Pragmatic Software Reuse: A View from the Trenches

Robert J. Walker and Rylan Cottrell

Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, AB, Canada
email: walker@lsmr.org

Technical Report 2016-1088-07

ABSTRACT

Software reuse has been a part of the software engineering field since its inception. Research on reuse has focused almost exclusively on pre-planned approaches. Relatively little has been written about reuse performed in the absence of its pre-planning: *pragmatic reuse*. While many academics have dismissed non-pre-planned reuse as ill-advised, very little evidence exists about it, especially with respect to industrial practice. We conducted a survey of 59 industrial software developers to capture the perception, frequency, motivations, difficulties, and execution of the practice of pragmatic reuse within their development activities. We found that the majority of developers surveyed perceive that: pragmatic reuse has an important place in their repertoire of techniques; pragmatic reuse tasks are a frequent part of their development activities; and that they face a variety of practical difficulties while performing pragmatic reuse tasks. Opinions vary on the range and scale of situations where pragmatic reuse is suitable.

1. INTRODUCTION

"We all know that copy-and-modify reuse should not be promoted." We have heard such sentiments expressed in similar form many times by academics. Two points are implied: (1) that copy-and-modify reuse is a necessarily bad practice, and (2) that everyone either accepts this "fact" or is ignorant. The evidence in the literature does not support either point: there is some evidence that copy-and-modify reuse can be a disciplined practice that is more appropriate in some contexts than the alternatives with no *evidence* that it is inherently negative; whether typical industrial practice accepts the purported "fact" remains an open question.

Traditional research into software reuse has focused on techniques that require explicit pre-planning and abstraction to some interface [9]. While the pros and cons of pre-planned* versus copy-and-modify approaches to reuse have been argued in the literature [14, 23], little evidence has been presented. Copy-and-modify is often mentioned briefly as a *pragmatic* choice in industry [e.g., 4, 6, 14, 18, 19, 25, 28], but we have little systematic information about the perception and practice of pragmatic reuse by software developers in industry. As a result, it is unclear whether cases of copy-and-modify described in the literature represent misguided or inexperienced developers, or common attitudes supported by disciplined practice.

*Note the difference between *planning* and *pre-planning*. A pre-planned approach looks to the future to design for reuse, whereas non-pre-planned approaches do not. The actual reuse of an artifact may or may not be planned by the developer as an initial phase of a specific reuse task, regardless of pre-planning (by someone else, most likely).

To address our poor understanding of the role that pragmatic reuse plays within the development activities of industrial developers, we conducted a survey to which 59 industrial software developers responded. The goal of the survey was to capture the perceptions, frequency, motivations, difficulties, and execution of pragmatic reuse tasks within the development activities of industrial software developers. We found that the majority of developers surveyed perceive: that pragmatic reuse has a useful place in their repertoire of techniques; that pragmatic reuse tasks are a frequent part of their development activities; and that developers face a variety of practical difficulties while performing pragmatic reuse tasks. Amongst our respondents, even the most ardent opponents of pragmatic reuse concede its appropriateness in some circumstances, and they themselves clearly perform pragmatic reuse sometimes. Overall, we found that our respondents had a surprisingly nuanced view of pragmatic reuse, recognizing its strengths and weaknesses, but approaching pragmatic reuse tasks with a disciplined attitude towards risk in its application.

The remainder of the paper is structured as follows. Section 2 details related work and how it does not adequately provide evidence as to industrial experience in this area. Section 3 describes our survey, methodology, and results. Section 4 analyses the data qualitatively and overviews a statistical analysis. Section 5 summarizes the key findings of the responses, the qualitative analysis, and the statistical analyses, in terms of our research questions. Section 6 discusses remaining issues.

2. RELATED WORK

Skepticism. The claim that all code duplication activities are inherently negative is a staple of folklore [e.g., 16]. Biggerstaff and Richter [3] warn that the payoff of pragmatic reuse is constrained by a low upper threshold, "probably less than 25%"; however, no evidence is presented to support this claim, nor any suggestion as to how these authors arrived at this value. Garlan et al. [10] present a case study on building a system from existing parts, reporting that conflicting assumptions were problematic. Others have pointed out that this is also the case in, for example, the integration of object-oriented frameworks that were individually pre-planned for reuse [27]. The software product-line literature [e.g., 20] often points to "clone-and-own" as the "ill-advised" alternative to software product lines, but without presenting evidence.

Principled use of pragmatic reuse. Any tool or approach can be misapplied, as many of us see from the questionable "object-oriented" designs often created by novices, but misapplication of a tool does not imply that the tool is fundamentally flawed. Krueger [19] appraises code scavenging (i.e., copy-and-modify) to have a potentially high payoff if a developer is able to find large source code fragments that are of high quality and that can be reused without significant modification; however, he claims that the effectiveness of code scavenging is severely restricted by its informality and lack of systematic support. Cordy [6] reports on his personal experience at Legasys Corporation identifying and systematically reprogramming Y2K risks for financial institutions. He claims that financial companies perceive the risk of new software to be so high that it is standard practice to copy an existing application and modify it to fit its new problem context. The overhead arising from maintaining an independent product is deemed acceptable here. Cai and Kim [5] found that clones are frequently short-lived, and when they are long-lived they are not easily refactored, suggesting that moving to a product-line architecture would not always be a simple proposition. Kapser and Godfrey [18] find that in 71% of cases, duplicate source code had positive impact on maintainability. They point out three rationales for why pragmatic reuse can be beneficial: refactoring does not aid the financial situation of an organization in the short-term; refactoring carries major risks that can be unacceptable; and redundancy provides better isolation of subsystems. Holmes and Walker [14] provide a process model for pragmatic reuse and a tool for systematizing it. They demonstrate that the use of their tool decreases time to perform pragmatic reuse, increases likelihood of success, and improves developers' sense of control over pragmatic reuse tasks, thereby addressing the concern of Krueger [19]. Makady and Walker [23] extend this work to leverage existing test suites to check whether a pragmatic reuse task has violated important properties of the reused code.

