# Towards Concept-Aware Programming Environments for Guiding Software Modularity

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

Patrick Rein
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Stefan Ramson
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
stefan.ramson@hpi.uni-potsdam.de

Jens Lincke
Software Architecture Group
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
jens.lincke@hpi.uni-potsdam.de

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

## Abstract

To design and implement a program, programmers choose analogies and metaphors to explain and understand programmatic concepts. In source code, they manifest themselves as a particular choice of *names*. During program comprehension, reading such names is an important starting point to understand the meaning of modules and guide the exploration process.

On the one hand, understanding a program in depth by looking for names that suggest a particular analogy can be a time-consuming process. On the other hand, a lack of awareness which concepts are present and which analogies have been chosen can lead to modularity issues, such as redundancy and architectural drift if concepts are misaligned with respect to the current module decomposition.

In this work-in-progress paper, we propose to integrate *first-class concepts* into the programming environment. We assign meaning to names by labeling them with a *color* corresponding to the metaphor or analogy this name was derived from. We hypothesize that aggregating labels upwards along the module hierarchy helps to understand how concepts are distributed across the program, collecting names belonging to a specific concept helps programmers to recognize which metaphor has been chosen, and presenting relations between concepts can summarize complex interactions between program parts. We argue that continuous feedback and awareness of how names are grouped into concepts and where they are located can help preventing modularity issues and ease program comprehension.

As a first step towards an implementation, we define criteria that help to detect names belonging to the same concept. We then investigate how techniques from natural language processing can be re-used and modified to compute an initial concept allocation with respect to these criteria. Eventually, we show design sketches how we plan to arrange and present concepts to programmers through tools, and what kind of information they can provide to help programmers make informed implementation decisions.

***CCS Concepts*** • **Software and its engineering** → *Abstraction, modeling and modularity*; **Integrated and visual development environments**; Object oriented architectures; • **Computing methodologies** → Semi-supervised learning settings; Mixture models;

***Keywords*** analogies, first-class concepts, program comprehension, topic models, integrated development environments

**ACM Reference Format:**
Toni Mattis, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2017. Towards Concept-Aware Programming Environments for Guiding Software Modularity. In *Proceedings of 3rd ACM SIGPLAN International Workshop on Programming Experience (PX/17.2)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3167110


*PX/17.2, October 22, 2017, Vancouver, BC, Canada*

ACM ISBN 978-1-4503-5522-3/17/10…$15.00
https://doi.org/10.1145/3167110

## 1 Introduction

Programming is an activity that extends far beyond typing source code into an editor and occasionally investigating execution state in a debugger. To design and implement a

program, programmers have to make a large number of decisions on how to map real-world ideas to concepts understood by the execution environment at hand. Naur described this process as *theory building* and the resulting mental model as the *theory* that allows its possessors to effectively create and manipulate programs[10]. Therefore, it is crucial that programmers cognize this theory to succeed at editing an existing program or module they have not yet worked with.

***Explaining concepts through analogies*** We assume that a significant portion of the theory building consists of finding *analogies*. Whenever we recognize the structure of an already understood concept in a new idea, it provides us with vocabulary to talk about the new idea as well as hypotheses on how to deal with and what to expect from it. Such analogies range from simple metaphors, e.g., an object named *stack* suggesting that things can be *pushed* onto and *popped* off its top, to composite domain concepts, e.g., that a *traffic network* consisting of *intersections* and *roads* is analogous to a *graph* with *vertices* and *edges*, that an asynchronous control flow can be understood as a *promise*, or that a structure should be traversed using the *Visitor* design pattern. This analogy-making process plays a central role in the *abstraction*[1] process that separates implementation details from external presentation. A recent, more detailed description of how metaphorical reasoning helps programmers can be found in [15].

***Problem statement*** Programming environments are oblivious of the fact that names carry meaning. Names are only aliases to be eventually resolved to executable code or memory locations using string matching. At the same time, programmers trying to recognize the underlying *theory* tend to rely on names and the analogies they suggest. They provide programmers with a heuristic to form testable hypotheses on the program's structure and behavior, e.g., seeing the messages *push* and *pop* successively sent to an object suggests that the latter resolves to the same object that was pushed before and lessens the urge to investigate further. The theory has been simplified by making an analogy. However, discovering concepts based on scanning for revealing names and how they relate to each other can be a time-consuming process [4]. Most programming environments dictate the way in which code can be arranged and modularized, often in a hierarchy, causing *cross-cutting* concepts to become scattered, and modules to lose their coherence by accommodating parts of a scattered concept. This comprehension process becomes even more challenging when the concepts gradually sheer from the initial module decomposition during program evolution, a process referred to as *architectural drift*. A lack of awareness of the chosen analogies can also lead to *inconsistent classification and naming*, e.g., by introducing synonyms instead of using a single name for the same idea, or *duplication* by failure to recognize a very similar concept somewhere else.

***Towards first-class concept support in development environments*** Our goal is to address these challenges in program comprehension and modularity by providing concept-awareness to programming environments.

In our proposed model, each occurrence of a name would carry one or more context-dependent *labels* that represent which abstract concept or analogy this particular term was derived from - imagine the variable named *vertex* to carry a *graph label* when used as the mathematical concept, or a *geometry label* when it is used in the context of a 3D renderer. These labels can be accumulated and propagated upwards in the hierarchical structure of the system, making it easier to recognize at top level how concepts are distributed.

