Documenting Pattern Use in Java Programs

Marco Torchiano
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Norway
Marco.Torchiano@idi.ntnu.no

Abstract

Design patterns are widely recognized as important software development methods. Their use as software understanding tools, though generally acknowledged has been scarcely explored. Patterns are most useful in understanding software when they are well documented. Sometimes they are described separately from code as design comments. Nevertheless they hold a strong relationship to the source code and thus they should be documented at the source level too. Unfortunately there is no agreement on how to document pattern use. This paper describes a structured approach to document pattern use in the Java language. Our solution is based on the standard Javadoc tool and it is able to generate HTML documentation. The approach has been implemented and tested with software that uses patterns.

1. Introduction

Since their first appearance in [1], design patterns have increased their popularity among software developers. In short a pattern is a reusable solution to a recurring problem.

In literature there is a distinction between generative and non-generative patterns. The former are used to build systems, while the latter are found in systems and used to explain them [2].

We are interested in investigating how an intentional and documented use of design patterns can help to understand a software system.

In particular we will focus on the documentation of patterns at the source code level. Our work addresses the use of patterns in the Java programming language. However the proposed approach can be adapted to other programming languages, such as C++, using a suitable code documentation system.

We developed a proof of concept implementation that is based on the standard Javadoc [3] tool, which generates HTML documentation starting from the source code and specially marked comments.

Few works propose a tool supported structured approach to pattern usage documentation; [4] addresses the tracing problem of enhancing the patterns visibility in the code.

The usefulness of patterns for software understanding purposes has been scarcely investigated. There is only one empirical study conducted by Prechelt and colleagues [5]. Their work suggests that there is empirical evidence of such usefulness; it has the merit of stressing two important aspects of design patterns. First they can be used as a means to understand or to explain a system. Second the mere presence of patterns is not enough; they should be documented to be effectively useful in understanding.

What lacks in their work is an analysis of how patterns are used in software systems and how they can be documented.

This paper addresses these drawbacks by providing four main contributions:

- an analysis of the use of pattern,
- a proposal of how to document the use of pattern,
- a proof of concept implementation by means of a standard code documentation system,
- an empirical validation of the approach.

The rest of this paper is organized as follows. Section 2 provides a discussion of the use of design patterns and the identification of the main features that should be documented. Section 3 describe the implementation of the pattern documentation using the Javadoc tool. Section 4 summarized the empirical validation of the approach. Finally section 5 draws some conclusions and describes future work.

2. Pattern use

The relationship between a general description of a pattern and its use is not as simple as it could appear. Design patterns usually describe an idea at a high level of abstraction. Although sample reference implementations are usually provided, there are several possible variations in a pattern implementation.

In addition, often patterns must be modified and adapted to serve the needs of the software system where they are used.

Often a design pattern is described by means of a class diagram. For instance Figure 1 shows the *composite* design pattern described in [1]. This pattern is used to represent part-whole hierarchies while minimizing the difference between composite and leaf objects.

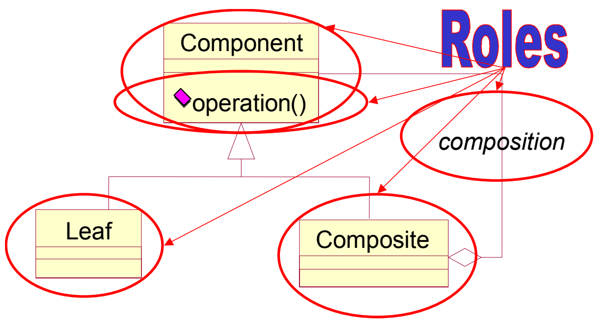

Figure 1: Composite pattern and its roles.

Design patterns are made up of several elements; we call them *roles*. The roles interact with each other to solve the specific problem addressed by the pattern.

During the development of a system design patterns are brought into use when problems are encountered, which are solved by a known pattern.

We call this use an *instance* of the pattern. While a design pattern is described in terms abstract classes[1], the instance consists of actual classes.

The pattern instance solves a system specific problem; we call *task* the purpose served by the instance, i.e. the reason why the pattern is used in the system.

Figure 2 shows an example of instance of the composite pattern. The task of this instance is to represent an arithmetic expression as a tree that makes it easy to evaluate its value.

An important relationship between the pattern and its instance is how the abstract roles defined in the former are mapped onto elements of the latter. For instance we can observe that class *Expression* in Figure 2 plays the *Component* role defined in Figure 1.

Since the instance of a pattern in part of a more complex system, it is used by the other elements. For instance there is a client that invoke the *evaluate()* method of class *Expression* to evaluate an expression.

So far we have analysed how a pattern is used. From this analysis we devise the main feature of an instance that should be documented.