Evidence about pragmatic reuse in industry. Few studies look at the industrial application of pragmatic reuse. Lange and Moher [21] conducted an observational study of a single industrial participant over a one week period, finding that 85% of methods were built using pragmatic reuse. Barnes and Bollinger [1] report that informal reuse of early work products occurs primarily when highly experienced developers use their familiarity with the functionality and design of a code module set to adapt those modules to new, similar uses. Rosson and Carroll [25] performed an exploratory study with four industrial developers and found that pragmatic reuse tasks involve one decision at a time, while a developer "debugs the code into existence". Brandt et al. [4] report on a study that logged the history of web searches conducted during 300 query sessions about the Adobe Flex web application and found that 51% were code-only queries; however, they could not distinguish if the queries were performed for purposes of pragmatic reuse. Selby [26] analyzed 25 software projects from NASA and found that 32% of modules were reused and of those 47% of them were reused with revisions—in essence, that pragmatic reuse was needed. Since NASA is an organization that has emphasized reuse as a key productivity goal, the fact that such a degree of pragmatic reuse was performed suggests that pre-planned reuse did not suffice for them and that resorting to pragmatic reuse was a necessary and viable alternative. Van Ommering [28] reports that pragmatic reuse was not very effective with respect to software development tasks at Philips, but provides no details in support of this statement. Jansen et al. [17] studied the use of opportunistic and pragmatic reuse by two start-up companies performing rapid development of medium- to large-scale projects and found that these techniques were cost-effective, but the paper is short on details. Haefliger et al. [11] studied reuse in six popular open source software systems mentioning the reuse of methods and algorithms to be frequent, but no further detail was reported, as the main focus of this work was on the creation of reusable (i.e., via pre-planning) entities. Holmes and Walker [12] conducted a survey with twelve developers from six companies. They found that these developers perform pragmatic reuse tasks as a means to save time and improve quality, that they have access to large amounts of source code which they can reuse, and that they want to understand the dependencies a feature has on its system before they attempt to reuse that feature. That survey did not consider the frequency with which pragmatic reuse occurs, the difficulties faced, or the perception of pragmatic reuse as positive or negative. Begel et al. [2] surveyed a very large group of developers at Microsoft to explore how they coordinate their work. When asked how they "mitigate the effects (potential or actual) of depending on other teams", 63% selected the option "minimize code dependencies" (i.e., on other teams' codebases) and 49% selected the option "eliminate code dependencies (e.g., 'clone and own')". These results were not elaborated upon further, as pragmatic reuse was not the focus of their study.

None of these studies satisfies our desire to understand the place of pragmatic reuse in industry. The observational studies are deep and interesting but small; the individuals studied are not necessarily representative of the larger population, or their behaviour could have been uncharacteristic of their normal habits. The larger studies were limited to individual organizations, in which behaviours could have been dictated by management. None of the existing studies examines whether developers actually perceive pragmatic reuse as a disciplined, reasonable approach, nor examines closely why they do or do not pursue pragmatic reuse in practice. None provides general information on the scale of tasks that are undertaken, their frequency, the risk-mitigation strategies employed, or the difficulties encountered.

3. SURVEY

The previous work on industrial use of pragmatic reuse leaves many questions unanswered, and does not immediately address the criticisms of it. To begin to fill-in the gaps in our knowledge and to seek evidence to support or to refute the criticisms, we identified seven more specific research questions of interest, listed below.

RQ1. Do industrial software developers perceive pragmatic reuse as a disciplined practice that is relevant to their development activities?

RQ2. How frequently and at what scale do industrial software developers perform pragmatic reuse tasks as a part of their development activities?

RQ3. How frequently do industrial software developers abandon pragmatic reuse tasks because of unforeseen problems?

RQ4. What motivates a developer to choose to perform a pragmatic reuse task?

RQ5. What preparations do developers take before performing pragmatic reuse tasks?

RQ6. What difficulties do developers face while performing pragmatic reuse tasks?

RQ7. What are the factors in choosing pragmatic reuse over re-implementing the desired functionality anew?

3.1. Methodology

To address these research questions we designed a survey in which we were concerned about three factors: (1) the time impact on respondents; (2) the risk of biasing respondents' opinions; and (3) a broad examination of the motivations and perceptions of pragmatic reuse, rather than other issues surrounding reuse in general or mechanisms employed in conducting pragmatic reuse.

We used SurveyMonkey,[†] a proprietary, web-based, survey management system. SurveyMonkey imposed a limit of 10 questions in its free form, acceptable given our concern about the time impact on respondents. To populate the survey questions, we considered positive and negative claims about pragmatic reuse in the literature, augmented from our own experience. We ran multiple dry-runs with colleagues to find problems and biases in the questions, resulting in multiple rounds of revision.

To reach a wide range of industrial software developers, respondents were recruited by posting a request for assistance on a variety of developer forums (Eclipse community forum, Microsoft Developer Network community forum, Nokia developer forum, Android developer forum, HP WebOS developer forum, and NetBeans forum), social networks (Twitter, Facebook, LinkedIn, and reddit), and developer groups hosted on social networks (Canadian developer connection, and Global software engineering). Our survey mechanism collected respondents' IP addresses to prevent likely duplicate responses; upon the closing of the survey, this information was deleted to help ensure the anonymity of the survey respondents, although we can report that we received responses (apparently) from every continent except Antarctica. The survey did not collect identifying information. The survey was first opened 29 December 2011 (with occasional reminders sent to the forums listed above) and closed 31 March 2012. The survey and recruitment process was reviewed and given clearance under the research ethics approval process of the University of Calgary.

3.2. Questions and Responses

To ensure that respondents had a common understanding of what we meant by "pragmatic reuse", we presented them with the following definition:

> *Pragmatic reuse refers to the reuse of source code that was not designed with reuse in mind, for example, copying segments of source code and modifying these to integrate within a target system (also known as copy-and-paste reuse, code scavenging, and ad hoc reuse).*

After additional explanations required for informed consent under the University of Calgary's research ethics approval process, the survey proper consisted of 10 questions, divided into a total of 59 sub-questions.[‡] Questions 1–4 were designed to collect demographic information on the respondent. We saw these as potentially useful for the sake of correlating with other opinions, for example, whether people with more industrial development experience were more or less likely to

[†] https://www.surveymonkey.com/

[‡] The survey responses can be found at http://lsmr.org/docs/pragmatic-reuse-dataset.csv.

have a positive opinion of pragmatic reuse. Questions 5–10 were designed to more directly address the research questions.

