Ambiguous, Informal, and Unsound: Metaprogramming for Naturalness

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
petrick.rein@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

Program code needs to be understood by both machines and programmers. While the goal of executing programs requires the unambiguity of a formal language, programmers use natural language within these formal constraints to explain implemented concepts to each other. This so called *naturalness* – the property of programs to resemble human communication – motivated many statistical and machine learning (ML) approaches with the goal to improve software engineering activities.

The metaprogramming facilities of most programming environments model the formal elements of a program (meta-objects). If ML is used to support engineering or analysis tasks, complex infrastructure needs to bridge the gap between meta-objects and ML models, changes are not reflected in the ML model, and the mapping from an ML output back into the program's meta-object domain is laborious.

In the scope of this work, we propose to extend metaprogramming facilities to give tool developers access to the *representations* of program elements within an exchangeable ML model. We demonstrate the usefulness of this abstraction in two case studies on test prioritization and refactoring. We conclude that aligning ML representations with the program's formal structure lowers the entry barrier to exploit statistical properties in tool development.

CCS Concepts • **Computing methodologies** → *Machine learning*; • **Software and its engineering** → *Abstraction, modeling and modularity*; *Object oriented languages.*

Keywords meta-objects, metaprogramming, machine learning, naturalness

META '19, October 20, 2019, Athens, Greece

ACM ISBN 978-1-4503-6985-5/19/10…$15.00
https://doi.org/10.1145/3358502.3361270

ACM Reference Format:
Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2019. Ambiguous, Informal, and Unsound: Metaprogramming for Naturalness. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (META '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358502.3361270

1 Introduction

The complexity of modern software systems requires collaboration among multiple programmers. Hence, the source code itself becomes an artifact of communication and persisted domain knowledge. The idea that programmers should "concentrate rather on explaining to human beings what we want a computer to do" has been a prominent hypothesis in Knuth's work on *Literate Programming* [9].

Even if modern programs are usually not "literate" in Knuth's sense (yet), machine learning and information retrieval techniques that worked well on literature and text documents have successfully been applied to source code. Examples include *semantic clustering*, a reverse-engineering approach by Kuhn et al. [10] that uses the semantic relatedness between identifiers to cluster software artifacts, or the extraction of latent *concepts* from large source code corpora using topic modeling by Linstead et al. [12]

More recent analyses performed by Hindle et al. in their work on the *Naturalness of Software* [6] systematically show that programs possess statistical regularities that correspond to those exploited by the natural language processing (NLP) community. Especially as the transition from formal (rule-based) methods to statistical (probabilistic) methods has proven very productive in the NLP domain, we can reasonably expect that software engineering can analogously benefit from statistical/ML methods augmenting the currently prevalent formal methods.

Problem An obstacle in applying these insights to software engineering is the heavyweight infrastructure needed to connect program and "black-box" ML models, and then mapping their output back into the program. The following problems play a role in this infrastructure and will be assessed in this work:

- The metaprogramming domain and the input domains of ML models are often misaligned. For example, NLP-based models will likely use *documents* or *n-grams* as primary abstraction, which need to be mapped to *classes*, *methods*, or *ASTs*.
- ML models have a resource-intensive training and relatively fast application mode, which works well on natural language with stable meanings of words. Abstractions and the corresponding identifiers in programs evolve much faster, requiring ML models to readjust on-line and within the granularity of programmers' changes.
- Introducing ML into a tool can introduce significant technical debt, since model-specific state needs to be passed through all interfaces that eventually need ML-determined properties. Re-use of ML models, intermediate representations, and results between multiple tools is not standardized and to the authors' knowledge rarely observed in practice.

Goal Our goal is to extend metaprogramming facilities to provide access to the *output* generated by a large class of ML models. Today's prevalent form of metaprogramming provides a reification of the formal elements of a program (e.g. classes and methods in object-oriented systems), which we call *meta-objects* in the scope of this work. Static analysis tools, test runners, frameworks, or language extensions can be based on meta-objects, and should be able to access probabilistic properties computed through ML models as well.

To address the problems stated above, our design follows the following principles:

Mapping Each meta-object should have a representation inside the ML model. This principle should eliminate the need to manage mappings between meta-object domain and ML domain in client code for using and updating the model.

Decoupling model from application Code using properties computed by the ML model should not interleave with code managing the ML model itself. This allows both to vary independently so that analysis tools can continue to focus on operating in the meta-object domain like they did for formal analyses.

Shared representation Different tools using the same ML-derived properties should access them through the same meta-objects. No separate pipelines need to be constructed.

This is not an exhaustive list of improvements, but a first take on overcoming misaligned inputs, outputs, and updates, while reducing the invasiveness of ML pipelines.

