First-Class Concepts: Reifying Architectural Knowledge beyond the Dominant Decomposition

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

Tom Beckmann
tom.beckmann@hpi.uni-potsdam.de
Software Architecture Group
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany

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

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

ABSTRACT

In software engineering, programs are ideally partitioned into independently maintainable and understandable modules. As a system grows, its architecture gradually loses the capability to modularly accommodate new concepts. While refactoring is expensive and the language might lack appropriate *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, they have no reified representation that can be explored and maintained through tools.

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. Our concepts are 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 and change programs from conceptual perspectives rather than scattering their attention across existing modules.

Our designs are geared towards facilitating multiple *secondary* perspectives on a system to co-exist alongside the original architecture, hence making it easier to explore, understand, and explain complex contexts and narratives not expressible in traditional modularity constructs.

COP '21, July 12, 2021, Virtual, Denmark

ACM ISBN 978-1-4503-8542-8/21/07…$15.00
https://doi.org/10.1145/3464970.3468413

CCS CONCEPTS

• **Software and its engineering → Modules / packages**; Software design engineering; **Integrated and visual development environments**.

KEYWORDS

software engineering, modularity, exploratory programming, program comprehension, remodularization

ACM Reference Format:
Toni Mattis, Tom Beckmann, Patrick Rein, and Robert Hirschfeld. 2021. First-Class Concepts: Reifying Architectural Knowledge beyond the Dominant Decomposition. In *Proceedings of the 13th ACM International Workshop on Context-Oriented Programming and Advanced Modularity (COP '21), July 12, 2021, Virtual, Denmark*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3464970.3468413

1 INTRODUCTION

Expressive programming languages offer constructs to partition systems into modules. In currently 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. 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. Deployment modules can help install and run different

[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.

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.

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 [12, 20].

A similar case can be made for design patterns [15], 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.

Advanced modularity constructs like aspects, layers [6], roles, or traits 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.

Primary and secondary modularity constructs. The 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* [2], 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 syntactic constraints are dropped, several other mechanisms help with modularity: free-form comments, additional linebreaks 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.

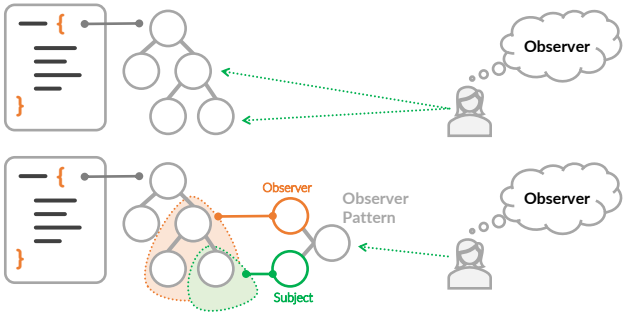

Figure 1: Reification of concepts that were previously implicit

Problem statement and research opportunity. Our working hypothesis is that *primary* modularity constructs can be hard to change to re-modularize existing systems and dominant decomposition subordinates potentially equally relevant concepts.

In contrast, existing *secondary* constructs contain little machine-readable information and mostly express additional concepts by means of natural language.

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 significantly re-organizing the underlying primary module composition. The existence of first-class concepts creates new tooling capabilities for exploration and maintenance.

2 REPRESENTING CONCEPTS

This section refines our notion of concepts and describes how we intend to represent them in a programming environment.

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. In particular, they should have the following properties (this list is not exhaustive):

Independent 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.

Non-exclusive 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, allowing technical and domain concepts to overlap.

Non-executable A concept should not interfere with compilation and run-time behavior, much like a comment or non-code cells in notebooks.

Reified In contrast to comments, the content of a concept should allow tools to automatically process them. Persistence

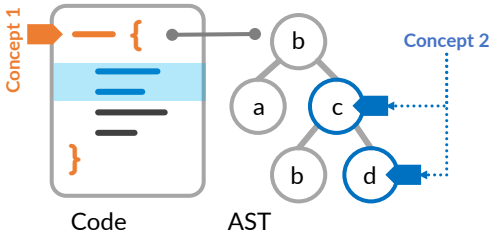

Figure 2: A concept tag can be attached to any meta-object, including expressions within one module's AST.

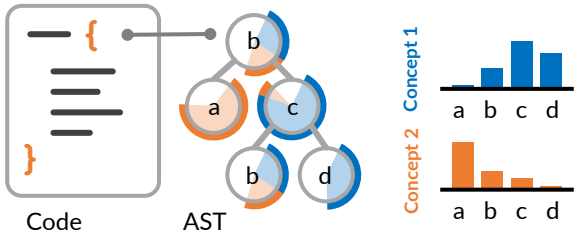

Figure 3: Two simple distributional concepts. a - d are syntactic features, e.g. specific names, types or call sites that occur in AST nodes.

