## **IV. EXTENDED IDENTIFIER READABILITY MODEL**

In this section, we extend our formal identifier readability model to incorporate two additional dimensions, yielding **six** components in total. Previously, the model evaluated four factors – semantic clarity, stylistic convention adherence, length appropriateness, and natural-language readability. We now introduce **Domain-Based Semantic Relevance (DR)** and **Syntactic Role Conformity (SR)** as new components. These additions integrate domain knowledge into naming and enforce linguistic expectations for different identifier types. We first update the overview and notation to reflect the expanded model, then define each new component with scoring methods, and finally present the revised combined score (with weighting). We also discuss how we calibrated these new factors and the potential applications of the enhanced model.

### **A. Overview and Notation (Updated)**

Let an identifier name $N$ be a string which we tokenize into a sequence of words $W = [w\_1, w\_2, \ldots, w\_k]$ (by splitting on casing boundaries, underscores, etc., as before). The readability model now evaluates **six** sub-scores, each in $[0,1]$:

* **Semantic Clarity (SC):** Measures how well $N$’s tokens convey the intended concept or purpose (meaningfulness of the name).
* **Stylistic Convention Adherence (ST):** Measures compliance with coding style guidelines and naming conventions (formatting, casing, etc.).
* **Length Appropriateness (LN):** Rewards names that are neither too short nor excessively long (optimal length range).
* **Natural-Language Readability (NL):** Measures how easily $N$ can be read as a natural phrase (pronounceability, familiar words, smooth word order).
* **Domain-Based Semantic Relevance (DR):** Measures alignment of $N$’s vocabulary with the application’s domain or ubiquitous language.
* **Syntactic Role Conformity (SR):** Measures whether $N$’s grammatical form (noun, verb, etc.) matches the expected linguistic role for its programmatic entity (e.g. class vs function).

These components are described in detail in the following subsections. Each component yields a score $X(N) \in [0,1]$ (for $X \in {SC, ST, LN, NL, DR, SR}$), with 1 meaning *fully readable* in that dimension. The overall Identifier Readability Score is then computed as a weighted sum of the six factors:

R(N)  =  wSC SC(N)+wST ST(N)+wLN LN(N)+wNL NL(N)+wDR DR(N)+wSR SR(N) ,R(N) \;=\; w\_{SC}\,SC(N) + w\_{ST}\,ST(N) + w\_{LN}\,LN(N) + w\_{NL}\,NL(N) + w\_{DR}\,DR(N) + w\_{SR}\,SR(N)\,,R(N)=wSC​SC(N)+wST​ST(N)+wLN​LN(N)+wNL​NL(N)+wDR​DR(N)+wSR​SR(N),

where $\sum\_{X\in{SC,ST,LN,NL,DR,SR}} w\_X = 1$ and each $w\_X \ge 0$. The weights $w\_X$ determine each factor’s relative influence. (We discuss weight selection and calibration in subsection H.) We next define the two new components (DR and SR), then describe how all six integrate into the model.

### **F. Domain-Based Semantic Relevance (Domain Alignment)**

**Definition:** *Domain-based semantic relevance* (DR) reflects the degree to which an identifier’s terms align with the vocabulary of the application’s domain or *ubiquitous language*. The intuition is that good names should resonate with domain concepts familiar to developers in that context. For example, in a healthcare system, an identifier like patientRecord or diagnosisList is more domain-relevant (and hence more immediately meaningful) than a generic name like dataList or a misnomer like clientRecord if “patient” is the established term for that concept. Leveraging domain terminology in code bridges the gap between code and the problem domain, helping developers trigger the appropriate domain knowledge when reading the name.