Retrieving all names that carry the same label allows to assess the analogy or metaphor that has been chosen for this particular concept. The concentration of labels within a module provides a metric of coherence, and architectural drift can be observed by a flux of labels between modules during program evolution. At run-time, these labels can be propagated to live objects and call frames, giving semantic meaning to dynamically emerging parts of a program.

The standard tools in a programming environment, e.g., an editor or graphical debugger, can view these labels or their summarized proportions on demand and provide additional *navigation by concept* to locate or recommend related artifacts. Run-time propagation and analysis of concepts allows to better retrieve relevant data and call frames during debugging sessions. Integration with version control allows to track concepts, quantify how concepts tend to misalign with modules over time, and make refactoring decisions to counter this trend.

As a first step, we acquire the labels using a statistical approach based on name co-occurrence. Later, this approach can be extended to include programmer feedback.

***Choice of environment*** In the scope of this work, we will discuss primarily object-oriented languages with class and package/module support, since they are widely used and serve as example for hierarchical decomposition. For experiments, we chose the Smalltalk environment *Squeak* [8], since it allows quick prototyping of tools and immediate feedback during exploration. In Smalltalk, all program parts are readily available as live objects, which greatly supports building tools that operate on the program, and allows us to extend this live object graph with support for first-class concepts.

## 2 Finding Concepts

In our approach, lexical tokens should carry semantic labels. Each label is uniquely defined by the position of the token it is attached to and the concept it denotes. Since we have no

[1] In the SICP sense of data and procedural abstraction[1].

```
City » planRouteTo: destination
  ^ (self vertex shortestPathTo:
     destination vertex) asRoute
```

**Figure 1.** A piece of source code with concept labels. Different colors indicate different concepts.

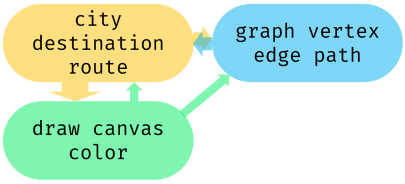


**Figure 2.** A concept overview with the most salient names. Concepts can refer to other concepts when they use them for their implementation. The strength of usage is denoted by arrow size.

clear definition what *concept* means, apart from the cognitive image induced by looking at its vocabulary and structure, we will refer to the label's concept as *color*. Instead of imagining the two variables *vertex* and *edge* having a label saying *graph* on it, just imagine they have a blue tag on them, like in Figure 1. The programmer might get the idea that *blue means graph* by looking at the parts of the program that are colored blue.

Concepts itself should not be standalone bubbles of vocabulary, but need to be linked to other concepts reflecting the way different modules associated with these concepts interact (see Figure 2). Concepts should be as independent from the current modularization of the program as possible, since they will be used to assess the modularity of the program from a more independent viewpoint.

Given the large number of labels, we propose to follow a semi-supervised learning approach: Most of the label assignments are found using an automated technique while giving programmers the chance to manually intervene and correct proposed labels or concepts. To bootstrap an initial label assignment, we need to investigate clues about which tokens should carry which labels.

### 2.1 Detecting Names

Our first goal should be to find the names that programmers have chosen. Typical places include the names of *variables*, *arguments*, *classes*, *methods*, *fields*, and *types*. Moreover, short *strings* or, in case the language has syntax for it, *symbols*, carry meaning. They can be found as dictionary keys, in metaprogramming, error messages, etc. *Operators* can be regarded as names when defining and overloading operators is common in that language. In Smalltalk and related languages there are *class categories* and *method categories* (*protocols*) for refined grouping of classes and methods, while other languages may have *package* names.

**Table 1.** Co-occurrence scores for a few pairs of names

| $t_1$ | $t_2$ | $\hat{F}(t_1, t_2)$ |
|---|---|---|
| visitor | accept | 70.1 |
| bounds | origin | 13.7 |
| collect | select | 6.8 |
| collect | color | 1.5 |
| visitor | color | 0.0 |

Each of these lexical tokens can be constructed from multiple names, which justifies the decision to attach multiple labels to a single lexical token. Typical examples are camel-case and underscore identifiers like `isEmpty` or `open_file`, or Smalltalk's multi-part messages `at:put:`, where each part can be camel-cased again. Our analysis is not constrained to nouns, or even actual words, but builds an exhaustive vocabulary of *potential* names used in the source code.

Most language keywords do not convey a programmer-chosen analogy and can be removed from the analysis. When confronted with pseudo-variables, such as *this* or *self*, one can either ignore them or attach context-dependent labels. In Smalltalk, we suggest to propagate labels from the class to *self*.

### 2.2 The Distributional Hypothesis

In semantics, the *distributional hypothesis* states that two lexical tokens that share meaning also share the same statistical distribution of where they occur. That means, they are more likely co-occurring (and co-missing) than two unrelated tokens.

If the lexical tokens in a program are derived from analogies, the distributional hypothesis should hold for program source code as well. For example, when a structure resembles a graph and programmers decide to name the parts *vertex* and *edge*, they will be more frequently occurring together than, e.g., *vertex* and *compile*.

