses. (4) *Receiver and argument features* describe receiver category and origin, argument counts, and argument characteristics such as ArgumentTypePattern. (5) *Architectural and graph features* capture layers inferred from annotations and package conventions, cross-layer calls, and caller fan-out and callee fan-in. (6) *Name-stereotype features* capture naming conventions through action prefixes and Boolean-like patterns, including NamePattern and BooleanLikeCall. (7) *Compact semantic summaries* combine lower-level observations into features such as CalleeNameAction and CalleeBodySummary.

Not all features have the same reliability. Syntactic observations, such as whether a result is assigned or returned, are strong static evidence. Naming-based, layer-based, and dependency-based features are heuristics that depend on project conventions. Body-summary features are factual only when the callee source code resolves; otherwise, they are marked as UNRESOLVED.

5.2. Annotation

The annotation is source-backed: for each selected call site, the annotator inspects the caller method, the invoked method, the enclosing statement, and how the result is used in the source code, with the extracted feature row serving as structured evidence. Each annotated call site records the label and a brief rationale indicating which feature evidence supports it.

The dataset was grown from an initial 500 call sites to the final 1,000 after the 500-site pilot showed weak recall for Compute/Transform, indicating that the pilot did not contain enough examples to separate this label from its semantically adjacent categories.

On the 100-call-site agreement subset (Section 4), the two annotators reached 79% raw agreement with $\kappa = 0.758$, which indicates substantial agreement. The disagreements were concentrated on adjacent role boundaries, in particular, Query/Retrieve versus Validate/Guard, Control/Orchestrate versus Command/Mutate, and Compute/Transform versus Creational. This mirrors the confusion pattern later observed in the learnt models.

5.3. Training and Evaluation

We chose rule learning over the two alternatives: manually weighted rules are hard to justify because the weights mostly reflect the designer's assumptions rather than evidence from labelled data. Black-box models, on the other hand, are unexplainable in a visualisation setting where the user should be able to inspect why a call site received a label. Learnt if–then rules are a compromise: they are explicit and readable, and each rule carries measurable support and error counts from the training data.

The three treatments (Section 4) were trained and evaluated in Weka. Default parameter values were used for all treatments.

The learnt JRip rule list ends with a default rule that predicts Control/Orchestrate. The default rule predicts the class that best covers the training examples left unmatched by the earlier, more specific rules; in our training data, these remaining call sites were most often orchestration-like.

The learnt rules were translated into an executable classifier script that applies the rules in the same order as the trained model. For each prediction, the script outputs the rule identifier, the rule condition, the training support and error counts, an estimated rule precision, and a proof trace that links the rule condition back to the extractor logic that produced the matched feature.

5.4. Visualisation

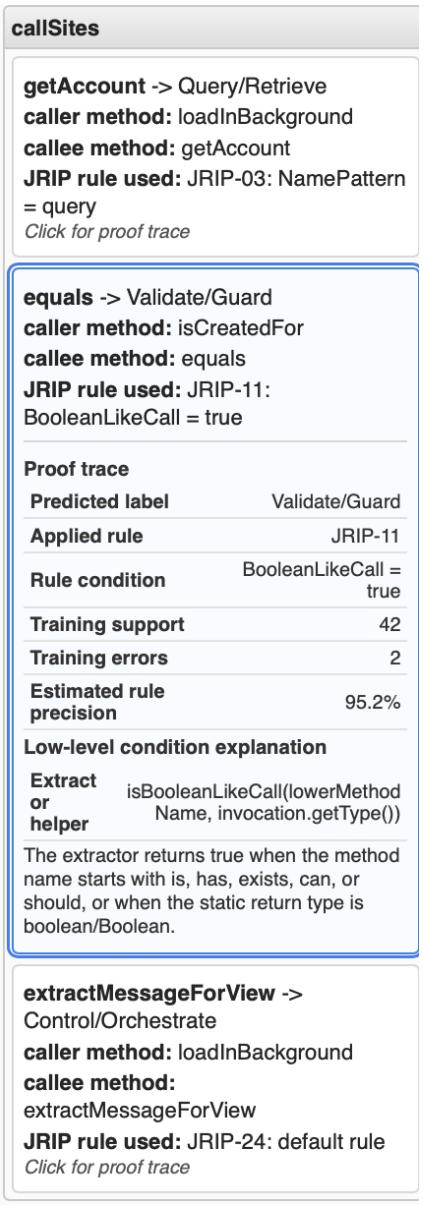
We extend ClassViz [11] by enriching the structural graph generated by Javapers [12] with call-site predictions. Each class-to-class edge aggregates multiple call sites and is coloured by its dominant semantic role (Figure 2). This colour is an overview signal, not a complete description of a potentially mixed-role dependency.

