Software Architecture Patterns: Reflection and Advances

[Summary of the MiniPLoP Writers' Workshop at ECSA'14]

Neil B. Harrison
Utah Valley University, USA
neil.harrison@uvu.edu

Uwe van Heesch
Capgemini Germany
uwe@vanheesch.net

Stefan Sobernig
WU Vienna, Austria
stefan.sobernig@wu.ac.at

Peter Sommerlad
Hochschule für Technik,
Rapperswil, Switzerland
peter.sommerlad@hsr.ch

Martin Filipczyk
paluno: The Ruhr Institute for
Software Technology, Germany
martin.filipczyk@paluno.uni-due.de

Alexander Fülleborn
University Duisburg-Essen
alexanderfuelleborn@hotmail.de

Angelika Musil & Jürgen Musil
Vienna University of Technology
{angelika,jmusil}@computer.org

ABSTRACT

Architectural software patterns capture successful designs for recurring problems in software architecture. For the first time, a workshop in the tradition of the software-pattern conference series (e.g. PLoP, EuroPLoP) was held jointly with the European Conference on Software Architecture (ECSA 2014) in Vienna, Austria. The main objective of this workshop called MiniPLoP was to bring software architecture researchers closer to the pattern community and to introduce the writers' workshop format to them. Writers' workshop at PLoP conferences provide authors of pattern papers with high-density feedback given by peers within a limited timeframe. In addition, our workshop provided a forum to reflect on the state of software architecture patterns and to discuss advances pattern research. This report contains an extended keynote abstract and summaries of the papers discussed during the workshop.

1. PREFACE

Patterns of software architecture are an important tool in software architectural design [1]. They can be observed in nearly every software architecture, even where the architects did not know they were using them. Papers and books on architecture patterns were first published nearly 20 years ago [2], and have been widely used. Architecture patterns continue to be an important area of research and practice, and have proven their worth for capturing and conveying architectural design knowledge and decisions.

Since 1994, the Pattern Languages of Programs conference (PLoP) has been held annually to bring together pattern authors for developing and improving (textual representations of) patterns in the computer science domain. Since then, many other PLoP conferences emerged, most importantly the European Conference on Pattern Languages of Programs (EuroPLoP), starting in 1996. The domains of the patterns submitted to *PLoP conferences vary, but software architecture has always been one of the major domains.

MiniPLoP was held jointly with the European Conference on Software Architecture (ECSA 2014) on August 25, 2014. The aims were to bring researchers in the software architecture field closer to the pattern community, to reflect on existing architectural software patterns, changing patterns and emerging patterns, and to advertise the writers' workshop format to to the software architecture research community. The writers' workshop format is adapted from the workshops of literary circles to provide highly concentrated feedback to authors [6]. At *PLoP conferences, the format has been extensively used to improve pattern writings, as well as other writings associated with patterns, such as research papers on the application of patterns.[1]

Apart from an introduction of *PLoP conferences and the writers' workshop format, MiniPLoP featured a keynote talk by Peter Sommerlad, and writers workshop sessions on four selected paper submissions. This report provides an extended abstract of the keynote and short summaries of all papers discussed during the workshop.

The organizers would like to thank the participating authors and non-author participants who contributed to the writers' workshop for providing their material for discussion and for giving constructive feedback to others. Our special thanks go to Peter Sommerlad for giving an inspiring keynote presentation. Additionally, we thank Paris Avgeriou and Uwe Zdun, the ECSA 2014 chairs, for supporting the workshop.

August 2014,

Neil Harrison
Uwe van Heesch
Stefan Sobernig

2. HOW PATTERNS SHAPED MY LIFE

In his keynote talk, Peter Sommerlad reflected on his personal history with Patterns and the Pattern community.

