First-Class Concepts: Reified Architectural Knowledge Beyond Dominant Decompositions

Toni Mattis, Tom Beckmann, Patrick Rein, and Robert Hirschfeld
Software Architecture Group, Hasso Plattner Institute, University of Potsdam, Germany

ABSTRACT Ideally, programs are partitioned into independently maintainable and understandable modules. As a system grows, its architecture gradually loses the capability to accommodate new concepts in a modular way. While refactoring is expensive and not always possible, and the programming language might lack dedicated *primary* language constructs to express certain cross-cutting concerns, programmers are still able to explain and delineate convoluted concepts through *secondary* means: code comments, use of whitespace and arrangement of code, documentation, or communicating tacit knowledge.

Secondary constructs are easy to change and provide high flexibility in communicating cross-cutting concerns and other concepts among programmers. However, such secondary constructs usually have no reified representation that can be explored and manipulated as first-class entities through the programming environment.

In this exploratory work, we discuss novel ways to express a wide range of *concepts*, including cross-cutting concerns, patterns, and lifecycle artifacts independently of the dominant decomposition imposed by an existing architecture. We propose the representation of concepts as first-class objects inside the programming environment that retain the capability to change as easily as code comments. We explore new tools that allow programmers to view, navigate, and change programs based on conceptual perspectives. In a small case study, we demonstrate how such views can be created and how the programming experience changes from draining programmers' attention by stretching it across multiple modules toward focusing it on cohesively presented concepts. Our designs are geared toward facilitating multiple *secondary* perspectives on a system to co-exist in symbiosis with the original architecture, hence making it easier to explore, understand, and explain complex contexts and narratives that are hard or impossible to express using primary modularity constructs.

KEYWORDS software engineering, modularity, exploratory programming, program comprehension, remodularization, architecture recovery

1. Introduction

Expressive programming languages offer constructs to partition systems into modules. In popular languages, these constructs include functions, methods, classes, modules, packages, namespaces, etc. We will generically refer to them as modularity constructs and their instances (e.g. a concrete class) as meta-objects.[1]

Most constructs are syntactic, some are given by the environment. For example, the file system serves as modularity construct in Python by providing module and package boundaries, while Smalltalk meta-objects exist at run-time and do not rely on syntax to delimit classes and methods.

Roles of modules Modules serve different purposes. To support program comprehension, they provide means of abstraction and facilitate chunking – a cognitive process that makes it easier to reason about large domains by grouping related information.

JOT reference format:
Toni Mattis, Tom Beckmann, Patrick Rein, and Robert Hirschfeld.
First-Class Concepts: Reified Architectural Knowledge Beyond Dominant Decompositions. Journal of Object Technology. Vol. 21, No. 2, 2022.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2022.21.2.a6

[1] In light of recent developments in projectional editing, we extend the definition of meta-objects to cover all reified executable elements of a system, including expressions, lexical tokens, and even sub-tokens.

To help with maintenance, modules encapsulate responsibilities and design decisions, such that revisiting individual decisions and changing a single responsibility rarely cascades into cross-module changes. Testability is improved by the capability to exercise and verify parts in isolation. Compilation units facilitate incremental and parallel builds. Deployment modules can help install and run different parts of a system on different host systems (e.g., services) or share components between multiple systems (e.g., libraries).

Ideally, each module corresponds to a single *concept* from the real-world domain or from its technical or architectural implementation. However, not all concepts relevant to understand a system can be demarcated by modules, which leads to impediments to program comprehension and maintenance later in the software evolution.

Limits of the dominant decomposition As of now, most modularity constructs enforce a dominant decomposition of the system because they are persisted in syntax or as a particular arrangement of files, directories, or run-time objects. Cross-cutting concerns (e.g. logging, authorization, handling I/O errors, etc.) cause the scattering of some concepts across multiple modules and, consequentially, multiple concepts get entangled within a single module (Mens 2001; Tarr et al. 1999). For example, a user profile of an application can be represented by a `User` class, however, several distant places in the application might check whether the currently authenticated user is allowed to perform an action or see some data. Hence, no single module implements the *authorization* concept. Any programmer trying to understand and change authorization needs to attend to multiple distant code locations, causing both mental strain and increasing the chance for errors.

A similar case can be made for design patterns (Mens et al. 2001), where different roles in the pattern are played by several objects, but there is rarely any modularity construct that coherently identifies and names the parts of a pattern. For example, the observer design pattern requires subjects to implement a subscription mechanism and observers to implement a notification protocol. While this decoupling is intentional, the *observer* concept itself has no modular representation and needs to be recognized by, e.g., naming conventions.

Advanced modularity constructs like aspects (Kiczales et al. 1997), layers (Hirschfeld et al. 2008), roles (Herrmann 2003), or traits (Schärli et al. 2003) alleviate some of these concerns but have not found widespread adoption. Even if they did, they would require significant re-engineering to show any benefits in legacy systems. For example, a cross-cutting concern like logging could be implemented as *aspect* that inserts the required method calls in all affected code locations. In this example, however, programmers would give up one dominant decomposition in favor of another, and now the readability of the refactored code might be impacted once programmers need to consider the logging aspect and base code without any logging statements simultaneously.

During *program comprehension* activities, programmers need to understand concepts not represented through language constructs by locating the relevant program parts, usually with little tool support other than search, following references, and keeping multiple editors open at the same time. These challenges are known as the concept assignment problem (Biggerstaff et al. 1993), have been approached using activities like software reconnaissance (Wilde & Scully 1995) and architecture recovery (Garcia et al. 2013; Lungu et al. 2014), and motivated the creation of numerous tools that attempt to recover conceptual knowledge beyond the dominant decomposition. Without a way to represent this knowledge within the program itself, such activities constitute one-shot efforts that have to be repeated when their results become outdated.

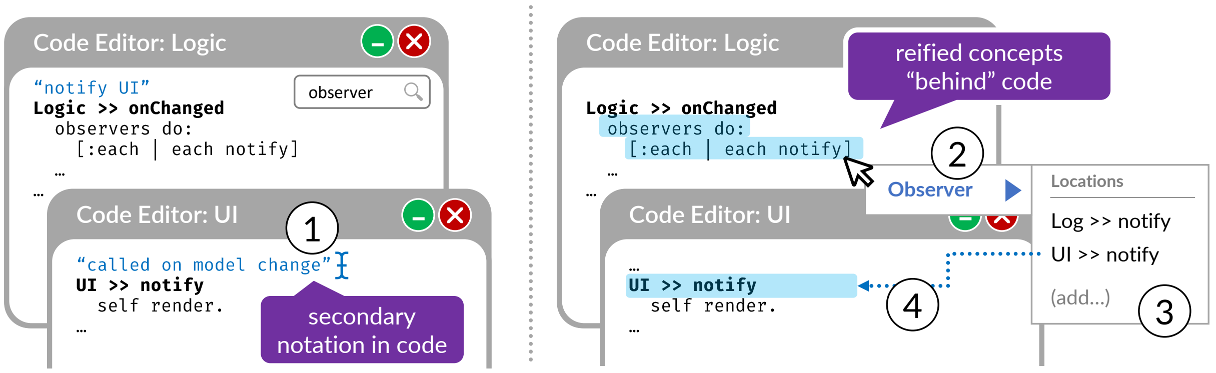
Primary and secondary modularity constructs The modularity constructs discussed above are behaviorally significant: they affect how the program behaves or represents its state. Changing module boundaries without impacting behavior, an important activity during *refactoring*, requires rewiring program parts to reproduce the old behavior within the new module structure. This can be expensive in legacy systems which have accumulated design decisions that are difficult to revert, and does not always lead to more maintainable decompositions in the presence of cross-cutting concerns.

In analogy to the dichotomy between primary and secondary notation from the *cognitive dimensions of notations* (Blackwell & Green 2003), we call them *primary modularity constructs* and distinguish them from *secondary modularity constructs* that only help perception and tooling, but never influence the program's run-time behavior.

Up to date, there are only a few examples of such secondary constructs in a narrower sense, for example categories in Smalltalk, comments for documentation generators (e.g. Javadoc) that form a small hypertext-like language to document code, or the `#region` pragma in C# that allows the editor to collapse code blocks regardless of their primary structure. In contrast to a refactoring, changing secondary constructs does not require cascading changes to the program and offers opportunities to document different architectural perspectives.