We can demonstrate this phenomenon using identifiers found throughout the Squeak/Smalltalk image. For two identifiers $t_1$ and $t_2$ we compute the proportion of methods[2] they occur in, $f(t_1)$ and $f(t_2)$. We also count in which proportion of methods both terms are present, $f(t_1, t_2)$. If the occurrence of both terms is statistically independent, then $f(t_1, t_2) \approx f(t_1)f(t_2)$ would hold, otherwise they would be co-occurring in a higher proportion. Hence, we can use the ratio $\hat{F}(t_1, t_2) = \frac{f(t_1, t_2)}{f(t_1)f(t_2)}$ to measure semantic relatedness of the following terms. Values close to 1 indicate random co-occurrences, larger values indicate shared concepts, smaller values suggest mutual exclusion.

[2] number of methods containing the term divided by the total number, here 74589, for Squeak 5.1 excluding empty methods and accessors.

The values computed in Table 1 indicate that *visitor* and *color* never occur together, *collect* and *color* at least share some usage, but not significantly more than if they were independent. The other pairs show increasing affinity, with *visitor* and *accept* being one of the most correlated name pairs in Squeak/Smalltalk. *collect* and *select* are part of the Smalltalk collection protocol.

Concerning our label-name model, we should ensure that having a label of the same color corresponds to a high co-occurrence score, using either the $\hat{F}$ metric or a similar measure, e.g., pointwise mutual information[3]. We will discuss formalizations of that optimization problem later when we discuss algorithms to solve it.

***Determining the number of concepts*** Our objective can be formalized as an optimization problem[4] that will generate color-name-assignments for any number $n$ of colors specified in advance. For smaller values, e.g., $n = 3$, there is not much information for programmers except a very rough decomposition. For larger numbers, such as $n = 300$, programmers can become lost and we would need a lot of data (gigabytes of code) to have sufficient support for distinguishing hundreds of different concepts.

We propose to address this issue by allowing experts to split, join, and create new concepts when the original partitioning seems unfit.

A second option would be a hierarchical approach, in which we establish a *tree of colors*, where each internal node represents a higher-level concept and its children smaller sub-concepts. This allows to increase the number of colors without sacrificing clarity, because a node that is too fine grained for the given task can be collapsed, its distinct colors are then represented as a single one again. In this scenario, the labels attached to each name would describe a *path* from the root of the tree to a leaf.

***Granularity of co-occurrence neighborhood*** Through defining co-occurrence as being part of the same *method*, we achieve a simple, yet effective baseline to automatically distribute labels. However, methods are a modularity concept themselves, which makes them susceptible to design defects and architectural drift. The optimization criterion still works reasonably well on a large number of methods, since incoherence within individual methods appears to cancel out statistically.

However, we should regard the neighborhood in which we define co-occurrence as interchangeable and explore further definitions of that relation, for example:

- Names in the same class. This works for large numbers of classes, but we lose information due to the coarser granularity. In the example above, $\hat{F}$(visitor, accept) would shrink to 18.7 and $\hat{F}$(collect, select) to 4.9, making their relative co-occurrence less salient.
- Names not more than $n$ edges apart in the abstract syntax tree (AST) of the method. This gives high locality and is stable against a lot of perturbations (e.g., the extract method refactoring mildly affects the neighborhood of most names). We have not implemented this scenario yet.

A graphical comparison between method-based and AST-based neighborhoods can be seen in Figure 3.

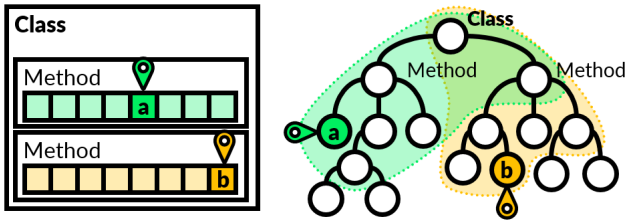


**Figure 3.** Illustration of two neighborhood definitions, green indicates **a**'s neighborhood, orange **b**'s neighborhood. On the left, everything inside the same method is defined as *co-occurring* with the reference names **a** and **b**; if both are in different methods, they have disjoint neighborhoods. On the right, co-occurrence is defined by maximum distance within the abstract syntax tree which may extend into method, class, and package nodes; neighborhoods may overlap.

### 2.3 Summarizing Abstractions and Dependencies

A method or other unit of modularity is typically not self-contained, i.e. it makes use of other units and is being used by units building on top of it. This naturally limits the coherence of names within a unit, since it tries to express itself consistent with one analogy while internally depending on vocabulary exposed by other concept's interfaces, e.g., a *stack* using a *list* internally.

Such abstraction barriers still manifests as highly correlated names using undirected co-occurrence metrics like $\hat{F}$, and therefore force different sides of the abstraction barrier into being labeled with the same color, although both might be different concepts.

To incorporate this phenomenon, we relate names that are provided publicly (e.g the method name *push* of a stack) to names that are prevalent in the respective implementation (e.g., a call to *add* in the underlying list).

