Guidance through Active Concerns

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
Producing usable documentation has always been a tedious task, and even communicating important knowledge about a system among collaborators is difficult. This paper describes an approach to creating documentation in the form of *guides*, which encapsulate passive information about important tasks along with active steps to be followed. The approach is concern-based, and introduces active steps into traditionally passive concerns. A developer can begin by creating a concern that identifies elements of importance in the context of a task, which, we believe, is easier and more natural than trying to formulate a process up front. S/he can then easily create a guide to the task based on this concern, and export it. Other developers can follow the guide, and, as they do so, their results are recorded as examples for future reference. As an early step towards validation, we created a guide for the complex task of creating an Eclipse editor.

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
User guidance, Separation of concerns, concern modeling, aspect-oriented software development

1. INTRODUCTION
Producing usable documentation has always been a tedious task and something developers have limited success with. Consider the following not-so-fictional scenario.

Being the brightest developer on your team, your boss asks you to explore the possibility of leveraging the new CoolGizmoUltra feature in Eclipse [4] in order to build multiple instances of SuperHotWidgets. That would be a wonderful assignment if it were not for the fine print in your contract that you did not read: you'll need to document your findings in order to teach your colleagues and future users how to create Widgets from Gizmos.

You open the CoolGizmoUltra feature in Eclipse, and begin your journey. As you look at the scarce documentation, you find that you need to make your project depend on a subset of the plug-ins offered by this feature and to create an extension from a particular extension point. Examining the extension point schema reveals that you have to implement the CoolGizmo java interface. After some reading, you find some Javadoc that gives an example of how to code the class required by the extension.

As you continue your journey, you want to build a GUI for your example Widget, but do not quite remember how to create a fully-functional TreeTable. You open your web browser and navigate to an SWT web tutorial to refresh your memory. After some work, you're ready to test and run your prototype. Unfortunately, even if you followed all the documentation you found, the application fails at runtime, throwing a UndocumentedAnnoyingException.

Taking a deep breath and clearing your mind, you load the supplied implementation example into your Eclipse development environment and launch the debugger. At some point in the execution (after following 1000 steps and an anger management problem), you realize that the registerYourGizmo() method is called and that the UndocumentedAnnoyingException would be thrown some time later if you forgot to code the call . You make a mental note of this and continue your exploration of the example, wary of other little undocumented quirks like this.

After working an entire day on this task, you try to assemble all your notes and newly-acquired knowledge in some form of readable and understandable documentation for your colleagues. You begin to struggle to make a usable process, or recipe. Which step should you start with? How do you emphasize the important and useful parts of the design? Where should you put references to important information? They might be useful for some developers, but you cannot separate two steps with a page full of references! As time goes on, you also find yourself constantly switching between your IDE and your guide, crossing your fingers that you did not forget something important in the process. Worse, two days later, just before submitting the documentation, you realize that the CoolGizmoUltra feature, in active development, just changed and you need to update the guide.

Such situations often arise. This one might be seen as fictional only because developers are often confronted with even more complex scenarios! Even with our powerful development tools and environments, such as Eclipse, developers are not well equipped to explore complex systems or—the focus of this paper—to communicate their newly-acquired knowledge to their

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

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.
OOPSLA 2006 Eclipse Technology Exchange (ETX), October 22-23, 2006, Portland, OR, USA.
Copyright 2006 ACM 1-58113-000-0/00/0004…$5.00.

colleagues and users. Moreover, current tools do not adequately leverage the information foraged by developers to actively support future users in using, modifying or extending their software artifacts. In this paper, we present a toolset, tightly integrated with the Eclipse platform, that addresses these problems.

2. RELATED WORK

There are currently existing tools that focus on helping developers to create usable documentation for users. One of these tools is the *cheat sheet* mechanism [3] in Eclipse. A cheat sheet is a form of tutorial that teaches the user how to interact with the user interface in order to accomplish a certain task. For example, there is a cheat sheet explaining how to use the Eclipse environment to check out a project from CVS. Cheat sheets can also include actions, such as a hyperlink that opens a dialog or launches a wizard. The main problem with cheat sheets is that they are cumbersome to create: you must write an XML file describing the tutorial and a java class for each action.

DocWizards [2] is another tool that teaches and guides the user in his interaction with the Eclipse user interface. Its main advantage over cheat sheets is the way a tutorial is created: DocWizards records your interactions with the UI, enabling other users to later replay the scenario to accomplish a similar task. It also allows some editing and generalization after capture.