In a broader sense, if syntax is disregarded, several other mechanisms help with modularity: free-form comments, additional line breaks to induce a sense of grouping by introducing distance, or the order of methods in a class (e.g. putting core responsibilities first and cross-cutting concerns last). The granularity of these structuring elements is still dominated by primary module constructs, e.g., the order of methods within a class remains linear and relatively stable no matter how many empty lines separate logical groups of methods, and it is challenging to express a comment that refers to multiple elements in different code files without duplication or (hypertext) links. Notebooks (e.g. Jupyter) invert primary and secondary constructs, allowing the executable program parts to exist in between headings, text, and figures, but only in a linear document-like form.

Problem statement and research opportunity Our working hypothesis is that *primary* modularity constructs can be hard to change to re-modularize existing systems and the dominant decomposition subordinates potentially equally relevant concepts, which impedes program comprehension. In contrast, existing *secondary* constructs contain little machine-readable informa-



```
"notify UI"
Logic >> onChanged
  observers do:
    [:each | each notify]
  …
…
```

```
"called on model change"
UI >> notify
  self render.
  …
```

```
Logic >> onChanged
  observers do:
    [:each | each notify]
  …
…
```

```
  …
UI >> notify
  self render.
  …
```

Figure 1 An example showing a delocalized implementation of the observer design pattern used to connect a program's UI to its business logic. *Left:* In traditional programming environments, comments and naming conventions (1) can identify coherent parts of the pattern, while textual search can help connect them. *Right:* We propose to augment programs with a reified *concept* model. In such a design, program parts can be directly selected and tagged as "Observer" by programmers instead (2). Extent and participating program parts (3) of the concept are immediately discoverable and navigable (4), in this example through context menus. This work explores how the underlying model can be designed and automated, and what improvements to the programming experience are rendered possible by first-class concepts.

tion and mostly express concepts by means of natural language, but are easy to change and subject to fewer constraints.

If the preferred modularization cannot be achieved by refactoring, programmers can document non-modular concepts through secondary means, but forfeit tool support.

To bridge this gap, we explore the design space of *first-class* secondary modularity constructs that allow a wide range of concepts to be expressed in a program without re-organizing the underlying primary module composition.

To this end, we take the *concept assignment problem* literally and discuss how multiple representations of *first-class concepts*, ranging from simple tags to statistical distributions, can be directly assigned to parts of a program.

First-class concepts create new tooling capabilities that offer the chance to view, navigate, and change programs from multiple, equally valid perspectives rather than only the dominant decomposition available through conventional IDEs. Due to their reified nature, programming environments have opportunities to support and automate concept location and assignment using machine learning.

First-class concepts allow programmers to view and edit programs as if they were modularized in the way that best fits their task, without actually refactoring the underlying system.

We implement some parts of the design space in Squeak/Smalltalk and run a small case study to demonstrate practical implementation considerations as well as insights on the programming experience provided by a preliminary implementation fist-class concepts.

In summary, we envision first-class concepts and tools based on them to help program comprehension and maintenance in several ways: First, authors of the program do not need to decide whether to refactor or to document concepts informally, but can directly link participating code locations and name them. Second, readers of the source code can understand, navigate, and eventually change legacy programs as if they were expressed more modularly from the beginning. Third, tool developers have access to a reified representation of architectural knowledge alongside existing meta-programming facilities and can use it to enhance their own tools or provide recommendations by means of statistical methods or machine learning.

2. Representing Concepts

To allow programmers to work with first-class concepts as if they were actual units of modularity, our programming environment first needs a representation of concepts. In this section, we first define their general properties and the scope of which knowledge they are supposed to represent. We then design two different ways to model concepts and their relationship to code, a discrete approach and a probabilistic one, both with different trade-offs, but compatible with each other.

2.1. Requirements

We are proposing secondary modularity constructs to represent concepts in a way that is more formal than comments, but less constrained with respect to their association with the underlying code as language extensions. In particular, we aim at preserving the following useful properties from both worlds, which we subsequently combine to our first-class concept model:

Independence of the dominant decomposition A concept can be connected to multiple code artifacts, meta-objects, or expressions regardless of their location in the package tree, file, or enclosing meta-object. Refactorings should cause concepts to move together with the moved code. This property ensures the capability to represent cross-cutting concerns or concepts hidden in interacting program parts (e.g. design patterns). It is present to some degree in advanced modularity constructs like aspects, which gain independence from existing modules by declaratively specifying their scope using pointcut languages, or teams (Herrmann 2003), where roles within a coherently expressed concept can bind to distant elements in the underlying program. In terms of secondary constructs, Commentary (Hirschfeld et al. 2018) allows to comment code across module boundaries.

Non-exclusiveness In contrast to hierarchical decompositions or categories, any part of the program should be able to play a role in as many concepts as needed. This property allows technical and domain concepts to overlap, e.g., a method can both implement a domain concept and play a role in a design pattern. Many advanced primary constructs including aspects, layers, traits, and teams allow the same method, field, or class to play a role in different explicitly modelled concepts. Commentary (Hirschfeld et al. 2018) and the concerns in ConcernMapper (Robillard & Weigand-Warr 2005) are secondary (IDE-integrated) constructs with the ability to refer to possibly intersecting sets of program elements.

Non-executability A concept should not interfere with compilation and run-time behavior, much like a comment or non-code cells in notebooks. This property ensures flexibility when compared to primary language constructs that contain behavior or declarative properties themselves and eliminate the costs associated with refactoring.

Reification In contrast to comments, the content of a concept should allow tools to automatically process them. Persistence and serialization are beyond the scope of this paper, so for our discussion we assume they exist as meta-objects with an interface that can be used by tools[2]. This sets first-class concepts apart from informal secondary constructs based on comments and makes them programmatically accessible to programming environments and their tool developers the same way files, classes, methods, etc. already are.

2.2. Type of Origin

The need to designate a part of the program as belonging to a concept can originate internally or externally:

Internal concepts are used to engineer the program itself. In the scope of this work, we consider three major sub-types: (1.) Domain concepts, e.g. a controllable character in a game, or a rectangle in a graphical editor; (2.) Technical concepts, e.g. a file handle, database transaction, or thread; and (3.) Architectural concepts, e.g. a pattern, invariant, or relevant design decision.

External concepts originate from the program's lifecycle, e.g. issues, which programmers sometimes link to code by stating the issue number in a comment. Other elements of the software lifecycle include relations to tests, dependencies, or build artifacts.

[2] For the purpose of this paper, imagine them being stored in a Smalltalk image without ever taking a "serialized" form.

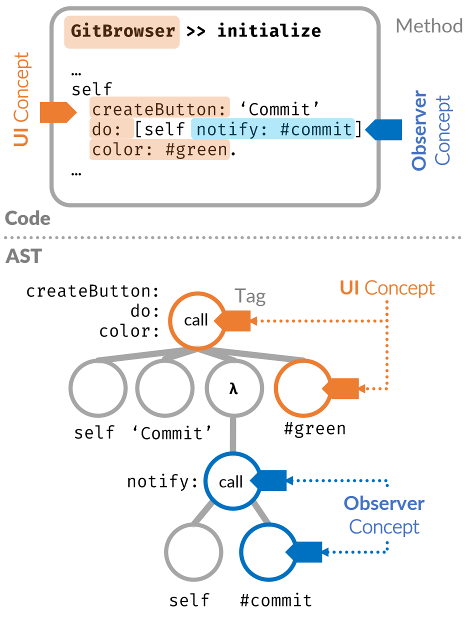


Figure 2 A concept tag can be attached to any meta-object, including expressions within one module's AST. In this example, a constructor method of a UI creates a button which has an associated closure responding to a click. The closure uses the notification mechanism known from the observer pattern, which can be tagged accordingly. Button-specific logic is tagged with the UI concept.

Most existing modularity constructs cover only internal concepts, leaving external concepts encoded in build configuration, issue trackers, and other systems outside the programming environment. In our design space, we try to cover both in a unifying way.

2.3. Tag-Like Concepts

In their simplest manifestation, a concept behaves like a tag (label) attached to one or more meta-objects or expressions as illustrated in Figure 2. Tagging code serves as mechanism to indicate the extent of a concept without resorting to informal means like comments.

This tag carries an identity and a (potentially ambiguous) descriptive name. For example, the parts of a design pattern can be tagged with the pattern name. This allows bidirectional identification of the concept: Displaying an "observer pattern" label near the code that invokes a notification mechanism signifies that this particular code is involved in said pattern, and the label allows programmers to navigate to all the observers that might subscribe to this notification. Meta-objects can carry multiple tags.

