Cross-cutting Commentary

Narratives for Multi-party Mechanisms and Concerns

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

Tobias Dürschmid
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
tobias.duerschmid@student.hpi.uni-potsdam.de

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

Marcel Taeumel
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
marcel.taeumel@hpi.uni-potsdam.de

ABSTRACT

Cross-cutting concerns are an inherent property of the implementation of non-trivial software systems. Their study led to the development of advanced modularity constructs, usually supported by meta-level frameworks and programming language constructs, to improve comprehensibility. Because of their invasive nature, systems need to be refactored or rewritten to take advantage of these constructs. However, practical considerations such as organizational or economical constraints often do not allow for such reengineering efforts, leaving those systems without explicit representations of their cross-cutting concerns.

We propose a lightweight, non-invasive approach to explicate and document cross-cutting, multi-party concerns called *Cross-cutting Commentary*, or *Commentary* for short. Our proposal is based on the observation that comments are co-located with the individual semantic units they are about and with that scattered and tangled in the absence of advanced modularity constructs for cross-cutting concerns and the assumption that well-crafted, informal explanations of system properties (their intents and the mechanisms they provide) improve comprehensibility. Commentaries are to help communicate narratives about system properties that involve multiple participants, both co-located in a single module or cross-cutting several of them, and allow for navigating to, from, and between them to explore the implementation artifacts involved.

Commentary was inspired by *layers* introduced with Context-oriented Programming to associate and manage partial definitions of system elements. While layers contribute to system comprehension during development and software composition at run-time, Commentary focuses on narratives for system exploration.

We present our first attempt to provide Commentaries in Squeak/Smalltalk. We explain implementation details and discuss several application scenarios considering the documentation of basic mechanisms of this programming and runtime environment.

COP'18, July 16, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5722-7/18/07…$15.00
https://doi.org/10.1145/3242921.3242927

KEYWORDS

cross-cutting concerns, documentation, modularity, exploratory programming, live programming, aspect-oriented programming, context-oriented programming, feature-oriented programming, Squeak/Smalltalk

ACM Reference Format:
Robert Hirschfeld, Tobias Dürschmid, Patrick Rein, and Marcel Taeumel. 2018. Cross-cutting Commentary: Narratives for Multi-party Mechanisms and Concerns. In *10th International Workshop on Context-Oriented Programming (COP'18), July 16, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3242921.3242927

1 INTRODUCTION

Software design involves the discovery and communication of domain-related entities, a formation of relationships among them, and their participation in meaningful scenarios. In this process, each such entity will be assigned a set of responsibilities to be fulfilled and other entities (or collaborators) to interact with in order to accomplish a larger goal.

There is usually an intent associated with a design decision that involves several participants and their links to each other. However, the intent is often lost because of its cross-cutting nature and the lack of advanced modularity constructs to capture such designs (and their intents) in the first place. Comments to remind the reader (the original programmers and the subsequent maintainers) of the purpose of implementation fragments are scattered (and often duplicated) throughout the system and recognized (if at all) as part of an overall system description only with great effort by the reader.

In Smalltalk, for example, there is a dependency mechanism (also known as changed/Update mechanism named after the two main method groups involved in this mechanism) implemented in classes `Object` and `Model` *"to coordinate activities among different objects. Specifically, its purpose is to be able to link one object, say A, to one or more other objects, say B, so B can be informed if A changes in any way. Upon being informed when A changes and the nature of the change, B can decide to take some action such as updating its own status. The concept of change and update, therefore, are integral to the support of this [...] kind of object dependence relationship. [...]"* [4]. The explanation goes on like that and eventually explains all methods that are part of this dependency mechanism and sample application scenarios built using them.

In actual Smalltalk systems such as Squeak [7], this account (or narrative) in its entirety is missing from the documentation. Instead, only pieces concerning a particular aspect of its implementation are placed as comments across several method implementations. For example, in `Object>>changed:` one can read *"Receiver changed. The change is denoted by the argument aParameter. Usually the argument is a Symbol that is part of the dependent's change protocol. Inform all of the dependents."* and in `Object>>update:` this is continued by *"Receive a change notice from an object of whom the receiver is a dependent. The default behavior is to do nothing; a subclass might want to change itself in some way."* [3]

While introductory courses recommend external documents such as the Smalltalk-80 BlueBook [4] for in-depth explanations like the one above, the system itself, which is known for explorative and interactive development tools, does not provide much more than individual and loosely coupled comments similar to the ones just mentioned.

Approaches such as feature-oriented programming (FOP [16]), context-oriented programming (COP [5]), or aspect-oriented programming (AOP [9]) help capture such design elements by offering dedicated language constructs or meta-level libraries. Features, layers, or aspects co-locate system parts (often partial class or method definitions) that contribute to the implementation and documentation of a particular concern. Thereby, they help both human readers and automated systems to better reason about that concern.

Development and runtime platforms that do not support such techniques continue to suffer from the scattering and tangling of both implementation and documentation artifacts. Holistic explanations, if available at all, are (too) removed from the system and require extra effort from readers to connect the description with what is described. In addition to this, immediate lack of benefit, a disconnect between artifacts such as an implementation and its documentation, can and will yield divergence also known as (architectural) drift.

In this paper, we propose a lightweight, and non-invasive concept called *Commentary* to make cross-cutting concerns and their documentation explicit and traceable, support co-evolution of related artifacts, avoid scattering and tangling of documentation, and improve system comprehension.