Those two tools are best suited to teaching and helping to perform UI interactions, but could be extended to guide the user in modifying software artifacts, with some important limitations. Furthermore, these tools are what we call *process oriented* since the author needs to think in terms of process when using them: the steps the user has to perform as opposed to the artifacts that are important in the software.

Finally, there are tools such as ConcernMapper [8] and JetEye [5] that help developers in their exploration. While the former is focused on gathering and organizing Java elements such as methods and fields, the latter mainly deals with web elements such as hyperlinks and images. Those tools only deal with one kind of artifacts, offer limited communication capability and do not provide any support for defining processes or guides.

3. SOLUTION

3.1 An active concern approach

We see the problems of exploring a system, communicating the findings and creating a process or a guide from these results as highly related. Furthermore, we strongly believe that information contained in the findings can be leveraged to provide interactive support to a user following a guide created from those findings.

In section 2, we presented some existing tools that were mainly process oriented or that focused only on helping developers to explore complex systems. We argue that it is quicker and easier to create usable documentation when thinking in terms of related artifacts rather than process. In other words, it is more natural for a developer to point out the important artifacts in a system than to think about the steps required to modify it.

Concerns [9] seem to correctly capture the intent of this approach: they encapsulate elements of different types that are of interest in a particular context. There are different types of concerns ranging from application-specific such as features to user-oriented such as performance. More explicitly, we see the exploration findings as the expression of a concern and the basis for a process or a guide. The addition of actions to concerns, moving from sets of elements to guides, is a key contribution of this work. We also use concerns to record and relate the elements produced when following guides.

3.2 The toolset

Our toolset is tightly integrated with Eclipse, especially the Java Development Toolkit. It supports the user in exploring a system, creating a guide to use/modify/extend the system, and following a guide.

The toolset is composed of four views and an editor, partially shown in Figure 1. The *Concern Explorer* view is used to represent the different concerns present in a user workspace. A user can create different types of concerns in this view and interact with them by adding, removing or restructuring references to elements. From this view, a user can activate a concern to indicate that s/he intends to work primarily in the associated area of interest. S/he can still manipulate other concerns, but some operations are restricted to the active concern.

The *Relationship* view shows the relationships between the active concern and other concerns. For example, if the user is currently following a guide, this view will show the existence of a relationship between the guide and the result being produced. The user can navigate among concerns by double-clicking on a concern. This activates the new concern and refreshes the relationship view to present the new relationships.

The *References* view shows information about the elements contained in a concern. The user can link this view with other views or simply drop or paste a concern into it to see its content.

The *Guide Editor* is a form-based editor used to create a guide. It allows the user to create steps from important artifacts, and to add related references to each step. The user can also add comments on each step or reference. Table 1 shows the supported references and steps. New types of references or steps can easily be added by creating an extension and providing a simple adapter.

The *Guide Result* view is displayed when a user wants to follow a guide in order to prodcuce a result, usually an extension or modification of the system. This view contains every step the user has to perform with a description and references for each step. It launches appropriate interactive support tools for each step.

3.3 Original scenario revisited

3.3.1 Exploring

Let's now revisit the scenario presented in the introduction, but this time we generously give to our bright developer an exclusive and unlimited license to our toolset.

When the developer receives his assignment, he creates in the Concern Explorer a new concern that he calls "New Gizmos". He begins by looking at the Eclipse Help Documentation. As he reads it, he copies and pastes the relevant links into his concern. As he discovers the dependencies, he adds references to the required plug-ins and extension point. Then he navigates the source code and drags and drops important classes and interfaces that he will need to use or extend in the concern. He also adds some comments to the references in order to clarify their role.