Smalltalk method categories already allow to group methods (e.g. a specific protocol implementing pattern interactions) into a named category, but they lack identity, making it difficult to distinguish different instances of the same pattern. Methods can only belong to one category, so overlaps between technical concepts (the pattern) and domain terms are inexpressible. The Smalltalk message send `self flag: #symbol` can be used to tag a method with a symbol. Tooling is able to display a flag icon next to such methods.

Closure property As tags become (secondary) meta-objects themselves, they can apply to other tags, thus creating hierarchies of tags. A situation in which a pattern exists in multiple instantiations throughout the program would be hard to navigate if the tags used for all instances looked the same. By creating different concepts for each instance, and tagging both of them with the more general pattern concept, the meta-object graph can be queried for both the instances of a pattern and the participants in each pattern. A generalization to ontologies is possible, but out of scope for this work.

Parametrized tags A different way to deal with multiple instances of a concept is parametrization, in which tags with the same name can be distinguished by additional parameters. For example, some open-source projects have a community process that discusses numbered "proposals" like *PEPs* (Python Enhancement Proposals) or *RFCs*. Code implementing such proposals can benefit from being tagged with the proposal number, thus making it part of the architecture while referencing the community consensus in a way that is explorable by contributors directly from their programming environment. Similar constructs work for issue or ticket numbers.

2.4. Distributional and Statistical Concepts

While a tag-like concept discretely refers to a set of meta-objects or expressions, we propose that concepts can also refer to either code locations or code features *with a degree of uncertainty* (see Figure 3). These types of concepts are not mutually exclusive, a concept should be able to both attach discretely to locations and still be able to carry statistical and generalizable data.

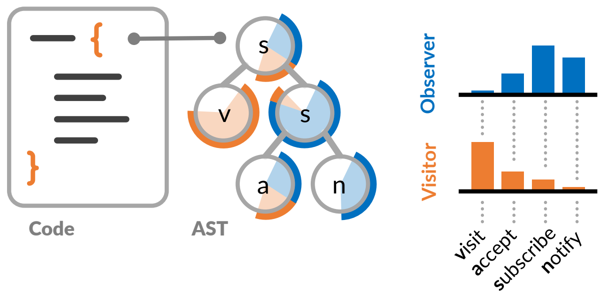


Figure 3 Two simple distributional concepts. In this case, identifiers are associated with two different design patterns, with the observer pattern having higher weight on "subscribe" and "notify" and the visitor pattern putting weight the "visit" and "accept" terminology. These natural language terms are still ambiguous but the combined presence of multiple correlated identifiers at once helps distinguish individual concepts.

Connection to natural language Most concepts show a distinct vocabulary in their choice of names. When tagging, e.g., the different methods involved in a large visitor pattern, it becomes apparent that the concept involves names like "accept" and "visit". This linguistic distribution can become part of the concept. This allows inference from new code, detection of inconsistent naming within a concept (e.g. using "logon" for a concept previously named "login") or ambiguous naming between concepts (e.g. using "client" to designate both a technical concept in a server-client setting and a domain concept in a customer relationship management system). Linguistic concepts can be inferred in both fully automated (Linstead et al. 2007) and programmer-supported ways (Saeidi et al. 2015) using topic modeling and have found their ways into programming environments already (Mattis 2017; Gethers et al. 2011).

Evolutionary concepts Besides linguistic features, concepts should ideally align with code evolution. Even if we are dealing with a cross-cutting concern and changes involve multiple primary meta-objects, the change should cover very few distinct concepts. If that is not the case, this could hint at either a missing concept and thus lead to a recommendation by the development environment, or misaligned changes, which could be used as feedback to motivate programmers to commit more fine-grained or more coherent changes in a version control system. If higher-resolution data is available, e.g., which code is read at the same time, we expect even better support in concept inference.

Concept expertise Some programmers tend to specialize in specific concepts. If concepts accumulated data about who reads, edits, and maintains program parts associated with them, programmers could seek out information in complex teams more effectively. Besides automatic data collection, programmers could be able to set themselves as contact persons for individual concepts. When teams change composition or workload over time, these expertise properties can be early warning signs for knowledge loss or bottlenecks.

Cross-system and global concepts The use of statistical properties opens up the opportunity to transfer data from one project to another, or train statistical models across a large set of projects. Global concepts identified across many projects can quickly be recommended to programmers working on another program and help bootstrapping the concept assignment process. Topic modeling has been shown (Linstead et al. 2007) to reveal relevant concepts, such as logging, parallelization, file path handling, database connection management, or event listener architectures, and the re-occurrence of their vocabulary in new projects indicates this concept is present again. Distributional concepts lend themselves to materialize such an a priori concept in the code base until programmers pin down individual constituents by confirming the suggested concept, thus effectively converting the probability into a discrete tag.

2.5. Lifecycle Concepts

To accommodate externally originating concepts, their reifications could be accessible by lifecycle tools, such as test runners, version control, continuous integration and delivery infrastructure, or deployment and monitoring tools. These tools would be able to consolidate data they generate at concept-level and express their output in terms of concepts (see Figure 4).

As an example, a particular artifact in which a meta-object ends up in the build cycle can be attached as concept. Especially

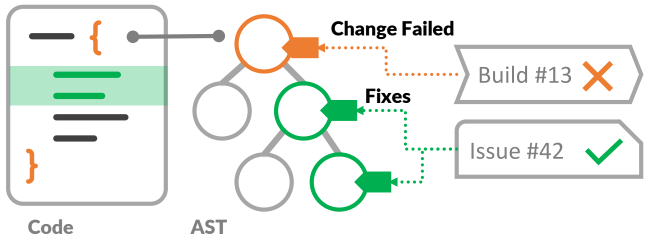


Figure 4 Lifecycle artifacts can be expressed using their own tag-like modularity constructs, in this example, a build failure and a fixed issue are (automatically) attached to the involved code. The concepts might be represented by a URL pointing to the issue tracker and CI infrastructure, where the programming environment can retrieve additional status information.

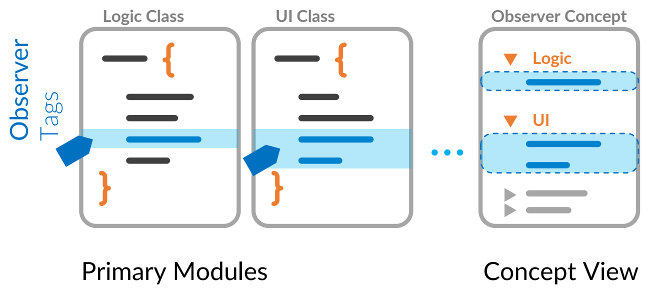


Figure 5 A concept expressed through tagged code can be rendered as single on-demand module with additional meta data explaining how the constituents relate to each other. In this example, the previously cross-cutting observer pattern is shown as on-demand module, helping programmers read and change all places that manage or handle subscription and notifications instead of sequentially or simultaneously open all source files.

utility code that might be included in multiple artifacts can be easily identified this way and both developers and operators of the resulting system now share a common understanding of the deployment concepts. Concept labels could also mark the build configuration or feature set each meta-object is included in, the relevance to certain subsystems, releases, customers, or whether a piece of code is part of a time-critical bottleneck or potentially deployed many times for scalability reasons.

3. Programs through the Lens of Concepts

First-class concepts enable multiple stakeholders to understand the system from their perspective, create opportunities to augment existing tools, and motivate novel tools to interact with them.

3.1. Automating Concept Allocation

Reified concepts allow different degrees of automation to help programmers designate and re-arrange concepts, as well as editing a program in such a way that it conforms to its concept structure.

Automation of secondary modularity constructs carries a lower risk to break the program than tools that manipulate primary modules (e.g. automated refactorings) and, as such, can more readily benefit from statistical methods. Although the degree of automation and support is a spectrum between manual and intervention-free automatic assignment, the following levels are of interest as they fit existing interactions in programming environments:

Manual basic tool support should allow manual concept assignment at meta-object level, either by tagging a selection of code with a concept (e.g. through selecting code with the cursor and selecting a concept from the context menu) or by adding a meta-object to a currently edited concept (e.g through a specific concept editor).

Reactive recommendations When opting to assign any meta-object to a concept or vice versa, the programming environment might use a recommender system to rank the most relevant concepts or meta-objects, or suggest to add a new one if no good match exists.

Proactive recommendations The programming environment might actively propose concepts while programmers work with code, programmers need to accept or reject such recommendations.

Partial inference of latent concepts Based on existing concepts and programmers' behavior in accepting and rejecting recommendations, they system might latently infer concepts for all remaining parts of the source code. Programmers must actively override the inferred concepts (which in turn can cause other inferred concepts to be retracted or re-computed to match the new constraint).

