Putting Information Foraging Theory to Work: Community-based Design Patterns for Programming Tools

Tahmid Nabi [1], Kyle M.D. Sweeney [1], Sam Lichlyter [1], David Piorkowski [1], Chris Scaffidi [1], Margaret Burnett [1], Scott D. Fleming [2]

[1] Center for Applied Systems and Software
School of Electrical Engineering and Computer Science
Oregon State University
Corvallis, OR, USA
{nabim, sweeneky, lichlyts, piorkoda, Christopher.Scaffidi, Margaret.Burnett}@oregonstate.edu

[2] Department of Computer Science
University of Memphis
Memphis, TN, USA
Scott.Fleming@memphis.edu

***Abstract*—The design of programming tools is slow and costly. To ease this process, we developed a design pattern catalog aimed at providing guidance for tool designers. This catalog is grounded in Information Foraging Theory (IFT), which empirical studies have shown to be useful for understanding how developers look for information during development tasks. New design patterns, authored by members of the research community for the catalog, concretely explain how to apply IFT in tool design. In our evaluation, qualitative analyses revealed the community-written design patterns compared well in quality to patterns that we had ourselves published in a smaller, peer-reviewed catalog.**

Keywords—tool design; software engineering; applied theory

I. Introduction

Tools play a central role in enabling developers to find information efficiently during development tasks. For example, such tools include search and recommendation functions that can help a developer find the location of a bug in order to fix it [20][21], or to leave and view notes for one another [30]. To date, designers have relied primarily on intuition and empirical study for tool ideas. For example, one tool embodied the insight that developers often need to navigate through code based on what lines of code *could* be called, and the tool included a novel static analysis to support navigation [19]. But this insight was gleaned only after lengthy empirical work [18].

Hence, tool designers could benefit from a synthesis of the literature in a form that highlights open areas and sparks insights. We took the first step toward this goal with a literature review [8] framed by Information Foraging Theory (IFT) [25], a theory that can explain and predict how developers seek information [20][21][24]. We examined software engineering papers and explained how programming tools revealed ways of applying IFT in practice [8], yielding 12 design patterns summarizing how those tools applied IFT concepts.

A key limitation of that preliminary catalog is that it only incorporated our own research group's perspectives. Our current paper therefore presents an expansion of this design pattern catalog through a community-based process. Researchers from around the world contributed 16 additional design patterns, which broadened and deepened the range of ideas for how to apply IFT for tool design. We evaluated these new design patterns through a qualitative analysis.

II. Background and Related Work

Information Foraging Theory (IFT) offers a framework for conceptualizing developer behavior [8][17][20][21][24]. To briefly summarize IFT as it applies to software engineering, developers hunt like *predators* for information in a *topology*, which consists of *patches* of code or other views connected by navigable *links*. Patches contain information *features* that have *value* in the context of the developer's current task. A link has a certain *cost*, often measured in time or effort. Links may be annotated with certain *cues* (e.g., labels) indicating where those links lead. Developers try to *maximize* value relative to cost.

Prior empirical work extensively validated the benefits of applying IFT. For example, tools motivated by IFT can predict where developers will navigate and offer links to reduce foraging cost [20][21][24], organize files visually to minimize navigation cost [12], summarize code to reduce effort of program understanding [2], help developers find appropriate versions of programs during reuse [16], and search for needed API documentation [31].

We seek to go beyond the design of individual tools and establish how IFT provides broad guidance for tool design in general. Our approach is to synthesize insights from tools into design patterns, which are general, reusable solutions to common design problems [4][9]. Design patterns are abstract enough to generalize solutions among multiple situations and approaches [10][23][29], but concrete enough to aid developers in everyday design work [26]. Design patterns exist for security-related features [32], agent-based and service-oriented architectures [15][28], object-oriented systems [3][9][34], embedded systems [1][7], and visualization tools [13]. Our literature review extended this list by showing that design patterns could describe insights to guide programming-tool design [8]. For example, our catalog included the Dashboard pattern, which refers to an information patch in which a developer can become aware of links that lead to continually changing patches that have high value.

