Work in Progress: Transitioning From Novice to Expert Software Engineers Through Design Patterns:

Is It Really Working?

Arbi Ghazarian
Department of Engineering, Arizona State University
Arbi.Ghazarian@asu.edu

Abstract— **Since their wide adoption by the software engineering community in the mid-90's, design patterns have become an important educational component in the training of novice software engineers due to the common belief that, as distilled experience of seasoned software designers, they can transform novice software engineers into skilled professionals in a relatively short time frame. This paper questions the validity of this commonly-held belief about the educational value of conventional patterns, arguing that although there is tremendous value in capturing and disseminating software engineering experience in the form of patterns, conventional design patterns cannot deliver on this expectation to significantly shorten the experience growth curve. We report initial results from an ongoing research project that aims to not only identify the difficulties in using conventional patterns as educational means for novices, but also to use such insights to develop new forms of patterns to help educators and experts to more effectively communicate and transfer their software engineering knowledge, experience, and understanding of best practices to novices.**

Keywords- Software Engineering Education; Design Patterns;

I. INTRODUCTION

Since their wide adoption in the mid-90's, design patterns [1] have become an important component in the education of novice software engineers; they have entered software engineering text books and are discussed and taught as part of various courses such as software architecture and design, software engineering, as well as courses on programming and software development. Design patterns are recurring software design problems along with their solutions, which are captured and documented in a reusable format. Since design patterns capture the design experience of seasoned software engineers, they are believed to help novice software engineers to become professionals in a shorter time frame. In other words, the underlying claim about the educational value of design patterns is that instead of taking the natural and lengthy route to becoming an experienced software designer through trial and error over many years, one can study design patterns that capture the essence of expert design knowledge and as a consequence become experienced in a significantly shorter amount of time. The common belief is that, after all, design patterns are supposed to help inexperienced software engineers to come up with the same design models as experts, without much need to gain experience over a long period of time. This paper questions the validity of this commonly-held belief about the educational value of conventional patterns, arguing that although there is tremendous value in capturing and disseminating software engineering experience in the form of patterns, conventional design patterns cannot make novice software engineers perform at the level of experienced software professionals in a short period of time. In this paper, we will argue that design patterns do not replace design experience; in fact, paradoxically as it may seem, one must be experienced to correctly identify opportunities for the application of patterns and implement them appropriately. Based on our observations, we provide an explanation as to why this is the case and accordingly propose criteria for alternatives forms of software patterns to make expert software design knowledge more accessible to beginning software engineers. This paper reports on an ongoing project that aims to improve software engineering education by understanding how expert software engineering knowledge can be made more accessible to novice software engineers.

II. OBSERVATIONS AND ANECDOTAL EVIDENCE

Our observations of software engineering students, who learn about design patterns as part of their program courses, are that a great majority of these students, when asked to apply the patterns they have learned to their course projects, misuse design patterns. This misuse of patterns takes various forms, perhaps the most notorious being the application of a pattern to a design problem that does not warrant the use of that particular design pattern. That is, a design pattern is forced, as a solution, to a design problem, for which the pattern was not originally intended as a potential solution. The ill-definedness of design problems in typical software engineering contexts, on the one hand, and the imprecision in the specification of the problem and the context components of a pattern, on the other hand, make the objective and correct recognition of opportunities for the application of a pattern a great cognitive challenge, especially for inexperienced software engineers. This inexactitude in distinguishing when, a chunk of software engineering knowledge, codified experience, or best practice, represented as a pattern, is the right solution to a software engineering problem is not peculiar to novices as we have seen numerous examples of applications of patterns in industrial software projects to problems to which they do not offer a solution, resulting in an unnecessary increase in the complexity of the system and consequently software comprehension and maintenance problems. Unfortunately, in software engineering, in the absence of hard physical laws, it is always possible to produce numerous functionally-correct solutions to a problem, most of which are deficient in several

978-1-4673-1352-0/12/$31.00 ©2012 IEEE

ways in terms of non-functional aspects of the solution including its understandability and the maintainability.

III. PROJECT STATUS, RESULTS, AND EXPECTED OUTCOMES

An underlying premise of this current research project is that, in order to address the above-mentioned problem with software engineering education, we must shift our research focus in the direction of investigating effective ways to more accurately specify both software engineering problems and the patterns that are intended to provide reusable solutions to these problems. In technical terms, the goal is to find a *match* function or operation that, given a problem and a list of predefined solutions (e.g., a catalog of patterns), accurately determines whether the conditions that warrant the application of a pattern (i.e., the problem description component of a pattern) match those of a given problem. It is precisely the complexity of the cognitive match operation that confronts the beginning software engineer. To make the cognitive match operation for a pattern accurately possible, we need to rethink how we specify software engineering problems and patterns.

