Evaluating the Use of Design Patterns during Program Comprehension – Experimental Setting

Yann-Gaël Guéhéneuc and Stefan Monnier
Department of Informatics
and Operations Research
University of Montreal, Quebec, Canada
{guehene,monnier}@iro.umontreal.ca

Giuliano Antoniol
Computer Engineering Department
Polytechnic Montreal, Quebec, Canada
antioniol@ieee.org

Abstract

In theory, there is no difference between theory and practice. But, in practice, there is.

Jan L. A. van de Snepscheut

1 Introduction

Design patterns [2] have been adopted quickly by the software engineering community and, since their introduction, several studies have been proposed to ease their choice [5], their use [1], and their recovery [10].

A design pattern is a literary form providing solution to recurring design problems. It decomposes in several sections: Intent, Motivation, Applicability, Structure, Participants, Collaborations, Consequence, Implementation, Known Uses, and Related Patterns. The solution advocated by a design pattern, mainly in the Structure, Participants, and Collaborations sections, is a design motif—a prototypical micro-architecture that describes the solution while abstracting context and implementation constraints.

Many studies (including studies by the authors) claim that identifying micro-architectures similar to design motifs in program architecture benefits software engineers by easing program comprehension. The rationale of the studies on design motif identification is:

- Design decisions are scattered in a program architecture and are often not documented.
- Design decisions help software engineers in understanding the design of a program.
- Micro-architectures similar to design motifs are clues on the design decisions.

Although the previous claim is logical and reasonable, some studies show that the use of design patterns does not ease program comprehension and, thus, question the benefits of design motif identification. In particular, the Wendorff's qualitative study [9] raises the issue of overuse and misuse of design patterns.

We believe that, indeed, design motif identification benefits software engineers by easing program comprehension but seek proofs of this claim. Thus, we suggest experimental settings with which to prove or to disprove the claim that design motif identification help software engineers' comprehension.

In Section 2, we elaborate on the research question related to the claim. In Section 3, we propose experiments to test our research question as well as experimental settings. Finally, in Section 4, we conclude on the research question and the experiments.

2 Research Question

2.1 General Question

Question. Our research question concerns the evaluation of the help that design motif identification really provides to software engineers during program comprehension.

Rationale of the Question. The cost of maintenance is evaluated to at least 50% of the overall cost of software development. Among those 50%, another 50% (for the least) is dedicated to program comprehension. Hence, any help in comprehending programs could reduce the cost of software development dramatically.

Many studies claim that design patterns, in general, ease program comprehension and that the documentation of used design motifs (possibly through semi-

automated identification), in particular, allows software engineers to grasp design decision quickly.

However, to the best of our knowledge, no systematic studies have been published yet to prove or to disprove the claim that design motif identification help in comprehending programs.

2.2 Hypotheses

Acquisition of the Information. Software engineers, like people in general, use their sight as the main mode for acquiring information. Indeed, software engineers spend long hours looking at computer-displayed models of programs in various forms and dimensions.

A well-known way of displaying program models is using the UML notation: 2D diagrammatic models of different aspects of a program (structure, behaviour, packaging...). Figure 1(a) shows a UML-like model of a sample program.

The use of sight suggests that many factor must be considered to understand how software engineers comprehend a program using a model such as shown on Figure 1(a). In particular, proximity, attention, object recognition and categorisation impact the quality and pertinence of the acquired information.

In the case of design motif identification, the use of visual models also raises the issue of the ease to identify visually design motifs with no a priori clues.

Use of the Information. Once design motifs have been identified (either visually or by other means), software engineers can use this information to comprehend a program better. They may use this information either to focus their attention towards the classes, methods belonging to an identified design motif (see Figure 1(b)), or to exclude from their attention those classes and methods to focus on other part of a program architecture (see Figure 1(c)).

2.3 Experimental Questions

How do software engineers look for design motifs? This question concern the way software engineers navigate through a program architecture when no design motif have been identified, documented, and displayed. Do the software engineers look for design motifs? Do they follow some specific path?

Do software engineers focus their attention on identified design motifs? This question focus on the use of identified and displayed design motifs over a classic model of a program architecture. Do software engineers focuses on the constituent of the design motifs or—on the opposite—focus on constituents of the program architecture away from the identified design motifs?

When do software engineers need to comprehend a program? What do they need to comprehend? Although not directly related to design patterns, these questions are clearly related to the two previous questions. It is most relevant to understand the software engineers' comprehension activities to contextualise the two previous question. Indeed, the context is important to set up relevant experiments. Unfortunately, to the best of our knowledge, no thorough studies of the software engineers' comprehension activities exist. An interesting study has been proposed by Murphy *et al.* [6] to analyse development activities. We hope this study will bear fruits that will help us in setting up our experiments.