Researchers have taken a variety of approaches for evaluating design pattern catalogs. Some applied qualitative analyses (e.g., [15][32]), others obtained feedback from pattern authors (e.g., [6][11]), and still others applied patterns and observed their benefits and weaknesses (e.g., [14][27]). We used the first of these three approaches in our evaluation.

This work was sponsored by the National Science Foundation

III. Community-Based Approach to Establishing a Catalog of Design Patterns

Below, we describe how we recruited and trained pattern authors, then reviewed their work. We also discuss new insights embodied in the design patterns that they contributed.

A. Recruiting researchers to serve as design pattern authors

To identify potential authors, we used the ACM Digital Library to search relevant conference proceedings (e.g. ICSE, FSE/ESEC, ASE and ICSME) for papers that described a software engineering tool. We used combinations of keywords related to software engineering and specific development tasks ("code," "debugging," "foraging," "maintenance," "navigation," "refactoring," "reuse," "software," "source," "tool," and "visualization"). We manually reviewed search results to confirm relevance, then emailed authors who were not graduate students. The response rate was 20%, which we consider high for such a time-demanding task as authoring patterns.

We collected design patterns via a wiki that provided introductory materials to help authors begin. The *Getting Started* page provided links to other pages in a logical progression from a basic understanding of IFT, to a familiarity with example IFT-based design patterns, and finally to the process of how to author a new one. The *IFT Primer* explained the key IFT constructs (e.g., patches, links, and value) and illustrated these with relevant examples. The *Walkthrough* consisted of a PowerPoint with embedded videos helping authors understand how our research team actually constructed one IFT-based design pattern. The *Pattern Description* page explained how to organize information into our design-pattern format (Table I), which extended the original object-oriented design patterns format [9] with a new "Connection to IFT" section. The *List of Patterns* provided a list of the extant design patterns, including icons indicating each pattern's completion status. The *Rules* page stated requirements that we enforced—principally, that each design pattern had to be on-topic, complete, not a duplicate, and relevant to tool design.

TABLE I. SECTIONS OF IFT-BASED DESIGN PATTERNS. SEE http://research.engr.oregonstate.edu/IFT FOR THE PATTERNS THEMSELVES.

Pattern Title: A memorable phrase summarizing the design pattern
Intent: A statement answering: What does the design pattern do? What is its rationale and intent? What is the information foraging issue?
Motivating Example: A real-life scenario illustrating an information foraging problem that tools implementing the design pattern solve.
Description: A description of how the pattern works, including: the pattern's input(s), how the pattern uses these, and its output(s).
Applicability: In what situations can the design pattern be applied? Include any assumptions made and all conditions that must be met.
Connection to IFT: Use IFT terms and constructs to explain how the tool aids with solving the information foraging problem.
Consequences: What are the tradeoffs and the results of this pattern? Subdivide these into benefits and liabilities (pitfalls of pattern misuse)
Known uses: Examples of the pattern found in real tools, including descriptions of how they implement or represent the pattern in action.
Related Patterns: What design patterns are related to this one? Note similarities and differences. With what other patterns can it be used?

B. Reviewing and revising of design patterns

Authors could start by either authoring new design patterns or by contributing Known Uses to existing patterns authored by other people. We frequently logged into the wiki and reviewed edits. We contacted each person by email if a week went by between posting new material. We also emailed authors suggestions for improving patterns, and we answered inquiries about how to finish the work. Authors either incorporated feedback or explained a rationale for declining suggestions. Several iterations of this feedback-modification process typically occurred until at least two members of the study team charged with approving the work both agreed that a given design pattern fulfilled all the rules. Authors received $100 for every completed design pattern, up to 3, as well as $10 for every Known Use written (for any design pattern, including those of other authors), up to 20, with a maximum of up to 3 Known Uses per design pattern.

C. Resulting catalog of IFT-based design patterns

Our 9 authors provided 16 design patterns between April 2015 and January 2016 (Table II).