2 COMMENTARY

A *Commentary* connects several parts of a system by providing narratives that explain their interconnection. It can range from mainly technical interactions to a merely conceptual point of view, extending from, for example, co-ordinated library and API calls to framework-driven interactions to use cases and beyond.

When building or maintaining a software system, there are several artifacts of interest that draw the attention of developers. In object-oriented systems, these artifacts typically involve two kinds of objects: (1) metaclasses, classes, instances, or (2) prototypes and methods. While both kinds of entities can be commented, such comments usually solely explain the individual artifact, but rarely try to connect related entities. This is mainly due to the lack of proper cross-referencing mechanisms and other overarching system elements.

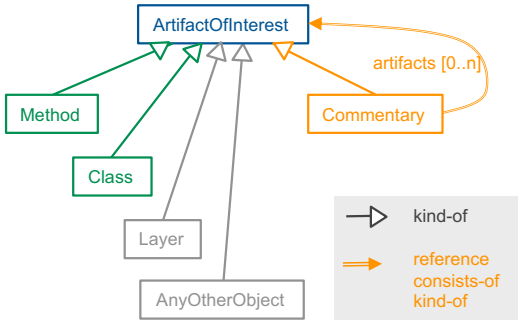


Figure 1: Commentary conceptual model.

The introduction of Commentaries offers an additional, lightweight, and non-invasive construct that allows to bridge this gap (Figure 1). A Commentary holds on to a narrative that ties together several artifacts and means to refer to these artifacts, from inside the narrative or otherwise. In general, Commentaries can maintain several relationships of different kinds to other artifacts, which includes pointers to other Commentaries as well. In the following, we employ references only from Commentaries to artifacts, but containment (consists-of) and refinement (kind-of) relationships are also worth exploring for *Commentary composition*.

In addition to regular methods, classes, and Commentaries, artifacts of interest include but are not limited to features, layers, aspects, and any other meta- or domain object reachable and worth mentioning.

Our model allows for a many-to-many relationship between Commentaries and artifacts. Thereby, artifacts can be part of more than one Commentary and, therefore, allow for overlaps between them. For example, class `Model` and its methods `>>changed:` and `>>update:` (inherited from class `Object`) can be (and in our system are) part of the two Commentaries "Changed/Update" and "Model-View-Controller." In this particular case, "Model-View-Controller" is based on "Changed/Update" (Figure 2).

Model-View-Controller or MVC is the user interface framework and paradigm of Smalltalk-80 [10] and still maintained as a UI alternative and fallback mechanism in the Squeak/Smalltalk environment. MVC makes heavy use of the Changed/Update mechanism to keep its components in sync. In the seminal MVC essay, one can read *"Model-View-Controller (MVC) programming is the application of this three-way factoring, whereby objects of different classes take over the operations related to the application domain (the model), the display of the application's state (the view), and the user interaction with the model and the view (the controller). [...] Models are those components of the system application that actually do the work (simulation of the application domain). They are kept quite distinct from views, which display aspects of the models. Controllers are used to send messages to the model, and provide the interface between the model with its associated views and the interactive user interface devices (e.g., keyboard, mouse). Each view may be thought of as being closely associated with a controller, each having exactly one model, but a model may have many view/controller pairs. [...]"* The essay goes on in explaining individual parts of this framework and their interplay.

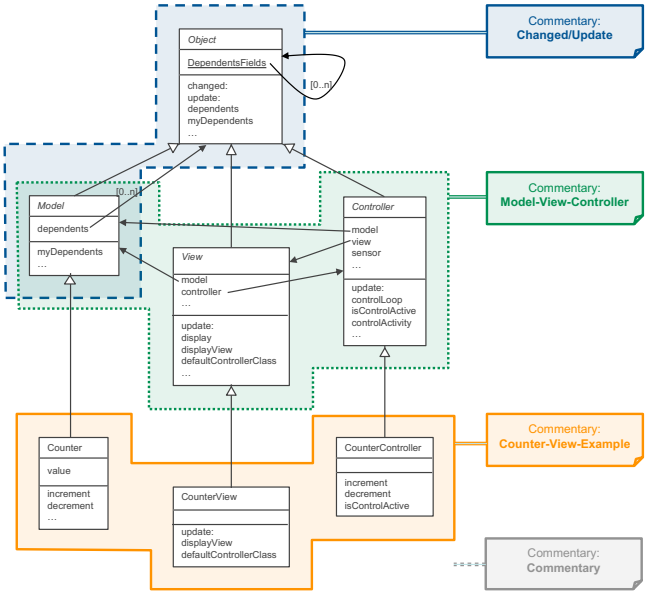


Figure 2: Cross-cutting Commentaries.

As with the Changed/Update mechanism, this account (or narrative) of MVC in its entirety is also missing from the documentation and, again, only parts of it are placed in comments close to selected implementation artifacts such as methods. This includes `Object>>addDependent:` to subscribe observers, `View>>defaultControllerClass` to configure a view's standard behavior, and `Controller>>controlActivity` to specify the first actions in the control loop.

Note that, while the Commentary of Changed/Update mainly focuses on a small subset of methods of `Object` (about 10 out of 450 methods implemented in `Object` [3]), the MVC Commentary not only cross-cuts elements of a single class, but several classes (here at least three of the base classes where most of the core architecture of the framework is defined and many more of their subclasses for ease of configuration and use).