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].

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, such as relations to tests, dependencies, or build artifacts.

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, least expressive manifestation, a concept behaves like a tag (label) attached to one or more meta-objects or expressions as illustrated in Figure 2. 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.

More expressive tags. As tags become (secondary) meta-objects themselves, they can apply to other tags, thus creating hierarchies of tags, e.g. by tagging individual roles of a pattern in code and tagging their tags as instance of that pattern like in Figure 1. A generalization to ontologies is possible, but out of scope for this work.

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.

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 should 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 [10] and programmer-supported ways [18] using topic modeling and have found their ways into programming environments already [5, 11].

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

[2]Imagine them being stored in a Smalltalk image without ever taking a "serialized" form.

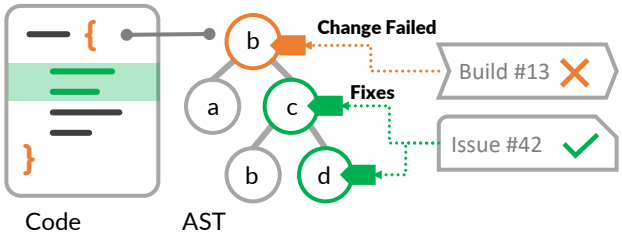

Figure 4: Lifecycle artifacts can be expressed using their own tag-like modularity constructs, in this example, a build failure and a fixed issue are (automatically) attached to the involved code.

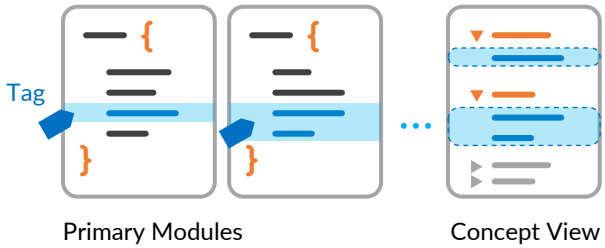

Figure 5: A concept expressed through tagged code can be rendered as single on-demand module with additional meta data explaining how the constituents relate to each other.

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.

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 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.

Manual basic tool support should allow manual concept assignment at meta-object level.

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.

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:

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.

3.2 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 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* [4], where techniques of object-oriented programming are applied to documentation to facilitate its reuse.

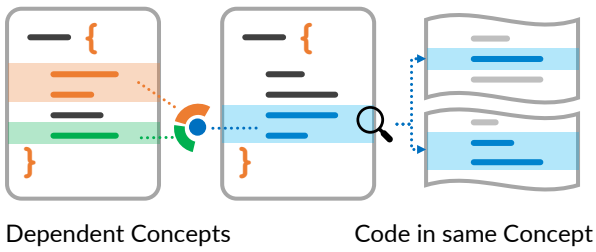

Figure 6: Concept-level navigation allows rapid navigation/preview of (scattered) code within the same concept or dependent concepts, thus forming a navigable graph beyond the dominant decomposition.

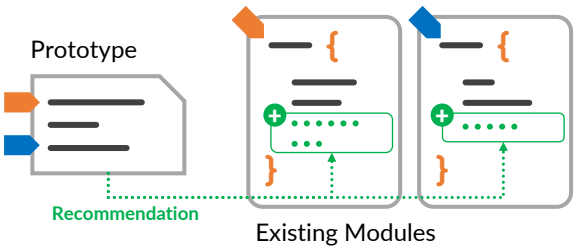

Figure 7: Concepts can help move exploratorily prototyped code into conceptually sound locations within the existing architecture.

3.3 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 [3]), 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 6.

3.4 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 7.

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 DISCUSSION AND OUTLOOK

Limits in 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.

With fully reified live-programming systems like Smalltalk in combination with modern projectional editors [1, 22], these designs are easier to implement than in traditional IDEs. Object-based version control like Git[3] and continuous integration[4] can work seamlessly on meta-object graphs, implementing our designs would add additional objects.

Traditional text-based IDEs and workflows require serialization of the concept model, e.g. as (likely much less readable) code comments.

Mental maps. A dominant decomposition might have the benefit of creating a stable mental map while working with the code. Even if concepts are scattered and entangled, programmers might eventually remember where to find them. On-demand navigation along conceptual relations and composable views that coherently display code of one concept lose 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. bread-crumb 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 [9].

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.

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.

5 RELATED WORK

ConcernMapper [17] and *FEAT* [16] 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.

