Aiding Evolution with Concern-Oriented Guides

Barthélémy Dagenais*
School of Computer Science
McGill University
Montreal, QC, Canada
bart@cs.mcgill.ca

Harold Ossher
IBM T. J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10590
ossher@us.ibm.com

ABSTRACT

Program documentation is often incomplete and out of date due to its tediousness and perceived low value. This requires evolution tasks to be preceded by time-consuming exploration. In this paper, we explore a concern-oriented approach to documentation that focuses on the code artifacts and their relationships to make the process of creating and using program documentation more efficient. As opposed to traditional documents or tutorials, guides created using this approach are interactive, almost wordless and automatically maintain implementation examples. We also present the rationale and the architecture of Mismar, a toolset tightly integrated in the Eclipse environment and implementing this approach. Moreover, since program documentation involves different artifact types, Mismar was build from the ground up to be extensible, and to support artifacts written in multiple languages or modeling approaches.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement;

General Terms

Documentation, Design, Experimentation, Human Factors.

Keywords

Software evolution, Documentation, User guidance, Separation of concerns, Concern modeling, Aspect-Oriented Software Development

1. INTRODUCTION

This paper approaches the relationship between aspects and evolution from a somewhat atypical perspective. Rather than addressing how the use of aspect-oriented technology within a body of software can enhance evolution, it describes the use of concern-oriented guides to aid developers in performing traditional evolution tasks. Our thesis is that documentation for developers is usually sadly deficient, causing developers, especially those new to a project, to have to do a great deal of exploration before they can perform evolution tasks, and that concerns can help. Indeed, documentation is often lacking because it is tedious to create, and we think that concerns represent a more efficient and natural way of approaching this issue.

Consider, for example, open source projects. Successful ones foster collaboration and innovation while benefiting from a faster pace of development resulting from the great variety of contributions. Unfortunately, most projects, even mature ones such as Eclipse, cannot fully leverage the power of the open source community for various reasons, one of them being the lack of proper developer's documentation. Contributors often have to reverse engineer parts of the application to understand its architecture and design before beginning to actively collaborate. While some developers are more talented in this area than others, all suffer from this lack of documentation. On the other hand, the user community usually expects high quality and new features that require most of the developers' time!

In [4], we introduced a new approach to easily and rapidly creating usable developer's documentation. Using this approach, implemented by a toolset now named Mismar, developers can create a usable and interactive programmer's guide in a matter of minutes. Moreover, as readers follow the guide, the artifacts they create are recorded, and thus can be presented as implementation examples to other users. Such a guide can save contributors hours, enabling them to start contributing right away by extending or modifying the system. In [4], we mainly focused on our contribution for developers: we presented a concern-oriented approach in traditionally process-oriented documentation and we introduced the concept of active concerns. We also argue that this approach is better than traditional documentation because it tightly links documentation to code and infers and automates common tasks.

In this paper, we focus on the technical aspects and the software engineering idioms that make this toolset innovative.

2. CONCERN-ORIENTED APPROACH

2.1 The approach

Developers (the *creators*) are usually expected to provide documentation of their software system so other developers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Workshop LATE'07, March 12–13, 2007, Vancouver, BC, Canada.
Copyright 2007 ACM 1-59593-655-4/07/03…$5.00.

* This work was performed during an internship at the IBM T. J. Watson Research Center.

(the *consumers*), be it colleagues, community or clients, can extend and modify it. Unfortunately, by doing so, creators must shift their focus from building a usable application to creating a repeatable process. In other words, they must think in term of steps needed to be performed, and often struggle with their order and specification. The consumers, for their part, need to constantly switch from the documentation to their development environment and repeat common tasks, and they can easily skip important steps buried in tons of text. In addition, it is not always obvious how current implementations map to the documentation and how they could be used as examples to follow.

In [4], we proposed a toolset, tightly integrated within the Eclipse development environment [5], to leverage software developers' knowledge by simply asking them to point out elements in the software system that are relevant to a particular evolution task, such as classes, methods, files or even web pages. From these elements, a guide with an appropriate step for each type of element is created. For example, if the user chooses the XYZ class, an "Extend XYZ class" step is created. Hence, a developer can create a complete guide by simply dragging and dropping (or copying and pasting) elements from the system to the guide editor. Once a step is created, comments and references to other artifacts can be added to provide contextual information. References are added simply by dragging the referenced items and dropping them on the references section of the guide editor. Moreover, the user can easily reorder the steps or the references to reflect their priority. We referred to this approach as concern oriented since it allows the user to focus on software artifacts and their relationships instead of the steps or process. Indeed, concerns [14] are usually said to capture every element in a software system that is relevant to a particular point of interest.