For each element of a software system that is part of a *pattern* instance we wish to document: the pattern that is instantiated, the *role* that is played by the element, the *task* of the pattern. In addition the *use* of a patterns should be documented.

Given a pattern instance, the knowledge of the base pattern provides understanding of the overall structure and behaviour. When maintenance is concerned this information make it easier to locate the spots where to apply changes.

The role that is played by each element of the pattern instance gives information on the specific behaviour and on the interactions with the other elements/roles.

The link between the abstract goal of a pattern and the concrete purpose in the actual system can be understood looking at the task of the pattern instance.

Recognizing that a part of a system is a client of a pattern instance let us understand its rationale. It is of paramount importance when such a part has to be maintained.

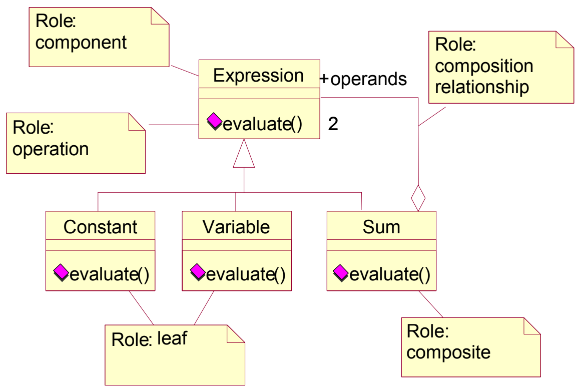

Figure 2: Instance of composite pattern.

3. Java Documentation

We focus now on the problem of documenting Java programs. The standard documentation of all the class libraries in the Java environment conforms to the Javadoc format. Thus we investigate this tool and its customization capabilities.

3.1. Javadoc

Javadoc[3] is a tool that parses the declarations and documentation comments in a set of source files and produces a group of cross linked HTML pages describing the classes, inner classes, interfaces, constructors, methods, and fields.

This documentation constitutes a layer over the source code. It can be used both as stand-alone and as a map to navigate more effectively through the code.

It is possible to include documentation comments in the source code, ahead of declarations for any entity (classes, interfaces, methods, constructors, or fields).

[1] Although patterns can be used within different paradigms and languages, here we assume object-oriented patterns, which are the most common form.

These are also known as Javadoc comments. A doc comment consists of the characters between the characters "`/**`" that begin the comment and the characters "`*/`" that end it. The text can continue onto multiple lines.

The documentation produced by Javadoc can be customized using tags. A *tag* is a special keyword within a doc comment that Javadoc can process. Javadoc has standard tags, which appear as `@tag`, and in-line tags, which appear within braces, as `{@tag}`.

Standard tags produce a dedicated section in the documentation. While in-line tags result into documentation elements that are embedded in the context where they are found.

The default output of Javadoc is a set of HTML pages linked to each other. There is an overview page for each package and a page for each class.

As of version 1.4, the Javadoc tool can be customized in two ways: using doclets or using taglets.

It is possible to use doclets to customize Javadoc output. A doclet is a program written with the doclet API that specifies the content and format of the output to be generated by the Javadoc tool. Doclets allow to define the overall structure and format of the documentation, for instance they can be used to produce documentation is other formats than HTML.

Taglets can be used to provide finer grained customizations. A taglet define a new tag, together with its meaning and appearance in the output documentation.

We decide to use the standard structure of Java documentation and add specific sections to document the use of patterns.

Since pattern documentation is an add-on to the standard documentation, taglets are used to customize the Javadoc tool.

3.2. Pattern taglets

Pattern in Java code are documented by means of additional tags. The programmer can use these new tags in addition to the standard ones.

The tags we propose to document the use of patterns are the following:

@pat.name: the name of the pattern. This is a standard tag that applies to an element of a pattern instance. For the time being only the patterns described in the GoF book [1] are valid. The name will be represented in the documentation as a link to on-line description of the pattern.

@pat.role: pattern role. This is an in-line tag that describes the role played by the element of the pattern instance; it must be nested inside a @pat.name tag.

@pat.task: pattern task. This is a standard tag that is used to describe the task performed by and instance of a pattern of by one of its parts.

@pat.use: pattern use. This is a standard tag that is used to describe the use of a pattern instance by one of its clients.

The first three pattern specific tags can be used to describe whole classes, methods, and fields. The last one typically is used to describe methods that use pattern instances. These tags are fairly generic, thus allowing to document both structural and behavioral patterns.

The following code is an example of documentation of a class that plays the role of *Leaf* in the *Composite* pattern.

```
/**
* This class purpose is…
* @pat.name Composite {@pat.role Leaf}
* @pat.task it represents a variable,
* {@link #evaluate()} gives the value
*/
class Variable { }
```

The above fragment of comment, when processed by the Javadoc tool enhanced with the proposed pattern specific tags produces the following documentation:

> The class purpose is…
> **Pattern:**
> **Composite**, role: Leaf
>
> **Pattern task:**
> It represents a variable, evaluate() gives the value.