A total of 69 respondents began the survey. We discarded all the responses from two respondents who reported zero years of industrial development experience, as we are only studying the opinions of persons with industrial development experience. We removed from consideration all blank responses of 8 respondents (R5, R8, R23, R40, R52, R53, R56, R57) who did not answer questions beyond Question 5, but report on their answers in this section. Our statistical analyses (Section 4.2) operate only on the 51 respondents who meet our criterion of industrial experience and who completed the survey.

Question 1. *What is your current employment status?* (59 total responses.) Employed primarily in software development: 34 (58%). Employed partially in software development: 11 (19%). Employed but without connection to software development: 1 (2%). Student: 12 (20%). Unemployed: 1 (2%). Retired: 0 (0%).

Question 2. *How many years' experience have you accumulated involving industrial software development?* (59 total responses, summarized by intervals.) 1–5 years: 29 (49%). 6–10 years: 15 (25%). 11–15 years: 5 (8%). 16–20 years: 6 (10%). 21–25 years: 4 (7%). The responses had a mean of 8.3 years, a median of 6 years, and a standard deviation of 6.8 years. In particular, we note that the 12 students report industrial experience with a mean of 5 years—perhaps these were graduate students or mature undergraduate students who returned to school after a period in industry.

Question 3. *How big is the biggest organization involving software development with which you have been employed?* (59 total responses.) $\leq$10 developers: 14 (24%). 11–100 developers: 15 (25%). 101–500 developers: 10 (17%). >500 developers: 20 (34%).

Question 4. *Please select the programming language(s) you use for development.* (200 total responses.) Java: 35 (59%). C++: 30 (51%). C#: 30 (51%). Objective-C: 11 (19%). C: 16 (27%). Python: 12 (20%). PHP: 17 (29%). JavaScript: 33 (56%). Other: 16 (27%).[§]

Question 5. *Pragmatic reuse refers to the reuse of source code that was not designed with reuse in mind, for example copying segments of source code and modifying these to integrate within a target system (also known as copy-and-paste reuse, code scavenging, and ad hoc reuse). Statement: Pragmatic reuse is a bad practice and should be avoided.* There were two radio buttons: *I agree with this statement* and *I reject this statement*, plus a freeform text field labelled *Please explain why*. Two-thirds of respondents (68%, 40 respondents) rejected the statement, while one-third (32%, 19 respondents) accepted the statement. We analyze the comments in Section 4.

Question 6. *Frequency and scale of pragmatic reuse, and abandonment.* This question was subdivided into 12 subquestions as explained in Table I. Each subquestion was associated with six radio buttons: "Always" = 6; "Very frequently" = 5; "Occasionally" = 4; "Rarely" = 3; "Very rarely" = 2; "Never" = 1. Cases without a response are indicated in the column "NR".

Question 7. *My motivations for pragmatic source code reuse are: ....* (218 total responses.) To save myself time: 43 (73%). The quality of the originating source code is high: 28 (48%). To increase the robustness of my source code: 9 (15%). To test out new ideas: 22 (37%). To add missing functionality I was previously unaware or unsure of how to implement: 19 (32%). To avoid modifying the originating system: 13 (20%). The design of the originating source code is not amenable to traditional forms of reuse: 14 (24%). To control the feature's future evolution: 1 (2%). I know portions of features I am developing already exist in other software systems: 32 (54%). I have access large amounts of source code available to be reused: 21 (36%). The originating system contains an automated test suite: 3 (5%). The reuse of source code is encouraged in my

[§]Languages reported under "other" were: ABAP (1), ABAP/4 (1), Access (1), APL (1), ASP (2), ASP.NET (1), BASIC (2), COBOL (3), Delphi (1), Erlang (1), Fortran (2), Haskell (1), Matlab (1), Pascal (2), Perl (3), PL/1 (1), RPG (1), Ruby (2), Symbian C++ (1), VB (1), VB/VBA (1), and VB Script (1). In all cases, they also selected at least one of the languages we listed.

Table I. Quantitative summary of responses to subquestions of Question 6.

	6	5	4	3	2	1	NR
I pragmatically reuse code	4	12	27	4	3	1	8
I pragmatically reuse source code functionality contained within a method	3	11	19	11	4	3	8
I pragmatically reuse a whole class	0	4	17	15	9	6	8
I pragmatically reuse a subset of a class	1	9	22	7	7	5	8
I pragmatically reuse a whole source code package	0	7	9	9	17	9	8
I copy and paste source code to be used in another location in the same software product	3	5	18	8	12	5	8
I copy and paste source code to be used in a different software product	2	10	23	8	3	5	8
I have had to modify source code I've pasted in from another location, to fit in its new location	7	27	12	1	2	2	8
I have abandoned an attempt to pragmatically reuse source code because the code had more dependencies on its system than anticipated	0	9	15	12	8	7	8
I have abandoned an attempt to pragmatically reuse source code because integrating the reused source code proved to be more problematic than anticipated	0	10	14	12	11	4	8
I have abandoned an attempt to pragmatically reuse source code because it later became apparent that the result was not appropriate for what I wanted to accomplish	0	6	11	12	15	7	8
I carefully investigate the originating system before attempting the reuse	14	15	13	6	3	0	8
Total	34	125	200	59	94	54	96

organization: 9 (15%). None; I do not pragmatically reuse source code: 0 (0%). Other: 4 (7%). R54 writes, "When the original code solves a hard problem that I need to solve again, but componentizing the code would be significantly more difficult than pragmatically copying it"; this is equivalent to the intent behind our given option "The design of the originating source code is not amenable to traditional forms of reuse" but R54 did not select this option. R33 responds, "Proof of concept," and R17 gives, "Just give it a try and see how 'this' piece of code fits in the new place"; these are both equivalent to the intent behind our given option "To test out new ideas" which R33 selected but R17 did not. R10 responds, "It is a policy to reuse code as much as possible": this is equivalent to the intent behind our given option "The reuse of source code is encouraged in my organization" which R10 selected.

Question 8. *Before attempting an pragmatic reuse task, ...* (74 total responses.) I construct a mental image (model) of the reuse functionality and its dependencies: 32 (54%). I record a detailed plan of the proposed reuse: 5 (9%). I only plan in advance for medium and large pragmatic reuse tasks: 17 (29%). I do not do advance planning before attempting to pragmatically reuse source code: 18 (31%). None; I do not pragmatically reuse source code: 0 (0%). Other: 2 (3%). R46 reports, "I typically only perform pragmatic source-code reuse (PSCR) when under tight time constraints, or the like. This context is generally antithetical to spending a lot of time in the planning phase. In short, if I have time to plan, I have time to write from scratch. PSCR in my experience generally happens

when I need something implemented in a time on the scale of an hour or two. (Once a feature gets sufficiently large/complex that PSCR can't be done in that time frame, I'm more comfortable insisting that more time be devoted to implementing said feature.)" R3 states, "I use pen and paper a lot."