Selecting an edge reveals its role and rule distributions and individual call sites with inspectable proof traces linking predictions to extracted evidence. For example, the highlighted Validate/Guard call site in Figure 3a is identified as Boolean-like because its method name starts with is and its static return type is Boolean.

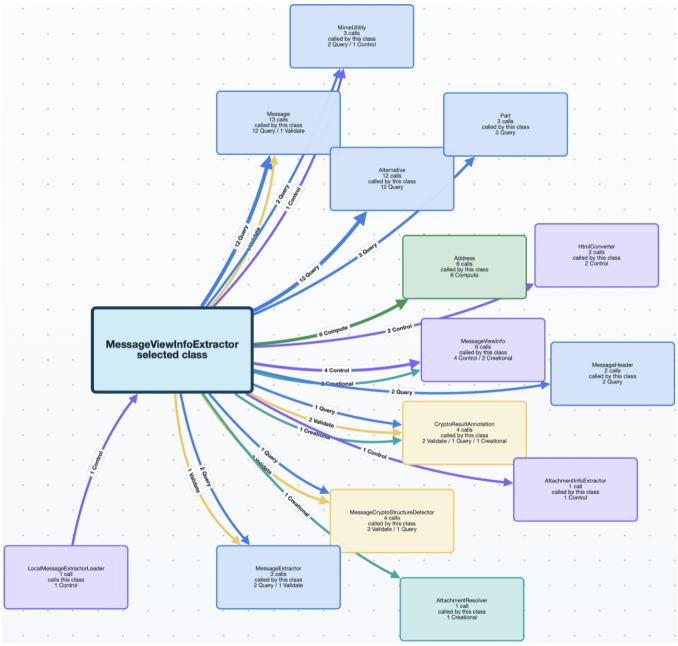
A class-focused mode shows a selected class's direct semantic neighbourhood, with call roles and counts encoded on edges (Figure 3b), and summarises its call-site role distribution in a pie chart (Figure 3c).

6. Results

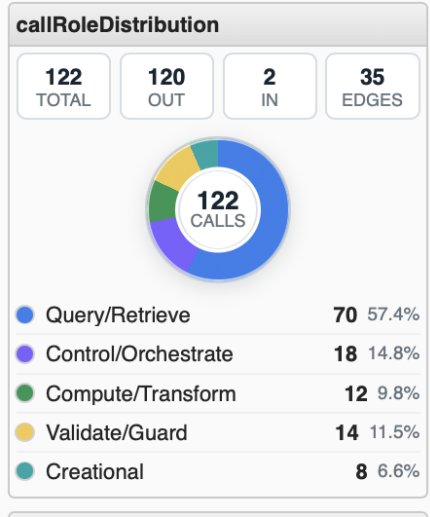
Table 2 compares model complexity and performance under 10-fold stratified cross-validation against the name-only baseline. All three learners outperform the baseline, with J48's small accuracy advantage coming at the cost of a much larger model.



(a) A portion of the sidebar showing the details of some call site classifications, one with the proof trace expanded (highlighted in blue).



(b) Class-focused visualisation mode centred on a selected class and its direct semantic neighbourhood.