TABLE II. NAMES OF DESIGN PATTERNS CONTRIBUTED BY PARTICIPANTS, WITH CORRESPONDING INTENTS (EDITED FOR SUCCINCTNESS)

Documentation Processing: Give developers a high level description of source code, without having to navigate through the code.
Extract Method Refactoring: Restructure the topology by extracting statements that are highly related into a separate method and creating a new patch.
Fault Localization: Identify the sections of code that are responsible for an undesired behavior of software.
Heuristics-based Code Completion: Group a set of functions by their relatedness to the current coding context
Impact Location: Identify source code affected by the alteration of a different section of code.
Online Feedback Miner: Extract from forum discussions API features that have caused problems for developers
Patch Prevalence: Provide information foragers more prevalent patches so as to more quickly arrive at a potentially profitable patch.
Patch Profitability: Indicate how much value an entire information patch yields for fulfilling information-seeking goals
Path Search: Search a path in a topology, collapsing the topology to a list of prey containing cues matching the predator's information goal.
Recollection: Find a previously known class or method that is relevant to the task at hand.
Reduce Duplicate Information: Reduce the size of the topology by eliminating nodes with duplicate information.
Rename Refactoring: Rename methods to reflect information contained, highlighting aspects relevant to expected future foraging
Shopping Cart: Allowing developers to accumulate a list of patches for extra vetting
Software Visualization: Characterize domain elements, e.g. structural program elements, by visualizing metrics and properties
Test Coverage: Monitor coverage of a unit test suite to ease software maintenance and evolution
Visualize Topology: Reveal the structure of the topology, helping developers to move along relationships and choose patches to visit

Their contributions covered a variety of topics that our earlier catalog of design patterns, published in TOSEM [8], had not addressed.

For example, our preliminary catalog included few patterns showing how to reduce the cost of processing information patches—an objective addressed by several novel design patterns from the new contributors. For instance, the *Documentation Processing* design pattern referred to tools that automatically parse and extract information from documentation into summative patches. The *Online Feedback Miner* pattern described tools, such as Haystack [33], that automatically digest online conversations to provide summative information to developers.

Our earlier catalog also lacked substantial coverage of patterns describing how tools could assist developers in making sense of large topologies, a topic covered better by the new design patterns. For example, the *Shopping Cart* cited TraCter, a tool for traceability analysts to collect patches in a topology so they can subsequently view those patches and examine them in detail [22], and the *Visualize Topology* design pattern referred to tools that depicted the relationships among patches. One pattern, *Patch Prevalence*, described an approach for decreasing cost by increasing the density ("prevalence") of high-value patches through, in essence, compressing or otherwise transforming the topology. It cited, as an example, Code Bubbles, which enables the visual juxtaposition of high-value patches within a window offering low-cost between-patch navigations [5]. The Patchworks code editor supports a related approach with a similar effect [12].

Finally, the new design patterns also discussed how tools can reduce cost in situations where developers face a sequence of foraging episodes—which our own patterns did not address. For instance, the *Recollection* design pattern explained how tools can help developers find their way back to places that they have visited before. The *Rename Refactoring* and *Extract Method Refactoring* design patterns discussed tools that enable the developer to modify the topology in order to reduce the cost of future foraging activities by improving maintainability. Although our preliminary work had explained how tools can aid developers in finding information needed before performing refactoring tasks, we (unlike our community authors) had not made the connection between the act of refactoring and the *future* cost of foraging.

IV. Evaluation

We assessed how well the 16 community-generated design patterns met general quality criteria, relative to our preliminary collection of 12 published in TOSEM [8]. Our logic was if the new patterns matched our peer-reviewed patterns in quality, then they were also suitable for publication.

A. Methodology

We performed a series of qualitative analyses that blended theoretically derived code sets with qualitative coding and focused on three areas.

a) *Coverage of IFT:* How well do design patterns, together, cover IFT-related objectives? For this analysis, we categorized design patterns based on a code set derived from the constructs of IFT.

b) *Abstraction & generalizability:* How abstract is each design pattern, and how well does it generalize across development tasks? Here, we categorized design patterns based on an adapted code set from Yskout et al [32] and a second code set developed through open coding.

c) *Evidence of usage*: To what extent has each design pattern found actual use? For this analysis, we categorized Known Uses using a code set from open coding.