In this paper, we first elaborate on the role of *naturalness* in software and give a high-level overview over techniques utilizing this information in section 2 (algorithmic details of specific machine learning models are out of scope). In section 3, we then present a novel extension to Smalltalk meta-objects for accessing this information. We proceed to use this framework in section 4 to improve two development tools: The automatic test runner and the refactoring capabilities of the code browser, each with minimal changes to the existing meta-object code. In section 5, we elaborate on design decisions and document the requirements an ML model needs to satisfy in order to work in this new framework. In section 6, we discuss related and possible future work.

2 Naturalness in Programs

Although it is possible to derive representations and machine learning input from the formal structure of a system, their power stems from their capability to capture statistical effects associated with natural language and structure, such as patterns emerging at run-time or in the development process.

2.1 Manifestations of Naturalness

In programs, *names*, often joined from multiple words via camel case or separators, provide the basis for communicating computational concepts. From a statistical viewpoint, words that appear in similar contexts likely share the same meaning ("difference of meaning correlates with difference of distribution" [5]), which can be used to measure semantic relatedness of program parts independently of their formal relations between each other.

A wider range of language can be found in *comments*, ranging from pure natural language documentation to commented-out code. Ordering, spacing, alignment, indentation, and other "*negative space*" can convey meaning by visibly grouping code passages and, at the same time, separating them from less related code.

Beyond source code, related *artifacts* like issues, discussions, pull requests, build logs, or documentation can convey meaning. In a software repository, the *version history* reveals how spatially unrelated program parts can still be semantically connected by being frequently changed together. *Dynamic* information generated at run-time plays a significant role in program understanding, as the widespread use of graphical debuggers or `printf`-style logging indicates. Concrete data and control flows can be observed and statistically mined.

2.2 Exploiting Naturalness in Software Engineering

There are multiple approaches to exploit natural information through statistical and ML models in software engineering. A comprehensive study by Allamanis et al. identifies three families of models [1]:

Code-generating Models statistically learn how code is composed from simpler parts (e.g. AST nodes or tokens). Examples are *grammars* with probabilities at each rule; or *n-gram* language models that record

which tokens likely follow a sequence of previous tokens. They can be replayed to generate new code that follows the learned conventions (code completion, program synthesis, project migration), locate unusual code (fault detection, code review), or generate unlikely code for fuzzing and obfuscation purposes.

Representational Models learn a compact, semantic representation of code (analogous to a lossy compression), e.g., *embeddings* into a vector space where conceptually related code is in close proximity and unrelated code more distant[1]. Such representations can be compared for "smart" code search or clone detection, or serve as input for classifiers that predict faults. Natural language generators can use semantic representations as input to generate names, commit messages, or code summarizations.

Pattern Mining extracts groups of statistically often co-occurring features from programs. These can be structural (e.g. design patterns, high-level relationships), lexical (e.g. topics), or mixed. A popular approach that belongs in this category is *topic modeling*, where topics are generated by grouping semantically related or often co-occurring features (e.g. identifier names), and each unit of code (e.g. class) is assigned a probability of being concerned with each topic. In many cases, topics can be directly mapped to domain concepts and help with reverse engineering, search, recommender systems, and clone detection.

In the scope of this work, we focus on expressive programmatic access to existing meta-objects rather than mechanisms to generate new program code or meta-objects from ML models. Representational as well as pattern mining models are prevalent in this domain.

2.3 Working with Representations

A milestone in dealing with the inherently ambiguous and redundant nature of human communication artifacts was the use of *representations* to model their meaning in a robust, machine-understandable way. A representation preserves the relevant semantic properties of each linguistic item and the relations between them without the need to store each individual, possibly noisy, relationship. Given two artifacts (e.g. words or texts), their relationship can be approximately re-computed from their representations, and unobserved relationships can be predicted effectively.

Obtaining a representation is usually preceded by *feature extraction*. During feature extraction, raw data is converted to an input representation suited for the model. Typical feature sets are *vectors* with each coordinate corresponding to the frequency of a specific word in an artifact, or *bags of words*. Stemming, weighting, and the removal of frequent words can be applied. In programs, for example, camel case can be split, and the range of features extends to syntactic and semantic elements of the programming language.

[1] *Word2Vec* is the analogous approach from the NLP domain [15]

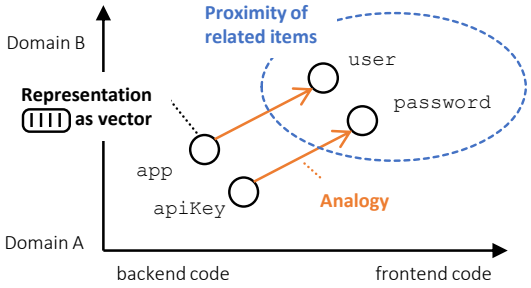

Figure 1. Illustration of a representational model embedding identifiers into a vector space, preserving their relationships approximately. In some models, the axes might reflect human-interpretable concepts (e.g. LDA), in others (e.g. word2vec) not. Typically, the embedding has more dimensions (up to 300) than depicted in this illustration.