3 Experiments

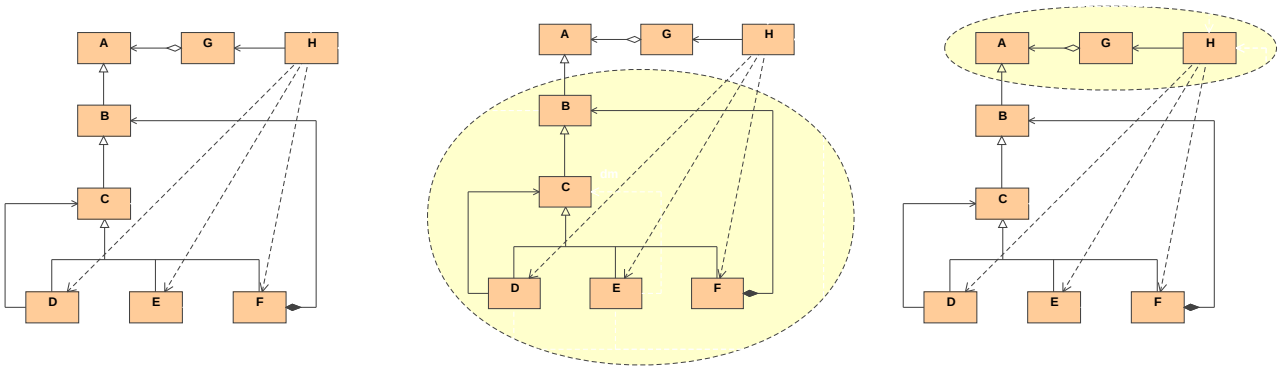
The use of new technology can help in setting up experiments to answer our experimental questions. We devise experiments to assess the identification of design motifs, on the one hand, and the use of identified design motifs, on the other hand, during the software engineers' program comprehension activity.

3.1 General Setting

Progress in non-intrusive monitoring of human behaviour makes it possible to follow the external behaviours of a software engineer involved in program comprehension activities without disturbing *too much* the performed activities.

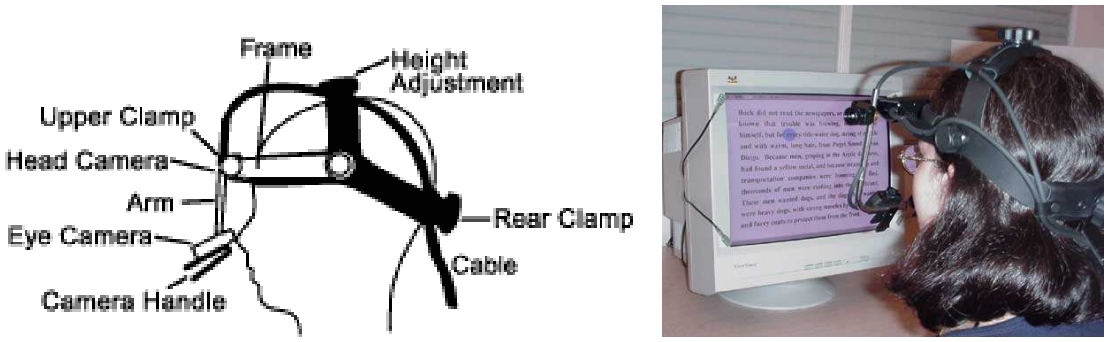
In particular, the use of video-based eye tracking systems allows following with enough precision a software engineer's eye movements while looking at a model of a program. A video-based eye tracking system records a subject's eye movements without much interferences with the subject's activity through the use of a special headband and an dedicated API.

In our experiments, we plan to adopt SR Research eye-tracking systems. SR Research is an international manufacturer of high quality eye-tracking systems with their EyeLink II systems. A EyeLink II system decomposes in a display computer, displaying the data to comprehend by the software engineer, a host computer storing the data related to a subject's eye movements, and a headband supporting cameras to track the eye movements.



(a) Sample UML-like diagram. (b) Focus on the Composite design motif. (c) Focus away from of the Composite design motif.

Figure 1. Sample UML-like diagram, with visual attention focused on or away from a micro-architecture similar to the Composite design motif.



(a) Headband and its different parts. (b) Use of the headband.

Figure 2. The SR Research EyeLink II eye-tracking system.

Figure 2(a) depicts the headband and its different parts, while Figure 2(b) shows a subject wearing such a headband. The headband mainly consists of a set of cameras recording the position of the head and the movements of the eyes with respect to the displayed image. Synchronisation with the image displayed on screen (being looked at by the subject) is perform through a dedicated API, which controls and synchronises the camera and generates the data. The generated data is stored on a dedicated host computer, hosting the eye-tracking system, and connected with the display computed by a high-speed ethernet connection. Figure 3 summarises the data acquisition process.

We plan to use such a system to study the eye movements of the software engineers on the constituents of class diagram-like models of programs and the dwell time on individual constituents such as class, relationships, identified design motifs. We shall adapt the PTIDEJ tool suite to synchronise the display of class diagrams-like models with EyeLink II systems. PTIDEJ [3] is a set of tools to evaluate and to enhance the quality of object-oriented programs, promoting patterns, at the language-, design-, and architectural-levels. With PTIDEJ, it is possible to create and to display models of AOL, C++, and Java programs using a unified meta-model, to infer inter-class relationships [4], and to identify structural design motifs.