The "Counter-View-Example" displayed in the CommentaryBrowser in Figure 3 is yet another kind of Commentary [10]. "Change/Update" describes a quite small and very local concern involving at least three system classes. However, "Counter-View-Example" exhibits application-level elements that are based on "Model-View-Controller" as shown in Figure 2. More-loosely coupled cross-cutting concerns or tutorials might also be good candidates to be captured, explained, or even deployed as Commentaries.

3 CROSS-CUTTING

Commentaries can describe ideas and mechanisms ranging from high-level to low-level, implementation-specific. They can incorporate and refer to modules and system units as supported by the programming language and runtime environments. The more such participants are involved the more likely it is that a Commentary cross-cuts the program or system.

Since Commentaries (can but) do not have to follow the dominant decomposition as dictated by the programming language, they help address *scattering* and *tangling*.

By bringing and keeping together what belongs together in a single and cohesive narrative, *scattering* can be avoided entirely. This one-to-many relationship between a Commentary and its participants is the most obvious of its benefits.

Also there is a one-to-many relationship between each individual participant and several Commentaries it is mentioned in or contributes to. Since different aspects of a particular participant can be emphasized by different narratives differently and, most importantly, those different comments are not co-located within the same unit of code but part of separate, yet cross-cutting narratives of multiple Commentaries and so avoid *tangling*.

In Figure 3 the one-to-many relationship between participants and Commentaries are expressed by the left list pane of the CommentaryBrowser and the one-to-many relationship between a Commentary and its participants by the browser's list pane on the right.

4 QUANTIFICATION

Another important observation is that Commentaries offer several means to describe or specify their participants. While we are planning to provide a more comprehensive report on such means in future work, here is an explanation of Commentaries as *homogeneous* and *heterogeneous* cross-cutting concerns.

Homogeneous cross-cutting concerns are well-known from and widely researched by the AOP community. Many AOP systems provide so-called *pointcut languages* to declaratively express sets of *join points* that exhibit specific and often repeating properties. Dynamically typed aspect-oriented programming languages with a rich reflection API often use metaprogramming to achieve the same.

Heterogeneous cross-cutting concerns, which are at the core of COP systems, often do not exhibit repeating properties and so explicitly enumerate their behavioral variations and the join points at which they are to occur.

While our examples emphasize documentation of heterogeneous cross-cutting concerns, Commentaries allow the application of all of the means of quantification (such as point-cut languages and predicates, metaprograms, and explicit enumeration) mentioned above to denote participants, including other Commentaries.

5 NAVIGATION

Many modern programming environments support simple, easy navigation between system elements such as classes, methods, and objects.

Commentary adds navigation support by providing convenient access to participants and other Commentaries referenced. All system elements can be used as starting points in tools like code browsers, object inspectors, or run-time debuggers to access Commentaries they already are or going to be involved in.

Figure 3 gives an impression of our integration of Commentary into the Squeak/Smalltalk environment and the navigation paths encouraged so far.

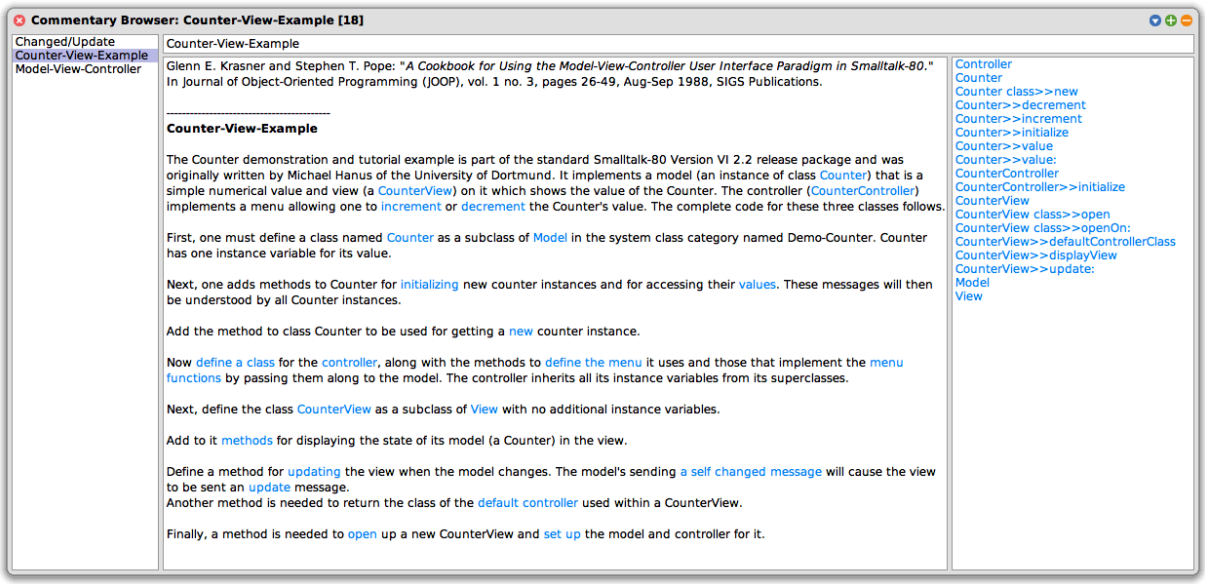


Figure 3: CommentaryBrowser showing the "Counter-View-Example".