(c) Side panel showing the call-site role profile of the selected class as a pie chart.

Figure 3: ClassViz views of call-site classifications, the selected class's semantic neighbourhood, and its call-site role profile.

Table 3 details the classification performance per role category under 10-fold cross-validation. The name-only baseline overpredicts Control/Orchestrate through its fallback and misses most Compute/Transform calls. JRip maintains high precision and recall across all classes. The few misclassifications by the rule learners are primarily concentrated among semantically adjacent labels. For example, in JRip, we observe minor confusion from *Query/Retrieve* to *Control/Orchestrate* (6 instances), *Compute/Transform* to *Query/Retrieve* (7 instances), and *Validate/Guard* to *Control/Orchestrate* (11 instances). Figure 4 shows the confusion matrix produced from the JRip 10-fold cross validation run.

Answer to RQ-1 Interpretable rule learners infer call-site roles with 90.9–92.4% accuracy under 10-fold cross-validation, approximately 20 percentage points above the 70.6% name-only baseline. The context

Table 2
Model complexity and overall classification performance.

Model	Size	Accuracy	Weighted F_1	Kappa
Baseline	12 mappings	70.6%	0.702	—
JRip	29 rules	90.9%	0.909	0.892
PART	43 rules	91.0%	0.910	0.893
J48	576 nodes(546 leaves)	92.4%	0.924	0.910

Table 3
Detailed classification performance (P=Precision, R=Recall, F-1 score) under 10-fold cross-validation.

Class	Name-Only Baseline			JRip			PART			J48		
	P	R	F_1	P	R	F_1	P	R	F_1	P	R	F_1
Control/Orchestrate (CO)	0.37	0.86	0.52	0.84	0.91	0.88	0.85	0.96	0.90	0.89	0.99	0.94
Command/Mutate (CM)	0.84	0.60	0.70	0.91	0.90	0.91	0.92	0.88	0.90	0.90	0.96	0.91
Creational (CR)	0.95	1.00	0.97	0.92	0.94	0.93	1.00	0.98	0.99	1.00	0.98	0.99
Compute/Transform (CT)	1.00	0.19	0.32	0.92	0.91	0.92	0.86	0.91	0.89	0.87	0.94	0.91
Query/Retrieve (QR)	0.86	0.86	0.86	0.90	0.90	0.90	0.88	0.83	0.86	0.92	0.83	0.87
Validate/Guard (VG)	0.98	0.67	0.79	0.95	0.89	0.92	0.97	0.91	0.94	0.95	0.89	0.92
Notify/Publish (NP)	0.83	0.96	0.89	1.00	0.92	0.96	0.91	0.88	0.90	1.00	0.88	0.94
Unknown/Mixed (UM)	1.00	1.00	1.00	1.00	1.00	1.00	0.67	0.57	0.62	1.00	0.71	0.83

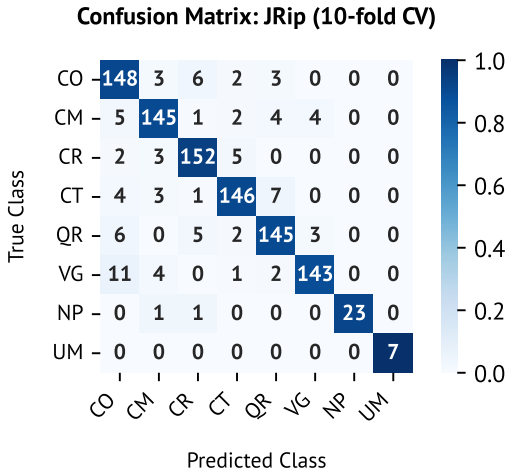


Figure 4: Confusion matrix of the JRip model under 10-fold stratified cross-validation on the 1,000 labelled call sites.

features (result usage, return type, receiver, control flow) carry substantial signal beyond the callee's name, most visibly for Compute/Transform, where the baseline's recall of 0.19 rises above 0.90.