In the late 1980s, when Peter learned about C++ as a C expert, he liked the possibility to directly model abstract data types with classes, but couldn't get his head around the good uses of C++'s keyword `virtual` to introduce dynamic polymorphism. While

[1] See also www.europlop.net for some background on the writers' workshops held at EuroPLoP.

working at Siemens from 1990 on, he got in contact with early drafts of Erich Gamma's PhD thesis and he also used Erich's and André Weinand's application framework ET++. Both of these experiences were an eye opener for Peter to better understand what object-orientation is about and how to make useful application of C++'s `virtual` keyword. He then gave a lot of OO courses using Design Patterns [7], like COMMAND, TEMPLATE METHOD, STRATEGY and COMPOSITE to motivate object-oriented design and development.

That contact with Design Patterns before the seminal book [7] was published and the collaboration with his Siemens colleagues led to POSA 1 [2]. Among many influencers, Jim Coplien (Cope) was the one who taught the Siemens POSA team the meaning and value of *forces* in a pattern description. His influence heavily shaped POSA's 2nd generation pattern format that inspired many other pattern authors with honest pattern descriptions that clearly show the applicability of a pattern through its forces as well as its limitations through its liabilities.

A personal reflection of a POSA author could not be without a reflection on POSA's REFLECTION pattern. In addition to language-supported reflection mechanisms, a Do-It-Yourself approach, as described patterns like TYPE OBJECT, PROPERTY LIST, or ANYTHING remains to be a valuable element in a developer's toolbox.

In addition to technical proficiency, patterns also inspired the pattern community that is celebrating its 20th anniversary this year with the 21st Pattern-Languages of Programming (PLoP) conference. Being part in this community gave Peter the opportunity to meet and discuss with many thought leaders of the software community, especially OO and Agile people, over the past 20 years. Peter is very thankful for the learning experience and inspiration he gained from the community at PLoP conferences. In a personal view, he learned to give and receive honest and constructive feedback through the PLoP's writer's workshops and shepherding; the latter both as a author (sheep) and coach (shepherd). Digging out concepts of a solution from a concrete design, as needed in pattern writing, helped hone his abstraction skills.

There are some regrets about what happened with patterns, in general, and POSA 1 [2] patterns, in particular. For example, the POSA authors weren't critical enough about SINGLETON, which turned out to become an anti-pattern once better understanding its implications; especially in the context of concurrent and parallel systems. Another regret of Peter is that POSA 1 did not separate a multi-tier architecture from the LAYERS pattern. This is because a multi-tier architecture does not get the abstraction and reuse benefits promised by LAYERS. What also happened with patterns and pattern-oriented software architectures especially was that many more people could now claim to be able to act as a software designer and architect. This ability often adds accidental complexity leading to many too complicated systems. He would like to see architects addressing *simplicity* in system designs as an overall design goal much more deliberately than it is today.

In the late 1990s, Peter moved to Switzerland to take over a small team from Erich Gamma in a small IT company after Erich left for founding the OTI (later IBM) research lab in Zurich. Peter grew that team again and created among other things security infrastructures for the Swiss banking and financial information industry. The experiences from this security-sensitive application domain led to his contribution to Security Patterns published in 2004 [16]. In between, Peter suffered his personal Y2K problem by being hit by a severe illness that took him almost up to 2004 for treatment and recovery therefrom. After that, he started a new live as professor for computer science at FHO HSR Rapperswil. He currently teaches patterns and C++ and continues to utilize in his own lectures and projects the lessons patterns and the pattern community taught him.

3. WORKSHOPPED PAPERS

This section contains summaries of the papers discussed at MiniPLoP, in chronological order of discussion.

3.1 SIS Pattern for Collective Intelligence Systems

Collective intelligence (CI) is an established phenomenon researched in several fields like sociology, biology, political science and economics [11]. *Stigmergy* is known as a coordination mechanism which provides computational systems with effective bottom-up, environment-mediated coordination capabilities [15].

Context. People organize themselves in collectives like groups, organizations, communities and societies for their common/mutual benefit. If people share certain knowledge and information, the collective will get more effective and efficient, which vice versa benefits each individual.