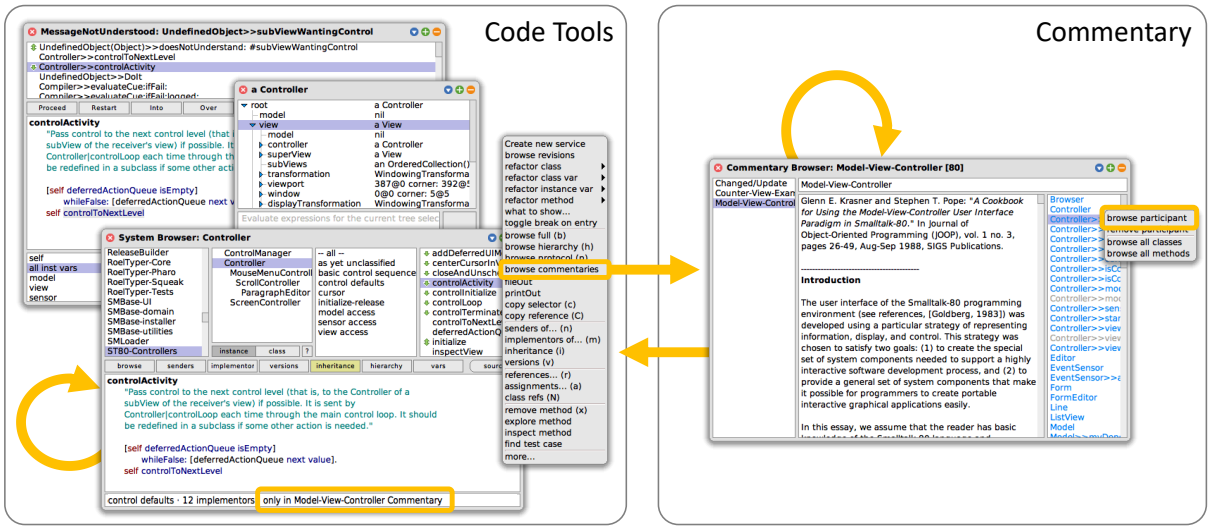


Figure 4: Navigation between programming tools like code browsers, object explorers, and debuggers and Commentaries.

6 IMPLEMENTATION

We have implemented a first prototype of Commentaries for the Squeak/Smalltalk environment [3]. This simple extension to Squeak consists of three light-weight classes, which use only a few hooks into the meta system (Figure 5).

`Commentary` is at the core of our implementation. As its name suggests, it represents a Commentary by storing its `title` as a brief summary, a `narrative` with all details, and a list of `participants` as fan-out. While the `title` helps locate and talk about a commentary, most of what makes a commentary is captured in the other fields.

A `narrative` is an instance of `Text` that stores both the story (a `String`) the commentary is about and text attributes (each an instance of `TextURL`) that allow to link text segments to a subset of the participants (metaobjects or regular objects) they refer to. If valid, a `TextURL` allow to directly navigate to the linked (meta-) object.

The list of `participants` contain all of the participants mentioned in the `narrative` via `TextURL`s and all other entities directly added to them. Allowing developers to add entities other than the ones mentioned in the `narrative` can help in several ways. For example, when creating a commentary, one could start with an empty `narrative`, add all entities potentially covered by the `narrative`,