Both representational and pattern mining models compute output representations that indicate how specific features or parts of the program relate to the latent concepts or patterns mined by the model. Often, these representations are *vectors*, like depicted in Figure 1, that exhibit a range of properties:

Topicality Individual vector dimensions indicate the degree to which the represented entity coincides with a *latent factor* or *topic*. One of the most widely used models that exhibits this property is the LDA model [2]. In software engineering, such concepts can represent parts of the implemented domain, clusters of technical vocabulary, or different coding styles among others.

Preserving semantic distance An *inner product*, $\odot$, of two representations can be defined, which results in a scalar that indicates how closely related the two represented features or meta-objects are. The semantic distance of units of software has many uses in software engineering, ranging from clone detection, over code search, to recommender systems. It can serve as code metric with respect to coupling (inner product between separate modules) and cohesion (inner product between module constituents). In section 4, we demonstrate its use in prioritizing tests that are semantically related to a change, and judging whether moving code to another class improves cohesion.

Composing meanings Some models have additive properties, i.e., if we compose two meta-objects or features with representations r_1 and r_2, their composition can be represented as $r_1 \oplus r_2$. The $\oplus$ operation may need to preserve certain constraints beyond just adding vector elements, e.g. that probabilities sum up to 1 or vector magnitudes stay

normalized. In software engineering, composition becomes more relevant and we need to define a standard protocol even for models whose representations are not directly composable. In this work, we rely on composition to compute the meaning of a class from its methods, e.g. in Listing 8.

Analogical reasoning Besides composition, a small family of representations provide semantics for subtraction, $\ominus$, of representations. With such an operation, the difference between meta-objects or features represent the relation in which they participate. Assume we have four classes, `IntArray`, `String`, `Integer`, and `Char`, then a representation with subtraction would approximate: $r_{String} \ominus r_{Char} \approx r_{IntArray} \ominus r_{Integer}$, indicating the system has learned that strings relate to characters like arrays relate to numbers. While this property is listed here for completeness reasons, implementing analogy in metaprogramming is left for future work.

3 A Metaprogramming-based Interface to Representations

Our goal is to extend meta-objects with properties that reflect the output of a (late-bound) statistical or ML model. We propose to introduce common abstractions for dealing with *semantic* information mined from natural language elements and other properties of a software system and attach them to the meta-object graph.

This way, meta-objects do not purely serve as input to an ML pipeline, but as both ends of an ML model. As a trade-off, this accessibility at meta-object level requires the ML model to no longer be a *black box* or run in a single-shot pipeline, but to match the structure of the program and react to changes at meta-object granularity (e.g. for individual classes or methods).

A note on notation We use the *Smalltalk* language as notation and the Squeak/Smalltalk [8] system as implementation platform. In Smalltalk, a program consists of live meta-objects that can be inspected and manipulated rather than merely source code. This simplifies dealing with the meta-object domain compared to languages where reflection and run-time metaprogramming has life cycles and object models that differ from code-based or binary-based metaprogramming. (In Java, for example, the Eclipse JDT infrastructure can provide tool developers access to a method's AST, while reflection can provide `Method` objects at run-time, but neither does the AST exist at run-time, nor does a change to the `Method` object affect the source code or class file. In Smalltalk, there is only a single `CompiledMethod` instance playing both roles.)

3.1 Models and Meaning

In one of our case studies, we sort unit tests according to their relevance to the most recently changed method as a way to prioritize their order of execution. To access representations at meta-object level, we introduce the `meaning` method on each meta-object. Obtaining the *meaning of a test method* can be done using `testMethod meaning`.

The resulting *meaning* can be compared to other meanings with respect to semantic similarity (`<->`) and helps us judge the relevance of a test method given a change: `relevance := testMethod meaning <-> changedMethod meaning`. The comparison returns larger values for more related methods with respect to the ML model. The model itself is abstracted away and not a concern at this point in the program, but can be accessed using a dynamically scoped variable `ActiveModel`.

In general, we provide the following entry points:

Meaning The *meaning* of a meta-object `m`, obtainable via the accessor method `m meaning`, encapsulates its representation with respect to the *model* that is *active* at the time of calling this method.

Model The *model*, accessible via the dynamically scoped variable `ActiveModel`, determines which representation is chosen for a *meaning* and provides implementation for comparison and composition operators. Different models for different machine learning algorithms can co-exist, but only one is *active* at a time.

The correspondence between meta-objects, their meanings, and the model is illustrated in Figure 2. By equipping *any* meta-object with a `meaning` method, we realize the mapping principle.

Meaning objects provide the following two operations:

Comparison The comparison `m1 <-> m2` between two meaning objects implements the inner product $\odot$ of their encapsulated representations and returns a numeric value indicating how similar the underlying meta-objects are.

Composition Creating a new meaning by composing two meta-objects' meanings `mc := m1 + m2` results in a representation for a (hypothetic) meta-object consisting of the two initial meta-objects. This implements the composition operator $\oplus$.