J48's large tree heavily memorises individual MethodName values, making it unsuitable for our concise, inspectable explanations. We therefore exclude it from the subsequent leave-one-project-out (LOPO) evaluation.

Under LOPO evaluation (Table 4), JRip outperforms PART on average, although PART performs better on PetClinic.

Table 5 reports per-class LOPO performance, with Figure 5 showing the confusion matrices. The cross-project accuracy drop reflects non-independent sampling under random cross-validation: call sites from the same project appear in both training and test sets, allowing rules to exploit project-specific naming and structure. Transfer is hardest to PetClinic, whose Spring idioms are absent from the other two projects, and easiest to JPacman, a small codebase with conventional Java naming practices. Notify/Publish never transfers across projects; recall is 0.000 in every LOPO fold for both learners.

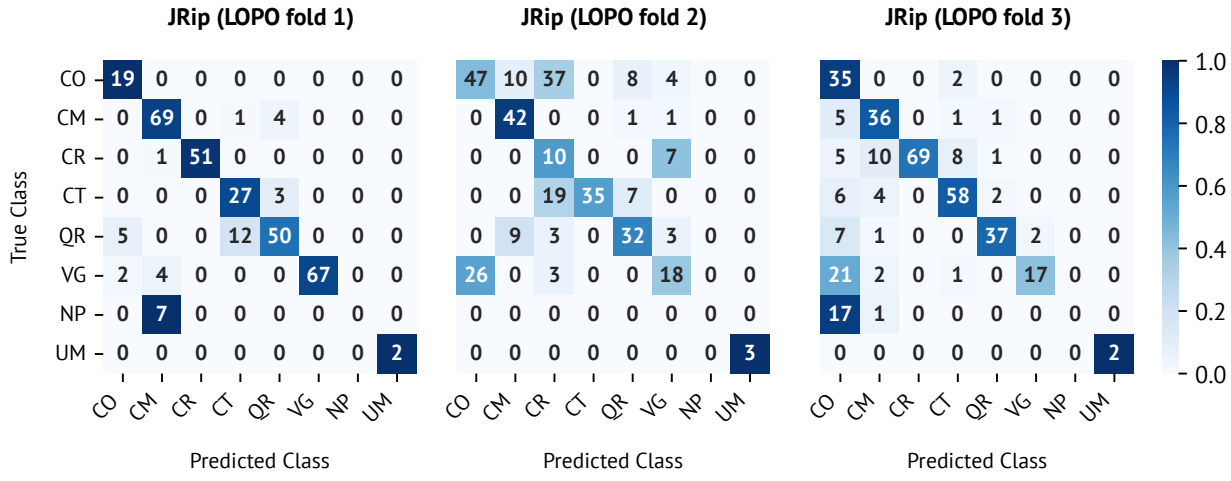
Table 4
Cross-project generalisation (LOPO) overall performance.

Held-out Project (Test)	n	JRip		PART	
		Acc.	Kappa	Acc.	Kappa
JPacman	324	88.0%	0.85	76.5%	0.71
PetClinic	325	57.5%	0.49	63.1%	0.55
Thunderbird	351	72.4%	0.67	64.7%	0.58
Mean	—	72.6%	0.67	68.1%	0.61

Table 5
LOPO per-class performance (Precision / Recall).