Question 9. *What difficulties have you faced while pragmatically reusing source code?* (120 total responses.) Keeping track of local dependencies to the source code you are reusing: 23 (39%). Keeping track of the external dependencies to the source code you are reusing: 29 (49%). Identifying, reusing, or testing source code associated with the functionality I am trying to reuse: 21 (36%). Current tool support: 11 (19%). Constraints imposed by the target system's logical or conceptual model: 26 (44%). I have not faced any difficulties: 6 (10%). Other: 4 (7%). R33 reports, "Technical debt incurred by the pragmatic reuse." R13 states, "I always did it manually i.e., I do not know whether any kind of tool supports this task and how. I mean I never searched for a tool to assist me." R28 writes, "If it's complex enough to even having to deal with these things for re-use, it sounds like a bad candidate for re-use – I'll often copy and paste a useful method, but rarely something with dependencies." R35 explains, "1. Understanding the complete details of the source code to reuse to know how to modify it. 2. Deciding what exactly needs to be changed for the reused code to fit in my target system, while still preserving the correctness of the reused functionality. 3. Verifying that the reused code works correctly in the new (target) system."

Question 10. *How do you decide to pragmatically reuse code rather than write it from scratch?* (108 total responses.) Having pre-existing knowledge of the desired reuse functionality implementation details: 37 (63%). Construction of a reuse plan that judges the potential success or failure of the reuse scenario: 3 (5%). The ability to easily reason about the source code level details of the pragmatic reuse task: 22 (37%). The estimated time cost to re-implement the desired functionality versus reusing source code.: 43 (73%). None; I do not pragmatically reuse source code: 0 (0%). Other: 3 (5%). R10 tells us, "Programming lead states the functionality already exists in product x and that I should locate and use that code block. I am told to specifically copy paste modify the code." R33 states, "Whether or not the code into which I'm pragmatically reusing will be used in production or not." R7 writes, "When I reuse code, which is rarely, I almost always do it with the intention of refactoring it. Code duplication is avoided in my company and my code would never pass the dev review if I do not ultimately refactor the reused code :-)."

4. ANALYSIS

4.1. Qualitative Analysis of Perceptions

We consider qualitative trends in the perceptions of pragmatic reuse. Below, we summarize the arguments made in the written explanations associated with Question 5; we indicate in parentheses which respondents made statements that are related to the summarized argument.

Sharing experience, avoiding known pitfalls. Several respondents argued for the great value in sharing existing knowledge represented by the copied code, and thereby avoiding the need to "reinvent the wheel" (R2, R3, R15, R31, R48, R25). Several more pointed specifically to the dangers of writing new code instead of reusing old code (R9, R24, R27, R35, R46, R59). Respondent 9 gives us a pithy quote: "Every new line of code I write is a liability; legacy code is stable and tested code."

Saving time outweighs the risks .... Many respondents felt that the potential to save time outweighs the risks involved in any alterations (R4, R6, R21, R24, R27, R47, R19). Respondent 46 tells us: "[...] when there are time etc. constraints, what can be produced by this method is rarely appreciably worse (and often considerably better) than what could be developed to do the same job from scratch. I'd rather spend 15% of my available time borrowing and 85% tweaking to fit the purpose, resulting in a '90%' solution, than 100% of my time generating something purpose-built, resulting in a '70%' solution." Respondent 59 says: "Using well tested software that does well what

it should, can be very useful even though there is more integration effort. Wrapping legacy code is a standard practice. If you want to SOA-enable COBOL code, you are basically reusing legacy code in a way that it wasn't designed to support. [The alternative of] rewriting 15 MLOCs of code is somewhat costly."

... Or the risks outweigh the savings. On the other hand, several respondents felt strongly the opposite. Respondent 39 tells us: "Under this definition of reuse it is seldom that anything other than a trivial sample of code can be reused without a significant adaptation. In my experience, trying to shoehorn code that was never meant to be reused results in a net-zero benefit—the time I spend integrating the new code (and trimming the extras) feels almost the same as coding it from the ground up."

"Modify" is the important part. Several respondents point to the adaptation aspect of pragmatic reuse as being key to its value (R17, R19, R20, R49, R54). Respondent 54 draws the distinction: "Copy/paste is bad and should be avoided, but copy/paste/modify is a very good way of accelerating similar style projects and leveraging existing programming experience that has been captured as code." Respondent 49 writes: "If the intent is appropriation, where the developer recontextualizes the borrowed elements to essentially express some new idea, pragmatic reuse has solid ground."

Design matters, but refactoring is not always feasible. Design was also a concern for many. A standard alternative to pragmatic reuse is refactoring to a common abstraction that can be refined in different contexts. Two respondents argue that these abstractions do not make sense in all situations (R22, R41). Several respondents differentiated their arguments on the basis of the scale of the pragmatic reuse being conducted, claiming that it is acceptable for "small amounts" or snippets of source code (R1, R14, R28, R30, R39, R58). This is consistent with the "reuse of uses" observations of Rosson and Carroll [25]. Respondent 36 states that "generalizing little by little from specific usage is much better than over-designing supposedly generic code." Respondent 22 claims that redundancy is sometimes acceptable, and that pragmatic reuse is usually conducted under stable conditions that makes it a reasonable prospect. Conversely, Respondent 12 argues, "[Pragmatic reuse] creates inconsistent and unstable software and most probably does not fit to the new software needs." Several point out that refactoring would be preferable, when it is feasible, due to a better design (R7, R25, R30, R36, R41). Respondent 53 cautions: "You are replicating code that was not designed for reusability in mind, making different variants of almost the same code, thus, making the problem [lack of reusability?] wors[e]." Respondent 40 is also concerned: "Most of the time that means that the person who blindly copies the code: (a) doesn't know what it does, [and] (b) doesn't care what [it] does."