Implicit models Instead of writing `model meaningFor: m`, we prefer the shorter notation `m meaning`, with `meaning` then invoking the dynamically scoped variable `ActiveModel` via a mechanism described in section 5. This design is controversial from a modularity viewpoint, because meta-objects are extended by ML-specific functionality rather than keeping the new functionality separate and importing it when needed. However, we found that it eliminates the need to pass a `model` object through several interfaces and discourages storing a (potentially outdated) model in some field. In Smalltalk, we have several mechanisms to extend existing abstractions in a modular way, e.g. through extension methods (defined at a class from one package but

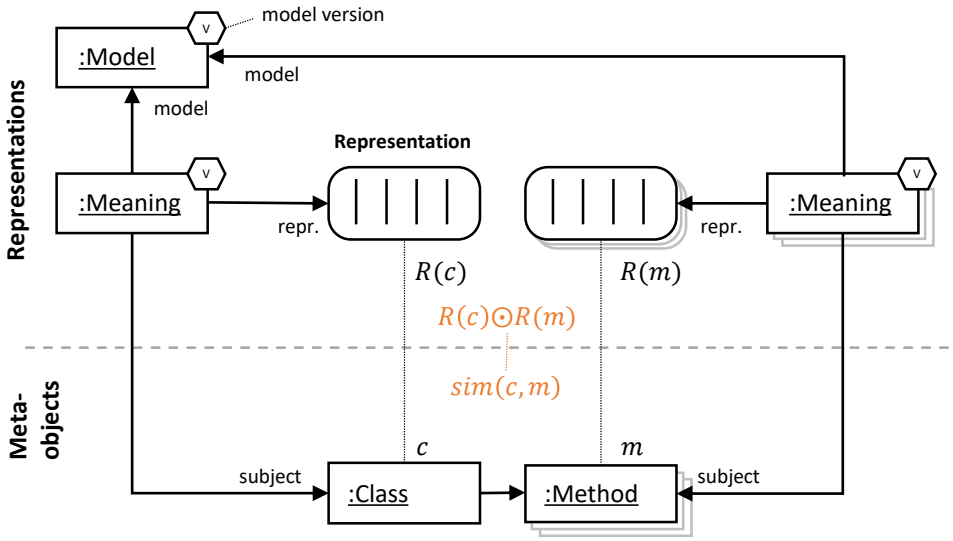

Figure 2. Correspondence between meta-objects and representations in an ML model, and the role of `Meaning` objects as abstraction over the representation details chosen by the model.

loaded and version-controlled by a different software package) or context-oriented programming (outlined in section 5), which keeps the development life-cycle of the mechanism separated from the core reflection API.

4 Case Studies

In this section, we show how a fully decoupled ML model can be used from within the meta-object domain, using two examples that make heavy use of metaprogramming and benefit from integrating ML-learned representations: test prioritization and automated refactorings.

4.1 Test Prioritization

Unit and regression testing is a widely practiced approach to detect faults during software evolution. However, test suites for larger software projects tend to run for several hours, thereby limiting how often they are run, how up-to-date their feedback is, and how easy a defect can be fixed when finally found.

The goal of test prioritization is to identify tests that most likely fail and run them earlier. This both reduces the time to obtain feedback in the presence of faults, and increases the confidence in test results when only a limited set of tests is run due to time constraints.

There are two major approaches: the *static* approach takes a program and a set of tests as input and tries to order tests in a way that large parts of the program are covered early on, while moving redundant, long-running, or highly specific tests into later stages. *Change-based* prioritization takes the most recent change as input and predicts which tests best detect faults introduced in this particular change. In this case study, we extend an existing change-triggered test runner to apply change-based prioritization before running tests.

Prioritizing by meaning Since changes and test cases can be considered collections of meta-objects as well, they have a conceptual meaning within the program. Since we defined an inner product on representations that yields a number of their relatedness, we can use this as a metric to compare tests. The priority p of a test t in response to changed meta-object m can be expressed as $p(t, m) = R(t) \odot R(m)$ and thus allows tests to be sorted beforehand.

The AutoTDD runner Smalltalk has a change-triggered test runner, *AutoTDD*, that runs a previously selected test suite after every individual method update. During configuration by the user, it creates a `TestSuite` object consisting of `TestCase` instances, each test case instance representing a test method (implemented at that specific class) to be executed. The runner then attaches itself to Smalltalk's `SystemChangeNotifier` and, whenever a method within the scope of the test suite is changed, all tests are re-run. The change event is an instance of `ModifiedEvent` that references the old meta-object and the new version replacing it.

Prioritizing tests To implement test prioritization, what needs to be done is parametrizing the `TestAutoRunner >> runSuite` method (which gets called in the event handler upon modification) to `runSuiteGiven: aMethodChange` in order to pass the current change to the test suite, and instead of only iterating over `TestSuite >> tests`, the runner

Listing 1. Sorting Tests

```
TestSuite >> prioritizedTestsGiven: aMethodChange
  | changeMeaning |
  changeMeaning := aMethodChange newMethod
    meaning.
  ^ self tests sorted:[:testMethod1 :testMethod2 |
      (testMethod1 meaning <-> changeMeaning)
        > (testMethod2 meaning <-> changeMeaning)]
```

now iterates over `TestSuite >> prioritizedTestsGiven: aMethodChange`, allowing it to yield sorted test methods in response to the current change.