Full inference No intervention is needed. This requires unsupervised machine learning and could be used to automatically analyze a yet un-tagged code base for the first time, then transition into one of the more interactive automation modes. Existing code-bases can be used to pre-train such models and discover common concepts in a novel code base.

Machine learning component Most workflows involving a recommender system - reactive or proactive - benefit from a statistical model that has not only access to the programmers' concept input but also other information available from the program. Especially code locations that share similar vocabulary, structure, edit history, navigation history, authors, test coverage, or run-time data might be part of the full extent of a concept. Such a model would need to run in an on-line mode, i.e., consume incremental updates and output incremental updates. Figure 6 illustrates a system incorporating user editing, concept representation, and statistical model interaction.

3.2. Co-maintaining Concepts and Architecture

Once the system is covered by either manually placed concepts or automatically inferred (latent) concepts, the programming environment is able to assist with programming tasks themselves. Three examples are given below:

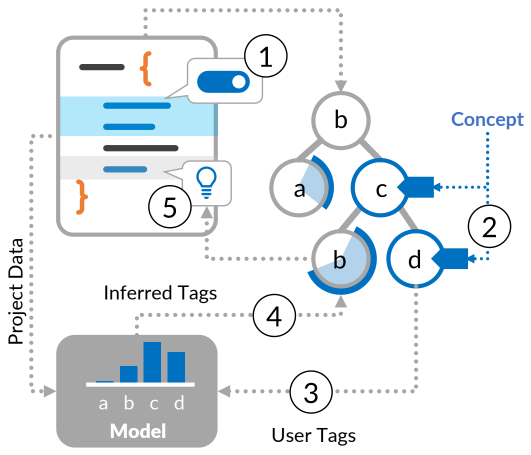


Figure 6 Semi-supervised concept allocation through machine learning: User assignment (1) updates the reified concept model (2). User-assigned tags are passed to a statistical model as constraints (3), while project data (e.g. call graph, navigation, and edit history) are available to infer likely participants in the same concept (4), which can be passed to users as recommendations (5).

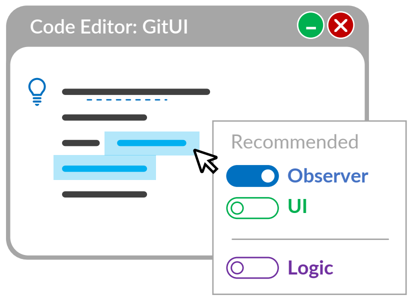


Figure 7 Sketch of a code editor using proactive recommendations by displaying a lightbulb next to code that signals an opportunity to attach a concept (top) and reactive recommendations (bottom) by displaying a ranked list of meaningful concepts in a context menu invoked on user-selected code.

Naming Distributional concepts can detect when an identifier is misplaced, suggest alternative names that match the concept, or suggest locations concerned with the concept this name should belong to.

Linting The mismatch between primary and secondary modularity constructs is an indicator for technical debt. While reified concepts reduce the "interest rate" of technical debt by enabling cross-module and cross-artifact units of modularity, a wide gap between both architectures helps to prioritizing refactorings.

Versioning Commits (and their commit messages) in version control systems benefit from including a coherent change, even if that change itself is scattered over multiple modules. The programming environment might help programmers join or split their change sets according to the underlying concepts.

Systems specifically designed for similar recommendation tasks already exist, e.g. MEntoR (Lozano et al. 2017) can detect missing characteristics in source code entities and Naturalize (Allamanis et al. 2014) can propose meaningful names following coding conventions. This opens up further opportunities to either feed additional information encoded in first-class concepts (e.g. concept names) into such models, or use the model output to, e.g., name a concept, thus providing implementations for the grey "Model" box in Figure 6.

3.3. Composable Perspectives

Concepts, by being independent of the dominant decomposition, provide the basis for re-arranging code in a way that coherently displays the elements of the concept. Instead of programmers having to open multiple editors, potentially even duplicating some editors to scroll to different positions, an individual concept could be "opened" and interacted with as if it were expressed as single unit in primary modularity constructs. The view would need to clarify "module breaks" and communicate the origin of each piece of code.

Concepts as presentation of scattered code As of now, meta-objects have order in their primary hierarchy (e.g. methods in a class) but not in concepts when following a tag-like approach. If we extend our notion of concepts to include a particular arrangement of primary meta-objects, e.g. an overall order, "opening" a single concept could resemble a Notebook, with individual cells displaying different slices from the underlying program as illustrated in Figure 5. If a cross-cutting concern is tagged, its perspective provides an overview over its usage throughout the system. When performing changes to a cross-cutting concern, programmers can then verify that the changes are implemented correctly in all occurrences without switching through various places in the code.

Editing in perspectives Editing in a conceptual perspective is more ambiguous, as any code removed from its primary construct is deleted, but removing code from its conceptual view must disambiguate between just removing the link between meta-object and concept, or deleting both. Adding code to a conceptual perspective can be done both by pulling in (non-tagged) meta-objects or creating a new meta-object within the view. For the latter, there needs to be a mechanism to re-attach

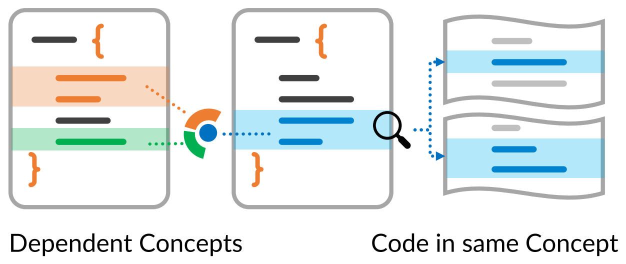


Figure 8 Concept-level navigation allows rapid navigation/preview of (scattered) code within the same concept or dependent concepts, thus forming a navigable graph beyond the dominant decomposition.

the newly added construct to the primary hierarchy (e.g. using a recommender that finds the conceptually most coherent place in the program tree), but this is not always necessary. If the new code is just an example invocation, it might as well exist within the concept only as part of its documentation.

Comments Composable perspectives should not be limited to functional code. Representing comments as meta-objects and including it in an (ordered, Notebook-style) view on a concept might help with documentation. Any comment would need a place in the primary hierarchy, thus being automatically updated when edited in either the primary hierarchy or the conceptual view. Similar ideas have been explored in the context of *Literate Programming* (Childs & Sametinger 1996), where techniques of object-oriented programming are applied to documentation to facilitate its reuse.

3.4. Spatial Arrangements

With no hierarchical constraints, concepts can generalize to two-dimensional arrangements of parts of a system. Composable perspectives could thus break from the linearity of code files or notebooks, and instead embed their structure in a canvas (similar to Code Bubbles (Bragdon et al. 2010)), potentially linking multiple concepts in a map. Zooming, panning, and searching facilitates exploration of larger systems and concept interactions.

Maintaining spatial arrangements Distributional concepts allow embedding algorithms from data science to cluster meta-objects based on their concepts, and concepts based on their vocabularies or co-evolution. Such an embedding does not need maintenance, but would update itself based on changes in the primary program structure and evolution.

Expressing concept interconnectedness Spatial views can use two-dimensional space to display adjacent concepts, e.g. code that belongs to another layer or a cross-cutting concern, in proximity. Instead of navigating to other concepts via an extra view or via hypertext, each meta-object could display indicators of other concepts it belongs to, and provide interactions to "open" this concept adjacent to the current concept as illustrated in Figure 8.

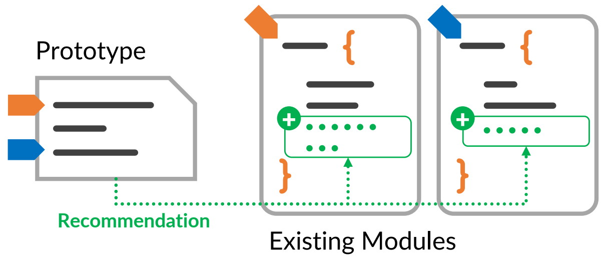


Figure 9 Concepts can help move exploratorily prototyped code into conceptually sound locations within the existing architecture.

3.5. Prototyping

When exploring solutions to a problem, a common approach is to rapidly prototype various ideas and gain a better understanding of the problem as one develops a solution. Often, the source code produced in this manner may not fit well into the primary hierarchy or could even have been written entirely in a REPL or (Smalltalk-style) workspace. Eventually, programmers may choose to discard it and rewrite the solution from scratch to fit the existing architecture.

If the existing architecture is enriched with first-class concepts, the concepts associated with the new code can be inferred statistically. This way, the programming environment can help integrate prototypical code more easily, or derive new architectural components.