**Method:** To quantify $DR(N)$, we incorporate an explicit *domain vocabulary* against which $N$ is compared. Let $D$ be the set of domain-specific terms (nouns, phrases, acronyms) that are relevant and approved for use in the project’s context. This domain lexicon can be constructed by analyzing domain documents, requirements, or a project glossary. (Prior work has advocated maintaining a “project dictionary” of concepts and their standard names, aligning code with the domain model.) We then define $DR(N)$ based on the overlap between $N$’s tokens and the domain lexicon $D$. One simple formulation is:

DR(N)  =  1k∑j=1kdomMatch(wj) ,DR(N) \;=\; \frac{1}{k} \sum\_{j=1}^{k} \mathrm{domMatch}(w\_j)\,,DR(N)=k1​j=1∑k​domMatch(wj​),

where $\mathrm{domMatch}(w\_j) = 1$ if token $w\_j$ (or its lemma or a common synonym) appears in the domain term set $D$, and $0$ otherwise. In other words, $DR(N)$ is the fraction of the identifier’s tokens that are recognized as domain-relevant. We assign partial credit in borderline cases – for instance, if $w\_j$ is a compound or acronym that closely matches a domain term, $\mathrm{domMatch}(w\_j)$ might be 0.5. Similarly, if $w\_j$ is a general word where a more domain-specific term exists (e.g., using “client” instead of the domain-preferred “customer”), we penalize it by giving a lower score. The goal is to reward identifiers that *speak the language of the domain*. An ideal score $DR(N)=1$ means every token in $N$ is drawn from the domain’s approved vocabulary, indicating a name firmly grounded in domain semantics. In contrast, a low $DR$ signifies that the name is either too generic or uses terminology inconsistent with the domain context, which could reduce clarity for developers familiar with the domain.

**Example:** Suppose a financial application has a domain lexicon including {account, balance, trade, portfolio, asset, …}. An identifier computePortfolioRisk would score high in $DR$ since *portfolio* and *risk* are domain terms (assuming *risk* is in the lexicon), whereas computeListStats (in a similar context) would score lower – its tokens are generic and not tied to financial concepts. By enforcing domain alignment, the model incentivizes choosing names that carry domain meaning, which literature suggests can improve comprehension by activating relevant background knowledge. This component complements **Semantic Clarity**: while SC ensures a name is meaningful in general, $DR$ ensures the name uses the *right* terminology for the given context.

### **G. Syntactic Role Conformity (Linguistic Consistency)**

**Definition:** *Syntactic role conformity* (SR) evaluates whether an identifier’s grammatical form matches the expected linguistic role given its programmatic usage. Different kinds of identifiers typically follow implicit part-of-speech conventions: for example, **class** names are usually nouns or noun phrases (they represent entities or concepts), **function/method** names are usually verbs or verb phrases (they represent actions), and **variable** names (especially non-boolean variables) are nouns or noun phrases (they represent objects or values). Violating these conventions can cause confusion or cognitive friction. Our SR component formalizes this by checking the identifier’s tokens against their expected syntactic category.

**Method:** We determine the *category* $c$ of identifier $N$ (e.g., class, function, variable, constant) based on its context in code or naming context (for instance, capitalization can hint at class vs variable in some languages, or we rely on parsed code metadata). We then define a predicate $\mathrm{matchesPattern}(N, c)$ that is true if $N$ conforms to the common naming grammar for category $c$. We use a part-of-speech tagging approach on the token sequence $W$ of $N$, or a simpler dictionary lookup for known verb and noun forms:

* If $c$ is a **function** or method, $W$ should start with a verb (or an imperative phrase). We maintain a list of common verb forms (e.g., *get, set, compute, update, calculate, is, has* etc.). If $w\_1$ (the first token) is a verb or a verb-like word, the pattern is satisfied; otherwise not. For instance, calculateDiscount or isEmpty conforms (starts with *calculate* – a verb, or *is* – a boolean predicate), whereas a function named discountValue would violate this expectation (it reads like a noun).
* If $c$ is a **class** or type, $W$ should form a noun phrase. Typically this means the *last* token (or the whole compound) is a noun that represents the entity. The name should **not** begin with a verb. For example, UserAccount or PurchaseOrder are noun phrases (conforming), whereas a class name ComputeEngine (starting with *compute*, a verb) is non-conforming – it would be clearer as ComputationEngine or similar. We check that no token in $W$ is an obvious verb form; if a verb is detected, that lowers the conformity.
* If $c$ is a **variable** (non-boolean), it should also read as a noun or noun phrase (describing the data). E.g. totalRevenue or userList are noun phrases and conform. A variable name should not be a verb or command. If a variable name starts with a verb (say, computeValue as a variable), it fails SR. (One exception: Boolean flags or predicate variables often start with verbs like *is* or *has*, as in isValid – this is acceptable because such names are read as assertions. Our implementation detects common boolean prefixes and treats them as conforming for variables of boolean type.)
* **Constants** (e.g., MAX\_COUNT) generally use nouns as well (often all-caps nouns). They are handled similar to variables in terms of expected POS (no verbs).