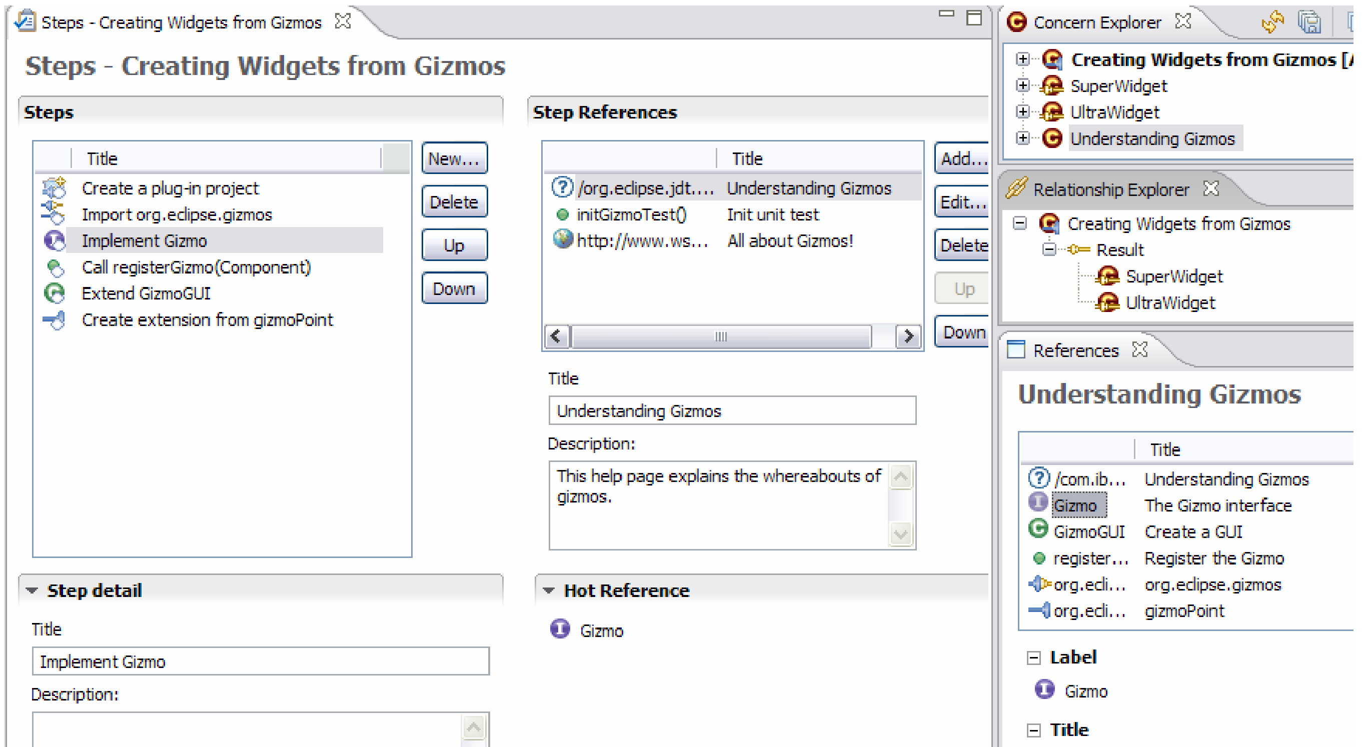

Figure 1. A quick glance at the various views and editors

Reference type	Step type	Behavior
-	Create a java/plug-in project	New Java/Plug-in Project Wizard
-	Generic step	By hand
Class or interface	Extend/Implement class/interface	New Class Wizard
Method	Call method	By hand
Plug-in	Add plug-in import	Add plug-in import in plug-in manifest
Extension point	Create extension	Create extension in plug-in manifest
Resources/Other java elements	Generic hot reference step	By hand
Eclipse help page / Web pages	Generic hot reference step	By hand

Table 1. Supported References and Steps

When he finds a suitable web tutorial, he copies and pastes the link into his concern. The reference title is retrieved from the hyperlink and the developer adds some comments to explain how this tutorial can be useful.

During the debugging of the implementation example, the developer drags and drops the important method into his concern. He also reorders the reference to put the newly-added method just after the interface reference.

When the developer has finished, he exports his concern and sends it to his colleagues. They will then be able to quickly understand the CoolGizmoUltra feature and will be somehow guided by the order of the references and their comments. Figure 1 shows a quick overview of his findings in the References view.

3.3.2 Creating a guide

Some time later, the valorous developer wants to start documenting his findings in a way that will help his colleagues and other users creating widgets from gizmos. He starts by opening his "New Gizmos" concern. From now on, he mainly has to decide whether a reference represents a step to follow or something that will help a user in performing a step.