Reliability: Two researchers defined rules for applying code sets, then independently coded 20% of instances. A code set was considered reliable if had an agreement of at least 80% using the Jaccard index. In situations where we did not meet this criterion, we revised our rules and repeated the evaluation until converging to a reliable coding scheme. Subsequently, one researcher coded remaining data. While coding the 28 design patterns, we did not pay attention to whether each was authored by us or by our recruited authors.

B. Results

1) Coverage of IFT

We constructed a code set by considering the constructs present in IFT and how a tool might modify these constructs to assist either in the present (as the developer is foraging) or in the future (when the developer might have to forage again). This led to identification of 10 IFT-related objectives:

- Improve alignment of expected value with actual value
- Decrease current cost of navigation
- Locate the prey of interest for the predator
- Decrease current cost of processing a patch
- Decrease future cost of navigation
- Increase the future value of information features
- Decrease the future cost of processing a patch
- Draw developer's attention to certain cues
- Increase current value of information features
- Improve alignment of expected cost with actual cost

In contrast to our TOSEM catalog, which only addressed 8 of the 10 objectives, the community-generated design patterns covered all 10. Moreover, for any given *level* of coverage (expressed as a number of design patterns addressing each objective), the community-generated patterns met or exceeded ours (Fig. 1). Thus, we concluded the community-generated patterns covered objectives as well as our preliminary catalog.

2) Abstraction and generalizability

Yskout et al. previously assessed their catalog of security-related design patterns in terms of 6 levels of abstraction [32]. When we applied the code set to our catalog, we found all of the design patterns were in only 2 of their 6 categories. If a design pattern had a well-defined context indicating one or more specific situation where the solution could apply, and if the design pattern also contained implementation details, then we refer to it as "Concrete." Otherwise, in the absence of a defined context and/or implementation details, we coded it as "Non-Concrete." (Yskout et al. referred to these codes as "Algorithm" and "Technique," but we feel that our own labels here are more reflective of the two codes' definitions.)

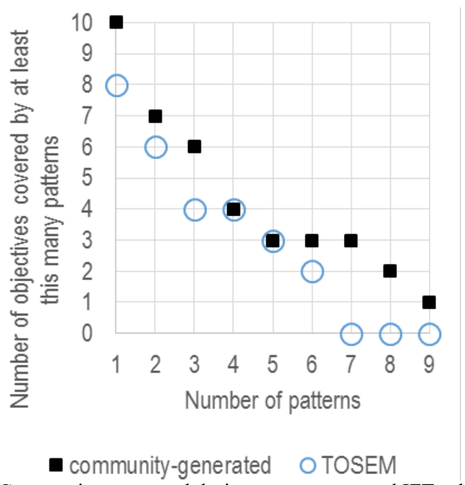

Fig. 1. Community-generated design patterns covered IFT-related objectives more thoroughly than did our own preliminary collection [8]

Based on these criteria, we categorized 25% of our TOSEM design patterns as Concrete and 75% as Non-Concrete. In contrast, we categorized 56% of community-generated design patterns as Concrete and only 44% as Non-Concrete. In the sense that a Concrete pattern is more specific than a Non-Concrete pattern, these results suggest the community-generated patterns offered some advantages our own lacked.

As a measure of generalizability, we categorized each design pattern based on whether it supported multiple development tasks. We identified 6 tasks through open coding: coordinating developers, understanding code, mapping functionality to code, refactoring, and testing/debugging. To qualify as applying to a task, a design pattern had to give an example scenario or a Known Use for that task.

We found 50% of our TOSEM design patterns generalized across multiple development tasks. In contrast, only 25% of community-generated design patterns applied to multiple tasks.

Thus, community-generated design patterns tended to be more concrete but less generalizable than our own. Our analysis suggests the complementary strengths of community-based and literature-review approaches.

3) *Evidence of use in practice*

For each Known Use, we searched online to find information about the use, which we classified as follows:

- *Same-group research tool:* Applied to a prototype created by same research group as the design pattern's author
- *Other-group research tool:* Applied to a prototype created by another research group
- *Industrial tool:* Available commercially, and/or open source but under continuous development and in use by an active online community

We considered these categories to represent increasing evidence of the design pattern's practical utility.