The *Archface* language [21] 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. Its exactness offers the capability to automatically verify architecture at the expense of ease of change.

Code Bubbles [3] are a style of programming environments that make use of spatial secondary notation to group conceptually related sections of code without imposing a strict dominant decomposition.

CodeTalk [19] reifies conversational comments at meta-object level and provides comprehensive tooling to access and manipulate them. *Cross-cutting Commentary* [7] 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.

Similarly, *UseCasePy* [8] makes use cases a first class concept that enables tracing requirements to an implementation via code annotations. 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.

To enable exploration and verification of aspects in software systems, the *intentional view model* [13, 14] allows programmers to create views on software that go beyond module boundaries. For example, programmers can get a mapping between test coverage and source code by outlining relevant language constructs in one view and tests that make use of the listed language constructs in another.

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

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.

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 legacy systems.

While these designs are high-level and preliminary, recent developments in machine learning, projectional editors, and live programming environments render prototypes and their evaluation feasible.

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

[1] Tom Beckmann, Stefan Ramson, Patrick Rein, and Robert Hirschfeld. 2020. Visual Design for a Tree-Oriented Projectional Editor. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming (<Programming> '20)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3397537.3397560

[2] Alan Blackwell and Thomas Green. 2003. Notational systems–the cognitive dimensions of notations framework. (2003).

[3] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. Association for Computing Machinery, New York, NY, USA, 2503–2512. https://doi.org/10.1145/1753326.1753706

[4] B. Childs and J. Sametinger. 1996. Literate programming and documentation reuse. In *Proceedings of Fourth IEEE International Conference on Software Reuse*. 205–214. https://doi.org/10.1109/ICSR.1996.496128

[5] Malcom Gethers, Trevor Savage, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. CodeTopics: Which Topic Am I Coding Now?. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 1034–1036. https://doi.org/10.1145/1985793.1985988

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

[7] Robert Hirschfeld, Tobias Dürschmid, Patrick Rein, and Marcel Taeumel. 2018. 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 (COP '18)*. Association for Computing Machinery, New York, NY, USA, 39–47. https://doi.org/10.1145/3242921.3242927

[8] Robert Hirschfeld, Michael Perscheid, and Michael Haupt. 2011. Explicit Use-Case Representation in Object-Oriented Programming Languages. In *Proceedings of the 7th Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '11)*. Association for Computing Machinery, New York, NY, USA, 51–60. https://doi.org/10.1145/2047849.2047856

[9] Carola Lilienthal. 2019. *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. dpunkt.verlag.

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

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

[12] K. Mens. 2001. Multiple Cross-Cutting Architectural Views.

[13] Kim Mens and Andy Kellens. 2005. Towards a Framework for Testing Structural Source-Code Regularities. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, USA, 679–682. https://doi.org/10.1109/ICSM.2005.93

[14] Kim Mens, Tom Mens, and Michel Wermelinger. 2002. Maintaining Software through Intentional Source-Code Views. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering* (Ischia, Italy) *(SEKE '02)*. Association for Computing Machinery, New York, NY, USA, 289–296. https://doi.org/10.1145/568760.568812

[15] Kim Mens, Isabel Michiels, and Roel Wuyts. 2001. Supporting Software Development through Declaratively Codified Programming Patterns. In *Journal on Expert Systems with Applications*. 236–243.

[16] M.P. Robillard and G.C. Murphy. 2003. FEAT a Tool for Locating, Describing, and Analyzing Concerns in Source Code. In *25th International Conference on Software Engineering, 2003. Proceedings.* 822–823. https://doi.org/10.1109/ICSE.2003.1201304

[17] Martin P. Robillard and Frédéric Weigand-Warr. 2005. ConcernMapper: Simple View-Based Separation of Scattered Concerns. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange (Eclipse '05)*. Association for Computing Machinery, New York, NY, USA, 65–69. https://doi.org/10.1145/1117696.1117710

[18] 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.

[19] Bastian Steinert, Marcel Taeumel, Jens Lincke, Tobias Pape, and Robert Hirschfeld. 2010. CodeTalk Conversations about Code. In *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing*. 11–18. https://doi.org/10.1109/C5.2010.11

[20] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. 1999. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)*. 107–119.

[21] Naoyasu Ubayashi, Jun Nomura, and Tetsuo Tamai. 2010. 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 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 75–84. https://doi.org/10.1145/1806799.1806815

[22] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards User-Friendly Projectional Editors. In *Software Language Engineering (Lecture Notes in Computer Science)*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, Cham, 41–61. https://doi.org/10.1007/978-3-319-11245-9_3