Tie Code And Questions: a Reengineering Pattern

Stéphane Ducasse and Oscar Nierstrasz

SCG-IAM, University of Bern

{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 [Duca99].

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 [Deme99] 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 [Duca99a]. Tie Code and Questions is a third kind of reengineering pattern, it is not only applicabel 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 — Link your questions or information about the code you are reengineering in the code itself to keep track of them and keep them synchronized with the code they refer to.

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.

You are reverse engineering the precise 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, a lot of assumptions are not clear for you and raise some questions about the reason of certain sequences of actions or about design.

Context

You need to have access to the code.

Solution

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

General Hints.

- Use conventions to identify your annotations. In a team context, include for example initials of the developer that made the comments. This way you can query them easily.
- Never use a language different that the one of the programming language (english in most cases) in the annotations. Otherwise, you create a different context and force the future reader to switch between them.
- When you know the answer of one of your questions add it near the questions for future readers.

Comment Based.

- Record your annotations by using the programming language comment (referred to as comment-based annotations). Programming environments like Eiffel's allow you to specify different level of visibility for your comments and code; in such as case give a private scope to the comments so that clients cannot see your annotations.

- If you are working with an IDE where you can query method senders use specific methods dedicated to the annotations (referred to as method-based annotations).

Discussion

The comment-based approach is better-suited for a text-based environment like the e-tags functionality with emacs [etags man]. The method based-approach is better suited for an integrated environment like the one of Smalltalk or Sniff+ that supports the query of method implementors or senders. In that case, the annotations can be queried like any other method send.

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

Keeping vs Removing the Annotations. What are the options that 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 let the comment-based annotations in the code. The Eiffel programming language environment provides several views of the code that are useful especially in such a situation.
- Method based annotations. If performance is an issue for the system you are working on, choose the comment-based approach or convert into comments the method calls using a perl script or similar tools.

However, in both cases you should always consider that if somebody got some problems with a piece of code and found the solution, it is likely that another developer will encouter the same problem. So when you will remove the annotations you will remove valuable information from your ystem.

Example

You can define a new method dedicated to the annotation in the common ancestor of the classes which you trying to understand. If your application does not share a common ancestor you may duplicate the method definition. You may also create a specific class to which the method is sent. The following Smalltalk code defines in the class MSEAbstractRoot (root of the Moose environment) the method strangeCode: that takes a string as argument. As per default this methods does nothing, its body is empty.

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

Then some annotations are included in some methods like the followingone. The annotations do not replace the methods comments but contain some specific questions that the maintainer asked himself 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 is checked to store
category?'.
    self saveCategory & isMetaClass not
        ifTrue: [aClassDef setNamedPropertyAt: #category put:
category].
aClassDef isAbstractKnown
    ifTrue: [aClassDef isAbstract: false]
```

Tradeoffs

Finding the Right Amount of Annotation. You should take care about the amount of information that you will introduce in the code. Too much annotations or too verbose ones will distract the reader from the code itself. Normally the code should communicate its intention and the methods or class comments are there to specify implementation details [Beck97a]. That's why the annotations should contain 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 use a blackboard system like a Wiki Web Server [http://c2.com/] to share them with your team. This way you could have a list of questions to ask to the original developers of the application and to discuss with the other members of your team. You would have a document with all your questions. However, Tie Code and Questions has the following advantages:

Minimise Context Description. While not applying Tie Code and Questions, 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 for documenting volatile information. By applying Tie Code and Questions, you will use as much as possible the context given by the language semantics and the classes themselves. This way you will minimize the need of describe the context of your questions and keep your effort low while documenting your questions and annotations.

Automatic Synchronization. While not applying Tie Code and Questions, 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 documentation it will be even more difficult to allocate time to keep it synchronized with the code. By applying Tie Code and Questions, as the code and their annotations are in close contact, you are improving your ability to keep the comments and the code changes in sync. While modifying the code, you will modify the annotations and remove them it they become obsolete.

Benefits of Locality. Having your annotations in a log file does not modify the code and does not introduce additional errors. However, every time you pass over the code that raised

some questions, these questions will not be directly present at the code level, so you will have to link the log and the code manually. Moreover, you will perhaps not have all the questions and annotations you asked yourself in mind and reading all your notes before starting to work is not possible. By applying Tie Code and Questions, the annotations are kept in the code at the place where the questions arose, you will read them at the exact place where they appear and only while you are reading this particular code. They will represent your reengineering context.

Improving Team Communication. A log file can be shared with other members of your team. However, you must keep control of the versions of the code to which it is related and every team member should pay attention that he is reading the right log corresponding with her/his version of the application. By applying Tie Code and Questions, team members will always read the annotations in sync with their versions and the code they are working on.

Rationale

This pattern has its roots in literate programming [Reen89a]. 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.

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.

References

- [Beck97] Kent Beck, Smalltalk Best Practice Patterns, Prentice-Hall, ISBN: 0-13-476904-X, 225 pages,1997
- [Deme99] Serge Demeyer, Stéphane Ducasse and Sander Tichelaar, A Pattern Language for Reverse Engineering, Proceedings of the 4th European Conference on Pattern Languages of Programming and Computing, 1999.
- [Duca99] The FAMOOS Object-Oriented Reengineering Handbook, Editors, Stéphane Ducasse and Serge Demeyer,See http://www.iam.unibe.ch/~famoos/handbook, University of Berne, 1999
- [Duca99a] Stéphane Ducasse, Robb Nebbe and Tamar Richner,Two Reengineering Patterns: Eliminating Type Checking, Proceedings of the 4th European Conference on Pattern Languages
- [Reen89] Trygve Reenskaug and Anna Lise Skaar, An Environment for Literate Smalltalk Programming, Proceedings OOPSLA '89, 337-346,1989.