Problem. There exists an information communication problem, where knowledge and information are distributed among individuals and thus are difficult to access on a collective level.

The problem is affected by the following forces:

1. Lack of structured coordination of collective action of users.
2. The information transfer (quality, timeliness) depends on the individual users and, therefore, is inconsistent and not well integrated.
3. Each user is situated in a specific context and, thus, has low awareness about available remote information from other contexts, its status, relevance and sources.

Solution. A Stigmergic Information System (SIS) consists of (1) human agents as proactive components, (2) a single, homogeneous, coordination artifact network as a passive component, and (3) a computational coordinator system as a reactive component [13, 12]. The SIS architecture (see Figure 1) creates a perpetual feedback loop between agent base (1) and coordination infrastructure (2,3), by instrumenting the actors' contributions to stimulate a subsequent reaction by other actors, resulting in a stigmergy-like process.

The **actor base** consists of human agents, who independently engage with the coordination environment that is formed by the artifact network and the coordinator system.

The **artifact network** consists of interlinked coordination artifacts, whereby each coordination artifact consists of a set of predefined attributes which is the same for all artifacts. Actors can create artifacts, modify the values of their attributes, and link artifacts together using artifact links.

The **coordinator system** is a reactive/adaptive computational system, which uses triggers to propagate changes of the coordination artifacts to actors. Triggers raise awareness among actors

about ongoing activities in the artifact network, and subsequently stimulate them to provide more contributions to an artifact.

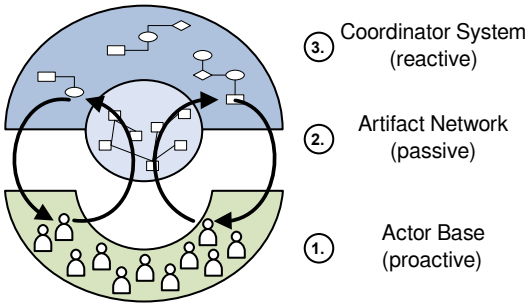

Figure 1: SIS architecture overview with stigmery cycle (adapted from [12]).

Example: Facebook. The coordination artifact in Facebook is the user profile. The actor base comprises people that can be described with the organizational structures and processes of a society. Each actor is the owner of only one profile. Users primarily contribute to their own profile pages, but they can also contribute directly to information shared by other users or their profile pages. Artifact links are represented by the friend relationship between two profiles and are defined by the actors.

Benefits. The SIS pattern has the following advantages:

- Harnessing the wisdom of crowds: The aggregated knowledge / information of a group of actors is better than knowledge / information of a single actor.
- Division of labour: Each artifact evolves by additive contributions and modifications of different actors.

Future Work. Based on the feedback of MiniPLoP participants, the following areas have been identified for future work:

- Extending the descriptions of examples and scenarios to include workflows making the abstract elements and processes more tangible.
- Exploring the relationships of the SIS pattern to other already existing architecture patterns (e.g. blackboard pattern), their commonalities, and differences.
- Describing the individual solution elements (i.e. coordination artifact, trigger) in greater detail and mapping them to well-known examples.

Paper title
The Stigmergic Information System Architecture Pattern for Socio-Technical Collective Intelligence Systems

Authors
Angelika Musil, Juergen Musil, and Stefan Biffl

3.2 Software Architecture Pattern Morphology

Software architecture patterns assist in the understanding of the structure of the architecture and the rationale of the decisions behind the architecture, and are found in virtually all complex systems [8]. As a system evolves, its architecture changes; this includes changes to the architecture patterns [1]. The changes make it more difficult to find the architecture patterns, greatly reducing their benefit. However, some architecture patterns are structurally similar to each other. Certain types of changes to a pattern may result in the pattern changing to another closely related pattern. Where this occurs, it helps preserve the benefits of using architecture patterns.