As shown in Fig. 2, our TOSEM catalog averaged 3.17 Known Uses per design pattern, slightly exceeding the 3.13 per design pattern that our community authors provided. However, we provided slightly fewer industrial use cases per design pattern, at 1.08 compared to 1.38 for community-generated design patterns. We did not provide any Known Uses regarding our own prior tools; in contrast, approximately 1/3rd of research tools cited by community authors were created by their own groups. Overall, these results suggest that community-generated design patterns compared well to our own in terms of their evidence for use in practice.

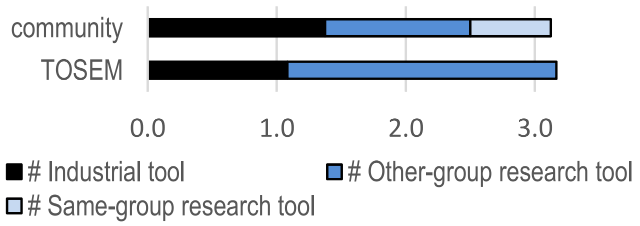

Fig. 2. Average numbers of Known Uses cited per design pattern (categorized according to whether each Known Use's tool was industrial, was created by the pattern author's research group, or was created by another research group)

V. Limitations and Threats to Validity

A key limitation uncovered by our work is that the community-generated design patterns tended to be of lower generalizability than our own. When applying a similar procedure with other theories in the future, a literature review or a post-processing step could yield complementary design patterns and ensure generalizability. Another risk to generalizability is that we recruited pattern authors based on software engineering publications; recruiting more pattern authors who have a background in HCI could be beneficial.

We did not ask programmers to implement tools using design patterns, so our evaluation of pattern quality might not match measurements of utility in practice. Future work could investigate the extent to which tool designers benefit from our design pattern catalog when turning designs into working code. Such additional evaluation could lead to valuable new insights for expanding and for applying the pattern catalog in practice.

VI. Conclusions

We have presented a community-generated design pattern catalog that expands our initial collection of IFT-based design patterns for programming tools. We now know (1) members of the research community can distill effective tool design patterns from literature when provided theory-based guidance; and (2) their patterns explained how to design tools aiding information-foraging objectives that our TOSEM patterns poorly covered. Most importantly, however, our work illustrates a path toward connecting a scientific theory of behavior with the practice of tool design.

Acknowledgement

This work is supported by an IBM PhD Fellowship and by the National Science Foundation (Grants 1101107, 1302113, and 1314384). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Armoush, A, Salewski, F, and Kowalewski, S. (2008) Effective pattern representation for safety critical embedded systems. *IEEE International Conference on Computer Science and Software Engineering*, 91-97.

[2] Athreya, B., and Scaffidi, C. (2014) Towards aiding within-patch information foraging by end-user programmers. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 13-20.

[3] Beck, K., Crocker, R., Meszaros, G., Coplien, J. O., Dominick, L., Paulisch, F., and Vlissides, J. (1996) Industrial experience with design patterns. *ACM/IEEE International Conference on Software Engineering (ICSE)*, 103-114.

[4] Borchers, J. (2000) A pattern approach to interaction design. *ACM International Conference on Designing Interactive Systems*, 369–378.

[5] Bragdon, A., Reiss, S., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., and Adeputra, F. (2010) Code Bubbles: Rethinking the user interface paradigm of integrated development environments. *ACM/IEEE International Conference on Software Engineering (ICSE)*, 455-464.

[6] Coplien, J., and Woolf, B. (1997) A pattern language for writers' workshops. *C Plus Plus Report*, *9*, 51-60.

[7] Fant, J. (2011) Building domain specific software architectures from software architectural design patterns. *ACM/IEEE International Conference on Software Engineering (ICSE),* 1152-1154.

[8] Fleming, S., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrance, J., and Kwan, I. (2013) An Information Foraging Theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Trans. Software Engineering and Methodology (TOSEM)*, *22*(2), 14.