A simple way is to adapt occurrence and co-occurrence measures $f$ and $\hat{F}$: First, we define $f_a(t_1)$ as the proportion of methods carrying $t_1$ in their signature including argument names, and $f_i(t_2)$ as proportion of methods that use $t_2$ in their implementation. Analogously, we define $f_{ai}(t_1, t_2)$ as the proportion of methods matching both criteria at the same time, and the directed co-occurrence $\vec{F}(t_1, t_2) = \frac{f_{ai}(t_1,t_2)}{f_a(t_1)f_i(t_2)}$. This

[3] $\text{pmi}(t_1, t_2) = log_2 \hat{F}(t_1, t_2)$, resulting in 6.13, 3.78, 2.77, 0.64 and $-\infty$ bits for the above example.

[4] Variants of what can be optimized are discussed in subsection 2.4.

would be approximately 1 if both terms were used independently and increase with higher correlation.

In our Squeak/Smalltalk image, we compute $\hat{F}$ and two directions of $\vec{F}$ to collect evidence for the following hypotheses:

- The drawing metaphor (*draw*, *canvas*, *color*, …) often makes use of the geometry concept (*bounds*, *corner*, *origin*, …):
  - $\hat{F}(\text{draw}, \text{bounds}) = 15.46$
  - $\vec{F}(\text{draw}, \text{bounds}) = 17.08$
  - $\vec{F}(\text{bounds}, \text{draw}) = 7.43$
- The parser concept often makes use of streams:
  - $\hat{F}(\text{parse}, \text{next}) = 2.18$
  - $\vec{F}(\text{parse}, \text{next}) = 3.29$
  - $\vec{F}(\text{next}, \text{parse}) = 1.43$

In this example, the directed measure is higher than the undirected version and largely exceeds the inversely directed measure, giving strong evidence for the orientation of the abstraction barrier.

We can extend the initial optimization problem that names with high co-occurrence should receive labels of the same color by allowing labels of different color if the directed co-occurrence is significantly higher in one direction. Additionally, we should keep track of the transition choices, i.e. how often which *abstraction color* has been linked to which *implementation color* and make sure each of these concept transitions have the maximum expected *directed* ($\vec{F}$) co-occurrence in addition to each individual color having maximum expected *undirected* ($\hat{F}$) co-occurrence between pairs.

Again, the $\vec{F}$ measure can be seen as a placeholder in this framework, and might as well be replaced by probabilistic or information-theoretic measures.

### 2.4 Algorithms

***Topic Models*** The distributional hypothesis is extensively exploited in natural language to extract *topics* of co-occurring words. Similar to our approach that starts with defining co-occurrence within methods, *topic models* typically operate on natural-language documents, often represented as an unordered histogram or multi-set of words, called the *bag of words* model. The colors of our labels correspond to topics, methods (or any co-occurrence neighborhood) can serve as documents.

Topic models are particularly useful because of their cold-start capability. Without any prior knowledge on real-world concepts, they are capable of guessing concepts from statistical co-occurrence only.

***Adapting LDA*** A prominent topic model is Latent Dirichlet Allocation (LDA)[6]. Related work [9][3][5][11] has frequently applied LDA to source code in order to interpret the discovered topics as concepts, so we are confident that LDA gives us a good starting point for computing an initial label assignment.

LDA describes a random process that serves as a model how documents are generated. This probabilistic model can be turned around (in the sense of Bayes' Rule) and instead of generating documents we can estimate the models parameters that most likely generated an observed set of existing documents.

Metaphorically speaking, the model equips us with a number $T$ of differently colored, unfair *word dice*, all of them having all the available words on their sides but with different chances of rolling them. They represent *topics* (e.g., the blue die would roll the words *push* and *pop* more frequently than *draw* and *canvas*, as opposed to the orange die). A document is the result of a set of subsequent dice rolls. Additionally, each document itself has a unique $T$-sided die which has each die color on its sides. It is rolled first to decide which of the $T$ dice should roll the next word.

The task is to find out the weights of the dice, i.e. which die rolls which word how often, and how the document-specific multi-colored $T$-sided dice are weighted. The weight assignment should maximize the probability of the observed documents.

Our labeled-name model fits well in the LDA framework, since the colors of the labels correspond exactly to the color of the die in the above analogy that generated that name. LDA does not maximize the mutual $\hat{F}$ measure, but a joint probability instead. A standard algorithm, which we can also use for our purpose, is known as *Gibbs sampling* [7], which we do not describe in detail in this paper. It is a randomized algorithm that generates a series of increasingly likely potential topic (or color) assignments to each word or name. By observing these assignments over a large number of iterations, the most frequent one can be chosen as our concept label, or even multiple ones if they are almost tied.

When we compute the average co-occurrence score ($\hat{F}$) of the ten most likely words in each topic, i.e., $\hat{F}_{10} = \frac{1}{45}\sum_{i=1}^{10}\sum_{j=i+1}^{10}\hat{F}(t_i, t_j)$ for $t$ being the topic's words sorted descending by frequency, we obtain values between 1.7 and 7.2 across six topics, averaging to 3.4. These values can be compared to other topic models we applied to our problem.

***Adapting BTM*** The Bi-term Topic Model (BTM)[16] does, compared to LDA, not model document-specific dice to select the next topic inside a document; it is much simpler. It models pairs of words as the result of rolling a topic-specific die *twice*. Inversely, we can take any two co-occurring names and estimate the most likely die that would have shown both when rolled twice, and propagate the label to both names. This causes names with many neighbors (e.g., all the other names in a large method) to accumulate a large number of labels, of which we chose a single one using majority vote.