For example, the Client-Server pattern [1] consists of a component called the Server, which receives requests and performs some service in response. The other component is the Client, which initiates requests to the Server. The Client and Server usually run on different computers. There are typically multiple Clients, which do not have to be homogeneous. Now consider the Broker pattern. This pattern also consists of Clients and a Server, but communication between them is mediated by a Broker component between them. One powerful use of the Broker pattern is to manage multiple identical Servers, forwarding requests from Clients as Servers become available. Multiple Servers can increase capacity and/or improve availability of the system. A system using the Client-Server pattern might easily change by the addition of a mediating component, causing the Client-Server pattern to change to the Broker pattern.

We have identified several pairs of closely related patterns, and describe the structural changes needed to change one pattern to the other. See [9] for a description of the types of structural changes that architectural patterns can undergo.

If a pattern is to change to another, it is not sufficient that there is a set of minor structural changes leading from one pattern to another. There must be a plausible external motivation to make those changes. The changes must make sense from a requirements point of view. In the Client-Server to Broker morphing described above, the addition of a mediating Broker component is entirely plausible when one considers the typical evolution of such a system.

For each pattern morphing we identified, we considered whether there is a plausible motivation to make such a change to a system using the original architecture pattern. In most cases, we found a plausible reason for such a change, which tends to support the pattern morphing. One case was notable for its lack of a reason: The Active Repository and Blackboard patterns are somewhat structurally similar; an Active Repository could become a Blackboard with the addition of one or more components. However, they are somewhat different in their intent, and we could not see an obvious reason for this morphing. Therefore, we considered this to be not a legitimate morphing.

The pattern morphing paths ease system extension in that they form natural system evolutionary paths. If the system changes from one pattern to another, the pair of morphing patterns form a design path. This is in fact following the sequence of pattern application found in pattern languages.

We intend to further this research through studies of pattern morphing in experimental settings. In addition, we hope to find examples of pattern morphing in industrial settings. We also intend to study the body of architecture patterns further to identify a comprehensive set of architecture pattern morphings.

Paper title
Software Architecture Pattern Morphology

Authors
Neil B. Harrison, Paris Avgeriou, and Uwe Zdun

3.3 Systematic Selection of Architectural Patterns

Even if the requirements for a software system are stated clearly, the design of a suitable software architecture that meets these requirements remains a challenging task for the software architect since both functional and quality requirements have to be considered. Reusing best practices and common knowledge captured in architectural patterns has shown to be valuable in this context. However, it is not always trivial to select the appropriate pattern for a given problem, e.g. because of interdependencies between benefits and liabilities of multiple patterns. Many existing approaches for deriving software architectures from a set of requirements therefore rely on skilled and experienced software architects.

We present a process to select appropriate architectural patterns [2, 1] supporting the design of a software architecture that meets its functional and quality requirements. To bridge the gap between requirements and software architecture, we established a question catalog that relates questions targeting the requirements and their answers to architectural patterns.

Prior to applying our process, the software engineer has to model requirements using *problem diagrams* [10], each being an instance of a certain problem frame. In a first step, a matching problem frame has to be found for each problem diagram given as input. From the set of all questions within the question catalog, the subset of questions that are related to the identified problem frames are instantiated and presented to the architect. Based on the answers given to the questions, appropriate patterns are filtered from the set of all patterns. The architect can then either select a pattern and finish the process or continue the process by answering open questions that are follow-ups to the initial questions. Finally, we envision the application of optimization techniques to rank patterns according to their benefits and liabilities in relation to the answers given in former steps. In this way, the architect is supported in the final decision making on an appropriate pattern.

In summary, we provide a problem-oriented approach for selecting architectural patterns addressing functional and quality requirements. The process we propose connects requirements and software architecture, supports even less experienced software architects in the design of architectures through a guided pattern selection process, and provides support for decision making and documentation of design decisions and their rationale.