Integration into pre-existing architecture We envision that the system should be able to propose an integration of new prototypical code into the existing architecture. Distributional concepts would provide the necessary heuristics: New code should be placed at locations already concerned with the newly prototyped concepts like shown in Figure 9.

Inferring the primary structure Automatic code formatters and pretty printers alleviate programmers from having to think about secondary notation of formatting, allowing them to exclusively enter syntactically valid notation for the structures they want to use and have the formatter provide an embedding in the project's source that is adequate. Similar to this, we envision that a first-class concept-driven system should allow programmers to enter code in arbitrary contexts as they think of it and alleviate the burden of organizing the code.

Rather than impeding creativity by forcing programmers to first locate an adequate place to position their code, such a system empowers them to start typing and introduce structure as soon as both the system and maybe also the programmers themselves understand the structures they are trying out better.

4. Case Study

We demonstrate a program change in a prototypical concept-aware programming system based on some of the mechanisms outlined above.

The purpose of this study is to answer two exploratory questions:

1. How can a simple tag-like concept model be implemented given the constraints of an existing programming environment?
2. Which changes in programming experience and architecture-related thinking can programmers observe while having first-class concepts at their disposal?

To this end, we first implement the concept model and two tools for managing the assignment of concepts to program parts and browsing individual concepts in Smalltalk. Subsequently, we report on our own experience using these tools to add a feature to a version control system for Smalltalk.

Tools presented in this section do not yet look like those proposed above because a full implementation and evaluation of the user interface is not yet part of this study. Nevertheless, the tools presented here allow a small set of interactions with the concept model through combining basic UI concepts already present in our environment.

4.1. Concept-aware Smalltalk

To explore concepts as first-class citizens of our programming system, we opt for the live programming environment (Rein et al. 2018) *Squeak/Smalltalk*. In such an environment, meta-objects are reified (e.g. as class objects and their method objects) and the *same* meta-objects that are edited by programmers exist at run-time. This is of great benefit to tool construction as there is no distinction between edit-time and run-time objects and representations and we do not need to construct a serialization format to represent concepts in source code.

This ubiquitous model of an *executable object graph* can be extended to carry conceptual data as `Tag` objects. Instead of modifying the meta-object hierarchy of the environment directly, we propose a less invasive implementation maintaining separate data structures holding an index mapping program parts to tags and vice versa which we call *concept store*.

Edit-oriented concept representation Since concepts should follow approximately the granularity of code comments, we require expression-level or token-level resolution.

While larger meta-objects like classes and methods persist during the program's life time, finer-grained meta-objects like AST nodes transiently exist for compilation or syntax highlighting purposes and are frequently re-generated when needed. Since program editing is still text-based on the lowest level, we chose to associate concepts either with *ranges of text* within a method or with an entire meta-object (method or class).

A text range requires certain maintenance when editing code, as its extent might move, expand, or contract between versions. There are two implementation strategies to track concept labels across code edits: A diff-based approach comparing the new code to the old version and computing the target location of the concept labels; and a keystroke-based approach instrumenting the editor itself and tracking the location of the tagged character range on a fine-grained level. We decided to re-use existing facilities in the text editor to track code locations through fine-grained edits: Code editors support text attributes that can style parts of the text or provide interactivity (e.g. click actions) on them. Their position is continuously managed by the editor itself the same way it manages the position of glyphs or words. This allows us to retrieve concept tags from the concept store upon loading a piece of source code, convert them into text attributes, and read them back once programmers save and compile the edited code. This makes for intuitive behavior, as newly inserted text before or after the concept tag does not get included, but typing new code inside a tagged range includes this code into the concept (analogously to typing into formatted text in rich text editors). Tagging code can be achieved via a context menu and just adds a new text attribute for the selected text.

Any downstream recommender that operates on AST nodes rather than text would only consider leaf nodes that intersect with the programmer's tagged ranges as belonging to the concept. Especially in longer identifiers or strings, the capability to assign *sub-strings* or *sub-identifiers* to concepts offers opportunities to explain the code base at a much finer granularity than code comments and helps with statistical inference of concepts at sub-string or sub-identifier level.

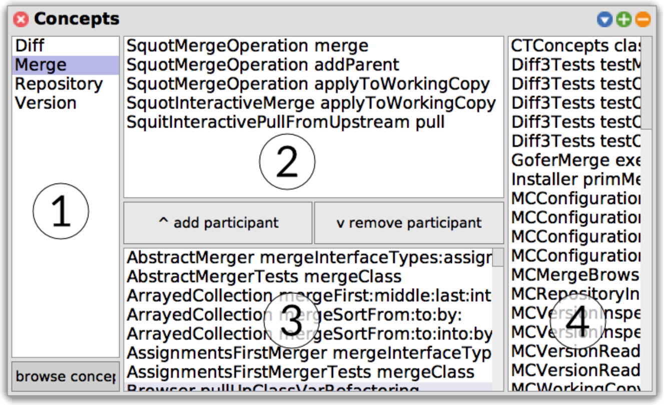
Concept-aware Code Editors Text editors allowing source code interaction have been extended to support concept assignment in its context menu, similar to Figure 7. To improve access to concepts, a bag-of-words heuristic is used to rank concepts by their vocabulary overlap with the selection. Two menu items are reserved for the most recently assigned concept, and the creation of a new concept. If the selection overlaps with an already assigned concept, this concept is present in the context menu as well and marked as active - clicking it removes the concept assignment. Syntax highlighting can be changed to concept highlighting, a mode where code tagged with a concept is typeset in the respective concept's color. Clicking any identifier in this mode opens the *concept editor*, from which navigation to other program parts of the same concept is possible. Highlighting concepts through color, however, made it hard to distinguish concepts with similar colors and overlapping concept.

Concept Editor To manage concepts through the UI, an editor which follows the multi-panel structure typical for Smalltalk allows concept management. A preliminary version has been implemented for this study, which is capable to list meta-objects associated with each concept, recommendations to quickly add meta-objects from the editor, and a list of methods that use (call into) the concept. For a screenshot, see Figure 10.

Concept Browser The browser is the standard code editor in Smalltalk and allows navigation by selecting class categories, classes, method categories, and methods in that order. Our variant limits the visibility of items when a browser is "scoped to a concept", e.g., when clicking *browse concept*. This allows faster navigation within a single concept. We plan to explore other filtering strategies like fading out unrelated meta-objects, putting them at the end of each list, or allowing to select concept unions and intersections. For a screenshot, see Figure 11.

4.2. Changing a Program

Study subjects As an example, we aim to resolve an open issue in a version control system in Squeak/Smalltalk:



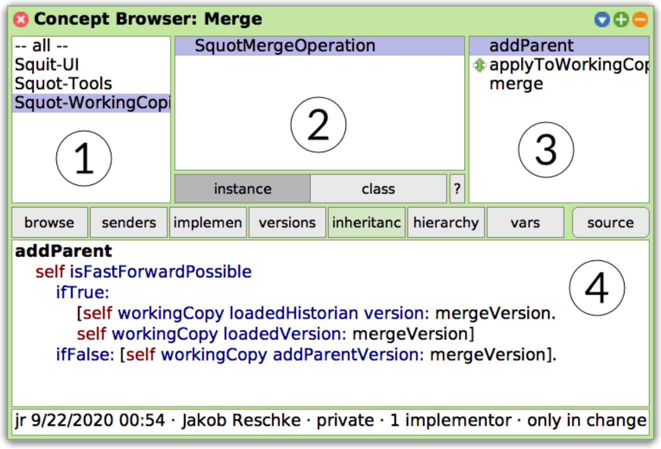
```
Diff
Merge
Repository
Version
```

```
SquotMergeOperation merge
SquotMergeOperation addParent
SquotMergeOperation applyToWorkingCopy
SquotInteractiveMerge applyToWorkingCopy
SquitInteractivePullFromUpstream pull
```

```
AbstractMerger mergeInterfaceTypes:assigr
AbstractMergerTests mergeClass
ArrayedCollection mergeFirst:middle:last:int
ArrayedCollection mergeSortFrom:to:by:
ArrayedCollection mergeSortFrom:to:into:by
AssignmentsFirstMerger mergeInterfaceTyp
AssignmentsFirstMergerTests mergeClass
```

```
CTConcepts cla:
Diff3Tests testM
Diff3Tests testC
Diff3Tests testC
Diff3Tests testC
Diff3Tests testC
GoferMerge exe
Installer primMe
MCConfiguratio
MCConfiguratio
MCConfiguratio
MCConfiguratio
MCMergeBrows
MCRepositoryIn
MCVersionInspe
MCVersionInspe
MCVersionRead
MCVersionRead
```

Figure 10 Screenshot of a simple concept editor viewing available concepts (1), items tagged with the selected concept (2), items most similar to the concept (3), and relevant users or clients of the concept (4).



```
-- all --
Squit-UI
Squot-Tools
Squot-WorkingCopi
```

```
SquotMergeOperation
```

```
addParent
applyToWorkingCop
merge
```

```
addParent
  self isFastForwardPossible
    ifTrue:
      [self workingCopy loadedHistorian version: mergeVersion.
      self workingCopy loadedVersion: mergeVersion]
    ifFalse: [self workingCopy addParentVersion: mergeVersion].
```

```
jr 9/22/2020 00:54 · Jakob Reschke · private · 1 implementor · only in change
```

Figure 11 Screenshot of a Smalltalk browser scoped to a single concept. The browser allows to navigate categories (1), classes (2), methods (3) and their source code (4) while only items tagged with or containing the selected "Merge" concept are visible.