*Gibbs sampling* is as effective in this model as it is with LDA, but while LDA stores a single die color per name occurrence, BTM stores a color for each possible pair, effectively squaring memory and run-time effort per method. However, compared to LDA, this model generates average co-occurrence scores of about $\hat{F}_{10} \approx 5.1$, outperforming LDA when it comes to intra-concept coherence. However, quantitative assessments of how well a topic model fits our concepts are to be taken with a grain of salt, since only a user study can tell us if the labels are meaningfully distributed.

What makes this model more attractive than LDA is that it does not take bags of words as input, but an extensionally defined co-occurrence relation, which does not even require transitivity. That means, we can fit this model on a syntax tree (or arbitrary graph) and define co-occurrence as being reachable in certain number of steps. This way, we can also include class-level co-occurrence, e.g., two method names in the same class can co-occur rather than only names within a method.

***Integrating Abstractions*** A shortcoming of both topic models is that they are unable to explain the abstractions and would mix abstraction with implementation details. Speaking in terms of the dice analogy, we need a second die, the *implementor's die* alongside each single-colored topic die, that rolls colors instead of words. This color indicates which die to roll to obtain an implementation-specific name. For example, given a blue die with a focus on the names *push* and *pop* and an orange die with a preference for *list* and *add*, the fact that stacks use lists internally would be encoded as the blue implementor's die preferably rolling orange.

Mathematically speaking, when we only consider the colored sides, we build a Markov transition matrix between concepts. The dice rolls that generate words are the output function of a hidden Markov model (HMM). When following a specific control flow, the concept transitions occurring on its way should reflect a Markov chain represented by this transition matrix, and the observed method names are the output of the underlying HMM. While this interpretation gives rise to a run-time algorithm to fit the model via estimating the HMM transition matrix from the dynamic call graph, we have not exploited this property yet.

## 3 Integrating Concepts into the Environment

Given a way of distributing differently colored labels to each name, and also knowing conceptual transitions and dependencies between label colors, we elaborate on our planned tool integration. The following section should be seen as future work.

### 3.1 Exploration

In accordance with Biggerstaff et al. [4], we suppose that understanding a program requires both relating concepts to parts of the program and relating concepts to one another. We will focus on one type of inter-concept relation, the *abstraction-implementation* relation capturing the recursive nature of defining a concept in terms of another concept, but anticipate that the *part-of* relation might be useful as well once we have arrived at a hierarchical concept model. Especially for explorative phases of program comprehension, we would like to address the following questions:

- Which concepts are there and which names have been chosen to communicate them?
- Given a concept,
  - Where is it located in the program?
  - Which concepts does it use for its implementation?
  - Which other concepts use this one for their implementation?
- Given a unit of modularity (e.g., class, file, method, …)
  - Which concepts is the unit concerned with at all?
  - Which sub-units (e.g. methods of a class) are concerned with which concept?

***Example*** We often encounter these questions in the context of reviewing students' group projects submitted at the end of a lecture, a use case in which it is crucial to quickly build a theory on how the software works and which analogies a group has chosen to solve the problem at hand. Since the modularity of submissions varies greatly between groups, and groups tend to invent new concepts we have never seen before, we cannot always rely on a class diagram as a first impression.

***Concept-augmented Class Diagram*** We propose to introduce a new tool to explore the program at concept-level and address the information needs stated above.

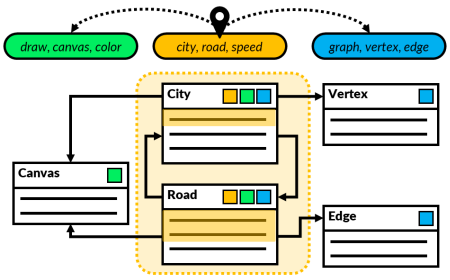


**Figure 4.** A class diagram with a selected concept. Concepts are represented as colored collection of relevant names, the corresponding parts of the program are highlighted using this color. Links to implementation-specific concepts are shown. Each module has colored indicators to show which additional concepts it is dealing with.

The design studies in Figure 4 and 5 show a class diagram with color-coded concepts. Selecting a concept from the

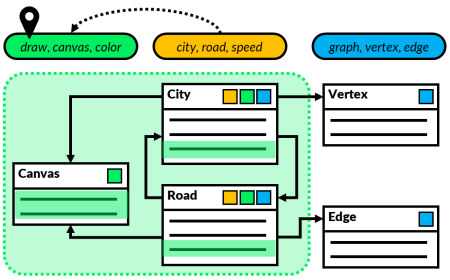


**Figure 5.** The same class diagram as in Figure 4 with a different selected concept.

top highlights methods which have a majority of names labeled with this concept, making clear where the concept is prevalent. It should be possible to collapse or hide classes not concerned with the selected concepts, since class diagrams tend to use up a lot of space for larger systems. Even if not selected, the other concepts are indicated by color markers alongside modules, they might as well show proportions rather than just a binary indicator, but with many concepts such miniature charts might get cluttered. A major challenge will be automated layouting, such that classes related by concept are arranged closer to each other. For a user-layouted class diagram, we can provide *linting*, e.g., assessing how well the spatial grouping of classes reflects concepts and help re-arrange a diagram for better readability.