We formalize $SR(N)$ as a binary or graded score based on these checks. In a simple form, we set $SR(N) = 1$ if $\mathrm{matchesPattern}(N,c)$ is true (the identifier conforms to its expected syntactic role), and $SR(N) = 0$ if not. We can also refine this to allow partial credit – for instance, if a name partially meets the expectation (perhaps a class name that has a noun but also includes an unnecessary verb), we might assign an intermediate score like 0.5. However, in most cases the rule is clear-cut. Thus, $SR(N)$ evaluates to 1 when a name uses the appropriate part-of-speech pattern for its kind, and 0 when it violates a naming convention (like a method named with a noun). This component enforces linguistic consistency with common coding guidelines (indeed, the rule “use nouns for classes and verbs for functions” is a well-known convention). By quantifying it, our model can flag names like a function Manager or a class ProcessData as less readable due to inconsistency in expected form.

**Example:** In a code review scenario, if a developer names a function bufferSize (noun phrase) instead of, say, computeBufferSize or getBufferSize, the SR score would be low because a function name lacking a verb is likely to confuse readers about its action. Conversely, a class named UploadManager (noun phrase) or a boolean variable hasStarted (verb prefix denoting a boolean condition) would score high on SR. This dimension, therefore, complements the stylistic convention check: ST covers format and casing, whereas SR covers the *grammatical role* of the words in the name.

### **H. Weighting, Calibration, and Applications**

After computing all six component scores $SC, ST, LN, NL, DR,$ and $SR$, we combine them into the final readability score $R(N)$ using the weight vector $\mathbf{w} = (w\_{SC}, w\_{ST}, w\_{LN}, w\_{NL}, w\_{DR}, w\_{SR})$. As noted, the weights sum to 1. Determining appropriate weights for the new six-dimensional model is important because not all factors are equally critical. In our extended model, we followed a procedure similar to the original four-factor calibration, augmented with considerations for the new dimensions:

* **Initial Weight Intuition:** Based on prior literature and our earlier results, **semantic clarity** remains the most crucial factor (we assign it a high weight, e.g. $w\_{SC} \approx 0.30$). A name that fails to convey meaning will hinder understanding even if it follows style or domain terms. We consider **domain relevance** and **stylistic conformity** as next-tier factors; each is important but somewhat less vital than core semantics. For example, a name that is meaningful but uses a slightly off-domain term or minor style violation is still largely understandable. We might give **DR** and **ST** weights on the order of ~0.15 each. **Natural-language readability** (fluency of the phrase) also remains significant (around 0.15) – it refines the clarity and ensures names are easy to read. **Syntactic role conformity** serves as a consistency check; while useful, a violation here (e.g., a verb-noun mix-up) might be a smaller readability issue than a completely obscure name. We assign SR a moderate weight (perhaps ~0.15 as well, comparable to style). **Length** is somewhat less critical in our findings (as long as it’s not extreme), so we give **LN** a slightly lower weight (around 0.10). These proposed weights sum to 1 (e.g., $w = (0.30, 0.15, 0.10, 0.15, 0.15, 0.15)$ for $(SC, ST, LN, NL, DR, SR)$). This reflects our hypothesis that *meaning* and *correct terminology* (SC+DR combined ~45%) are paramount, followed by *readability/consistency* factors (ST+NL+SR ~45%), and finally length (~10%).
* **Pilot Calibration:** We then calibrated these weights empirically. We extended our earlier survey of developers (where they rated identifier names on readability) to include cases illustrating domain and syntactic-role effects. For instance, we presented pairs of names for the same concept – one using a proper domain term vs. one using a generic term – and checked if human raters found the domain-specific name more readable. We also included examples of function names with correct verb phrasing vs. noun phrasing, to gauge the impact on perceived clarity. Using these new data points, we adjusted $w\_{DR}$ and $w\_{SR}$ to better align the model score with the human judgments. Similarly, we verified on a sample of domain-specific code (e.g., a medical software module) that our model assigns higher scores to identifiers that use medical terminology correctly, and tweaked the scoring function and weight if needed to ensure $DR$ had a measurable effect. This calibration confirmed, for example, that introducing domain relevance improved the model’s correlation with expert judgments in domain-heavy code, and that the SR factor helped catch names that developers found “oddly worded” even if they were otherwise clear. The final weights were chosen to maximize agreement with the survey ratings and intuitive rankings, subject to the constraint $\sum w\_i = 1$. (In practice, small variations in weights around the chosen values did not significantly change the model’s ordering of good vs. bad names, indicating the model is robust.)
* **Revised Formula:** The resulting combined score is as given earlier, $R(N) = \sum\_{X \in {SC,ST,LN,NL,DR,SR}} w\_X X(N)$. For implementation convenience, we scale $R(N)$ to a percentage or 0–10 scale in some tools, but all analysis is based on the 0–1 normalized score.