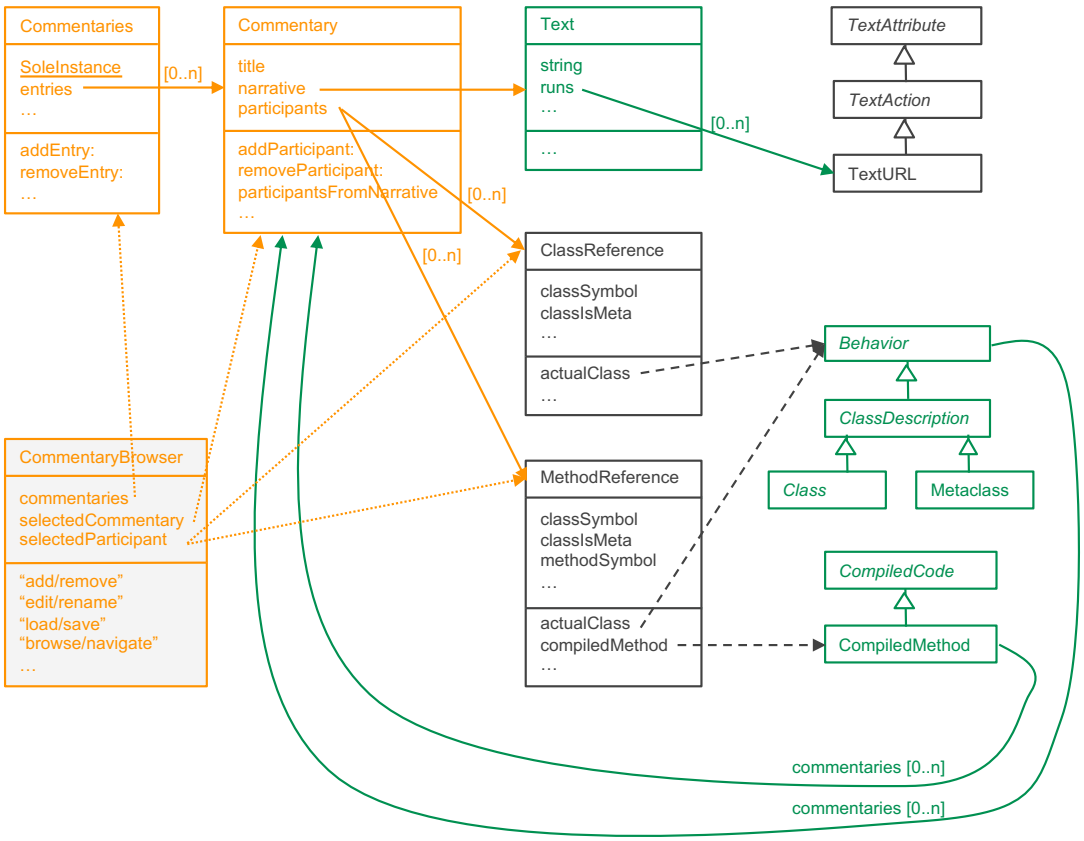


Figure 5: A first implementation of Commentaries in Squeak/Smalltalk.

and then evolve the narrative to cover as many of the already identified entities as possible. Or one could treat the entities referred to from the `narrative` as core participants and provide all other participants as seeds for system exploration encouraged by the commentary.

Squeak's `ClassReferences` and `MethodReferences` are lightweight proxies for `Behaviors` or `CompiledMethods` that help both to decouple a reference to from the objects referenced and to establish a proper reference when needed and if possible. *Their use makes it possible to refer to participants mentioned in the commentary but not (yet or anymore) present in the system so that commentaries and the system can evolve decoupled and independently.*

On the same account, these proxies help developers get a good understanding of the completeness of the system with respect to a commentary or, the other way around, the completeness of a commentary with respect to the system it is to explain.

The `CommentaryBrowser` shown in Figure 3 and mentioned in Figure 5 is based on Squeak's `ToolBuilder` framework to fit Squeak's standard tool suite [2]. In its four panes it lists all commentaries available and displays for a selected commentary its `title`, `narrative`, and list of `participants`. One can add, remove, load, and store individual commentaries and change their `titles`, `narratives`, and list of `participants`. `participants` are both collected from the `narrative`, added via drag and drop from the Squeak environment, or maintained via a context menu.

7 DISCUSSION

Cross-cutting Commentary provides advantages across many dimensions of software development.

At first, the proposed concept supports developers in making cross-cutting concerns explicit by providing a tool for documenting them closely to the related source code.

Additionally, by systematically linking commentaries about general system aspects with their related implementations, cross-cutting Commentary provides *bidirectional traceability*, which in general can increase the software quality as well as productivity and accuracy of maintenance [15, 17]. Developers can directly find the methods and classes that implement a requirements by navigating to the participants of the corresponding Commentary. Additionally, the requirements that influenced the design decisions of a development artifact can be found by browsing the commentaries linked to the artifact.

Often comments related to cross-cutting concerns are duplicated across the involved methods or classes [11]. This might result in inconsistencies or overhead when changing documentation. Such duplicate documentation can be reduced by providing a many-to-many mapping between commentaries and development artifacts.

However, the concept might result in some challenges. At first, additional infrastructure is required for exchanging Commentary information between development environments. Furthermore, the

system works only if all developers have a CommentaryBrowser installed because our new artifacts cannot easily be viewed or edited with existing tools. Similarly, additional tool support for code reviews and other social coding web sites are needed to display commentaries attached to the code. These issues can complicate the technology stack of the developers.

Nonetheless, we argue that the advantages gained by having the possibility to directly connect source code with documentation of cross-cutting system aspects justify this additional effort.

8 RELATED WORK

API Documentation Tools

There are many tools that assist developers in writing or generating API documentation. This overview covers only a small selection.

Javadoc. Using the API documentation tool Javadoc, developers can refer to methods and classes in comments with the `ref` tag[1]. Programming environments such as IntelliJ IDEA provide tool support for the Javadoc syntax to facilitate the navigation along linked metaobjects and to update these paths when a referenced metaobject is changed. Furthermore, developers can easily generate hyperlinked documentation from Javadoc-conform source files using Doxgen[2]. However, Javadoc comments are associated with only a single metaobject. So to document concerns that are scattered over multiple metaobjects, comments have to be split or replicated. These comments would tangle the other comments that concern the main (and other) responsibilities of a metaobjects. Therefore, in practice, cross-cutting concerns are difficult to document using Javadoc. In contrast, Cross-cutting Commentary provides support to modularize scattered/tangled documentation and reference it from all participants.

Jadeite. The problem of finding the right API and determining how to use it is addressed by Jadeite [19]. Building on the Javadoc syntax, Jadeite displays commonly used classes based on usage statistics and automatically shows the most common ways to instantiate them. While Jadeite significantly speeds up the usage of unknown APIs, it does not handle cross-cutting concerns either.

SpyREST. Cisco's SpyREST tool [18] automatically generates web pages for REST API documentation from test code. Thereby, it improves the usage of APIs by providing examples similar to Jadeite, but lacks support for cross-cutting concerns.

Documentation Reuse Concepts

To the best of our knowledge, only few approaches aim to modularize comments.

Literate Programming. Childs and Sametinger [1] described a concept for documentation reuse utilizing literate programming. They transferred object-oriented concepts such as inheritance, visibility modifiers, abstraction, inclusion, and references to comments. Thereby, reuse of entire or partial comments is supported by adding specific language constructs for documentation. Hence, the definition of documentation differs from its compiled view. In contrast, Commentaries provide a direct manipulation view of documentation. Furthermore, referencing metaobjects in comments is not targeted by their approach.