### 3.1.1 Concept-Aware Browser and Editor


**Figure 6.** Schematic code browser and editor with color indicators for two active conceps and a warning when names from unrelated concepts have been introduced.

For less explorative tasks or tasks requiring programming, the standard code browser and editor can be augmented to support first-class concepts in multiple ways. Similar to the class view, we can use color-coding to indicate relatedness of certain modules to a given concept, illustrated in Figure 6. Concepts should be available anywhere, e.g., by selecting any identifier within the source code the editor should explain which concepts this identifier is labeled with (e.g., by showing a colored indicator near the identifier and some names associated with it). Interacting with any colored concept indicator, e.g., by hovering or clicking, should highlight every part of the program that belongs to the same concept.

Besides color-coding, the editor may warn programmers when they use names that are inconsistent with the context.

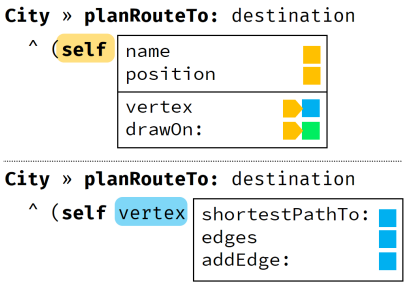


**Figure 7.** Code completion is scoped to a concept and additionally proposes completions from related concepts. When a concept transition is detected, code completion updates its concept context accordingly.

Concerning code completion, each typed name indicates what concept programmers are dealing with. Combined with the context (e.g., concepts in the currently active method and class), code completion can prefer those names that are either inside the concept, or in frequently used related concepts, like illustrated in Figure 7 (top). When a transition into another concept is detected, e.g., because a specific implementation detail uses a different concept, syntax completion can snap into that concept again, as in Figure 7 (bottom). This can be combined with traditional code completion ranking mechanisms, such as type inference and most-recently-used heuristics, but details on the exact mechanism need to be explored and tested yet.

## 3.2 Debugger

A common challenge with debugging complex systems in graphical debuggers is that the call stack does not reflect abstractions in the original program design. Especially highly modular architecture expands to a call stack that slices through a high number of abstractions which are likely unrelated to the problem at hand.

First-class concepts can help to structure a debugging session, since they can collapse call frames that belong to unrelated concepts, or at least mark the relevant ones.

Since modern debuggers have access to live objects in the running program, there is the opportunity to arrange or retrieve them by their concept.

## 3.3 Version Control

With modern version control systems, changes can be tracked in small increments and attributed to their author. When reviewing individual change sets, the difference in terms of concepts (e.g., how many labels of each concept have been

removed or added) may already give away what the code change was about. By aggregating the number of concept labels affected per author, we can compute what each author is an expert for.

### 3.4 Assessing Modularity

The distribution of concept labels in each module can be used to assess the module's coherence as well as the locality of the respective concept. Related work dealing with an LDA-based modularity assessment proposes to use entropy measures [9]. We can adapt this idea to the labeled names and define per-method entropy as:

$$H(m) = -\sum_{c} p(c|m) \log_2(p(c|m))$$

With $p(c|m)$ being the proportion of concept $c$ with regard to all labels in module $m$. The larger this number, the less coherent a module is with regard to naming. As a refinement we could eliminate implementation-specific concept labels from the analysis and only focus on the labels given to the method signatures. This would give us a measure for the entropy in a module's interface and not consider the fan-out caused by implementation details.

Using the same mechanism, we can quantify cross-cuttingness of a concept by measuring its entropy across all modules:

$$H(c) = -\sum_{m} p(m|c) \log_2(p(m|c))$$

With $p(m|c)$ given by Bayes' Rule as $p(m|c) = \frac{p(c|m)p(m)}{p(c)}$, and estimating $p(c)$ as the global proportion of concept $c$ in all labels, and $p(m)$ as the global proportion of labels used by module $m$ of all the labels available.

***Quantifying architectural drift*** In combination with version control, we could track the entropy measures across change sets and score each change according to whether it was an improvement (decreasing $H(m)$) or a sign of deterioration (increasing $H(m)$). This difference might become more apparent when versions with many changes in between are being compared.

Using visualizations similar to the ones discussed above, we can visualize the drift, as in Figure 8. Browser and editor should be aware of those metrics and track them for programmers to stay informed.

### 3.5 Including Programmers' Feedback

All of our designs rely on the fact that concepts detected using topic-modeling and similar techniques group names in a way that they are recognizable as coherent concept by humans. Since they only make use of artifacts created by programmers, they can only better arrange information that is already there, but not recover parts of the programmers' mental model that were never encoded in the program.

