Tie Code And Questions: a Reengineering Pattern

Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz

Serge.Demeyer@uia.ua.ac.be

{ducasse,oscar}@iam.unibe.ch

Abstract. The reengineering pattern presented in this paper shows how you can support your understanding during system reengineering by linking your questions or information about the code in the code itself.

1. Introduction

Legacy systems are not limited to the procedural paradigm and languages like Cobol. Even if object-oriented paradigm promised the building of more flexiblesystems and the ease in their evolution, nowadays object-oriented legacy systems exist in C++, Smalltalk or Java. These legacy systems need to be reengineered to meet new requirements. The goal of the Famoos Esprit project was to support the evolution of such a object-oriented legacy systems towards frameworks.

In this context, we used patterns as a way to record reengineering expertise. We wrote reverse engineering patterns that record how to extract information of the legacy systems from the code, the organization or the people [Deme99n] and reengineering patterns that present how code can be transformed to support new requirements, to be more flexible or to simply follow object-oriented design [Duca99c]. Tie Code and Questions is a third kind of reengineering pattern, it is not only applicable during the reverse engineering phase but can also be used during the reengineering of a software system.

Acknowledgements. This work has been funded by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015 as well as by the European Union under the ESPRIT program Project no. 21975. We would like to thanks the attendees of the SCG internal workshop writer at the Universitu of Bern: Serge Demeyer, Pietro Malorgio, Tamar Richner,Matthias Rieger and Sander Tichelaar.

Tie Code and Questions

Intent — Keep all your questions and answers about the code you are reengineering synchronized with the code by storing them directly in the source files.

Problem

You want to *keep track of your understanding* about a piece of code and the questions that you have, keep these *remarks synchronized with the code* during its future evolution, and *share them* with the other members of your team.

This is a hard problem because:

- Understanding code you do not write is really difficult due to the fact that we have to understand symbolic information and models that can be represented in various ways and styles.
- You want to record your understanding or questions about a piece of code *as soon as* they appear. Otherwise you will forget them because at the time they may seem too simple, or because later you will be concentrating on another part of the code.
- You need to record your understanding *as close as* possible to the code element it refers to avoid to spend time describing the context of the problem you have. Moreover the information you are interested in only makes sense to a maintainer who has the code available.
- You want to write your understanding in a simple way with your favourite code browser tools, so recording the information in a design document is not a practical solution.
- You want to *share* the information you found principally with other team members in the *future* because this is likely that you or your team will pass over the same piece of code or you may get into the code later and you do not want to forget what you learned. However you only want to have this information available when you will read the same piece of code and you do not want to spend meeting time reporting detailed information to unconcerned people.

Context

You are reverse engineering the functionality of an application. You may have applied *Refactor to Understand* and started to refactor the code when you identified *Duplicated Code*. You may also have used *Step Through The Code* to understand a functionality. However, as you did not develop the original code, there are many design decisions which are not clear to you, and numerous questions arise as you proceed.

Solution

While you are working on the code annotate it directly and immediately with the questions you are facing.

General Hints

- Use conventions to identify your annotations. In a team context, include, for example, the initials of the developer that made the comments. This way you can easily query them.
- Never write your annotations in a language different from that in which the source code is programmed (English in most cases). Otherwise, you create a different context and force the reader to switch between them.
- When you discover the answer to any one of your questions, immediately update the annotation for the benefit of future readers.

Annotations

- Record your annotations by using the commenting convention of the programming language (referred to as comment-based annotations). Some programming environments, like that of Eiffel, allow you to specify different levels of visibility for your comments and your code; where possible, assign a private scope to your comments so that clients cannot see the annotations.
- If you are working with an IDE where you can query method senders, use special methods dedicated to the annotations (referred to as *method-based* annotations). These methods take a single annotation string as an argument, and typically have an empty implementation. You can then use the querying and browsing facilities of your IDE to identify classes containing annotations, or specific locations where the annotations occur, without the need for any additional tools or special text pre-processing.

Discussion

The comment-based approach is better-suited for a text-based environment like the e-tags functionality supported by emacs [etags man page]. The method-based approach is better suited for an integrated environment like that of Smalltalk or Sniff+ that supports querying of method implementors or senders.

The less you change the code, the less likely it is that you will introduce errors. This makes the comment-based version safer than the method-based version. However using a method-based approach allows you to easily produce a log file.

Keeping vs Removing the Annotations

What options do you have when you want to release a new version?