The first sentence in the documentation is used as the generic class description. Then the pattern specific tags generate their output. The *Composite* link brings to a on-line description of the composite pattern. The *evaluate()* link, obtained through the default in-line tag link, brings to the documentation of the method evaluate within the current class.

4. Empirical validation

To validate the proposed approach we decided to replicate the experiment presented in [5], which is described in detail in [6]. Since a replication package is available[2] together with the collected data it will be used as a reference.

The experiment consists of two maintenance tasks performed on two different Java programs. The hypotheses can be summarized as:

H1. the presence of pattern documentation makes the task completion quicker;

[2] http://www.ipd.uka.de/~prechelt/packages/patdoc_package.zip

H2. the presence of pattern documentation reduces the number of errors committed.

We tried to keep all the details as close as possible to the original experiment. The only variations were:

- the use of the HTML documentation, produced by Javadoc, instead of paper code listings;
- the introduction of pattern specific tags, instead of unstructured pattern documentation;
- the adoption of a web-based interface for the questionnaire.

We used the same programs, but we adapted the original documentation to both of the Javadoc style and the pattern specific tags.

The subjects of the experiment were 28 students at the fourth year in computer science degree at Norwegian University of Science and Technology (NTNU).

One third of the students wrote less than 300 lines of code in their career and none of them ever wrote more than 3000 lines of code.

The students had no previous experience with design patterns; they were given a two hours lesson on the topic by the author.

We ran the Mann-Whitney test on the collected data, the results are shown in Table 1 and compared against the results obtained running the same test on the reference data.

		H1	H2
Task 1	Our	p = 0.0835	p=0.0376
	Reference	*opposite*	p=0.0944
Task 2	Our	p = 0.3225	p=0.4253
	Reference	p = 0.0799	p=0.3728

Table 1: Significance of tests.

The results we obtained running our tests on the reference data are very close to those reported in [6].

The only strong support we obtained is for hypothesis H2 on task 1, but it is completely unsupported for task 2.

Both our results and the reference ones do not fully support H1; actually the reference data for task 1 indicate the opposite of the hypothesis.

Compared to the reference experiment the main difference is the improvement in time obtained with the web-based exercise compared to the paper based.

In task 1 the average time reduced from 55 to 47 minutes, while in task 2 it went from 55 to 54; the average reduction in time is 8%.

The post questionnaire revealed that the students judged it positively and liked the online modality.

5. Conclusions

This paper proposes a structured approach to document pattern use in the Java programs. The approach extends the Javadoc tool with pattern specific tags.

Our proposal is based on a simple extension of a widely used standard; therefore it can be easily adopted. The overhead required to document pattern is very low: a few lines in addition to the usual documentation.

The resulting documentation is well structured and is linked to pattern description, thus making it easy to understand the program.

The web-based context emulated more closely a real professional development environment such as an IDE. Thus the results deriving from our approach are more likely to be applicable to real system development. From the empirical validation we can conclude that:

- both our experiment and the reference one seem to indicate that patterns are useful in some maintenance tasks,
- since our subjects were less expert and trained than the reference, our support is stronger,
- the enhanced documentation provided by Javadoc improves efficiency of maintenance.

The preliminary results are positive, but there is a lot of future work, in particular:

- further experiments should be performed with both students and professionals,
- the analysis of the use of pattern should be refined based on the feedback from the experiments,
- more detailed experiments should be designed to validate the pattern specific tags.

We believe the proposed approach is both useful and easy to adopt. Given the ever-increasing pattern importance the proposed tags can become part of the standard set of Javadoc.

6. References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[2] J. Coplien, "Software Design Patterns: Common Questions and Answers," in *The Patterns Handbook: Techniques, Strategies, and Applications*, L. Rising, Ed. New York: Cambridge University Press, 1998.

[3] "Javadoc Tool Home Page": Sun Microsystem, 2002, available at http://java.sun.com/j2se/javadoc/.

[4] A. Cornils and G. Hedin, "Statically checked documentation with design patterns," in Proc. of 33rd International Conference on Technology of Object-Oriented Languages (TOOLS 33), Mont-Saint-Michel, France, 2000.

[5] L. Prechelt, B. Unger, M. Philippsen, and W. F. Tichy, "Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance," *IEEE Transactions on Software Engineering*, vol. 28 (6), pp. 595-606,June 2002.

[6] L. Prechelt, "An experiment on the usefulness of design patterns: Detailed description and evaluation," Faculty of Informatics, University of Karlsruhe 9/1997, June 1997.