The resulting guide is also tightly integrated into the development environment, providing interactive support to the end user. For example, when the user performs the "Extend XYZ class" step by double-clicking on it, the "Create new class" wizard in Eclipse is launched. The wizard is customized by the information provided by the guide author: the dialog title and description are taken from the step title and description fields and the wizard automatically proposes to extend the XYZ class. This is what we called active concerns, in contrast to traditionally passive concerns, since new behaviors are inferred from the elements of a concern.

Because terminology is scarce when talking about developer's documentation, we tried our best to select meaningful terms. Hence, a *guide* is a list of *steps* that need to be performed in order to complete a task. When a user follows a guide, s/he creates a *result*. When a user performs a *step* (see Figure 1), the *output* of the step is known as an *output example* or *implementation example*. A *reference* is a link to an artifact such as a class, a method, a file or a web page. A step can include a *hot reference*, which is a reference to an element on which the user needs to act; for example, if the hot reference of a step refers to class XYZ, it means that the user will need to use or extend this class. Finally, *step references* are references to any artifacts that might help the user in performing a particular step.

For documentation consumers, code examples have often been recognized as one of the most useful sources of information when trying to understand a program [9][11]. Our guide leverages the power of examples by recording every output resulting from a step execution. These outputs are then proposed as implementation examples when a user performs the same step in another context, i.e., when following the same guide to create another result.

The examples are retrieved dynamically each time the user looks at a guide, so there is no need to maintain them as in the case of usual documentation. Also, it is possible to find complete examples of the application of an entire guide quickly, by viewing the guide in a relationships view: it will show every result that was created from the guide. Loading a result gives the user access to a complete example presented with a familiar structure (i.e., the guide steps). Since implementation examples are simply references, the consumer or the guide creator can also add comments to help future consumers understand a particular feature.

Handling code examples in this way differs from current research that tries to infer code examples by mining source repositories [2][9]. One of the advantages is that it is easy to retrieve a complete example: given any artifact, the toolset can traverse all guide results to retrieve the one that produced this artifact, if any. We are also currently investigating how we could build examples by selecting any artifact and then using the guide structure to infer a complete example that was not originally created with the guide.

2.2 User experience

Programmer's guides, tutorials or cheat sheets usually contain steps to be performed, described with complete sentences. If the guide is relatively complex and involves more than a few steps, the process of creating and following the guide can become a tedious task simply because there is a lot of text to write or read. Such guides may include pictures such as screenshots or diagrams, but they are mainly text based. It is then hard for the reader to quickly capture important concepts and artifacts and to understand their relationships.

We argue that guides created by Mismar are worth a thousand words and embody a lot of knowledge carried by the user interface elements instead of by words. When creating a guide, the user typically does not have to enter a single word. Steps' titles and references' labels are inferred from the elements selected by the user: we go so far as to parse the clipboard to retrieve the title of a pasted web link. As another example, if the user selects the ABC interface, the guide will create an "Implement ABC interface" step and will use its javadoc as a draft for its description.

Relationships between artifacts and steps are carried by their proximity. When the user selects a step, the view is refreshed to show the step references and output examples. We prevent