Documentation Refactoring Toolkit. Luciv et al. [12] presented a method for finding fuzzy repetitions in documentations as well as a toolkit (DRT) for refactoring the documentation using an XML representation of comments defined by the DocBook and DocLine technologies. The DRT modifies the internal XML representation of comments while preserving its text representation such as PDF. Direct manipulation of comments is not facilitated by DRT.

Cross-cutting Concerns

Context-oriented Programming. Context-oriented programming (COP [5]) proposes dynamic layer composition for system adaptation at run-time. Layers are a modularity construct to help represent cross-cutting concerns at both the language level and in the runtime system. Inspired by COP layers, Commentaries are to help improve modularity of system documentation. So far, Commentaries do not affect system behavior.

Mylar. Mylar [8] improves navigating source code by encoding *degrees-of-interest* of program elements through monitoring the programmer's browsing activity. Views show such navigation traces to help extend and modify related code. Such tracing can complement the specification of participants in Commentaries. While Mylar relies on external natural-language documentation such as bug reports, our linked Commentaries integrate more fine-grained explanations into the software system.

Intentional Views. Many cross-cutting concerns follow explicable rules that programmers have in mind when talking or thinking about them. The Intentional Views [14] approach uses *logic predicates* to capture such rules in the context of documentation and verification, supplemented with short text comments [13]. Tools can then visualize predicate matches to support code navigation or guide through rule violations. We think that such predicates can enhance the way Commentaries specify participants. Our linked, formatted narratives provide more freedom for programmers to explain their intent and decisions. Yet, both formal and informal means of documentation serve different kinds of tasks and users.

UseCasePy. To reduce scattering and tangling of use cases and improve their traceability from analysis to design to implementation, UseCasePy [6] makes use cases first-class entities at development-, deployment, and run-time. Compared to UseCasePy, Commentaries are a more light-weight and less invasive approach to connecting executable and non-executable artifacts.

9 SUMMARY AND FUTURE WORK

We proposed *Commentary* as an approach to address the cross-cutting nature of documentation of implementation and runtime artifacts. Based on the assumption that concerns cross-cut these artifacts and the observation that their documentation follows their structure, we introduced Commentary as a light-weight, non-invasive system element to capture and narrate sets of related artifacts to support and improve system comprehension.

[1] `{@ref<classpath>#<methodname>}`
[2] www.doxygen.org

So far artifacts of interest were classes and methods. We will extend Commentary to involve other metaobjects like features, layers, and aspects or any object feasible. Furthermore we will explore the interaction between Commentary and (unit) tests, refactoring, topic and concept discovery and maintenance, and requirements engineering.

We will also investigate how Commentary can be developed into lightweight or even proper modules to better support modularity in less open and flexible programming and runtime environments.

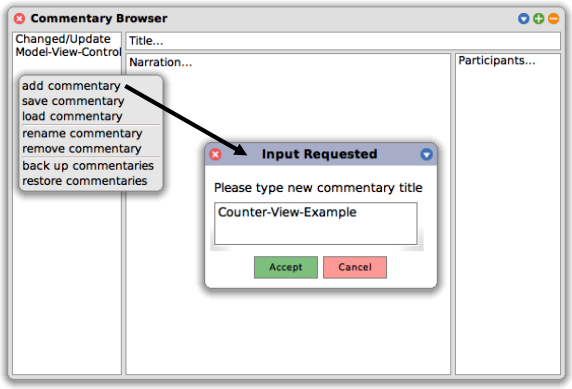
ACKNOWLEDGMENTS

We gratefully acknowledge the financial support of the HPI's Research School (https://hpi.de/forschung/research-school.html) and the HPI-Stanford Design Thinking Research Program (https://hpi.de/en/dtrp/).

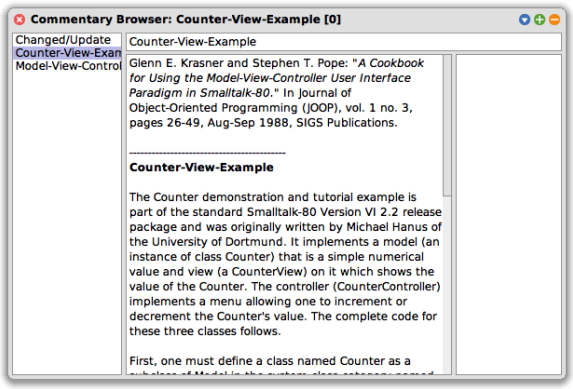
A APPLICATION SCENARIO

In this section, we show the interaction with the CommentaryBrowser to add the MVC-based "Counter-View-Example" [10] as a Commentary to the system. We interleave screenshots and text to explain most interactions step by step.