Maintenance matters. Maintenance was a concern for several. Developers will not always predict what a future developer's needs will be, so pragmatic reuse is inevitable (R17, R31, R55). Respondent 55 humbly states: "I'm not psychic. It would be nice to be able to always know in advance when code will be needed again. I am not so mighty." On the other hand several worry about the classical code clone maintenance problem (R5, R10, R19, R42, R50).

Other points. A few respondents extrapolate from personal experience, claiming that pragmatic reuse works for them and they have seen others use it well so it is a reasonable practice (R1, R24, R37), or conversely that they had problems and so pragmatic reuse should be avoided (R39, R42).

A few unique points were expressed. Respondent 42 cautions: "A common repository of reusable code imposes a lot of administrative and documentation overhead on the developer." Respondent 13, while positive about pragmatic reuse, sees the downside as being its current lack of structured process, echoing Krueger [19]. Respondent 2 expresses concerns with the dangers of taking dependencies on external interfaces in safety critical contexts—the alternative being copy-and-paste of features—echoing an industrial practice reported by Cordy [6].

4.2. Statistical Analysis

Since the data collected via the 59 responses should be viewed as exploratory, the presence or absence of statistically significant associations could mean little for the greater population.

Nevertheless, we statistically analyzed the responses to consider evidence of associations or correlations in the responses. The results should be viewed as support for or against pursuing narrower (and more expensive) hypothesis testing with other methodologies.

We rigorously applied statistical methods appropriate to the kinds of data being analyzed. To determine significance and strength of associations between a pair of ordinal variables, we utilized Spearman's rank-order correlation. To determine the significance of associations between a pair of nominal variables, we used Fisher's exact test as our sample sizes are small. To determine the strength of an association between nominal variables, we used Cramér's V which yields a value in the interval $[0, +1]$ and hence provides no information about directionality; when the directionality was of interest, we resorted to visual inspection of the contingency table. At times we were also interested in determining whether there was a difference in ordinal data collected between independent groups (e.g., those with positive versus negative perception of pragmatic reuse): we used the Kruskal-Wallis one-way analysis of variance by ranks (which reduces to the Mann-Whitney-Wilcoxon test without a continuity correction for only two groups); Kruskal-Wallis assumes that the two populations have the same shape and scaling. When a test is needed to compare variables of differing measurement levels, the test for the weakest of the levels must be used; for example, to determine the presence/absence of an association between a nominal and ordinal variable, Fisher's exact test is appropriate while Spearman's rank-order correlation is not.

We constructed a script in the R statistical language to conduct the appropriate tests for associations between pairs of variables, as described above. We utilized built-in functions for Spearman's rank-order correlation (cor.test(method="spearman", alternative="two.sided", ...)), Fisher's exact test (fisher.test(simulate.p.value=TRUE, ...)), and the Kruskal-Wallis one-way analysis of variance by ranks (kruskal.test(...)); we added the vcd package for computation of Cramér's V (assocstats(...)). A total of 3,371 tests were performed.

We performed the full analysis twice: once with the raw data and once by replacing the binary responses to Question 5 with a synthetic variable taking on a four-valued ordinal scale ("strongly agree", "agree", "disagree", "strongly disagree") based on an interpretation of the respondents' qualitative remarks; when these remarks were absent, the binary response was interpreted as "strongly agree" or "strongly disagree" as appropriate. Few significant (and interesting) associations resulted in either analysis, and when they did, they were generally weak to very weak. To save the reader from a long and tedious exposition, we point to the most interesting results in Section 5.

5. REVISITING THE RESEARCH QUESTIONS

RQ1: Do industrial software developers perceive pragmatic reuse as a disciplined practice that is relevant to their development activities? Two-thirds of our respondents rejected the statement "Pragmatic reuse is a bad practice and should be avoided". Three-quarters of respondents that provided some sort of explanation as to their perception, rejected the statement. A small majority (55%) of all respondents providing explanations take a balanced view, either believing that pragmatic reuse is an important approach but that there are situations where it is not appropriate, or conversely believing that pragmatic reuse is an overall dangerous approach but that there are situations where it is reasonable or unavoidable.

The specific explanations for perceptions largely conformed to issues expressed in the literature. Issues *in favour* of pragmatic reuse were: pragmatic reuse saves time; pragmatic reuse leverages past experience effectively; pragmatic reuse of mature code leads to safer code than reimplementation; refactoring is not always a viable alternative; predictive approaches to reuse can demand too much of an investment; pragmatic reuse is a good first step prior to refactoring the target system; and the redundancy resulting from pragmatic reuse within a system can sometimes be reasonable. Issues *against* pragmatic reuse were: bad design can result; code clones can be a maintenance problem; the current industrial practice of pragmatic reuse is too unstructured; the time it takes to integrate the copied code eats up the savings from not reimplementing it; and it promotes ignorant surgery.

No statistically-significant, demographic basis for the respondents' opinions was found, with the exception of a weak relationship between the use of C# and perception (those using C# have a

slightly greater chance of a positive than a negative perception of pragmatic reuse; those not using C# are much more likely to have a positive than a negative perception). Most importantly, we see no statistically significant indication that those with more experience or currently working in industry are more likely to view pragmatic reuse negatively. While there was a vague hint of such a relationship between perception and experience (a weak correlation with $p > 0.05$), this disappears entirely when we repeated the analysis in quantitative terms of the written explanations.

Interestingly, amongst those with a negative perception of pragmatic reuse, the majority report occasional or more frequent performance of pragmatic reuse. Even those very few who claim to never perform pragmatic reuse provide motivations, planning actions, and deciding factors for it—suggesting that "never" does not match the reality.

RQ2: How frequently and at what scale do industrial software developers perform pragmatic reuse tasks as a part of their development activities? As is to be expected, as the scale of pragmatic reuse increases from individual methods to entire packages, the reported frequency of performing such tasks drops. This downwards trend appears less pronounced for those with a positive perception of pragmatic reuse. We were surprised that package-level pragmatic reuse would have much frequency at all, since we would expect greater modularity to lead to more standard reuse approaches. Perhaps this is a sign of the "clone-and-own" approach to product families, mentioned in Section 2.

With more industrial experience, one is slightly less likely to perform pragmatic reuse of individual methods and somewhat less likely to perform copy-and-paste within the same product.

Not surprisingly, those with positive perceptions of pragmatic reuse have a moderate tendency to report performing it more frequently than those with negative perceptions, both overall and at the level of single methods, and they have a moderate tendency to report modifying the pasted code more frequently. No significant relationship exists between perception and frequency of copy-and-paste either within or between products.