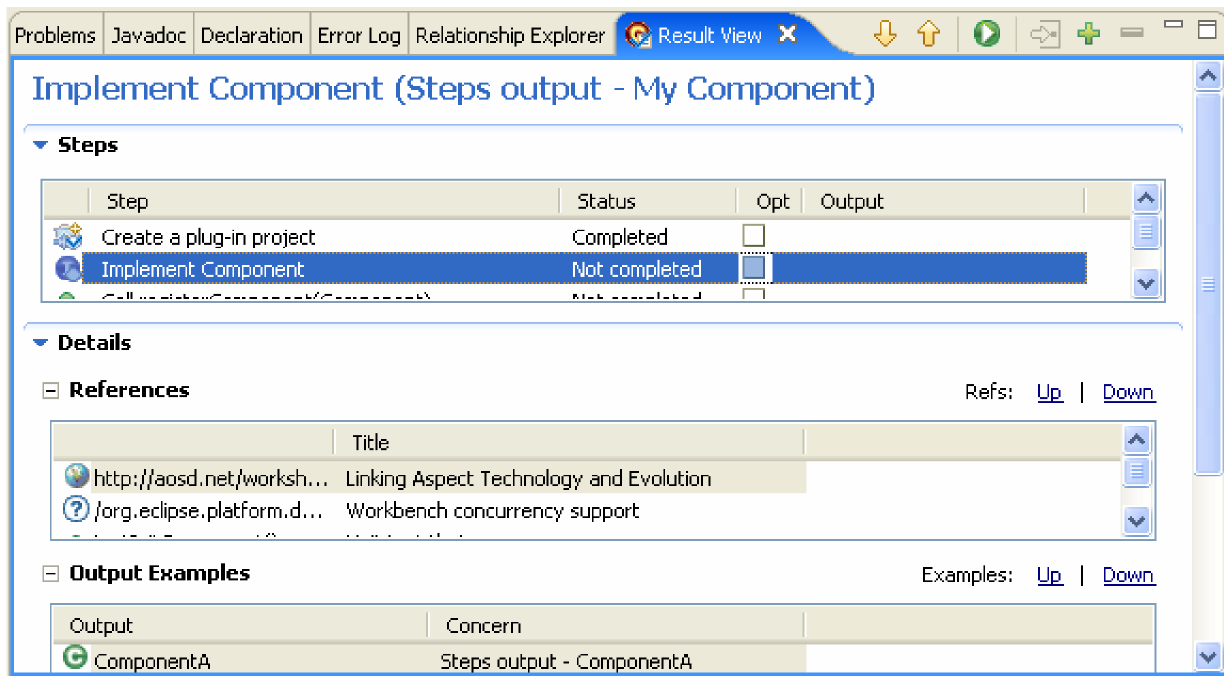

Figure 1. The Result view showing a step in a guide along with its references and output examples.

information overload [10] by showing only relevant information for a task and prevent user interface cluttering by using proximity instead of words or arrows to represent relationships.

Furthermore, icons, taken from the Eclipse environment, are used to represent the different types of steps and artifacts. The user, familiar with these icons, is then able to quickly assess the types of steps and elements involved when following the guide.

The order of the steps or references also carries important information, ranging from dependence to priority and relevance.

This all means that the guide author can focus only on selecting the important elements, without thinking about writing sentences with coherent format and language. Because step execution is assisted and sometimes fully automated, the guide author does not have to worry about explaining how to deal with the development environment. Thus, s/he stays in the same state of mind and uses an environment and metaphors similar to those s/he is used to when working on the code. For the end user, a quick look at a guide can reveal the magnitude of the task, the types of artifacts involved and the relevant references and examples that could help with the task.

Of course, the guide author can change titles and provide comments to clarify some steps or the relevance of particular artifacts, but if an artifact is already well named, this will probably carry more meaning and be more useful than a full-fledged description.

Documentation maintenance is also highly facilitated since few, if any, words are involved. The author can simply change a reference or drag and drop a new one to modify a guide: the icons, titles and labels are refreshed accordingly. Each time a guide result is loaded by the end user, it is refreshed to reflect the latest changes in the underlying guide. Any steps that were deleted from the guide are put at the bottom of the step list; steps are not deleted from existing results since they can still contain important information about implementation, but their priority is certainly lowered since they were deleted from the guide.

2.3 Limitations

There are currently two main limitations in Mismar. First, if the underlying software changes (e.g., through refactoring), the references are invalidated and not usable any more. Although this remains an area for future research, this risk is mitigated by Mismar's ease of use: it is easy and fast to create or rename a step or reference and copy all of its linked references and description. The other limitation is that we limit the number of tasks and outputs per step to one. For example, if an interface is selected as a hot reference, the inferred behavior is to ask the user to implement it: it is not currently possible to dynamically select a different behavior (e.g., create an instance of this interface at run time). In addition, even if the user decides to split the implementation into multiple classes using inheritance or composition, only one of those classes will be recorded as an output. Although we hope to introduce these features in later versions of Mismar, it is still possible to get around most cases. E.g., if you want to create an instance of an interface, it may be because you want to use it as a parameter to another method call. You can then simply select the method call as a step and put the interface in the step references.

3. ARCHITECTURAL OVERVIEW

3.1 A concern model

The Mismar architecture provides opportunities for other researchers by being extensible and offering an underlying model ready to be mined and analyzed.