Held-out project	Model	CO	CM	CR	CT	QR	VG	NP	UM
JPacman	JRip	0.73 / 1.00	0.85 / 0.93	1.00 / 0.98	0.68 / 0.90	0.88 / 0.75	1.00 / 0.92	— / 0.00	1.00 / 1.00
	PART	0.11 / 0.11	0.84 / 0.87	1.00 / 0.98	0.46 / 0.57	0.87 / 0.70	0.76 / 0.92	— / 0.00	— / 0.00
PetClinic	JRip	0.64 / 0.44	0.69 / 0.96	0.14 / 0.59	1.00 / 0.57	0.67 / 0.68	0.55 / 0.38	n/a*	1.00 / 1.00
	PART	0.64 / 0.39	0.70 / 0.96	0.65 / 1.00	0.92 / 0.75	0.40 / 0.92	0.94 / 0.34	n/a*	— / 0.00
Thunderbird	JRip	0.37 / 0.95	0.67 / 0.84	1.00 / 0.74	0.83 / 0.83	0.90 / 0.79	0.90 / 0.42	— / 0.00	1.00 / 1.00
	PART	0.25 / 0.38	0.81 / 0.81	1.00 / 0.82	0.57 / 0.43	0.78 / 0.81	0.59 / 0.83	— / 0.00	0.00 / 0.00

* No Notify/Publish instances in the PetClinic test dataset.
"— indicates precision is undefined (class never predicted).



(a) JPacman held out. (b) PetClinic held out. (c) Thunderbird held out.

Figure 5: Confusion matrices of the JRip model under LOPO validation, trained on two projects and tested on the held-out third.

Because the naming and receiver cues for notification calls are project-specific, rules from two projects do not match the third project's notification idioms. Thus, the strong Notify/Publish scores under 10-fold cross-validation reflect within-project consistency rather than a generalisable signal.

The effect of the default-rule fallback is also visible in the LOPO precision values. When PetClinic is held out, JRip's precision for Creational drops to 0.14, meaning that many PetClinic call sites are incorrectly absorbed by creation-related rules. When Thunderbird is held out, the precision for Control/Orchestrate drops to 0.37, as unmatched call sites fall through to the default rule into Control/Orchestrate. PART additionally loses Unknown/Mixed completely in every fold.

Answer to RQ-2 The learnt rules only partially generalise. Mean accuracy on unseen projects drops to 72.6% for JRip (68.1% for PART), 18–23 percentage points below cross-validation, and transfer varies

strongly with the held-out project (88.0% on JPacman versus 57.5% on PetClinic). Notify/Publish does not transfer at all, showing that its cues are project idioms rather than a generalisable signal.

PART's model size drops from 43 rules on the full dataset to 23–26 in the LOPO folds, whereas JRip's size varies less across evaluation setups.

Answer to RQ-3 JRip offers the best balance between accuracy and interpretability. Its 29 rules reach 90.9% accuracy, within 1.5 percentage points of J48's 546-leaf tree, while remaining compact enough for per-call-site proof traces; it also attains the best cross-project mean and the most stable model size across evaluation setups (29 rules on the full corpus, 23–25 in LOPO folds).

7. Threats to Validity

Construct validity: The taxonomy is a working taxonomy: its completeness, role orthogonality, and alignment with practitioners have not been independently established. The experiment therefore measures classifier performance *relative to the taxonomy*, not the taxonomy's validity, which is left for future work. Many call sites permit multiple plausible interpretations; the assigned label reflects the annotator's judgement of the dominant role in context. The Unknown/Mixed label mitigates but does not eliminate this. The labels are static semantic classifications and ignore runtime behaviour: the extractor does not observe dynamic dispatch, reflection, framework configuration, or dependency-injection resolution. Several features, including layer inference, name–action categories, and dependency detection, are heuristic and may degrade in projects with unconventional naming or package structures.

Internal validity: The annotator used the extracted feature row as structured evidence, and many features are name-derived, so the annotator and learner partly rely on the same surface signals. The name-only baseline bounds the callee-name contribution at 70.6%, but does not remove circularity for richer features. A single primary annotator created the corpus; a second annotator only measured agreement (79%, $\kappa = 0.758$), thus residual label noise of that magnitude propagates into both the training and test data. In 10-fold cross-validation, call sites from the same project appear in both training and test folds; the LOPO gap of about 18–23 percentage points (Table 4) quantifies the resulting inflation.