- Comment-based annotations. If your client does not have to see the code, then you can leave the comment-based annotations in the code. The Eiffel environment provides several views of the code that are especially useful in such situations.
- Method-based annotations. A good compiler will not generate any code for unambiguous calls to messages with empty bodies. Nevertheless, if performance is an issue, and the overhead of the empty calls cannot be tolerated for the system you are working on, choose the comment-based approach or convert the method calls into comments using, for example, a perl script.

In either case you should seriously consider simply leaving your comments in the code. If the software is valuable enough for you to invest effort into reengineering it, the likelihood is great that someone in the future will again have to extend and modify it. That person will almost certainly benefit from the questions and answers you have identified.

Examples

You define a method dedicated to the annotation in the common ancestor of the classes you are trying to understand. (If your application classes do not share a common ancestor you might duplicate the method definition, or, better yet, you may define a separate class for this method.)

In the Moose Environment. The following Smalltalk code defines in the class `MSEAbstractRoot` (root of the Moose environment) the method `strangeCode:` that takes a string as argument. The default implementation is empty.

```
MSEabstractroot>>strangeCode: aString
    "empty method body"
```

Annotations are then included in selected methods. These annotations do not replace the methods' comments but rather contain specific questions that you asked yourself while trying to understand the system:

```
assessClassAttributesFor: aClassDef smalltalkClass: aSTClass
"Try to find out the properties of the given class (i.e., category
sourceAnchor, declaredAbstract, ...)"
    | category |
(self saveComments and: [aSmalltalkClass comment isEmpty not])
        ifTrue: [aClassDef addComment: aSmalltalkClass comment].
category := self assessClassCategoryFor: aSmalltalkClass
            isMetaClass: isMetaClass.
self saveSourceReference
    ifTrue:
        [aClassDef sourceAnchor:
            (MSEUtilities browserCategoryToSourceAnchor: category)
        ].
self strangeCode:
    'SD:3/12/99.Why is metaclass checked to store category?'.
self saveCategory & isMetaClass not
    ifTrue: [aClassDef setNamedPropertyAt: #category put: category].
aClassDef isAbstractKnown
    ifTrue: [aClassDef isAbstract: false]
```

In Squeak. The following code shows the definition of the `flag:` method in Squeak 2.7. Here the developers use the fact that the Smalltalk environment supports also the browsing of symbols (here passed as parameters).

```
Object>>flag: aSymbol
    "Send this message, with a relevant symbol as argument, to flag a
message for subsequent retrieval. For example, you might put the
following line in a number of messages:
    self flag: #returnHereUrgently
```