The model behind this toolset was inspired by prior work aimed at defining an extensible and customizable model for concerns [7][13]. Like more recent tools such as ConcernMapper [12] and Mylar [10], our toolset hides the concrete model from the end-user to some extent, but makes it available for third-party tools. The end user is thus not aware that by creating and using a guide, s/he is building a complete concern model that can then be analyzed and transformed.

The concern model is implemented using the Eclipse Modeling Framework (EMF) [6], which provides three major advantages: it is a subset of the standardized OMG Meta Object Facility, it provides transparent XML persistence, and it offers runtime data modification awareness, i.e., it is possible to be notified of any changes to any attributes and objects specified in the model.

The basis of the model is the concern class that embodies an intension. Relationships such as "is guided by" or "is used by" can exist between two concerns. An intension specifies a set of artifacts. The structure of an intension can range from an internal set of explicit references, to a query. The intension is responsible for computing the concern's extension (the actual set of artifacts). The client program is usually not aware of the structure of the intension: it only receives the extension. In section 3.2, we present how we can define new types of elements, including intensions, which would allow support for any types of query language, such as JQuery [8].

Everything in our toolset is related to a concern. As illustrated in Figure 2, a guide is a concern with an intension that contains a list of steps. A step is a concern with an intension that contains a reference to the step information, a (hot) reference to its main artifact (e.g., the XYZ Class to be extended), and a reference to a concern, which, in turn, contains a list of step references (such as a eclipse help page). When a step is performed, its output is encapsulated in a concern that has an "is a result of" relationship to the step concern. One can thus access any element in a generic way by following relationships and querying intensions.

Presenting results as implementation examples is only one of many possible usages of a concern model.

3.2 Extensibility

Extensibility was the top priority during the design of the toolset. Indeed, we initially wanted to support any type of reference (i.e., dragging and dropping any type of element, even from third party tools or plug-ins), but this capability soon extended to the possibility of customizing the interactive support and being able to create new types of concerns, intensions and steps.

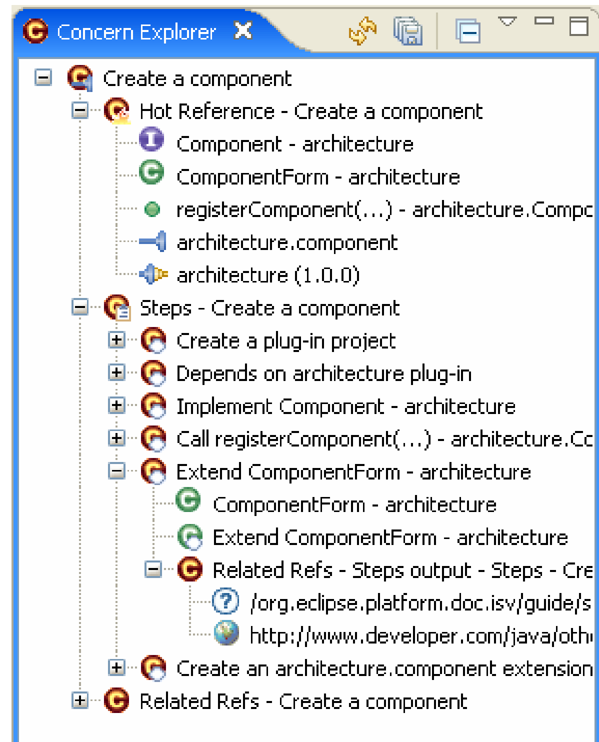

Figure 2. The Concern Explorer showing the underlying concern model.

Extensibility is provided through the extension point mechanism in Eclipse that enables any component (plug-in or bundle) to define ways for other components to extend it. Our toolset mainly provides two extension points named type and extender. To create a new type of element, the developer must create a type extension and specify a unique identifier, an icon representing the type and the kind of element that is extended (e.g., a new type of concern or a new type of reference). Then, the developer needs to specify extenders, Java classes implementing interfaces that are called by the toolset in various contexts. The model is also extensible: it is possible to define a new kind of element that has its own, new internal structure.

In summary, not only can you define new types of elements that already exist, but you can also define new kinds of extensible elements. For example, we are now working on integrating AspectJ elements and also on defining new types of concerns, such as code walkthrough, that require different structures and UI interactions. Our toolset can leverage and integrate with almost any third party tools in Eclipse, and hence support other programming languages, such as C++ or PHP.