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

[10] Gamma, E., Helm, R., Johnson, R. and O'Brien, L. (2009) Design patterns 15 years later: An interview with Erich Gamma, Richard Helm, and Ralph Johnson. *InformIT*. http://www.informit.com/articles/article.aspx?p=1404056

[11] Harrison, N. (1999) The language of shepherding. *Pattern Languages of Program Design, 5*, 507-530.

[12] Henley, A., Singh, A., Fleming, S., and Luong, M. (2014) Helping programmers navigate code faster with Patchworks: A simulation study. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 77-80.

[13] Hundhausen, C. (2005) Using end-user visualization environments to mediate conversations: A "Communicative Dimensions" framework. *Journal of Visual Languages and Computing, 16*(3), 153–185.

[14] Iacob, C. (2012) Using design patterns in collaborative interaction design processes. *ACM Conf. Computer Supported Cooperative Work Companion (CSCW)*, 107-110.

[15] Juziuk, J., Weyns, D., and Holvoet, T. (2014). Design patterns for multi-agent systems: A systematic literature review. *Agent-Oriented Software Engineering*, Springer Berlin Heidelberg, 79-99.

[16] Kuttal, S. (2013) Variation support for end users. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 183-184.

[17] Kuttal, S., Sarma, A., and Rothermel, G. (2013) Predator behavior in the wild web world of bugs: An Information Foraging Theory perspective. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 59-66.

[18] LaToza, T., and Myers, B. (2010) Developers ask reachability questions. *ACM/IEEE International Conference on Software Engineering (ICSE)*, 185-194.

[19] LaToza, T., and Myers, B. (2011) Visualizing call graphs. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 117-124.

[20] Lawrance, J., Bellamy, R., and Burnett, M. (2007) Scents in programs: Does Information Foraging Theory apply to program maintenance? *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 15-22.

[21] Lawrance, J., Bellamy, R., Bumett, M., and Rector, K. (2008) Can information foraging pick the fix? A field study. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 57-64.

[22] Mahmoud, A., and Niu, N. (2011) TraCter: A tool for candidate traceability link clustering. *IEEE International Requirements Engineering Conference*, 335-336.

[23] May, D., and Taylor, P. (2003) Knowledge management with patterns. *Communications of the ACM, 46*(7), 94-99.

[24] Piorkowski, D., Fleming, S., Scaffidi, C., John, L., Bogart, C., John, B., Burnett, M., and Bellamy, R. (2011) Modeling programmer navigation: A head-to-head empirical evaluation of predictive models. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 109-116.

[25] Pirolli, P., and Card, S. (1995) Information foraging in information access environments. *ACM Conference on Human Factors in Computing Systems (CHI)*, 51-58.

[26] Prechelt, L., Unger, B., Tichy, W., Brossler, P., and Votta, L. (2001) A controlled experiment in maintenance: Comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering, 27*(12), 1134-1144.

[27] Ramirez, A., and Cheng, B. (2010) Design patterns for developing dynamically adaptive systems. *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 49-58.

[28] Rischbeck, T., and Erl, T. (2009) *SOA Design Patterns*, Prentice Hall.

[29] Rising, L. (2007) Understanding the power of abstraction in patterns. *IEEE Software, 24*(4), 46-51.

[30] Storey, M., Cheng, L., Singer, J., Muller, M., Myers, D., and Ryall, J. (2007) How programmers can turn comments into waypoints for code navigation. *IEEE International Conference on Software Maintenance*, 265-274.

[31] Stylos, J., and Myers, B. (2006) Mica: A web-search tool for finding API components and examples. *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 195-202.

[32] Yskout, K., Heyman, T., Scandariato, R., and Joosen, W. (2008) *Security patterns: 10 years later*. Technical Report CW 514, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, https://lirias.kuleuven.be/bitstream/123456789/183886/1/CW514.pdf

[33] Zhang, Y., and Hou, D. (2013) Extracting problematic API features from forum discussions. *IEEE International Conference on Program Comprehension (ICPC)*, 142-151.

[34] Zimmermann, O., Zdun, U., Gschwind, T., and Leymann, F. (2008) Combining pattern languages and reusable architectural decision models into a comprehensive and comprehensible design method. *IEEE/IFIP Conference on Software Architecture (WICSA)*, 157-166.