External validity: The corpus draws on three purposively sampled open-source Java projects, with availability weighted over representativeness (Section 4); other domains, frameworks, and naming conventions may yield different feature patterns. The LOPO results quantify this concretely. The rationale for capping Notify/Publish (i.e., its consistent feature shape) holds within projects, but its LOPO recall of 0.000 in every fold shows that the identifying cues are project-specific idioms. Its cross-validation scores therefore reflect within-project consistency, not a generalisable signal.

Conclusion validity: The two low-support labels warrant separate caveats. Notify/Publish (25 instances) is small by design, so its per-class metrics should be read as within-project evidence; Unknown/Mixed (7 instances) is small because such cases are genuinely scarce, so its per-class metrics are not stable estimates. We did not test the differences between the learners for statistical significance; our model-selection argument rests on interpretability and stability instead. All models used default Weka parameters.

8. Discussion

8.1. Interpretation of the Results

Naming features are strong but insufficient on their own. The roughly 20-percentage-point gap between the learnt rules and the name-only baseline quantifies the added value of contextual evidence: result usage, return type, receiver information, fan-in and fan-out, and control-flow context.

The labels that are easy to learn are those with clear and distinct static signals: Command/Mutate combines call sites with a void return type, an ignored return value, and a mutator-like name; Validate/Guard combines Boolean-like calls with conditional usage; Creational is marked by factory or builder-like prefixes. Conversely, the labels that are hard to learn overlap conceptually: a call whose result is subsequently returned by the calling context may indicate data retrieval (Query/Retrieve), result derivation (Compute/Transform), or a step in a workflow (Control/Orchestrate), so the distinction relies on subtler contextual cues.

The model interpretability–accuracy trade-off resolves in favour of JRip. With our explainability objective, accuracy per-rule is a better measure than accuracy alone, and JRip's 90.9% accuracy per 29 rules beats J48's 92.4% per 546-leaf tree and PART's 91.0% per 43 rules.

The learnt rules are also empirically grounded. Support and error counts replace hand-picked weights, and proof traces make each edge colour inspectable rather than something to be trusted blindly. Cross-project generalisation is the real open problem. Rules learnt within a project partially memorise its conventions, and rare classes need more data before their LOPO numbers mean anything.

8.2. Vision for Interaction-Aware Software Analysis

Call-site classifications turn a dependency graph into a *typed dependency graph*, enabling analyses that distinguish the design roles of interactions rather than treating all dependencies alike. Two applications illustrate this agenda.

First, role-typed edges could support architecture conformance checks [25], such as detecting calls that violate a rule that the view layer must not mutate the domain. They could also provide semantic evidence for smell detection [14, 26], rather than relying solely on structural proxies. Second, coupling metrics [27] could distinguish queries from mutations and orchestration, helping developers assess which dependencies merit inspection.

These applications remain hypotheses, not evaluated capabilities of the present classifier. Their usefulness depends on validating the taxonomy and improving cross-project transfer. The immediate contribution is an inspectable representation that connects design vocabulary to evidence at individual call sites.

9. Conclusion and Future Work

This study demonstrates the feasibility of classifying method call sites against a working taxonomy of semantic roles. Using 1,000 labelled call sites from three Java projects, JRip achieves 90.9% accuracy under 10-fold cross-validation with 29 rules. Integrating these predictions into ClassViz provides per-call-site proof traces, extending semantic classification from declarations to interactions. Mean leave-one-project-out accuracy falls to 72.6%, identifying cross-project transfer as the main empirical limitation.

The next priorities are to validate the taxonomy's coverage and distinctions with practitioners, and to replicate the study across more projects and languages. A larger corpus should strengthen rare-label coverage and use multiple annotators with adjudication. More project-independent features and naming normalisation are needed to improve transfer.

References

[1] M.-A. Storey, Theories, Methods and Tools in Program Comprehension: Past, Present and Future, in: 13th International Workshop on Program Comprehension, 2005, pp. 181–191. doi:10.1109/WPC.2005.38.

[2] E. Yourdon, L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice-Hall, Inc., 1979.