4. RELATED WORK

The internal design of our toolset was driven by past work on concern models [7][13], and the ease of use and the drag and drop capability were inspired by ConcernMapper [12]. Although ConcernMapper allows users to group code elements in a "concern", it is limited to Java methods and

fields, it does not provide any behavior or other communication capabilities such as comments and references and only offers one structure (unordered set of elements), whereas Mimsar offers richer structures. The idea of building an implicit concern model and supporting any kind of elements is strongly related to Mylar [10]. Interactive documentation is also not a new topic: Cheatsheets in Eclipse [3] and the DocWizard project [1] can perform some steps automatically, but these actions must be manually selected by the author: they are not inferred from the guide content, requiring the author to focus on process. Also, these solutions lack the rich contextual information Mismar provides with its references and automatic implementation examples.

5. CONCLUSION

With Mismar, software developers get the chance to easily and rapidly document their systems, which could encourage collaboration and contribution and thus ease and speed evolution. In this paper, we presented the rationale and the extensibility mechanisms behind the toolset. Using a concern-oriented approach and a concern model led to interesting features. For example, offering a wordless guide was a natural way of representing and dealing with such a model. We are hoping to release Mismar with an academia-friendly license in the near future.

As an early step toward validation, and to ensure that our approach scales, we designed a complete guide to the complex task of creating an Eclipse text editor. We compared it with usual textual tutorials available on the Internet, including one written by one of the authors. The time involved to create and follow the guide was reduced by at least an order of magnitude. More complete validation remains an important area for future research.

Another key theme for future work is to support automation of various sorts: updating of guides as the underlying software changes, finding existing examples that were created by hand and tying them to the guide as implementation examples, and constructing a draft guide automatically from user actions when performing a task.

6. ACKNOWLEDGMENTS

We thank Steve Abrams and the anonymous reviewers for many insightful comments and discussions.

7. REFERENCES

[1] Bergman, L., Castelli V., Lau T., Oblinger D. “DocWizards: a system for authoring follow-me documentation wizards.” In Proceedings of the 18th annual ACM symposium on User interface software and technology, pages 191-200, 2005.

[2] Bruch M., Schäfer T., Mezini M., FrUIT: IDE Support for Framework Understanding, Proceedings of the Eclipse Technology Exchange at OOPSLA, 2006.

[3] “Building cheat sheets in Eclipse V3.2.” http://www-128.ibm.com/developerworks/library/os-ecl-cheatsheets/

[4] Dagenais, B., Ossher H., Guidance Through Active Concerns, Proceedings of the Eclipse Technology Exchange at OOPSLA, 2006.

[5] Eclipse. http://www.eclipse.org/

[6] EMF. http://www.eclipse.org/emf/

[7] Harrison W., Ossher H., Sutton S., Tarr P., Concern modeling in the concern manipulation environment, In Proceedings of the workshop on Modeling and analysis of concerns in software, 2005

[8] Janzen, D. and De Volder, K. Navigating and querying code without getting lost. In Proceedings of Aspect Oriented Software Development, Boston, 2003, 178-187.

[9] Holmes, R and Murphy G. C. Using structural context to recommend source code examples. In Proceedings of the International Conference on Software Engineering, pages 117–125. ACM Press, 2005.

[10] Kersten, M. and Murphy G. C. Mylar: a degree-of-interest model for IDEs. In Proceedings of the 4th Conference on Aspect-Oriented Software Development, pages 159-168, 2005.

[11] Nykaza, J., Messinger R., Boehme, F. Norman C. L., Mace M., Gordon M. “What programmers really want: results of a needs assessment for SDK documentation.” In Proceedings of the 20th annual international conference on Computer documentation. Pages 133-141, 2002.

[12] Robillard, M. and Weigand-Warr, F. ConcernMapper: Simple View-Based Separation of Scattered Concerns. In Proceedings of the Eclipse Technology Exchange at OOPSLA, 2005.

[13] Robillard, M. P. Representing Concerns in Source Code. Ph.D. Thesis. Department of Computer Science, University of British Columbia. November 2003

[14] Tarr, P., Ossher, H., Harrison, W. and Sutton, Jr., S. M., “N degrees of separation: Multi-dimensional separation of concerns.” In Proceedings of the 21st International Conference on Software Engineering (ICSE '99), 107–119, IEEE, May 1999.