There are several improvements to our process that we aim at realizing in the future, partially based on the constructive feedback we received during the MiniPLoP workshop. Currently, we are elaborating the relationships between architectural tactics and patterns and integrate tactics into our process. We have already evaluated the proposed process on a subset of the CoCoME example [14], but we will extend the evaluation to cover the full example. Further, we plan to conduct controlled experiments as well. Since we identified several steps that are suited for automation, we are currently developing tool support that guides the architect through our process. Finally, we plan to embed the presented process into our GenEDA[2] method, as a first step to generate architectural alternatives from quality requirements.

[2] http://www.geneda.org/

Paper title
Towards Systematic Selection of Architectural Patterns with Respect to Quality Requirements

Authors
Azadeh Alebrahim, Stephan Faßbender, Martin Filipczyk, Michael Goedicke, and Maritta Heisel

3.4 Problem-oriented Pattern Retrieval

It is still a challenge for a software engineer to handle patterns in her daily practical work. At first, it is still difficult for a software engineer in her role of a pattern consumer to retrieve and to apply a suitable pattern for the problem at hand. Secondly, it is difficult for a software engineer in her role of a pattern provider to develop new patterns, being sure that such a pattern does not yet exist already. To support domain experts to take advantage of analysis patterns as well as design patterns across expert domains without leaving their field of expertise, we provide our approach of *Problem-oriented Pattern Management Methodology* (ProPMan). ProPMan consists of a set of methods that serve the mentioned requirements of finding, understanding and applying patterns.

In this paper, we present a component of ProPMan that we developed to offer tool-supported cross-domain pattern retrieval, the *Problem-oriented Pattern Retrieval* (ProPRet) method. ProPRet consists of a process description as well as a tool implementation. The method makes use of a graph matching approach, applied to graphs we call *Problem-Context Pattern* (PCP) graphs. It enables a software engineer in her role of a pattern consumer to perform a similarity search for the artefacts that represent the problem part of patterns called *problem-context patterns*. Problem-context patterns are the output of applying another ProPMan method, the *Problem-oriented Pattern Generation* (ProPGen) method, to domain-specific UML models. See Figure 2 for an overview.

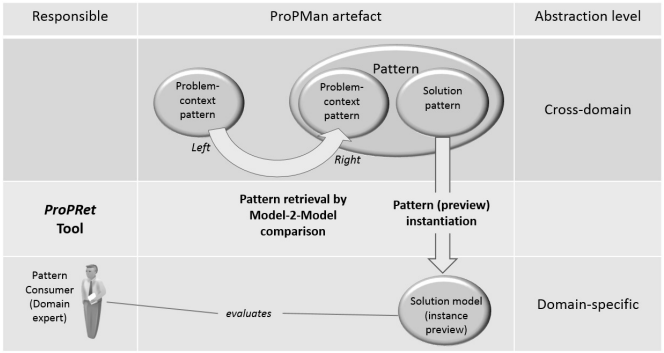

Figure 2: Conceptual overview of the ProPRet approach.

In the ProPRet similarity search approach, the prerequisite is that a pattern consumer generates a problem-context pattern from a problem-bearing, domain-specific UML model beforehand. In order to do this, he or she applies the ProPGen method and hence provides such a problem-context pattern as a search key. This problem-context pattern is then compared with problem-context patterns that have so-named "solution patterns" assigned and which both have been stored in a pattern library by a software engineer in his or her role of a pattern provider - using ProPGen as well. The result list of a pattern retrieval run contains items that are still cross-domain, as these items are pairs of generic problem-context and solution patterns. However, ProPRet enables domain experts to understand and hence compare the retrieval results by

transforming them to a domain-specific context - by providing previews on solution pattern instances.