[3] N. Dragan, M. L. Collard, J. I. Maletic, Reverse engineering method stereotypes, in: 2006 22nd IEEE International Conference on Software Maintenance, IEEE, 2006, pp. 24–34.

[4] N. Dragan, M. L. Collard, J. I. Maletic, Automatic identification of class stereotypes, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.
[5] L. Moreno, A. Marcus, Jstereocode: automatically identifying method and class stereotypes in java code, in: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 358–361.
[6] A. F. Al-Ramadan, J. A. Behler, M. J. Decker, N. Dragan, M. L. Collard, J. I. Maletic, Stereocode: A tool for automatic identification of method and class stereotypes for software systems, in: 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2024, pp. 898–902.
[7] A. Nurwidyantoro, T. Ho-Quang, M. R. V. Chaudron, Automated classification of class role-stereotypes via machine learning, in: Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering, 2019, pp. 79–88.
[8] R. J. Wirfs-Brock, Characterizing classes, IEEE software 23 (2006) 9–11.
[9] T. Ho-Quang, A. Nurwidyantoro, S. A. Rukmono, M. R. V. Chaudron, F. Fröding, D. N. Ngoc, Role stereotypes in software designs and their evolution, Journal of Systems and Software 189 (2022) 111296.
[10] T. Ho-Quang, A. Bergel, A. Nurwidyantoro, R. Jolak, M. R. V. Chaudron, Interactive role stereotype-based visualization to comprehend software architecture, in: 2020 Working Conference on Software Visualization (VISSOFT), IEEE, 2020, pp. 122–132.
[11] S. A. Rukmono, ClassViz: From Inspection Tool to Research Vessel, in: The 24th Belgium-Netherlands Software Evolution Workshop (BENEVOL 2025), 2025.
[12] S. A. Rukmono, Javapers: Extract Knowledge Graph from Java Source Code, https://github.com/rsatrioadi/javapers/releases/tag/v2.0, 2026. Version 2.0.
[13] B. Meyer, Object-Oriented Software Construction, volume 2, Prentice hall Englewood Cliffs, 1997.
[14] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional, 1999.
[15] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2008.
[16] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Interative Development, Pearson, 2004.
[17] E. Gamma, Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995.
[18] K.-J. Stol, B. Fitzgerald, The ABC of Software Engineering Research, ACM Transactions on Software Engineering and Methodology (TOSEM) 27 (2018) 1–51. doi:10.1145/3241743.
[19] S. Baltes, P. Ralph, Sampling in Software Engineering Research: A Critical Review and Guidelines, Empirical Software Engineering 27 (2022) 94.
[20] X. Xu, E. Frank, Class JRip, Weka Javadoc, package `weka.classifiers.rules`, ???? URL: https://weka.sourceforge.io/doc.dev/weka/classifiers/rules/JRip.html, implements the RIPPER rule learner (Repeated Incremental Pruning to Produce Error Reduction), based on William W. Cohen's algorithm.
[21] W. W. Cohen, Fast Effective Rule Induction, in: Machine learning proceedings 1995, Elsevier, 1995, pp. 115–123.
[22] E. Frank, I. H. Witten, Generating Accurate Rule Sets Without Global Optimization (1998).
[23] J. R. Quinlan, C4.5: Programs for Machine Learning, Elsevier, 2014.
[24] J. Wood, Understanding and Computing Cohen's Kappa: A Tutorial, WebPsychEmpiricist. Web Journal at http://wpe. info/ (2007).
[25] G. C. Murphy, D. Notkin, K. J. Sullivan, Software reflexion models: Bridging the gap between design and implementation, IEEE Transactions on Software Engineering 27 (2001) 364–380.
[26] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings., IEEE, 2004, pp. 350–359.
[27] S. R. Chidamber, C. F. Kemerer, A Metrics Suite for Object Oriented Design 20 (1994) 476–493. doi:10.1109/32.295895.