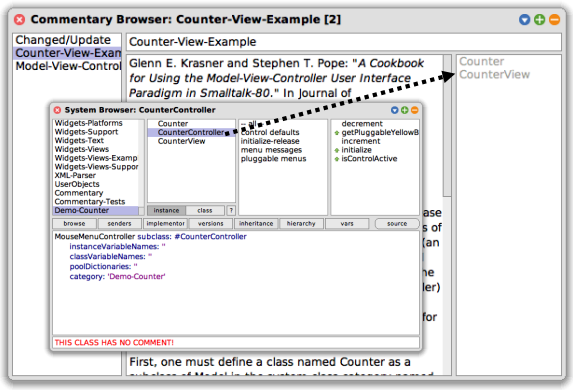
First, a new commentary is created via a context menu in the left pane of the browser listing all Commentaries available at the moment. We name our new Commentary "Counter-View-Example":



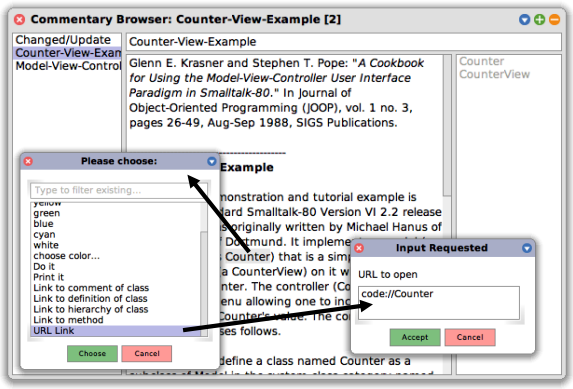
Now we copy and insert the text explaining the "Counter-View-Example" from the aforementioned paper [10] in the browser's middle pane, which looks and behaves like a regular text editor, providing means to change and format the text at hand:



Then, we start adding participants to our Commentary. Here, we use Squeak's drag and drop mechanism by grabbing the two classes `Counter` `CounterView` (we already implemented "Counter-View-Example" in our Squeak image) from the system browser and dropping them in the right pane of the CommentaryBrowser listing all participants of the selected Commentary:

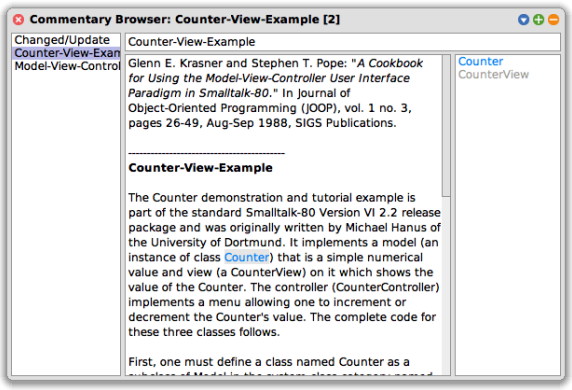


The gray color of participants in the right pane indicate that our narration (middle pane) does not (yet) refer to those participant. We now add such a reference from the text to participant `Counter` by selecting "Counter" (any selection would be possible) and associating this selection with a hyperlink (a `TextURL`) referring to `Counter`:

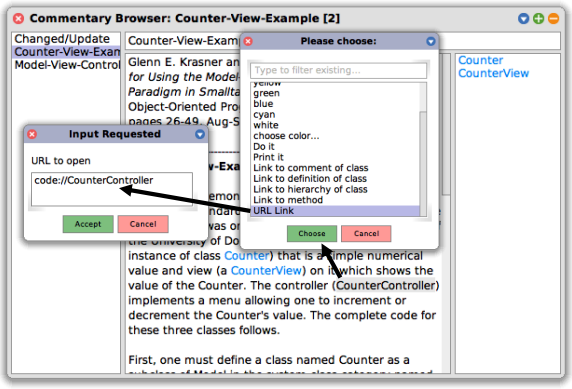


Now our narration refers to `Counter` at least once and so the color of `Counter` in the list of participants has been set to blue (the

color of our hyperlinks in the narration pane). We do the same for `CounterView` afterwards:

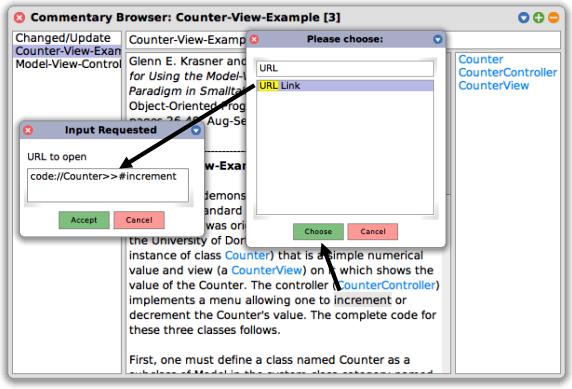


This time we add a new participant (`CounterController`) starting from the narration pane by adding a hyperlink from the "CounterController" text selection to `CounterController`:

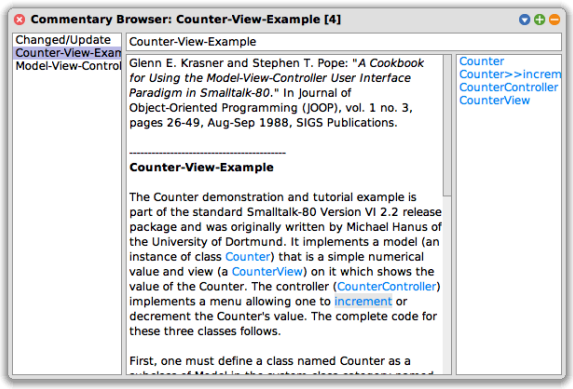


Since `CounterController` was not yet part of this Commentary, inserting this new hyperlink will be reflected in an extension of the list of participants on the right.