3.2 Experimental Setting

We want to understand how software engineers use design motifs during program comprehension activities. Thus, the subjects of our experiments are software engineers while the objects are models of programs highlighting or not design motifs.

Consequently, in the perspective of generalising the results of our experiments, we must choose our subjects carefully. We must choose subjects and form two

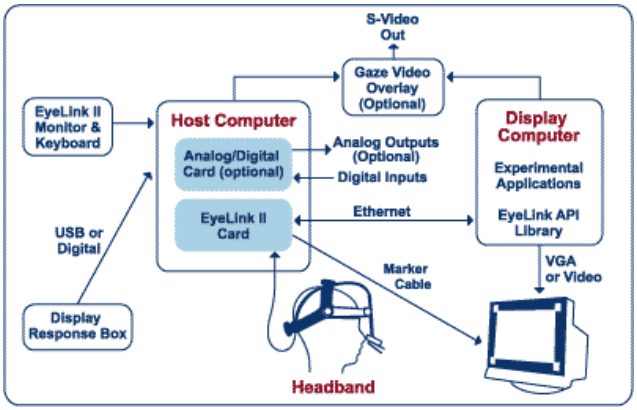


Figure 3. Use of a EyeLink II system.

groups: Subjects with no knowledge of design patterns and subjects with a deep knowledge of design patterns. We must perform our experiments with sufficient numbers of subjects in both group to avoid experimental biases. We plan to use students at the Department of Informatics and Operations Research as subjects. First or second year bachelor students would belong to the group with no knowledge of design patterns, while Master or Ph.D. students would belong to the group with good knowledge of design patterns.

The objects of our experiments must be programs in which developers use design patterns. However, we can limit ourselves to a small set of programs providing that we limit the reinforcement learning process [8]. We plan to use the JHotDraw and JUnit programs because those are well-know programs with limited numbers of classes and an average complexity, in which developers used design patterns.

The program comprehension activities we shall ask software engineers to perform must involve both part of the program architectures related to design motifs and not related to design motifs. A typical task involving a design motif would be to add a class to the design motif (*i.e.*, a functionality to the program). A typical task not involving a design motif would be to modify a class to change the program behaviour.

3.3 Identification of Design Motifs

Hypothesis. We make the hypothesis that expert software engineers look for micro-architectures similar to design motifs during program comprehension while novice software engineers do not.

Expected Results. The expected results of the experiments is that expert software engineers spend time, at the beginning of the program comprehension activity, in searching for design motifs, while novice software engineers just follow in rand orders the relationships among classes to perform their tasks.

3.4 Use of Identified Motifs

Hypothesis. Our hypotheses is that the knowledge of the design motifs used in the design of a program decrease the time for program comprehension. This knowledge allows software engineers to focus on constituents in the micro-architectures highlighted as design motifs or away from these constituents.

Expected Results. Depending on the task at hand and the knowledge (or lack thereof) of the existing design motifs, we expect to measure the difference of time spend by expert and novice software engineers on the constituents of micro-architectures similar to design motifs. We expect that expert software engineers will use their knowledge and the knowledge of the used design motifs.

Discussion. Our hypothesis is similar to the idea of "intentionally ignored information", such as experimented by Rock and Gutman [7].

4 Conclusion and Future Work

We presented experimental settings using eye-tracking systems to understand how software engineers (novice and expert) use knowledge of the design patterns used in the design of programs.

We believe these experimental settings will help in understanding the use of design patterns and answer questions related to their wide adoption and their far-reaching use in practice.

We now plan to refine these experimental settings and to perform the actual experiments as part of the development of the LaiGLE laboratory (Laboratory for Experimental Software Engineering).

References

[1] Jan Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, February 1998.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1^{st} edition, 1994.

[3] Yann-Gaël Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1^{st} *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005. **Submitted for publication.**

[4] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the* 19^{th} *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[5] Olivier Motelet. An intelligent tutoring system to help OO system designers using design patterns. Master's thesis, Vrije Universitët, 1999.

[6] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubraniś. The emergent structure of development tasks. In Andrew P. Black, editor, *proceedings of the* 19^{th} *European Conference on Object-Oriented Programming*, pages 33–48. Springer-Verlag, July 2005.

[7] Irvin Rock and Daniel Gutman. The effect of inattention on form perception. *journal of Experimental Psychology: Human Perception and Performance*, 7:275–285, 1981.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1*st* edition, March 1998.

[9] Peter Wendorff. Assessment of design patterns during software reengineering: Lessons learned from a large commercial project. In Pedro Sousa and Jürgen Ebert, editors, *proceedings of* 5^{th} *Conference on Software Maintenance and Reengineering*, pages 77–84. IEEE Computer Society Press, March 2001.

[10] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the* 26^{th} *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.