We use the `<->` operator, which asks the ML model to compute the inner product ($\odot$) between a test method's representation and the changed method's representation (see Listing 1).

Choosing a model Test prioritization itself already works with simple lexical models. For our initial prototype, the model extracts identifier names from each meta-object. The *representation* is a *multi-set* of these features, with the inner product defined as the relative size of their intersections. Composition is realized as (multi-)set union. More sophisticated lexical models for test prioritization, including *TF-IDF*-based and *LDA*-based variants have been discussed in a previous study [14] and implemented in Smalltalk using the new abstractions. In the LDA case, the model needs an initial setup to configure hyper-parameters and train for several iterations. The model registers as change listener, so any new method or change in the specified package causes an incremental re-training. Using such a model in test prioritization can detect introduced defects within the first few unit tests, as shown in [14].

Discussion At this point, no direct reference to any ML model has been made, as our entry points to access the ML representations are fully defined by the `meaning` property of the meta-objects. The meta-objects themselves, however, must be passed to a suitable location, which required most of the changes in adapting the *AutoTDD* runner.

The sort method uses merge sort, requiring a number of comparisons in $O(n \log n)$. Each comparison requests representations of two method meta-objects. If the model itself provides lazy *caching*, there is only the need to compute n representations (with n being the number of test methods). An eager caching strategy, with the model listening for any changed test method and updating their representation proactively, requires no computation of representations during test runs, which might even be the preferable option given our goal to reduce feedback time during tests.

The quality of the prioritization results and their performance now depends on which model is being set up. This raises the question of the scope in which a model is active - in this case, we might want to have either a *global* scope, i.e., the test runner uses the system-wide default model, or a *tool*/*module* scope, where a model optimized for test prioritization is active anytime the control flow originates from any module in the test runner package.

Listing 2. Checking a refactoring

```
Browser >> dropOnMessageCategories: method at:
    index
  ... "drag/drop handling"
  destinationMatchesSemantically :=
    (destinationClass meaning <-> method meaning)
      > (sourceClass meaning <-> method meaning).
  destinationMatchesSemantically ifFalse: [
    ... "let user confirm the move"]
```

4.2 Refactoring

Refactoring often aims at improving the modularity of a program. With a representational model, we can quantify modularity in terms of semantic similarity (inner product). If, for example, our aim is to move a method to another class, we can check if the destination class provides a more similar environment to that method than the original class.

In Squeak/Smalltalk, we can instrument the code browser that handles a simple variant of the *Move Method Refactoring* by drag and drop: in Listing 2, we compare the similarities of the dragged method to both classes and request user confirmation if the change would move the method to a less related class.

Reified refactorings The Smalltalk Refactoring Browser [16] was one of the earliest tools to help programmers perform a range of common behavior-preserving modifications without manually editing the code. In this tool, refactorings are *reified*, i.e., they store the relevant configuration and provide a method to eventually re-write the source code. We can add an analogously simple code snippet to the `transform` method of a `MoveMethodRefactoring` to check the above constraint, see Listing 3.

Most refactorings can be extended by an `effectiveness` metric that takes the difference between modularity before and after the refactoring. For example:

Analogously, renaming a class or extracting parts of an AST into another method can easily compare the fitness of their new name given the code.

Discussion Refactorings, especially the reified family used by the Refactoring Browser, are formal and meta-object-heavy operations that can benefit from insights generated by machine learning. The new capability to not only rely on formal modularity metrics (e.g. tight class cohesion or

Listing 3. Computing the effectiveness of a refactoring

```
RBChangeMethodNameRefactoring >> effectiveness
   | methodMeaning |
   methodMeaning := (self class >> oldSelector)
     parseTree body meaning. "includes comments"
   ^ (newSelector meaning <-> methodMeaning)
      - (oldSelector meaning <-> methodMeaning)
```

cyclomatic complexity [11]) but to factor in naturalness, allows these *refactoring metrics* to quantify how well the *intent* to refactor towards semantically coherent modules is met. Even if improvement of cohesion is insignificant in most smaller refactorings, a scoring mechanism can prevent code relocations to completely unrelated classes by mistake or misunderstanding.

In programming environments, suggesting names based on their adequacy for the module they describe, or highlighting the most promising targets for code-moving refactorings, can now be implemented based on the refactoring effectiveness. Combining natural and formal modularity metrics gives rise to future work on evaluating and recommending refactorings.

Shared representation Both test prioritization and refactoring can rely on the same model and even re-use the representations. The `Meaning` instances that become updated following the change event triggered by the refactoring are immediately available to the prioritizing test runner. This demonstrates our third principle – *shared representation* – in action.