Squot (Reschke et al. 2018)[3], the *Squeak Object Tracker*, and *Squit*, its Git backend. This project is well suited for a concept-oriented approach as it consists of several abstraction layers linking Smalltalk meta-objects to native Git structures and operations, but several concerns cut across architectural layers and abstractions use a different vocabulary than Git (to support multiple version control backends).

Using the tools outlined above, we attempt to add a new feature. As of now, merge conflicts cannot be resolved by opening an editor and manually creating the merged version.

Switching to the concept perspective To add this enhancement, we need to locate the *merge* concept. This is challenging as it cross-cuts several architectural layers, including the UI (for providing means to invoke the new merge mode and displaying the editor), the generic version control abstraction of Squot, and the specific implementation in Squit/Git.

We have previously annotated code relevant to merge functionality. Figure 10 shows the concept editor in the state where we manually added a few meta-objects and can now add additional merging functionality based on the recommended list below. A limitation of vocabulary-based recommenders becomes apparent as it recommends *merge sort* on arrays. At the same time, the list of items that use the merge concept on the right quickly suggests *unit tests* that allow us to understand the concept by studying the test examples.

Browsing the concept opens the scoped concept browser shown in Figure 11 and allows to focus on implementing the feature. By doing so, we need to add a few more methods to the concept which are currently unreachable in the filtered view. Escaping into a non-filtered browser is as easy as clicking the *browse* button.

4.3. Key Insights

Constructing short-cuts over selecting relevant slices The most salient observation while changing a cross-cutting concern was that modifying the program through a conceptual filter is a good way to grow a concept and discover its actual extent after starting with a few "seed items". With each new requirement implemented using this workflow, programmers refine the concept model, thus providing more and more short-cuts to understand scattered concerns. We conclude that, while traditional cross-cutting edits require programmers to ignore irrelevant parts or constantly search for relevant parts (*selective* workflow), our concept-oriented workflow motivates a *constructive* approach where items of interest are added to the currently edited conceptual view, starting with an empty set.

Discovering tests By having an overview over methods and classes which use (but do not constitute) a concept, it becomes relatively easy to locate relevant unit tests. Alternatively, we could have added the relevant tests directly to the concept. When confronted with the choice whether test code is a user or a part of the concept, programmers need to make sure they decide consistently for all tests.

[3] https://github.com/hpi-swa/Squot

Ambiguity of natural language Our simple word-based recommender is effective at suggesting ways to extend a concept, but suffers from ambiguities as the confusion between a merge in Git and a merge in the merge-sort algorithm illustrate. Hence, we suggest to use a language modeling approach that considers the semantic meaning of each word, or a statistical approach that considers past interactions, changes, and project structure. The latter would have identified merge sort as completely unrelated part that never shared any edits, authors, or distribution units (packages) with Squot.

Limitations of the Prototype As of now, we cannot share and version-control our concept model and have neither used more expressive tags nor distributional concepts. To collectively maintain a concept model, a way to commit, diff, and merge concept assignments is needed that should be as easy as version-controlling code comments.

Furthermore, we found that the elements of the primary decomposition still dominate navigation and concept management as it is easier to recommend and tag whole meta-objects rather than specific expressions or arguments inside methods, fields of classes, and other program items that tend to grow and lose cohesion as software evolves.

This finding appears to disagree with our hypothesis that fine-grained source code annotation is needed, but might as well be rooted in the program we were editing: We did neither encounter exceptionally large methods where concepts were entangled in a way that impedes program understanding, nor were we unable to discern which part of a method contributes to the concept we were interested in. Concepts as a way to slice the dominant decomposition were still useful, and sub-method granularity can be useful once long methods occur, which is rare in a Smalltalk project.

5. Discussion and Outlook

5.1. Integration into Programming Environments

Limited applicability to traditional IDEs Our designs rely on the existence of programing environments that go far beyond traditional text editing and diff-/patch-based versioning and assume the ground truth representation of the system is the graph of fine-grained meta-objects rather than a set of text files.

Traditional text-based IDEs and workflows require serialization of the concept model, e.g. as (likely much less readable) code comments or separate files, while mixed environments like Smalltalk can benefit from extensions to their meta-object model down to method granularity until we are confronted with limited text-based concept maintenance like described in our case study. Serialized or comment-embedded conceptual data is always prone to becoming outdated or inconsistent when any environment unaware of its meaning modifies the program's source code.

Fine-grained reification One limiting factor is the granularity of the object model that represents the full program and its accessibility to tool developers. With fully reified live-programming systems (Rein et al. 2018) like Smalltalk in combination with modern projectional editors (Beckmann et al. 2020; Voelter et al. 2014), these designs are easier to implement than in traditional IDEs. In such an ideal setting, the smallest expressions can have identity, which allows them to be tagged, tracked through the program evolution, and re-used in novel compositional views and editors.

Object-based distributed version control like Squot (Reschke et al. 2018) and continuous integration[4] have been shown to work seamlessly on meta-object graphs and integrate with real-world toolchains based on Git and popular CI services like *TravisCI* or *GitHub Actions*. Implementing our designs would add additional objects at a finer granularity level.

Live programming environments have the ability to make live examples part of the programming workflow (Rauch et al. 2019; Niephaus et al. 2020) and turn them into first-class entities in code editors. This offers novel opportunities to include them into modularity constructs (both primary and secondary) as a form of interactive documentation.

5.2. Navigability and Learnability

Dominant decompositions are not inherently an impediment to program comprehension and maintainability: They provide stable landmarks and help programmers create a mental map while working with the code. However, this mental map might only be accurate for specific areas, so any tacit knowledge acquired by navigating one part is less useful in previously unseen areas (e.g. legacy components, different teams, different companies/vendors), and fast high-level navigation can become harder.

Mental map stability and concepts Even if concepts are scattered and entangled, programmers might eventually remember where to find them or which term to search for in familiar parts of the dominant decomposition. On-demand navigation along conceptual relations and composable views that coherently display code of one concept run the risk of losing these landmarks. Further research might be needed to determine whether programmers might "get lost" when viewing the system from the perspective of its secondary structure or miss the big picture, and which techniques (e.g. breadcrumb navigation or showing sufficient primary context) alleviate this problem. A similar problem is frequently observed in the *C#* language that allows physical code files to exist independently of their namespace hierarchy (Lilienthal 2019).

In our case study, we focused on filtering existing code browsers, thus maintaining the (sparse) primary structure. This way, getting disoriented was less likely but we did not fully leverage the concept space. Instead of getting used to conceptual navigation, the tools were perceived as means to learn the dominant decomposition. In light of these observations, we plan to study in how far this effect is desirable or impedes program comprehension, and which interface designs present alternative conceptual views in such a way that using the dominant decomposition does not feel like a necessary fall-back but just another complementary perspective.

High-level navigability and desire paths Petricek (Petricek 2021) draws parallels between cities and software architecture.

[4] SmalltalkCI: https://github.com/hpi-swa/smalltalkCI (retrieved 2021-04-26)

Even in an imperfectly engineered part of a system, such as a city district or program module, its inhabitants (experts) can navigate efficiently, but for visitors it is important that movement between districts is easy, e.g. through public transit, and certain landmarks are present to orient themselves.

Applying these insights to our problem, a dominant decomposition would correspond to the historical road network and might be locally familiar to experts. Concepts can be used to construct alternative, but equally valid perspectives on our software analogously to public transport maps that omit roads and are geometrically simplified, yet provide useful guidance to outsiders, potential contributors, or "inhabitants of a different district", e.g., experts from another team.