RQ3: How frequently do industrial software developers abandon pragmatic reuse tasks because of unforeseen problems? With more experience, once having started a pragmatic reuse task, one is slightly less likely to abandon a task because of unexpected dependencies and somewhat less likely to abandon a task because of unexpected integration issues.

As the frequency of abandonment increases, it is somewhat more likely that the developer is performing pragmatic reuse because they need help with their task. It seems probable that, in a situation where the developer is uncertain of how to proceed with a development task, they are less likely to be able to judge *a priori* whether pursuing it via pragmatic reuse on a specific artifact is ill-advised or not; thus, they are more likely to get into trouble that forces them to abandon the task.

The fact that those reporting less frequent abandonment due to poor results are slightly more likely to decide in favour of pragmatic reuse when they know that the functionality already exists, could be interpreted as a tendency to stick to the task even when it becomes difficult or ill-advised. This conforms to anecdotes reported by Holmes and Walker [14].

RQ4: What motivates a developer to choose to perform a pragmatic reuse task? Every one of the motivations that we provided was selected by at least one respondent. In two cases, the low level of response surprised us, as these motivations were argued convincingly in the literature as being important: to control the feature's future evolution [6] and the presence of an automated test suite [12]. In the first case, this may be a motivating factor for an under-represented segment of the population. In the second case, the reason is unclear, since many respondents (28) indicated that high quality of the originating code is important. Apparently, either these respondents do not consider automated test suites to be important, or the systems that they have at their disposal for pragmatic reuse simply do not have automated test suites. We favour the latter explanation, as our experience suggests that automated testing is less prevalent in industry than it should be.

Those who report more frequent pragmatic reuse overall are somewhat more likely to indicate organizational support as a motivating factor. At the level of single methods, those who report more frequent pragmatic reuse are somewhat more likely to indicate exploration/experimentation as a motivating factor. At the level of classes, those who report more frequent pragmatic reuse are somewhat more likely to indicate as motivating factors both a desire to save time and the knowledge

that the functionality already exists elsewhere. At the package level, those who report more frequent pragmatic reuse are somewhat more likely to indicate the high quality of the originating source code as a motivating factor; this is also a somewhat more likely motivating factor for those who report more frequent copy-and-paste between products.

There is a strong, increasing tendency to avoid modification to an original system as industrial experience increases, which is well in keeping with our personal experience and anecdotal evidence.

RQ5: What preparations do developers take before performing pragmatic reuse tasks? A large majority (82%) of respondents claim to perform careful investigation of the originating system before attempting pragmatic reuse, at least occasionally; a small majority (57%) claim to perform it very frequently or always. All four of the planning activities (or lack thereof) that we suggested to respondents were chosen by some of them. Only 11 respondents (19%) chose only "I do not do advance planning." As no respondents claim to never undertake careful investigation prior to pragmatic reuse, these 11 respondents clearly interpreted investigation and planning as distinct.

Evidence in support of the hypothesis that those pursuing pragmatic reuse are more incautious than those that avoid it is weak or non-existent. While a significant negative correlation exists between frequency of pragmatic reuse overall and frequency of careful investigation prior to conducting pragmatic reuse, we emphasize that the correlation is weak (Spearman's $\rho = -0.292$) and visual inspection calls into question the meaningfulness of the result. In addition, frequency of class-level pragmatic reuse is significantly associated with planning at this level, and high quality is reported as a concern amongst those who report very frequent pragmatic reuse at the package level and between products, with a significant association.

At low levels of industrial experience, there is a strong tendency to construct mental models but as experience increases, this tendency evens out. Examining the contingency tables for other planning options, we see that at higher levels of industrial experience, the tendency to apply other approaches increases but without an obvious pattern. Our interpretation is that more experienced developers recognize that mental models are not sufficient for pragmatic reuse in many cases.

Respondents who want to avoid modifying the original system are slightly less likely to construct a mental model before conducting pragmatic reuse. This seems paradoxical, since avoiding modification to the original system is a sign of caution, while not bothering with a mental model could be taken as a lack of caution. However, more experienced developers tend to replace mental models with alternative planning approaches. Furthermore, analysis of respondents' justifications for their perception of pragmatic reuse points to the fact that several see pragmatic reuse as being worthwhile only where such preplanning is not necessary; this is clearly not a standard sentiment.

RQ6: What difficulties do developers face while performing pragmatic reuse tasks? Every one of the 6 suggested difficulties (including a lack thereof) was selected by respondents. Only 6 respondents claimed to never have difficulties during pragmatic reuse; these did not also select other options. 5 of these 6 appear to be rather frequent performers of pragmatic reuse displaying medium-to-high levels of industrial experience (5–24 years) and occasional or more frequent need to modify the pasted code, so it is surprising that they never encounter troubles.

Beyond this curious minority, there is evidence that those pursuing pragmatic reuse do indeed encounter troubles, but particularly when they are motivated by being lost in their current task. We have postulated that this can be explained by their inability to judge the task well before engaging in it, since they are, after all, uncertain about how to proceed.

Those who are currently employed primarily in software development are much more likely to *not* have difficulties in coping with local dependencies, while other employment categories are somewhat *more* likely to have such difficulties. Perhaps those with greater day-to-day exposure have coping mechanisms that others forget.

The fact that those reporting more frequent pragmatic reuse at the single method level are somewhat more likely to report difficulties with external dependencies might be explainable simply as follows. Imagine that any given pragmatic reuse task has a fixed probability of encountering difficulties with external dependencies; the probability that a given developer encounters such difficulties at least once then increases as the number of pragmatic reuse tasks performed increases.

The lack of relationships between the absence of investigation/planning and encountering difficulties is surprising. We speculate that developers who do not bother with investigation/planning deal with simple, small-scale cases that do not require it and where problems cannot possibly arise.

RQ7: What are the factors in choosing pragmatic reuse over re-implementing the desired functionality anew? All four factors we suggested were selected by respondents, though only 3 chose the existence of a detailed plan—not surprisingly since no industrial tool supports such plans.

Respondents reporting robustness as a motivation for pragmatic reuse are much more likely to also report easy reasoning about the source to be an important deciding factor in pursuing pragmatic reuse. This is a reassuring sign, since making one's own code more robust on the basis of someone else's certainly would require the ability to reason about that other code. But beyond that, it does not seem to point towards a general observation.