He begins by creating a new guide in the Concern Explorer that he names "Creating a Widget". The Guide Editor is opened. Then, he selects the plug-in dependency reference from his first concern and promotes it to a *hot reference* by selecting the corresponding option in the context menu. A hot reference refers to an artifact on which a developer will work when following a step of the guide. The guide editor then asks the developer if he wants to create a step from that reference. The developer agrees and a new step, "Add this plug-in dependency", is created in the guide. He moves on and does the same for the classes, the interfaces and the extension point present in his first concern. Each reference creates an appropriate step such as "Create an extension from extension point X", "Extend class Y" and "Implement interface Z".

When the developer encounters references (such as a web page, a JUnit test or an Eclipse help page) that might be helpful to other users but that do not represent a step, he simply drags and drops or copies and pastes those references into the *Step References* section of the appropriate step in the Guide Editor. For each step and reference, the developer can enter some comments in the description text box.

If the developer finds other elements that are important but were not part of his original concern, he can simply select them in

Eclipse (e.g., in the Package Explorer) and promote them to hot references to create steps. He can also drag and drop (or copy and paste) any other elements into the Step References section.

Alternatively, the user can create a step by pressing the "New" button in the Guide editor. Every step that can be created from selecting a hot reference can also be created this way; sometimes it is quicker to use one of the available wizards to create such steps. Moreover, some steps cannot be created from hot references. For example, the "Create a new Plug-in Project" step can only be created using the "New" button: it does not make sense to select a particular plug-in project to create this kind of step.

Once the guide is finished, the developer can export it, like his first concern, and distribute it to his colleagues. The resulting guide is shown in Figure 1.

3.3.3 Following a guide

A user is now ready to create a widget from the gizmos. He imports the guide and any accompanying examples.

In the Concern Explorer, the user right-clicks on the guide and selects the context menu option "Create result". He decides to name his result, "SuperWidget". This action creates, as its name implies, a result and opens the Guide Result view.

In this view, the user can see all the steps that he needs to perform, with their descriptions. When he selects a step, he can also see the associated references with their comments. Double-clicking on a reference will open it in the Eclipse environment (e.g.,:a Java class in the Java editor) or in a web browser.

For each step, the user has three choices: he can skip the step, he can perform the step, or he can select the output for the step. Let's take the example of the step "Extend class XYZ". If the user chooses to perform this step by double-clicking on it, a "New class" wizard is launched, initialized with the step information: the dialog title is the step title, the dialog description is the step description and the class XYZ is added to the "Superclass" field. On the other hand, it is perfectly plausible that the user has already created a class that extends XYZ in another project. He can then select this class as the output for this step.

As the user progresses, the output of each step is recorded. This allows future users to get relevant examples for each step. For example, when a user performs the step "Extend class XYZ" in future, he will be able to see in the Guide Result view that this step was performed when creating "SuperWidget," and that a particular class was created as a result. By double-clicking on the example, the class will open in a Java editor.

4. DISCUSSION

In the last section, we demonstrated how a developer could use our toolset to explore a system and create usable documentation that actively guides other users in using, modifying or extending software artifacts. This toolset is greatly inspired by concerns and therefore allows the user to easily select important and related elements according to a particular focus.

4.1 Seamless integration with Eclipse

Our toolset is tightly integrated with Eclipse: in fact, some people who assisted with our demos could not differentiate functions contributed by our toolset from those provided by Eclipse.

The integration with the development environment is essential to bring concerns to a new level, i.e., active concerns. It is by inferring the kind of support that the user will need for a step involving a particular type of reference and by selecting the appropriate mechanism in the environment that concerns can be truly active. This interaction also works the other way: the existing support mechanisms are enhanced by the information contained in a concern. For example, the new class wizard was initialized with the reference information.

Another benefit of this integration is that the user, as the guide author or as the guide follower, does not constantly have to switch between the documentation and his development environment. Even the references that help him understand new concepts or the task at hand are mostly integrated with his environment.

4.2 Unobtrusive

Developers have access to a phenomenal variety of tools, but only use a few, for numerous reasons (e.g., the limited size of short term memory). Any new tool is in direct competition with hundreds of other tools and must provide great value in order to be considered. This barrier of entry can be lowered by presenting high usability and having a low learning curve.

We argue that our toolset fulfills these requirements by introducing only a few new concepts and leveraging existing tools and techniques, such as drag and drop, copy and paste and existing dialogs and wizards, with which the user is already familiar. There are also different ways of achieving the same goal to suits the different tastes and habits of developers. For example, it is possible to create a step through a wizard or simply by selecting, dropping or pasting an element. So, even if a user has forgotten how to create a step or add a reference, it will be easy to deduce how to do it.