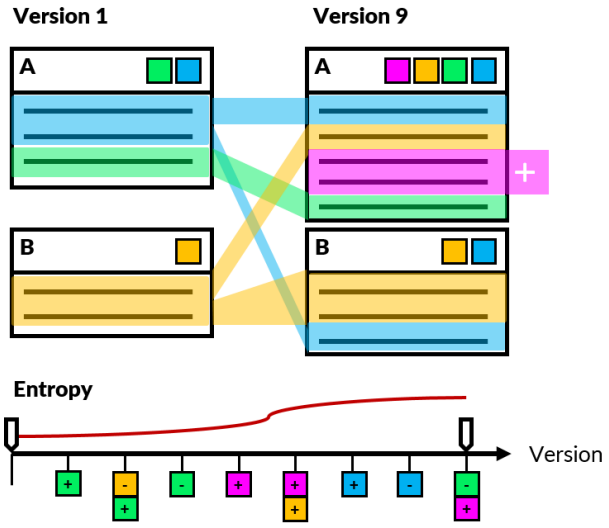


**Figure 8.** A visualization of architectural drift for two classes. Two versions, 8 change sets apart, are inspected and the flow of concept labels is summarized. The timeline in the lower half plots entropy over time and which concepts have been modified in each version.

At any point where concept labels appear, experts should be able to correct the current label. This may trigger a cascade of automated updates trying to optimize the remaining, automatically chosen labels to reflect the change made by the expert. Other operations on concepts that we want to support include:

- Merge two concepts. This is simple, as each label of the second concept can just flip its color to the first concept.
- Automatically split a concept. This requires computing an optimal split, where each sub-concept retains the highest possible coherence while making sure that the names with the lowest semantic similarity are put into different sub-concepts.
- Manually split a concept. Experts drag and drop names into a new concept (similar to [11]). The environment may recommend additional names to move to the new concept, e.g., those that have high co-occurrence with the ones already moved.
- Shuffle a set of concepts. This might be the last resort when automated concept allocation fails. The good concepts remain fixed and the (randomized) algorithm is re-run on the remaining concepts, probably finding a different solution[5].

[5] Gibbs samplers are randomized algorithms. Especially when a large range of near-optimal, but completely different solutions exist, re-running a Gibbs sampler with a different starting configuration can give a better solution.

### 3.6 First-class Concepts as Reflection Facility

All of the above tools motivate a common interface to access concepts programatically, which is yet to be designed. As it reflects on a program itself, we propose to integrate it with existing meta-programming facilities.

For example, where the name of a variable or argument is accessible through reflection, we would add accessors for reading and modifying the concepts attached to that identifier. Especially for the explorative part, meta-objects like *class objects* or *method objects* would need a method that reports on the proportion each concept occupies within the respective meta-object's code. Concept objects itself should behave like nodes in an edge-weighted directed graph, with each outgoing edge pointing to an implementation-specific concept and the edge-weight indicating the relevance of the target concept for the source concept's implementation. In later stages, the model might evolve to include *part-of* relationships or other inter-concept relations. Each concept should be able to return the names it is most frequently attached to.

If code itself is present as abstract syntax tree, the leaves of the AST could be refined to store not only *names*, but also concepts associated with lexical tokens present in such a name, e.g. the identifier "shortestPathTo" would cause the respective AST node to have children for "shortest", "path", and "to", with at least "path" having the *graph* concept attributed to its node. Upper levels of the AST could then recursively aggregate the concept labels to obtain a concept distribution for each inner AST node up to the full unit of modularity represented by a single AST.

An interesting challenge manifests itself when concept-aware reflection facilities or client tools are used by multiple, collaborating developers. In a fully automated setting, the assignment algorithm may be tuned to compute the same results for each identical working copy of a shared repository. When decentralized expert feedback is taken into account, we need a way to synchronize concept assignments, for example, by serializing them into code comments or files managed by version control. Alternatively, a repository of concepts and their lexical features can be maintained independently from version control, feedback would be submitted to this repository, and automated analyses on different code bases can re-use this "crowd-sourced" knowledge.

Providing a meta-programming interface to interact with concepts through code and vice versa would not only help implementing the above tools, but also encourage future tool builders and programmers interested in code analysis to make use of concepts, and potentially become a source of collecting valuable expert feedback on the meaning of identifier names.

## 4 Related Work

In the following section, we shortly discuss earlier examples dealing with "concepts" at tooling level, as well as related statistical models which have potential applications to automated concept inference.

***DESIRE and DM-TAO*** An early instance of program-assisted concept assignment was the *DESIRE* system [4]. It was founded on a model that distinguishes programming-oriented concepts (formal, unambiguous, data-manipulating) from human-oriented concepts (informal, possibly ambiguous, domain-based). The authors identified the mapping between them as a crucial part in program comprehension similar to Naur and highlighted that natural language tokens and proximity of statements (including grouping, e.g., by linebreaks) are important to automatically discover these links. Particularly interesting is their first approach to an *intelligent agent* called DM-TAO that can both locate concepts in code and explain code in terms of domain-model concepts, thereby serving two of our main use cases. It relied on a semantic graph that, similar to a neural network, computes the likelihood of a domain-model concept given observable syntactical and term-like features, can be trained by user feedback and bootstrapped from an existing domain model.