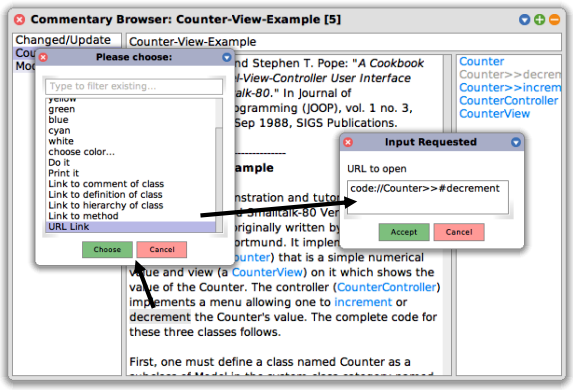
In the following, we select "increment" in our narration and link this selection to the method `Counter>>increment` in our implementation:



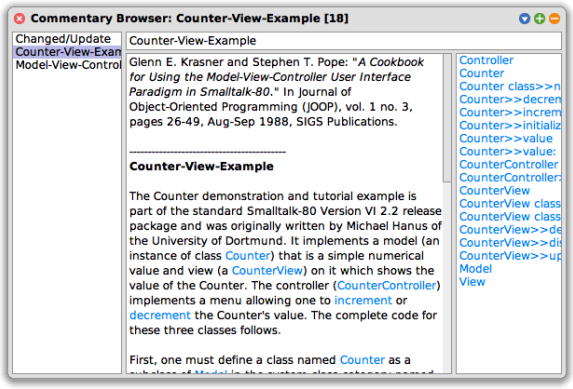
Since `Counter>>increment` was not yet mentioned in our list of participants, this list is updated as expected:



In a similar way, we now add `Counter>>decrement` by first dropping this method from a system browser into our commentary (`Counter>>decrement` will first appear in gray) and then hyperlink it from our narrative (`Counter>>decrement` now appears in blue in our list of participants):



What follows are a few more interactions with the CommentaryBrowser until there is no more to add, change, or remove for now:



Note that any part of a Commentary can be changed at any time in the process, and that both Commentaries and the objects they refer to can and will evolve over time.

REFERENCES

[1] Bart Childs and Johannes Sametinger. 1996. Literate Programming and Documentation Reuse. In *IEEE International Conference on Software Reuse (ICSR)*. 205–214. https://doi.org/10.1109/ICSR.1996.496128

[2] Squeak Community. 2018. Squeak ToolBuilder Framework. http://www.squeaksource.com/ToolBuilder/. Accessed: 2018-03-11.

[3] Squeak Community. 2018. Squeak/Smalltalk v5.1. http://files.squeak.org/5.1/Squeak5.1-16549-32bit/Squeak5.1-16549-32bit-All-in-One.zip. Accessed: 2018-03-11.

[4] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman, Boston, MA, USA.

[5] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. 2008. Context-Oriented Programming. *Journal of Object Technology (JOT)* 7, 3 (2008), 125–151.

[6] Robert Hirschfeld, Michael Perscheid, and Michael Haupt. 2011. Explicit Use-case Representation in Object-oriented Programming Languages. In *Dynamic Languages Symposium (DLS)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2047849.2047856

[7] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 318–326.

[8] Mik Kersten and Gail C. Murphy. 2005. Mylar: A Degree-of-interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD '05)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/1052898.1052912

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (ECOOP)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, 220–242.

[10] Glenn E. Krasner and Stephen T. Pope. 1988. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal on Object-Oriented Programming(JOOP)* 1, 3 (1988), 26–49.

[11] Dmitry Luciv, D.V. Koznov, George A. Chernishev, and Andrey N. Terekhov. 2017. Detecting Near Duplicates in Software Documentation. *arXiv preprint arXiv:1711.04705* (2017).

[12] D. V. Luciv, D. V. Koznov, H. A. Basit, and A. N. Terekhov. 2016. On Fuzzy Repetitions Detection in Documentation Reuse. *Programming and Computer Software* 42, 4 (01 Jul 2016), 216–224. https://doi.org/10.1134/S0361768816040046

[13] Kim Mens and Andy Kellens. 2005. Towards a framework for testing structural source-code regularities. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*. 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 (SEKE '02)*. ACM, New York, NY, USA, 289–296. https://doi.org/10.1145/568760.568812

[15] Patrick Mäder and Alexander Egyed. 2012. Assessing the Effect of Requirements Traceability for Software Maintenance. In *IEEE International Conference on Software Maintenance (ICSM)*. 171–180. https://doi.org/10.1109/ICSM.2012.6405269

[16] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming (ECOOP)*, Mehmet Akşit and Satoshi Matsuoka (Eds.). Springer Berlin Heidelberg, 419–443.

[17] Patrick Rempel and Patrick Mäder. 2017. Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *IEEE Transactions on Software Engineering* 43, 8 (Aug 2017), 777–797. https://doi.org/10.1109/TSE.2016.2622264

[18] S M Sohan, Craig Anslow, and Frank Maurer. 2017. Automated Example Oriented REST API Documentation at Cisco. In *International Conference on Software Engineering (ICSE): Software Engineering in Practice Track (SEIP) (ICSE-SEIP '17)*. IEEE Press, Piscataway, NJ, USA, 213–222. https://doi.org/10.1109/ICSE-SEIP.2017.11

[19] Jeffrey Stylos, Zizhuang Faulring, Jeffrey Yang, and Myers Brad A. 2009. Improving API documentation using API usage information. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 119–126. https://doi.org/10.1109/VLHCC.2009.5295283