Those reporting difficulties arising from external dependencies or unexpected constraints in the target system are somewhat more likely to report easy reasoning about the source as a deciding factor in the pursuit of pragmatic reuse. This could be an indication either that their perception of easy reasoning can turn out to be false, or that these are separate cases of pragmatic reuse. No respondents report both a lack of difficulties and easy reasoning about the source being a deciding factor. This suggests that respondents can be misled as to the true ease of reasoning, and better tool support (e.g., [14]) could help here.

6. DISCUSSION

6.1. Methodological Matters

Our results must not be taken as definitive, as the sample is apparently small and clearly self-selected. Lethbridge et al. [22] point out that response rates to software engineering questionnaires are often very low, as compared to other fields, so arriving at larger samples would be problematic. Compared to other studies in this area (with 1, 4, 6, 12 participants), 59 respondents is a large sample. Whether their opinions are right or wrong, their opinions inform us about industrial experience and industrial problems for which, it would seem, pragmatic reuse is often regarded as an appropriate option within a range of alternatives.

One may think that our use of SurveyMonkey with its limit of 10 questions is problematic. However, simply switching to another alternative without such limits is unlikely to lead to better results, in whatever sense. More questions would *reduce* the likelihood of receiving a response, since they would demand more time from a potential respondent, and developers are notoriously frugal in donating their time [22].

A typical methodology [e.g., 8] to performing a survey would narrow the scope drastically, asking multiple questions as cross-checks, and would target specific individuals/organizations, in order to arrive at precise and statistically valid results on very specific questions. These questions would arise from a meta-analysis of multiple primary studies already existing in the literature. However, in our context, such primary studies are largely absent from the literature and those that do exist provide only vague hints on small topics. Thus, we feel that such a methodology would be premature and thus unreasonable to demand. Given the results we present in this paper, narrower research questions can now be formulated and pursued more deeply; this iterative refinement approach to investigating an untouched topic has been promoted previously [24].

6.2. Threats to Validity

In general, we could expect that individuals with strong opinions about pragmatic reuse (whether positive or negative) would more likely have donated their time and energy to the survey. Statistical analysis indicates that the respondents lack significant associations in their demographic data, suggesting their independence. We have no way to ensure that respondents' answers conform to their real beliefs or that those beliefs conform to an objective reality. Their responses may have been intentionally deceptive, but there are two factors that should have reduced this possibility:

(i) respondents were given no rewards, regardless of their answers; and (ii) the survey was anonymous, so there was no danger that any unpopular opinion could be attributed to them, nor that any popular opinion could be attributed to them. Purely vandalistic responses would have resulted in no particular gratification, since they would not have been able to see any havoc wrought.

Their interpretation of the questions may have differed from ours, and hence our interpretation of their answers may have differed from theirs. We provided a definition of pragmatic reuse to reduce this communication barrier, but we did not attempt to provide definitions of all terms and quantities about which we asked. As a positive indication that such effects were not large, we found only one case where the respondent's acceptance of the statement of Question 5 contradicts their purely positive explanation of the value of pragmatic reuse. There were only two cases where respondents added textual responses (under "Other") that coincided with existing options but where they had not also chosen those options.

Although we have used it as a means of characterizing expertise, years of industrial experience is a potentially problematic variable. Each year is clearly not equivalent, as evidenced by any learning curve, and the expertise gained from a set of years is likely not equivalent for different individuals. On the other hand, there is the fact that everyone with the same amount of experience is likely to have been trained at about the same time, seen the same set of trends grow and fade, etc. and thus would be inclined towards the same biases. As we have no means for controlling for such effects in this kind of study, we merely point out this fact for other researchers. A visual inspection of the data shows that even developers at high levels of experience display a random inclination towards positive or negative perception of pragmatic reuse.

6.3. No Correlation between Employment Status and Perception

Our interest was in responses from participants with industrial development experience regardless of their current employment status. The industrial development experience of respondents who marked themselves as students ranged from 1 to 15 years with a mean of 5 years, a median of 3 years, and a standard deviation of 4.33 years. Mapping employment status to an ordinal scale 0–5, where 0 signifies "Employed primarily in software Development" and 5 signifies "Retired" and then applying a two-tailed Spearman's rank correlation results in a coefficient value of -0.081 and a p-value of 0.541; therefore there is no correlation found between employment status and perception of pragmatic reuse.

6.4. Reducing the Risk of Pragmatic Reuse

We found several indications that the myth of only careless hackers embarking on pragmatic reuse is a false one: abandonment is not correlated with planning, years of experience is not correlated with perception. Nevertheless, there was widespread recognition of the risks involved.

One semi-automated approach for small-scale reuse exists called Jigsaw [7] that is designed to remove all the tedious aspects of small-scale reuse tasks to allow developers to focus on the high-level details (e.g., inheritance hierarchies, control flow idioms). place.

For medium to large scale tasks, Holmes et al. [12–15] present an approach that helps developers manage dependencies by accepting and rejecting functionality of the desired reuse source, through a tool called Gilligan. Though Gilligan attempts to minimize developer effort by systematizing the process, developers still face investigative costs before being able to decide if pragmatic reuse is the best way forward. Both these tools are unable to help developers with the reuse of associated test code and testing of the reused functionality within its new target location.

7. CONCLUSION

We conducted a survey of the perceptions, frequency, motivations, difficulties, and execution of pragmatic reuse tasks from the perspective of individuals with industrial software development experience. We received responses from 59 people with an average of 8.3 years of industrial development experience, from a wide range of company sizes, and using a wide variety of

programming languages. We performed a qualitative analysis of respondents' explanations for their perceptions of pragmatic reuse, and a statistical analysis of their other responses.

Two-thirds of respondents had an overall positive perception of pragmatic reuse, and even amongst those reporting a negative perception, the majority report at least occasional acts of pragmatic reuse. No statistically-significant relationships were found between perception and any other demographic factor. In particular, this contradicts the notion that pragmatic reuse is a "bad habit" of junior developers while senior developers "know that it is a mistake." An overall picture of careful management of risks emerges, though some individuals display greater risk tolerance or aversion. A large majority (82%) claim to perform careful investigation of the originating system before attempting pragmatic reuse, at least occasionally. The strongest result was the desire to avoid modification to the originating system, getting even stronger with greater experience. In general, the respondents are aware of the strengths and weaknesses of pragmatic reuse, and approach any non-trivial instance of its application with care. Most respondents see pragmatic reuse as valuable at small scales, for prototyping and learning purposes, and where alternative approaches are not feasible. Some see it as valuable at larger scales too, despite their recognition of risks. More senior developers express somewhat greater consciousness that mental models of pragmatic reuse tasks are often insufficient, moving to more robust mechanisms. Dependency management and ease of reasoning about the source appear to be the most common factors that need tool support.