The toolset does not impose any process or way of thinking or working. It is resilient to errors and changes, in that it allows the user to easily reorganize references and concerns. The user is free to start with general concerns and can promote them to a guide if it suits his needs. Of course, he can also start right away with building a guide.

Another interesting feature is the automatic updating of results when a guide is changed. If the user adds, modifies or deletes steps in a guide, the results are modified accordingly when they are loaded: new steps are added and deleted steps in the guide are moved to the end of the step list in the existing results (they might contain important contributions so they are not automatically deleted). This enables the guide author to modify the guide without breaking existing results or examples.

Because this toolset is unobtrusive, it silently helps the developer in structuring his thoughts. At the end of this structuring task, the toolset not only allows him to produce a structured guide, but it also provides interactive support and a basis for further software development activities.

4.3 Agile Documentation

Agile practitioners often stress the fact that documentation should be kept to a minimum in a project [1]. Indeed, good and useful documentation is time-consuming to write, hard to maintain and usually does not provide great added value to the most important artifact: the working software.

Practices such as pair-programming and test-driven development (or softer practices derived from these) tend to produce clearer code and comprehensive test suits that are self-documenting. It is often argued that clear code and working examples, including tests, are often the best documentation a developer can get. For example, a study on the relevance of different types of documentation resulted in eight recommendations, the first four of them being related to source code examples [7]. Moreover, those are the first artifacts that get updated, so they always reflect the most current state of the working software. Our toolset enables developers to identify these important elements easily, quantify their relevance by ordering them, provide some comments about them, and communicate the resulting concern.

For example, JUnit test references for each step can easily be recorded. The number of tests in the system can be daunting, and organization helps the developer to focus on the relevant ones. The tests will likely be updated as the software evolves, but since the concern contains only references, navigation will lead to the latest version. A current limitation of our toolset is that references can become invalid over time and then need to be manually updated. Support for updating concerns and guides in the face of structural changes to the software remains an area for future work.

Documentation is often required for people external to the agile team and is also essential for distributed teams such as in open source projects. For example, a developer can be in charge of some part of the architecture and as soon as it is usable, he might want other contributors to start building on it. Again, documentation must be light, because the system is constantly evolving and it is better to spend most of the developers' time writing tests and features than updating the documentation.

As a guide is being used by developers, relevant implementation examples are recorded and automatically contribute to the documentation. Hence, this is really in the spirit of agile documentation: documentation that is as near as possible to the source code, light, quick to create, and cheap (mostly free) to maintain.

5. CONCLUSION AND FUTURE WORK

To quote Scott Ambler on agile documentation, "Software developers have the knowledge, technical writer have the skill" [1]. Since most developers can't afford to have their own technical writers producing the documentation in parallel with their work, they must often rely only on their knowledge.

Our toolset leverages that knowledge and makes it easy to create usable, integrated and interactive documentation. This is mainly done by taking advantage of concerns, since they conceptually represent the situation at hand.

Concerns are usually passive in the sense that they are representations that do not add any behavior to their environment (though valuable activities can be performed on or using them, such as relationships discovery or user interface filtering [6]). With our tool, we bring concerns to a new level by incorporating active steps that guide developers in their tasks.

As an early step towards validation, we created a guide for a complex task: creating a text editor in Eclipse. One of the authors had already written a complete tutorial on the matter using Microsoft Word and could easily compare the effectiveness of both approaches. The biggest advantages of our toolset were the natural way of adding references for each step and the ease with which an implementation example was created from the guide. In this case, the example was the Java editor in the Eclipse SDK examples bundle, and it also allowed the author to validate the guide.

We intend to perform usability tests in future to validate and improve the toolset. We also want to investigate the possibility of creating external documentation such as XML, HTML or Microsoft Word documents from guides, for users who prefer to read traditional documentation. Finally, we also want to explore the integration with repository or versioning systems. This integration could enable guide authors and users to deal more effectively with changes to concerns, guides and the underlying software.

6. ACKNOWLEDGMENTS

We would like to thank Steve Abrams for several insightful discussions.

7. REFERENCES

[1] Ambler, S.W., "Agile Documentation: Strategies for Agile Software Development." http://www.agilemodeling.com/essays/agileDocumentation.htm

[2] 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.

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

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

[5] Jeteye. http://www.jeteye.com/

[6] 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.

[7] 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.

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

[9] 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.