5 Implementation

This section elaborates on design and implementation decisions underlying the meaning/model protocol.

ML models as context The primary entry point is the method `meaning` on any meta-object. The returned object encapsulates the implementation details of the meta-objects' representation in the currently active ML model. A design decision is how the model can be properly separated to not interfere with code operating in the meta-object domain, as required by our principle of *decoupling model from application*.

Both *dynamic variables* [19] and *context-oriented programming* [7] are suitable to separate ML model from meta-object code:

As a dynamic variable, a model can be brought in scope during certain control flow. Any code requiring the model asks the dynamic variable `ActiveModel`. At the point of configuration, programmers can use `ActiveModel value: model during: [code]` to run code under the specified model `model` (see Listing 4).

In context-oriented programming (Listing 5), the model can become a *layer* that provides method adaptations when activated. In contrast to a dynamic variable, client code does not need to address the active model, but provide variation points (methods), which the layer will dynamically override.

If the host language supports dispatch on multiple arguments, which is not the case in Smalltalk, a multimethod can serve the same role.

Listing 4. Dynamic variable and double dispatch

```
CompiledMethod >> meaning
  ^ ActiveModel value meaningOfMethod: self

Class >> meaning
  ^ ActiveModel value meaningOfClass: self
  ...


Model >> meaningOfMethod: aMethod
  ^ self represent: aMethod parseTree

Model >> meaningOfClass: aClass
  "compute representation of aClass"
  ...

ActiveModel value: myModel new during: [
  metaObject meaning].
```

Listing 5. Context-oriented Programming

```
Object >> meaning
  "default implementation"

Model (layer) >> CompiledMethod >> meaning
  "override meaning if active"
  ^ thisLayer represent: self parseTree

Model (layer) >> Class >> meaning
  "compute representation of class"
  ...

myModel withLayerDo: [
  metaObject meaning].
```

While the context-oriented variant provides stronger separation between model and meta-objects and avoids *double-dispatch* to obtain specific representations for each type of meta-object, it raises complexity by using a non-standard language concept. With dynamic variables, only a single model can be active, while layers allow multiple models to be stacked and provide composition semantics. Dynamic variables are sufficient for the use cases in this paper, so we continue to use them and leave a context-oriented implementation for future work.

5.1 Meaning Objects

Our `Meaning` objects require information about the model that generated them, the meta-object they represent, and operations to compare and combine them.

Structure of Meaning instances Each `Meaning` instance holds the following private state:

- The *model* that owns this representation,
- The *features*, i.e., the input representation to the ML model, per default a `Set`,
- The output *representation*, whose type is specific to what the model's implementation uses
- The *version* the model had when generating this instance, since re-training/updating the model invalidates all existing representations,
- The *meta-object* represented by this instance, to avoid duplication of data already present in the meta-object.

Listing 6. Implementing the similarity operator

```
Meaning >> <-> other
  ... "(ensure same model and version)"
  ^ self model
      similarityBetween: self representation
      and: other representation
```

The construction of such an instance involves the following steps:

1. The model is asked to provide the meaning for the requested meta-object, double-dispatching according to its type.
2. The model retrieves the meta-object features (e.g. words), again a double-dispatch protocol is used that leaves variation points for different feature extraction routines and re-use.
3. The model retrieves its representation of the given set of features (e.g. a vector) and constructs the `Meaning` instance.
4. Features and representations remain cached. The representation cache must be discarded if the model is retrained, both must be discarded when the meta-object changes.

Inner Product In common scenarios where naturalness is utilized, the representations of meta-objects are used to compute *similarity*. Examples include *recommender systems*, where similarity to a context or task decides which item is proposed, *information retrieval*, where similarity to a query is used to rank results, or software *modularity*, where similarity of items between modules relate to *coupling* and within a module to *cohesion*.

The analogical mathematical concept underlying similarity is the *inner product* between two representations, $R(m_1) \odot R(m_2)$ with R being the representing function and m_i any meta-object. In Smalltalk, the `<->` operator (Listing 6) is used.

It is only defined between `Meaning` instances of the same model having the same version.[2].

Composition When a meta-object is composed of other meta-objects (e.g. classes consist of name, superclass, class comments, fields, and methods), representations of the constituents can be reused to infer the meaning of their composition. However, not all representations lend themselves to simple composition: While models like *word2vec* [15] can approximate the meaning of compound sentences from adding individual word vectors, a probabilistic topic model would need to recompute the most likely representation not just from the constituents' representations but from their features.