The notion that proponents of pragmatic reuse are ill-informed, inexperienced, or foolhardy is not borne out. Pragmatic reuse is an industrially valuable practice that carries risks and difficulties known to developers, but clearly represents a current best practice in some circumstances.

ACKNOWLEDGMENTS

We thank Brad Cossette, Reid Holmes, and Jonathan Sillito for their feedback on this manuscript and the survey design. This work was supported by a Postgraduate Scholarship and a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] B. H. Barnes and T. B. Bollinger. Making reuse cost-effective. *IEEE Softw.*, 8(1):13–24, Jan. 1991. doi: 10.1109/52.62928.

[2] A. Begel, N. Nagappan, C. Poile, and L. Layman. Coordination in large-scale software teams. In *Proc. Int. Wkshp. Coop. Human Aspects Softw. Eng.*, pages 1–7, 2009. doi: 10.1109/CHASE.2009.5071401.

[3] T. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Softw.*, 4(2):41–49, Mar. 1987. doi: 10.1109/MS.1987.230095.

[4] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer. Two studies of opportunistic programming: Interleaving web foraging, learning, and writing code. In *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, pages 1589–1598, 2009. doi: 10.1145/1518701.1518944.

[5] D. Cai and M. Kim. An empirical study of long-lived code clones. In *Proc. Int. Conf. Fundamental Approaches Softw. Eng.*, volume 6603 of *Lect. Notes Comp. Sci.*, pages 432–446, 2011. doi: 10.1007/978-3-642-19811-3_30.

[6] J. R. Cordy. Comprehending reality: Practical barriers to industrial adoption of software maintenance automation. In *Proc. IEEE Int. Wkshp. Progr. Comprehen.*, pages 196–205, 2003. doi: 10.1109/WPC.2003.1199203.

[7] R. Cottrell, R. J. Walker, and J. Denzinger. Semi-automating small-scale source code reuse via structural correspondence. In *Proc. ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, pages 214–225, 2008. doi: 10.1145/1453101.1453130.

[8] T. Dybå. An empirical investigation of the key factors for success in software process improvement. *IEEE Trans. Softw. Eng.*, 31(5):410–424, 2005.

[9] W. B. Frakes and K. Kang. Software reuse research: Status and future. *IEEE Trans. Softw. Eng.*, 31(7):529–536, July 2005. doi: 10.1109/TSE.2005.85.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, Nov. 1995. doi: 10.1109/52.469757.

[11] S. Haefliger, G. von Krogh, and S. Spaeth. Code reuse in open source software. *Manage. Sci.*, 54(1):180–193, Jan. 2008. doi: 10.1287/mnsc.1070.0748.

[12] R. Holmes and R. J. Walker. Supporting the investigation and planning of pragmatic reuse tasks. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pages 447–457, 2007. doi: 10.1109/ICSE.2007.83.

[13] R. Holmes and R. J. Walker. Lightweight, semi-automated enactment of pragmatic-reuse plans. In *Proc. Int. Conf. Softw. Reuse*, pages 330–342, 2008. doi: 10.1007/978-3-540-68073-4_35.

[14] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, Nov. 2012. doi: 10.1145/2377656.2377657.

[15] R. Holmes, T. Ratchford, M. Robillard, and R. J. Walker. Automatically recommending triage decisions for pragmatic reuse tasks. In *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 397–408, 2009. doi: 10.1109/ASE.2009.65.

[16] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999. Section 2.7: The Evils of Duplication.

[17] S. Jansen, S. Brinkkemper, I. Hunink, and C. Demir. Pragmatic and opportunistic reuse in innovative start-up companies. *IEEE Softw.*, 25(6):42–49, Nov./Dec. 2008. doi: 10.1109/MS.2008.155.

[18] C. Kapser and M. W. Godfrey. 'Cloning considered harmful' considered harmful: Patterns of cloning in software. *Empir. Softw. Eng.*, 13(6):645–692, Dec. 2008. doi: 10.1007/s10664-008-9076-6.

[19] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992. doi: 10.1145/130844.130856.

[20] C. W. Krueger. Towards a taxonomy for software product lines. In *Revised Papers Int. Wkshp. Softw. Product-Family Eng.*, volume 3014 of *Lect. Notes Comp. Sci.*, pages 323–331, 2004. doi: 10.1007/978-3-540-24667-1_25.

[21] B. M. Lange and T. G. Moher. Some strategies of reuse in an object-oriented programming environment. In *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, pages 69–73, 1989. doi: 10.1145/67449.67465.

[22] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empir. Softw. Eng.*, 10(3):311–341, July 2005. doi: 10.1007/s10664-005-1290-x.

[23] S. Makady and R. J. Walker. Validating pragmatic reuse tasks by leveraging existing test suites. *Softw. Pract. Exper.*, 43(9):1039–1070, Sept. 2013. doi: 10.1002/spe.2134.

[24] G. C. Murphy, R. J. Walker, and E. L. A. Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Trans. Softw. Eng.*, 25(4):438–455, July/Aug. 1999. doi: 10.1109/32.799936. Special Section on Empirical Software Engineering.

[25] M. B. Rosson and J. M. Carroll. The reuse of uses in Smalltalk programming. *ACM Trans. Comput.-Hum. Interact.*, 3(3):219–253, Sept. 1996. doi: 10.1145/234526.234530.

[26] R. W. Selby. Enabling reuse-based software development of large-scale systems. *IEEE Trans. Softw. Eng.*, 31(6):495–510, June 2005. doi: 10.1109/TSE.2005.69.

[27] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: Concepts and guidelines. *Softw. Pract. Exper.*, 31(3):277–300, Mar. 2001. doi: 10.1002/spe.366.

[28] R. van Ommering. Software reuse in product populations. *IEEE Trans. Softw. Eng.*, 31(7): 537–550, July 2005. doi: 10.1109/TSE.2005.84.