For the ProPRet similarity-search approach, we introduce a similarity metric for PCP graphs based on the findings described in Dijkman et al. [3]. Furthermore, to give tool support for ProPRet and also for the remaining methods of our overall ProPMan methodology, we use the Eclipse framework. For the comparison of problem-context patterns, we use the Epsilon Framework as an extension of Eclipse. Epsilon is a family of languages and tools for code generation, model-to-model transformation, model validation, comparison, migration and refactoring that works out-of-the-box with EMF and other types of models. We are currently implementing an Epsilon comparison language (ECL) program to realize the ProPRet retrieval tool.

In the context of the overall ProPMan methodology, we presented an initial draft of an approach for cross-domain reuse of problem solutions via analysis patterns before [4]. A proposal for problem-oriented documentation of design patterns as a first draft of another member of the ProPMan method family, namely the *Problem-oriented Pattern Extension* (ProPExt), has been published in [5].

Paper title
A Method for Cross-Domain Problem-oriented Pattern Retrieval

Authors
Alexander Fülleborn and Maritta Heisel

---

4. REFERENCES

[1] P. Avgeriou and U. Zdun. Architectural patterns revisited: A pattern language. In *Proc. of the 10th European Conference on Pattern Languages of Programs (EuroPloP'05)*, pages 1–39. Hillside, 2005.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 2000.

[3] R. Dijkman, M. Dumas, and L. García-Bañuelos. Graph matching algorithms for business process model similarity search. In *Proc. of the 7th International Conference on Business Process Management (BPM'07)*, volume 5701 of *LNCS*, pages 48–63. Springer, 2009.

[4] A. Fülleborn and M. Heisel. Methods to create and use cross-domain analysis patterns. In *Proc. of the 11th European Conference on Pattern Languages of Programs (EuroPLoP'06)*, pages 427–442. Universitätsverlag Konstanz, 2007.

[5] A. Fülleborn, K. Meffert, and M. Heisel. Problem-oriented documentation of design patterns. In *Proc. 12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, number 5503 in LNCS, pages 294–308. Springer, 2009.

[6] R. P. Gabriel. *Writers' Workshops & the Work of Making Things: Patterns, Potery ...*. Addison-Wesley, 2002.

[7] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley, 1994.

[8] N. B. Harrison and P. Avgeriou. Analysis of architecture pattern usage in legacy system architecture documentation. In *Proc. 7th Working IEEE/IFIP Conference on Software Architecture (WICSA'08)*, pages 147–156. IEEE, 2008.

[9] N. B. Harrison and P. Avgeriou. How do architecture patterns and tactics interact? A model and annotation. *J. Syst. Softw.*, 83(10):1735–1758, 2010.

[10] M. Jackson. *Problem Frames: Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[11] T. W. Malone, R. Laubacher, and C. Dellarocas. Harnessing crowds: Mapping the genome of collective intelligence. Working Paper 2009-001, MIT Center for Collective Intelligence, 2009.

[12] J. Musil, A. Musil, and S. Biffl. Towards a coordination-centric architecture metamodel for social web applications. In *Proc. of the 8th European Conference on Software Architecture (ECSA '14)*, volume 8627 of *LNCS*, pages 106–113. Springer, 2014.

[13] J. Musil, A. Musil, D. Winkler, and S. Biffl. A first account on stigmergic information systems and their impact on platform development. In *Companion Proc. of the Joint 10th Working IEEE/IFIP Conference on Software Architecture & 6th European Conference on Software Architecture (WICSA/ECSA'12)*, pages 69–73. ACM, 2012.

[14] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil. *The Common Component Modeling Example: Comparing Software Component Models*. Springer, 2008.

[15] A. Ricci, A. Omicini, M. Viroli, L. Gardelli, and E. Oliva. Cognitive stigmergy: Towards a framework based on agents and artifacts. In *Proc. of the 3rd International Conference on Environments for Multi-Agent Systems (E4MAS'06)*, volume 4389 of *LNCS*, pages 124–140. Springer, 2007.

[16] M. Schuhmacher, E. Fernandez-Buglioni, D. Hypertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. Wiley Series in Software Design Patterns. John Wiley & Sons, 2000.