We therefore provide two ways to compose meaning:

[2]Not storing derived properties from meta-objects for a long time is advisable in general, and in this case not only the meta-objects but also the model can change, invalidating the stored representation

Listing 7. Implementation of the composite meaning

```
CompositeMeaning >> features
  ^ self meanings gather: [:meaning |
      meaning features]

CompositeMeaning >> representation
  ^ self model collapseComposite: self

Model >> collapseComposite: aCompositeMeaning
  ^ self representationForFeatures:
    aCompositeMeaning features
```

Immediate Composition $R(m_c) = R(m_1) \oplus R(m_2)$, where $\oplus$ is provided by the model

Destructuring Composition First, $R(m)$ is explicitly modeled as $R(m) = r(f(m))$ where r computes the representation of a set of *features*, while f extracts these features from meta-object m. Then, $R(m_1) \oplus R(m_2) = r(f(m_1) \cup f(m_2))$, i.e., the combined set of features from both metaobjects is jointly represented. In contrast to immediate composition, which may apply normalization or other non-associative operations, this operation is associative and commutative. The implementation of this mechanism is visualized in Figure 3.

Most models from the NLP domain support destructuring composition, because they originally worked on *documents*. Joining the features of two documents treats them like a single concatenated document. In our implementation, destructuring composition is performed by default, unless a hook (`Model >> collapseComposite:`) in the model is overridden to directly compose two `Meaning` instances.

To model intermediate results during composition, we introduce the `CompositeMeaning` class, which holds a collection `Meaning` objects, including other composites, and a `NullMeaning`, which will return the other `meaning` object whenever composed and yields an empty set of features if queried.

When asked for the representation, the `CompositeMeaning` recursively collects the features from child meanings (implemented by `Meaning >> features`, see Listing 7) and passes them to the model to be converted to the final representation. A composite meaning can be associated with a meta-object or a collection of meta-objects, in which case the representation will be cached and retrieved once the same meta-object or collection occurs again.

5.2 Model Objects

To implement an own model, `Model` needs to be subclassed and the following interface implemented:

- For each type of meta-object, a method `featuresFor<T>: a<T>` with `<T>` being the class name (double dispatch). This should return a collection of features the model needs to represent the particular meta-object.

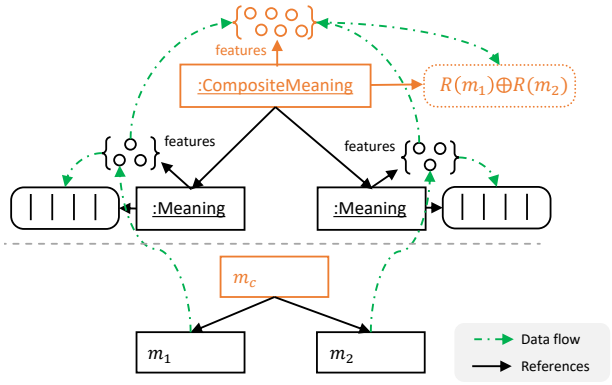

Figure 3. Compositional structure of meanings over their corresponding meta-objects. Data flow via *features* is highlighted. All data flows are supervised by the model.

Listing 8. Representing a class as composition of its methods

```
Model >> meaningForClass: aClass
  ^(aClass selectors
    collect: [:selector | (aClass >> selector)
    meaning])
     inject: Meaning null
     into: [:methodMeaning1 :methodMeaning2 |
       methodMeaning1 + methodMeaning2].
```

- `representationForFeatures:` which takes a collection of features and returns their representation within the model.
- `similarityBetween: r1 and: r2` computing the inner product $r_1 \odot r_2$ from two representations.

Around this minimal core, which already covers a wide range of simple models with word-like or n-gram features like *word2vec* or LDA-style topic models, a number of hooks can be overridden:

- `representationFor<T>:` can be overridden for specific types of meta-objects `<T>` to bypass feature extraction and directly emit a representation. This is useful if the model has already pre-computed a representation as part of the training phase.
- `meaningFor<T>:` can analogously be overridden to provide specific `Meaning` instances or custom subclasses, e.g. as part of an optimization that requires the `Meaning` to cache additional information, or to override the composition operator.
- `collapseComposite:` to implement a custom resolution for composite meanings.

The default implementation of `meaningForClass` in Listing 8 illustrates how composition can be used to express the meaning of a class as the sum of their methods.

Observer protocol To keep a model up-to-date, it needs to subscribe to the `SystemChangeNotifier` after initial training and implement an event handler to deal with meta-objects changing. Different types of changes (add, remove, change) of methods and classes need to be handled, but not necessarily immediately.

6 Related Work and Outlook

ML in Software Engineering State-of-the-art models used in software engineering go beyond the meta-object domain: Saeidi et al. [17] make use of kernel-based methods to learn representations of software modules that encode relationships in the evolutionary and dynamic domain as well by considering version history and call graphs.

The idea of representing changes as first-class meta-objects, e.g. via *Change Boxes* [4] and representing them inside a model that is capable of dealing with evolutionary information is appealing – it allows the model to "understand" what a change is about and, at the same time, learn new semantic correlations from co-changed program parts.