Consequently, we expect concepts to be especially useful for such outsiders as a form of documentation close to the code, as well as a way to communicate aspects of the system at a higher level analogously to describing routes by pointing at a public transport map.

An analysis by Cataldo et al. (Cataldo et al. 2009) shows that sets of modules which are highly connected with each other (analogously to a neighborhood in a city) exhibit lower defect rates, i.e., experts know their "code neighborhood" and can change it more effectively compared to changes that depend on the cooperation with other neighborhoods. Concepts might help mapping such cross-neighborhood dependencies explicitly, thus supporting coordination efforts across remote code locations and making implicit "units of work" explicit. Supported by a (machine-learning backed) recommender with access to version history and code authorship, concepts could reconstruct the real boundaries of such neighborhoods which tend to match the social network of their maintainers, according to Conway's law, rather than the legacy architecture.

Apart from high-level navigability, the urban planning metaphor lends itself to describe another mechanism which we plan on exploring through first-class concepts:

Desire paths are trails left by people that took the path of least resistance instead of designated infrastructure (e.g. walking diagonally over a lawn between two paved ways) (Kohlstedt 2016). Equating the dominant decomposition with pre-existing infrastructure, programming environments should leave similar footprints of paths taken frequently but never represented explicitly. Such navigational data could serve as additional input to recommender systems or statistical models inferring concept allocation and serve as novel input alongside already persisted repository data.

From both perspectives, concepts provide both a way to "map" existing and "pave" new infrastructure when the historic pathways have become inefficient for the purpose of maintaining and extending a system.

5.3. Path to fully integrated lifecycle

We proposed to link lifecycle artifacts like build information, issues, or history to the affected code as a reified concept. This gives rise to fully integrate these artifacts into the program's meta-object graph. For example, issues would be a (cross-cutting) meta-object linked to code and tools could treat them in the same fashion as, e.g., a class or method with respect to version control, navigation, and even debugging. *URLs* can provide simple means to refer to lifecycle artifacts from the programming environment, while the issue tracker or continuous integration platform would have access to the code annotated with a specific artifact once the annotated version is committed.

Extrapolating from there, concepts could replace other currently syntactic mechanisms, such as access modifiers (public/private) or, in a more controversial proposal, (static) types, when they are only used for correctness checks but are not behaviorally significant.

6. Related Work

Tagging code The idea that multiple meta-objects can be sorted into (cross-cutting) concepts has been explored before:

ConcernMapper (Robillard & Weigand-Warr 2005) and *FEAT* (Robillard & Murphy 2003) are Eclipse plug-ins that use a reified concept model allowing to link concerns to Java meta-objects. They demonstrate that IDE features, e.g. navigation and code search, can benefit from knowing which "concern" the code belongs to and coherently display search results or rank them if they fit the currently edited/viewed concern. These "concerns" are effectively the first implementation of tag-like concepts at a high level of granularity and make the Eclipse programming environment partially concept-aware.

Similarly, *UseCasePy* (Hirschfeld et al. 2011) makes use cases a first-class entity that enables tracing requirements to an implementation via code annotations, effectively "tagging" meta-objects with their use case. In this way, programmers can make use of specific views on the source code that are centered around use cases. Further, the authors propose a semi-automatic manner to recover use case annotations from legacy code bases by using trace data from acceptance tests – an approach also feasible for concept discovery. In contrast to *ConcernMapper* and our first-class concepts, these annotations are part of the source code.

Extensional vs. intensional While first-class concepts are primarily *extensional*, i.e., their extent is defined by enumerating all links between code and concepts, there are approaches that define concepts intensionally by expressing its *characteristics* in a query language. Meta-objects are included automatically if they match the query and the formality allows verification:

The *Archface* language (Ubayashi et al. 2010) can model the correspondence between architectural elements and their implementation using language concepts from aspect-oriented programming. Archface supports multiple concurrent and cross-cutting views on the system, similar to those we suggest for a concept-annotated system. Its exactness offers the capability to automatically verify architecture at the expense of ease of change.

The *intentional view model* (Mens et al. 2002; Mens & Kellens 2005) allows programmers to create views on software that go beyond module boundaries. Through their use of a logic query language, multiple views can be verified for consistency, and programmers have the opportunity to model their knowledge about a concept in said language.

We propose a different trade-off by not requiring programmers to abstractly define the properties of their concept, but annotating concrete code passages that belong to a concept. Hence, our extensional approach is driven by *examples* rather than rules, while generalization and scaling are delegated to statistical models and recommender systems.

Cross-cutting secondary constructs *CodeTalk* (Steinert et al. 2010) reifies conversational comments at meta-object level and provides comprehensive tooling to access and manipulate them. *Cross-cutting Commentary* (Hirschfeld et al. 2018) solves the problem that comments are often tied to individual code locations and thus scatter when they explain a cross-cutting concern. By designing meta-objects that tie together cross-cutting comments and tools to interact with them, they support system exploration from a "secondary" viewpoint. An integration of Commentary with first-class concepts can be the basis for powerful notebook-like views on the program similar to those illustrated in Figure 5.

Code Bubbles (Bragdon et al. 2010) are a style of programming environments that make use of spatial secondary notation to place and group related sections of code on a “canvas” without imposing a strict dominant decomposition. Although concepts are not explicitly reified, conceptual relatedness can be expressed by proximity in two dimensions.

Conclusion

In this exploratory work, we illustrated *first-class* concepts as a way to express multiple competing conceptual perspectives on a system without the need to refactor the underlying module structure. Analogously to how a public transit map of a city serves different stakeholders than the historical road network, we see conceptual perspectives as a way to make large software systems more navigable for experts and non-experts alike by representing how important parts are connected outside their dominant decomposition.

We discussed requirements for the model that supports linking concepts to code with varying degrees of granularity, ranging from sub-expression level to large modularity constructs. We extended the notion of *concepts* to include both discrete phenomena (like patterns and cross-cutting concerns) as well as implicit concepts associated with uncertainty that can be managed with the help of statistical methods. These concepts can originate internally from the desire to structure and explain the system as well as from external lifecycle tools, thus offering diverse domain-oriented and technical perspectives to view the system.

Building on these ideas, we opened up the design space for tools that operate on the *conceptual* architecture and provide better ways to assist program comprehension, documentation, maintenance, and exploratory programming in architecturally convoluted systems. Our case study suggests that concept-aware tooling can provide a more focused programming experience, while simple recommender systems drastically accelerate concept allocation. Maintaining the concept model while simultaneously changing the system is a surprisingly effective way at discovering and persisting architectural connections in a way that can make future changes to the same concept easier. While our case study used only a small subset of the design space, recent developments in machine learning, projectional editors, and live programming environments render a more capable implementation and their evaluation feasible.

We identified user interactions that benefit from recommender systems where we see novel research opportunities in the intersection of programming experience and ML. Of particular interest are ways to convert the continuous stream of user input and navigation data into recommendations to co-maintain first-class concepts and the underlying architecture, distributed maintenance of an ML-assisted concept model among multiple contributors, "explainable AI" that makes its reasoning about a program’s structure accessible to programmers, and how such systems interact with live programming environments where run-time data is omnipresent and users expect immediate feedback while working on running programs.

Acknowledgments

This research has been supported by the Federal Ministry of Education and Research of Germany (BMBF) in the KI-LAB-ITSE framework (project number 01IS19066) and the HPI Research School for Service-Oriented Systems Engineering.

References

Allamanis, M., Barr, E. T., Bird, C., & Sutton, C. (2014, November). Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (pp. 281–293). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2635868.2635883

Beckmann, T., Ramson, S., Rein, P., & Hirschfeld, R. (2020, March). Visual design for a tree-oriented projectional editor. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (pp. 113–119). New York, NY, USA: ACM. doi: 10.1145/3397537.3397560

Biggerstaff, T., Mitbander, B., & Webster, D. (1993, May). The concept assignment problem in program understanding. In *[1993] Proceedings Working Conference on Reverse Engineering* (pp. 27–43). doi: 10.1109/WCRE.1993.287781

Blackwell, A., & Green, T. (2003). Notational systems–the cognitive dimensions of notations framework.