***Topic models on code*** The discipline of topic modeling, originally from the domain of natural language processing, has been applied to software in several circumstances. Linstead et al. [9] successfully applied LDA to a large repository of Java source files to discover globally prevalent concepts. They also introduced entropy as a method to measure tangling and scattering of topics in the sense of aspect-oriented programming. Saeidi et al. [11] refined LDA and included expert feedback in the form of manually defined *must-link* and *cannot-link* constraints between names. Asuncion et al. [3] extend topic modeling to related artifacts surrounding the programming environment. This allows information retrieval and software traceability across large projects. Thomas et al. [14] propose to track topic evolution in source code, especially the points in time where new concepts were introduced or removed, by a specifically designed topic model. Binkley et al. [5] investigated LDA on source code in detail and provided relevant insights that helped parametrize our own LDA implementation as a first step towards automated concept assignment.

***Other statistical models*** Beyond topic modeling, the discipline of graph clustering and community detection researches similar structures, such as the mixed-membership stochastic block model [2], which can be understood as a topic model on graphs.

The problem of structuring concepts in a hierarchy has been addressed for natural language and images using a hierarchical Dirichlet process [13], but not yet applied to source code.

*Computational biology* is concerned with detecting subsystems (analogous to concepts) in reaction pathways to understand and classify lifeforms in terms of these subsystems. The models used to summarize reaction pathways into subsystems resemble modern topic and graph clustering models [12]. The fact that abstractions (implementation details, interface) in programming resemble reactions in chemical pathways (reagents, products) makes these models partially applicable to our domain.

## 5 Conclusion

In this work-in-progress paper, we have explored a framework for integrating first-class concepts into a programming environment at both the source code and tooling level, and have given a bottom-up plan how to realize such an environment.

At the bottom level, we decomposed source code into names and attached concept-specific labels to each name. Subsequently, we looked into co-occurring pairs of names to approach a formal model of which names should receive the same label, thereby defining the term *concept* implicitly as both the result of an optimization procedure as well as the metaphor recognized by programmers when dealing with names of the same concept. We have shown that topic modeling provides us with already well-researched tools to derive label assignments from, but more research on topic modeling in the context of programs is needed.

Based on this code-level model and a set of high-level information needs, we show tool designs that summarize potentially large numbers of labels at several levels of modularity. With regard to the feedback programmers receive during programming, we consider the co-evolution and alignment of names and concepts with the hierarchical decomposition chosen by the programmer as an important part to steer modularity and comprehensibility for other programmers or their future selves. To better co-design topic models with programming environments and test these working hypotheses, we are currently in the process of implementing adequate tools and must defer this evaluation to a later stage.

## Acknowledgments

Sincere thanks also go to all PX workshop participants, who provided valuable feedback by discussing this topic thoroughly. We gratefully acknowledge the financial support of HPI's Research School(www.hpi.uni-potsdam.de/research_school) and the Hasso Plattner Design Thinking Research Program.(https://hpi.de/en/dtrp)

## References

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs - 2nd Edition.* MIT Press.

[2] Edoardo M Airoldi, David M. Blei, Stephen E. Fienberg, and Eric P. Xing. 2009. Mixed Membership Stochastic Blockmodels. In *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou (Eds.). Curran Associates, Inc., 33–40.

[3] Hazeline U. Asuncion, Arthur U. Asuncion, and Richard N. Taylor. 2010. Software Traceability with Topic Modeling. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, Cape Town, South Africa, 95–104.

[4] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. 1993. The Concept Assignment Problem in Program Understanding. In *Proceedings of 1993 15th International Conference on Software Engineering*. 482–498.

[5] David Binkley, Daniel Heinz, Dawn Lawrie, and Justin Overfelt. 2014. Understanding LDA in Source Code Analysis. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC 2014)*. ACM, Hyderabad, India, 26–36.

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

[7] Tom Griffiths. 2011. Gibbs Sampling in the Generative Model of Latent Dirichlet Allocation. (2011).

[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, 318–326.

[9] 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.

[10] Peter Naur. 1985. Programming as Theory Building. *Microprocessing and Microprogramming* 15, 5 (May 1985), 253–261.

[11] Amir M. Saeidi, Jurriaan Hage, Ravi Khadka, and Slinger Jansen. 2015. ITMViz: Interactive Topic Modeling for Source Code Analysis. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC '15)*. IEEE Press, Piscataway, NJ, USA, 295–298.

[12] Mahdi Shafiei, Katherine A. Dunn, Hugh Chipman, Hong Gu, and Joseph P. Bielawski. 2014. BiomeNet: A Bayesian Model for Inference of Metabolic Divergence among Microbial Communities. 10, 11 (2014), e1003918.

[13] Yee Whye Teh, Michael I. Jordan, Matthew J. Beal, and David M. Blei. 2004. Hierarchical Dirichlet Processes. *J. Amer. Statist. Assoc.* 101 (2004).

[14] Stephen W. Thomas, Bram Adams, Ahmed E. Hassan, and Dorothea Blostein. 2011. Modeling the Evolution of Topics in Source Code Histories. In *Proceedings of the 8th Working Conference on Mining Software Repositories* (2011) *(MSR '11)*. ACM, 173–182.

[15] Alvaro Videla. 2017. Metaphors We Compute By. *Commun. ACM* 60, 10 (2017), 42–45.

[16] Xiaohui Yan, Jiafeng Guo, Yanyan Lan, and Xueqi Cheng. 2013. A Biterm Topic Model for Short Texts. In *Proceedings of the 22Nd International Conference on World Wide Web (WWW '13)*. ACM, 1445–1456.