Tool building frameworks Frameworks that aim at building development tools, like Vivide [18], can benefit from added capabilities of meta-objects. Since building tools with Vivide requires the implementation of relatively small transformation steps that extract the desired information from meta-objects, dealing with ML models would have been prohibitively complex before. Having access to the meaning of each object, however, allows individual parts of a tool to quickly sort by relatedness to another meta-object, cluster, or visualize large collections of meta-objects by projecting their representations in 2D.

Mirrors Mirrors are a reflection concept that inspired our dual architecture. Mirrors are separated from the objects they reflect on and thus provide what Bracha et al. [3] call *encapsulation*, *stratification*, and *ontological correspondence*. Our meaning infrastructure realizes both encapsulation by separating the meaning interface from a concrete ML model implementation, and to a limited degree ontological correspondence by following the compositional structure of programs through composite meaning objects. Stratification is not fully achieved, as each meta-object is directly extended by the `meaning` method. A context-oriented approach where access to meanings is modularly separated into a layer would be more stratified.

Concept-aware IDEs Programming environments that "understand" what code is about [13] can help with a wide range of program comprehension and engineering activities. Mining a code base for concepts that constitute semantically connected source code locations, e.g. via topic models, and identifying their distinguishing vocabulary, can help with reverse engineering activities and highlight refactoring opportunities. Code completion can sort suggestions by semantic relatedness to the current context, changes can trigger recommendations for related code locations that are likely relevant, and code search is more robust when based on

conceptual similarity rather than exact lexical matches. Providing access to *concept-level* information at meta-objects minimizes the changes required at tooling level to implement this functionality.

7 Conclusion

Applications of machine learning in software engineering are often approximative and unsound, as they make probabilistic statements about the system rather than deducing formal properties. Similar approaches have proven powerful in the domain of natural language, where ambiguity, redundancy, and noise would make formal models too brittle. Analogously, many parts of programs that are essential for their comprehension, such as names, have meaning beyond their formal role. This "natural" aspect can be utilized by natural-language inspired ML techniques and put to use for a variety of development activities, often in liaison with formal methods.

Metaprogramming and its corresponding meta-object domain have been representing formal elements of programs in the past and are frequently used in analyses and programming tools. Introducing the capabilities of machine learning by giving programmers access to the natural meaning of each meta-object as determined by an ML model serves as important starting point to better integrate ML models with programming environments. We could demonstrate in our test runner and refactoring case studies that meta-object code can, without major changes, directly prioritize tests or judge the quality of a refactoring based on their similarity in the natural domain.

We conclude that much of the architectural overhead, which is rarely addressed by the research of ML in software engineering, can be eliminated through such a framework. With the possibilities of machine learning readily available for metaprogramming, future ML-based programming tools might also integrate better into the programming workflow, like code completion, syntax highlighting, or graphical debugging already do today.

References

[1] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4 (July 2018), 81:1–81:37. https://doi.org/10.1145/3212695

[2] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3 (March 2003), 993–1022.

[3] Gilad Bracha and David Ungar. 2004. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 331–344. https://doi.org/10.1145/1028976.1029004

[4] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. 2007. Encapsulating and Exploiting Change with Changeboxes. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007 (ICDL '07)*. ACM, New York, NY, USA, 25–49. https://doi.org/10.1145/1352678.1352681

[5] Zellig S. Harris. 1954. Distributional Structure. *WORD* 10, 2-3 (Aug. 1954), 146–162. https://doi.org/10.1080/00437956.1954.11659520

[6] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 837–847.

[7] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich* 7, 3 (2008), 125–151. https://doi.org/10.5381/jot.2008.7.3.a4

[8] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*. ACM, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[9] D. E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (Jan. 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[10] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. 2007. Semantic Clustering: Identifying Topics in Source Code. *Information and Software Technology* 49, 3 (March 2007), 230–243. https://doi.org/10.1016/j.infsof.2006.10.017

[11] Michele Lanza and Radu Marinescu. 2007. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media.

[12] Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. 2007. Mining Concepts from Code with Probabilistic Topic Models. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, Atlanta, GA, USA, 461–464. https://doi.org/10.1145/1321631.1321709

[13] Toni Mattis. 2017. Concept-Aware Live Programming: Integrating Topic Models for Program Comprehension into Live Programming Environments. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)*. ACM, New York, NY, USA, 36:1–36:2. https://doi.org/10.1145/3079368.3079369

[14] Toni Mattis, Falco Dürsch, and Robert Hirschfeld. 2019. Faster Feedback Through Lexical Test Prioritization. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/3328433.3328455

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]* (Jan. 2013). arXiv:cs/1301.3781

[16] Don Roberts, John Brant, and Ralph Johnson. 1997. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems* 3, 4 (1997), 253–263. https://doi.org/10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.0.CO;2-T

[17] Amir Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen. 2019. Applications of Multi-View Learning Approaches for Software Comprehension. *The Art, Science, and Engineering of Programming* 3, 3 (2019), (to appear).

[18] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-Driven Tool Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/2661136.2661150

[19] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. 2007. Context-Oriented Programming: Beyond Layers. (2007).