Bragdon, A., Zeleznik, R., Reiss, S. P., Karumuri, S., Cheung, W., Kaplan, J., ... LaViola, J. J. (2010, April). Code bubbles: A working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 2503–2512). New York, NY, USA: ACM. doi: 10.1145/1753326.1753706

Cataldo, M., Mockus, A., Roberts, J. A., & Herbsleb, J. D. (2009, November). Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, *35*(6), 864–878. doi: 10.1109/TSE.2009.42

Childs, B., & Sametinger, J. (1996). Literate programming and documentation reuse. In *Proceedings of fourth ieee* *international conference on software reuse* (p. 205-214). doi: 10.1109/ICSR.1996.496128

Garcia, J., Ivkovic, I., & Medvidovic, N. (2013, November). A comparative analysis of software architecture recovery techniques. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 486–496). doi: 10.1109/ASE.2013.6693106

Gethers, M., Savage, T., Di Penta, M., Oliveto, R., Poshyvanyk, D., & De Lucia, A. (2011). CodeTopics: Which Topic Am I Coding Now? In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 1034–1036). New York, NY, USA: ACM. doi: 10.1145/1985793.1985988

Herrmann, S. (2003). Object Teams: Improving Modularity for Crosscutting Collaborations. In M. Aksit, M. Mezini, & R. Unland (Eds.), *Objects, Components, Architectures, Services, and Applications for a Networked World* (pp. 248–264). Berlin, Heidelberg: Springer. doi: 10.1007/3-540-36557-5_19

Hirschfeld, R., Costanza, P., & Nierstrasz, O. (2008). Context-Oriented Programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, *7*(3), 125–151. doi: 10.5381/jot.2008.7.3.a4

Hirschfeld, R., Dürschmid, T., Rein, P., & Taeumel, M. (2018, July). Cross-cutting Commentary: Narratives for Multi-party Mechanisms and Concerns. In *Proceedings of the 10th International Workshop on Context-Oriented Programming: Advanced Modularity for Run-time Composition* (pp. 39–47). New York, NY, USA: ACM. doi: 10.1145/3242921.3242927

Hirschfeld, R., Perscheid, M., & Haupt, M. (2011). Explicit use-case representation in object-oriented programming languages. In *Proceedings of the 7th symposium on dynamic languages* (p. 51–60). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/2047849.2047856 doi: 10.1145/2047849.2047856

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. (1997). Aspect-oriented programming. In M. Akşit & S. Matsuoka (Eds.), *ECOOP'97 — Object-Oriented Programming* (pp. 220–242). Berlin, Heidelberg: Springer. doi: 10.1007/BFb0053381

Kohlstedt, K. (2016). *Least resistance: How desire paths can lead to better design.* https://99percentinvisible.org/article/least-resistance-desire-paths-can-lead-better-design/ (Retrieved 2021-09-14).

Lilienthal, C. (2019). *Sustainable Software Architecture: Analyze and Reduce Technical Debt.* dpunkt.verlag.

Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., & Baldi, P. (2007). Mining concepts from code with probabilistic topic models. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering* (pp. 461–464). Atlanta, GA, USA: ACM. doi: 10.1145/1321631.1321709

Lozano, Á., Kellens, A., Mens, K., & Arevalo, G. (2017, April). MEntoR: Mining entities to rules.

Lungu, M., Lanza, M., & Nierstrasz, O. (2014, January). Evolutionary and collaborative software architecture recovery with Softwarenaut. *Science of Computer Programming*, *79*, 204–223. doi: 10.1016/j.scico.2012.04.007

Mattis, T. (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* (pp. 36:1–36:2). Brussels, Belgium: ACM. doi: 10.1145/3079368.3079369

Mens, K. (2001). Multiple cross-cutting architectural views..

Mens, K., & Kellens, A. (2005). Towards a framework for testing structural source-code regularities. In *Proceedings of the 21st ieee international conference on software maintenance* (p. 679–682). USA: IEEE Computer Society. Retrieved from https://doi.org/10.1109/ICSM.2005.93 doi: 10.1109/ICSM.2005.93

Mens, K., Mens, T., & Wermelinger, M. (2002). Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on software engineering and knowledge engineering* (p. 289–296). New York, NY, USA: ACM. Retrieved from https://doi.org/10.1145/568760.568812 doi: 10.1145/568760.568812

Mens, K., Michiels, I., & Wuyts, R. (2001). Supporting software development through declaratively codified programming patterns. In *Journal on expert systems with applications* (pp. 236–243).

Niephaus, F., Rein, P., Edding, J., Hering, J., König, B., Opahle, K., . . . Hirschfeld, R. (2020, November). Example-based live programming for everyone: Building language-agnostic tools for live programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (pp. 1–17). New York, NY, USA: ACM. doi: 10.1145/3426428.3426919

Petricek, T. (2021). Programming as architecture, design and urban planning. In *Onward! 2021 (to appear).*

Rauch, D., Rein, P., Ramson, S., Lincke, J., & Hirschfeld, R. (2019, February). Babylonian-style Programming. *The Art, Science, and Engineering of Programming*, *3*(3), 9:1-9:39. doi: 10.22152/programming-journal.org/2019/3/9

Rein, P., Ramson, S., Lincke, J., Hirschfeld, R., & Pape, T. (2018, July). Exploratory and live, programming and coding. *The Art, Science, and Engineering of Programming*, *3*(1), 1:1-1:33. doi: 10.22152/programming-journal.org/2019/3/1

Reschke, J., Taeumel, M., Pape, T., Niephaus, F., & Hirschfeld, R. (2018). *Towards version control in object-based systems*. Universitätsverlag Potsdam.

Robillard, M. P., & Murphy, G. (2003, May). FEAT a tool for locating, describing, and analyzing concerns in source code. In *25th International Conference on Software Engineering, 2003. Proceedings.* (pp. 822–823). doi: 10.1109/ICSE.2003.1201304

Robillard, M. P., & Weigand-Warr, F. (2005, October). ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (pp. 65–69). New York, NY, USA: ACM. doi: 10.1145/1117696.1117710

Saeidi, A. M., Hage, J., Khadka, R., & Jansen, S. (2015). ITMViz: Interactive topic modeling for source code analysis. In *Proceedings of the 2015 IEEE 23rd International Conference* *on Program Comprehension* (pp. 295–298). Piscataway, NJ, USA: IEEE Press.

Schärli, N., Ducasse, S., Nierstrasz, O., & Black, A. P. (2003). Traits: Composable units of behaviour. In L. Cardelli (Ed.), *ECOOP 2003 – Object-Oriented Programming* (pp. 248–274). Berlin, Heidelberg: Springer. doi: 10.1007/978-3-540-45070-2_12

Steinert, B., Taeumel, M., Lincke, J., Pape, T., & Hirschfeld, R. (2010, January). CodeTalk: Conversations about code. In *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing* (pp. 11–18). doi: 10.1109/C5.2010.11

Tarr, P., Ossher, H., Harrison, W., & Sutton, S. (1999). N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 international conference on software engineering (ieee cat. no.99cb37002)* (p. 107-119).

Ubayashi, N., Nomura, J., & Tamai, T. (2010, May). Archface: A contract place where architectural design and code meet together. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (pp. 75–84). New York, NY, USA: ACM. doi: 10.1145/1806799.1806815

Voelter, M., Siegmund, J., Berger, T., & Kolb, B. (2014). Towards user-friendly projectional editors. In B. Combemale, D. J. Pearce, O. Barais, & J. J. Vinju (Eds.), *Software language engineering* (pp. 41–61). Cham: Springer International Publishing. doi: 10.1007/978-3-319-11245-9_3

Wilde, N., & Scully, M. C. (1995). Software reconnaissance: Mapping program features to code. *Journal of Software Maintenance: Research and Practice*, *7*(1), 49–62. doi: 10.1002/smr.4360070105

About the authors

Toni Mattis is a PhD student in the Software Architecture Group of the Hasso Plattner Institute at the University of Potsdam. His research interests are software modularity, machine learning for live programming environments, and code repository mining. You can contact him at toni.mattis@hpi.uni-potsdam.de.

Tom Beckmann is a PhD student in the Software Architecture Group of the Hasso Plattner Institute at the University of Potsdam. His research interests include visual languages, projectional editors, and live programming environments. You can contact him at tom.beckmann@hpi.uni-potsdam.de.

Patrick Rein is a PhD student in the Software Architecture Group of the Hasso Plattner Institute at the University of Potsdam. His research interests include live and exploratory programming systems as well as personal information management systems. You can contact him at patrick.rein@hpi.uni-potsdam.de.

Robert Hirschfeld leads the Software Architecture Group at the Hasso Plattner Institute at the University of Potsdam. His research interests include dynamic programming languages, development tools, and runtime environments to make live, exploratory programming more approachable. Hirschfeld received a PhD in computer science from Technische Universität Ilmenau. You can contact him at robert.hirschfeld@hpi.uni-potsdam.de or visit https://hpi.de/swa.