**Applications:** The enhanced six-factor readability model opens up new possibilities for improving code quality in practice. The inclusion of domain knowledge in the model enables **domain-sensitive static analysis** tools. For example, a static analyzer can use the $DR$ score to flag identifiers that do not align with the established project terminology. If a codebase has a glossary of terms (as recommended by Deissenböck et al.), our model would detect names using out-of-domain or inconsistent terms and suggest replacements, thus enforcing conceptual consistency across the code. The $SR$ component, on the other hand, can be integrated into linters or code review bots to automatically enforce naming conventions about grammatical form. For instance, during code reviews, a bot could warn “Function name should be a verb phrase” if it encounters a low $SR$ score for a method name. This provides immediate feedback to developers, encouraging best practices.

More broadly, the composite readability score $R(N)$ can be used in **continuous integration** pipelines – for example, failing a build or at least issuing a warning if new code introduces very low-scoring names (similar to how some teams set minimum code coverage or use linters to enforce style). This ensures naming quality is maintained over time. We foresee integration with modern IDEs and code review platforms where our model serves as an automated “naming advisor.”

Another exciting application is in **automated naming assistance and refactoring tools**. With the rise of AI code generators and large language model assistants, our readability model can act as a post-processing filter to choose among candidate names. For example, an AI pair programmer could generate multiple suggestions for a variable name; our model can rank these suggestions by $R(N)$ and recommend the most readable option. Likewise, a **renaming tool** could scan an existing codebase, identify the lowest $R(N)$ scores (names that likely hinder comprehension), and suggest better alternatives (perhaps by drawing from domain vocabulary or fixing the syntactic form). Such a tool could automatically improve legacy code by aligning it with current naming standards and domain language. Indeed, prior work on project glossaries and identifier dictionaries shows the benefit of consistent naming, and our model provides a quantitative framework to implement those ideas.

In summary, by extending the identifier readability model with Domain-Based Semantic Relevance and Syntactic Role Conformity, we capture more context in the scoring – *what* language the code is written in (the domain), and *how* that language is structured (the grammar of names). Our calibrated six-dimensional model yields a holistic readability score that correlates with human judgments and addresses gaps left by considering only general factors. We expect this enhanced model to be valuable for developers and tool builders alike, enabling better naming checks in code reviews, domain-aware linting, and intelligent assistants that help maintain a clean, comprehensible codebase. The next sections will evaluate this extended model empirically and demonstrate its effectiveness in real-world projects.

**Sources:**

1. Deissenböck, F., & Pizka, M. (2006). *“Conciseness, consistency, and convention in identifiers: A formal model and its application.”* Software Quality Journal, 14(1).
2. Recent Literature Review on Identifier Readability (Mane, 2024) – highlights importance of domain terms and POS conventions in naming.
3. Our initial model calibration study (Section 4.6 of this thesis) – methodology for weight tuning.
4. Discussion on tool integration of readability metrics (Section 7 of this thesis) – potential for code reviews and AI assistant usage.