Our research project draws on both theoretical and empirical software engineering research to address this problem. On the theoretical side, our goal is to develop a formal foundation for the specification of software problems and their corresponding patterns. Formal specification can be a fruitful avenue of research to address the problem as the observation is that the informal and inaccurate definition of problems and patterns make them subject to individual interpretations, which, in turn, makes it difficult to achieve an objective mental match operation on the part of the software engineer. Our work in formalizing the specifications of software engineering problems and patterns has resulted in two formalisms, namely the Problem Decomposition Scheme (PDS) [2] and Systematic Translation Scheme (STS) [3]. The former formalism provides a way to accurately and objectively classify software problems into requirements categories or problem dimensions, while the latter links each problem category to a set of design regularities [4] that form a predefined solution, resulting in a form of pattern known as traceability patterns [5][6]. Design patterns are, for the most part, meant to address non-functional problems such as extendibility and maintainability. An accurate characterization and definition of such non-functional concerns, due to their abstract and somewhat vague nature, is difficult to achieve. Design patterns, in effect, attempt to link from such non-functional problems to a predefined solution and are consequently subject to difficulties arising from the lack of accurate problem definitions. In contrast, traceability patterns link a category of functional problems, such as business rules category of requirements in enterprise information systems, to a predefined solution. Initial results from a previous study shows that functional problems, when stated as atomic requirements statements, can be objectively recognized [2]. This suggests the hypothesis that, software engineering knowledge, represented as traceability patterns, should be easier for novice software engineers to consume. We plan to test this hypothesis as part of our ongoing research effort.

Traceability patterns aim to solve some of the issues associated with conventional patterns through tracing problem dimensions that can be objectively recognized to their corresponding predefined solutions. This predefined solution contains the wisdom we try to transfer to the software engineer. This capability to objectively and consistently recognize problems that have predefined solutions, without much reliance on experience on the part of the software engineer, makes traceability patterns a more appropriate educational tool for software engineers, at least for novices. To enrich the theoretical foundation underlying traceability patterns, we are currently investigating the use of ontologies [7][8][9][10][11] and domain models as a formal means to capture and communicate software engineering knowledge.

On the empirical side, we have conducted a user study to empirically verify whether indeed software problem defined in PDS-based functional specifications are objectively recognizable by software engineers (i.e., is an accurate mental match operation possible?). Initial results have been reported in [2] and are promising. The technical feasibility of traceability patterns in software projects has been demonstrated in a proof of concept system [5][6]. We plan to further evaluate traceability patterns by putting them into practical use in industrial software projects. We expect that these efforts will positively impact educational outcomes for software engineers. We also expect that results from our project will be of importance to the software engineering education community as they will afford us the knowledge and insight necessary to devise better approaches to bridge the gap between novice and skilled software engineers.

REFERENCES

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

[2] Ghazarian, A., Sagheb-Tehrani, M.; Ghazarian, A., "A Software Requirements Specification Framework for Objective Pattern Recognition: A Set-Theoretic Classification Approach", Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems (CECCS 2011). pp. 211-220. Las Vegas, USA, 2011.

[3] Ghazarian, A., "A Formal Scheme for Systematic Translation of Software Requirements to Source Code", Proceedings of WSEAS Applied Computing Conference (ACC 2011), pp. 44-49. France. 2011.

[4] Ghazarian, A., "Coordinated Software Development: A Framework for Reasoning about Trace Links in Software Systems", Proceedings of the IEEE's 13th Int'l Conference on Intelligent Engineering Systems (INES 2009). pp 236-241, IEEE Computer Society, Barbados. April 2009.

[5] Ghazarian, A., "Traceability Patterns: An Approach to Requirement-Component Traceability in Agile Software Development". Proceedings of the 8th WSEAS International Conference on Applied Computer Science (ACS 2008). Venice, Italy. November 2008.

[6] Ghazarian, A., "A Matrix-Less Model for Tracing Software Requirements to Source Code", International Journal of Computers. NAUN. ISSN: 1998-4308. Issue 3. Volume 2. pp. 301-309. 2008.

[7] Borst, W. "Construction of Engineering Ontologies for Knowledge Sharing and Reuse". Doctoral Dissertation, Enschede. NL-Centre for Telematics and InformationTechnology. University of Tweenty. 2006.

[8] Gruber. T. "A translation approach to portable ontology specifications". Knowledge Acquisition. 5(2):199-220. 1993.

[9] Studer, R., Benjamins, V., and Fensel, D. "Knowledge engineering: Principles and methods". Data Knowledge Eng. 25(1-2):161-197. 1998.

[10] Musen, M. "Domain ontologies in s/w eng: Use of protege with the eon architecture". Methods of Info. in Medicine. 37(4-5):540-550. 1998.

[11] Falbo, R., Menezes, C., and Rocha, A. "A systematic approach for building ontologies". Proceedings of the IBERAMIA'98. Lisbon. 1998.