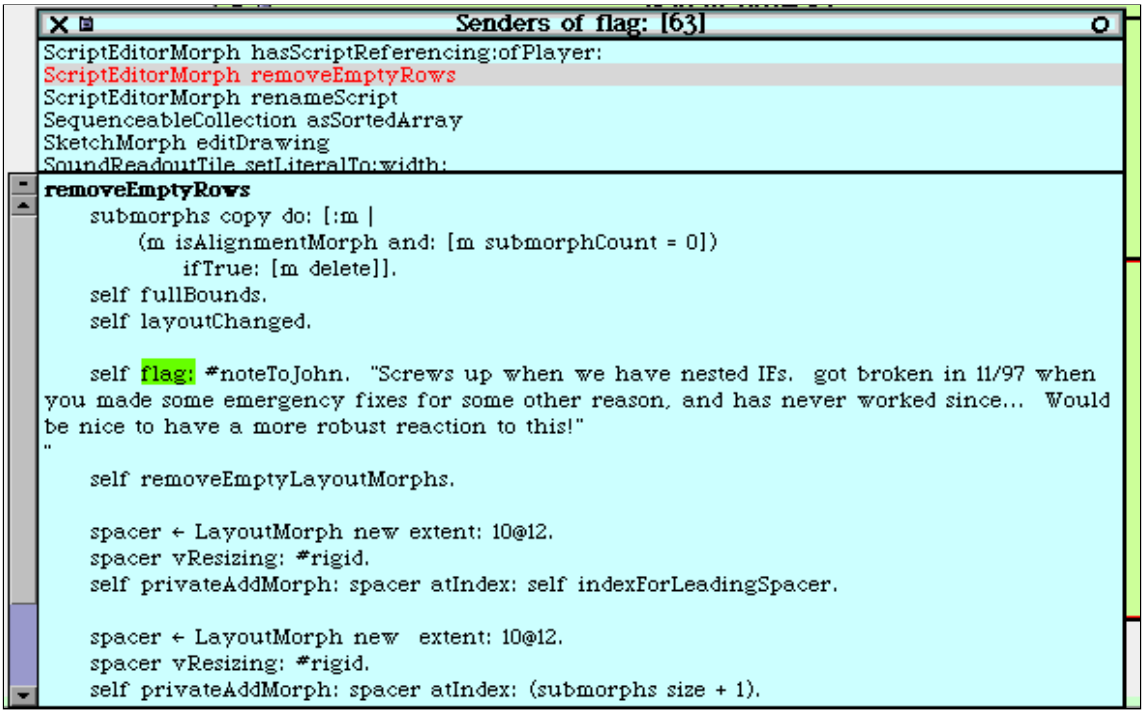

Figure 1 Finding all senders of a message in Squeak.

```
    Then, to retrieve all such messages, browse all senders of
#returnHereUrgently."
```

Figure 1 shows all the senders of the `flag:` message in the Squeak2.7 environment. Here we see both the method-based, mainly used for cross-references integrated into the environment, and comments.

Tradeoffs

Finding the Right Amount of Annotation. As with any kind of comments, you should take care to introduce just the right amount of detail. Terse or cryptic annotations quickly lose their value, and verbose annotations will distract the reader from the code itself. Normally the code should communicate its intention, and the methods or class comments are there to clarify implementation details [Beck97a]. That's why the annotations should contain only specific and precise remarks.

To see the advantages of applying Tie Code and Questions, let's compare it with the alternative solution of writing your questions and information into a separate log file or using a bulletin board system like a Wiki Wiki Server to share them with your team. With a bulletin board you can easily prepare a list of questions to ask to the original developers of the application and discuss them with the other members of your team. However, Tie Code and Questions has the following advantages:

Minimise Context Description. By applying Tie Code and Questions, you will exploit the context given by the programming language and the code. This way you will minimize the need to describe the context of your questions and keep your effort low while documenting your

questions and annotations. With other approaches you will have to spend an extra effort to describe the context of your annotations.You will certainly include some method bodies and this will be redundant with the code itself. Moreover, you will spend time documenting volatile information.

Automatic Synchronization . By applying Tie Code and Questions, you keep the code and the annotations in close physical proximity, and you thereby improve your chances of keeping them in sync. While modifying the code, you will more naturally modify the annotations, or remove them it they become obsolete. With other approaches, you will have to really take care to keep the code and the questions in sync. You will have to update the log file each time the related code changes. Moreover, as your log will be not an official document it will be even more difficult to allocate time to keep it synchronized with the code.

Improving Team Communication. Tie Code and Questions ensures that team members will always read the annotations in sync with the current version of the code they are working on. A log file can be shared with other members of your team. However, you must manage different versions of the log file for each version of the code, and every team member must spend extra effort to be sure that he or she has the right log file for the working version of the application.

Rationale

This pattern has its roots in literate programming. Literate programming puts the emphasis on keeping the code and its documentation physically close. The physical proximity reduces the effort spent in keeping the code and its documentation in sync.

Related Patterns

[PlopD4 page 632] is a pattern proposing to help newcomers to feel like home when they arrive in a new project. The pattern solution is : "An adopter should be encouraged to ‘move in’ by cosmetically arranging the code". The present pattern is more about how can maintainers or developers use simple code annotations to improve their understanding about code details and keep it close to and synchronized with the code elements they refer to.

Known Uses

- The Squeak development team used this technique not to keep track of questions but to communicate between developers. This way every developer had an understanding of the status of strange aspects of the code. In this team the comments were introduced by invoking method `flag:` defined in the class `Object`.
- During the development and the maintenance of the Moose environment, the pattern has been applied to register questions about the strange aspects of the system. The team used the methods `codeToBeChanged:` and `strangeCode:` implemented into the application root class to annotate with two different meanings.

- During the development of the game Skweek in assembler possible improvements were tagged using dummy labels named `gorbi`. Hence the editor and the debugger could identify them easily.
- A slightly different but related use of the pattern is applied by the company MediaGeniX. A systematic code tagging mechanism was introduced. The idea is to include in method comments information identifying the motivation of the code changes (bug fixes, new development, new release), the name of developer, the time of the actions. From this information the dependencies between the applications were extracted [OOPSLA 98 poster]. To increase the acceptance of the tagging procedure with the developers, a free field was added to the tag where the developers could write what they want.

Resulting Context

You are registering the questions or the aspects of the system you are maintaining inside the system thus reducing the effort spent to keep the code and your questions or early understanding of the application in sync. However, success is not garanteed: First, the team should stay motivated to annotate the code and second, you should pay attention to keep a similar level of annotation.