



Understanding reuse of software examples: A case study of prejudice in a community of practice



Ohad Barzilay^{a,*}, Cathy Urquhart^b

^a Tel Aviv University, Israel

^b Manchester Metropolitan University, Manchester, UK

ARTICLE INFO

Article history:

Received 7 April 2013

Received in revised form 25 February 2014

Accepted 26 February 2014

Available online 27 March 2014

Keywords:

LinkedIn

Qualitative research

Grounded theory

Virtual focus group

Human aspects

Open source development

ABSTRACT

Context: The context of this research is software developers' perceptions about the use of code examples in professional software development.

Objective: The primary objective of this paper is to identify the human factors that dominate example usage among professional software developers, and to provide a theory that explains these factors.

Method: To achieve this goal, we analyzed the perceptions of professional software developers as manifested on LinkedIn online community. We analyzed the data qualitatively using adapted grounded theory research procedures.

Results: The research yields an initial framework of key factors that dominate professional developers' perception regarding example usage. We use the theoretical lens of *prejudice theory* to put these factors in a broader context, and outline initial recommendations to address these factors in professional organizational context.

Conclusion: The results of this work, in particular the use of qualitative techniques – allowed us to obtain rich insight into key human factors that affect professional software developers, and enrich the body of literature on the issues of reuse. These factors need to be taken into account as part of an organizational reuse strategy.

© 2014 Published by Elsevier B.V.

1. Introduction

This paper examines an important issue for software developers – that of software reuse, and how this might be perceived in relation to code duplication. In the software engineering community, *code duplication* has been widely acknowledged as a bad practice [57]. When copying and pasting code snippets found elsewhere in the code base, the developer misses opportunities to abstract the functionality in question, and in turn, reduces code quality. In addition, redundant code often results in consistency issues, which also affect the quality of the overall system [50].

Code reuse, on the other hand, is widely accepted as a good practice. The literature advocates the reuse of existing code, and mentions several benefits of it including: increased productivity [89], improvement of code quality [68] enforcement of design

consistency [31,77] and of coding standards [17], and the establishment of an effective knowledge transfer mechanism both within and outside the organization [92].

In some cases, however, code reuse may be perceived by software developers as code duplication. Given the recent massive availability of code online, in open source projects, technical blogs, and Q&A websites [10,97], this issue is worthy of investigation. On the one hand, incorporating this online code in production may be considered as a modern manifestation of code reuse – already written code that can spare the software developer the effort to “re-invent the wheel”. On the other hand, using this code involves ‘copying and pasting’ it, an action that on itself, serves for many developers as a warning sign, an indicator, of the banned code duplication activity.

In this paper, we focus on developers' perception of (re)using code examples – existing code snippets that are used in a new context. Our definition of a code ‘example’ is broad; some of the code examples which appear on the Internet were not written in order to be reused. Code examples may accompany answers on Q&A sites [12], illustrate and idea in an online tutorial, or even be extracted from an open source project [98].

* Corresponding author. Address: Tel Aviv University, Recanati Building, Ramat Aviv, Tel Aviv 69978, Israel. Tel.: +972 3 6406292; fax: +972 3 6409983.

E-mail addresses: ohadbr@tau.ac.il (O. Barzilay), c.urquhart@mmu.ac.uk (C. Urquhart).

Research questions:

- How do software developers perceive example usage in their work, and in particular a usage that involves copy and paste? What arguments do they use to justify their position? Which concerns do they consider?
- How can these perceptions be explained?

In this study, a community of software developers on LinkedIn was asked to describe its practices when using code examples. Developers' responses revealed a variety of perspectives on using code examples. We outline those different perspectives by dividing these developers into three groups according to their example usage characteristics: those who use examples habitually, those who avoid using examples, and those who make limited use of examples. Further analysis of the findings using grounded theory [102] methods revealed nine human factors that dominate developers' perceptions. It is suggested that developers' approach is dominated by their personality, and affected by concerns such as their community identity, ownership and trust. We find that developers' perception of such reuse goes beyond the activities and practices, and that some developers associate the use of code examples with negative character. Some of these developers stereotype habitual example users as inferior and unprofessional.

In this paper we use the *prejudice perspective* [86,23] in order to examine how certain software developers are 'prejudiced' against the use of code examples, and consequently against developers who do not share their views on this matter. The term prejudice is of course somewhat loaded; in the social sciences it is commonly used in the context of stereotyping, and discrimination against certain groups, which results in racism, nationalism and sexism. While we would not claim that in the context of software development the implications of prejudice against use of examples are as severe, we do find this perspective supremely useful to examine the adoption and avoidance of using certain professional practices, for reasons which will be explained in more depth later.

It should be noted that not only human aspects are associated with example usage – there are some other issues involved in this activity such as engineering aspects (e.g. search techniques and tools) and legal issues (e.g. copyright and licensing). These issues are outside of the scope of our discussion; however, we believe that these challenges can be mitigated with proper training and organizational support (e.g. teaching developers which code could they use, and on what circumstances). This approach is also supported by Rotenberger et al. [91] and Morisio et al. [75].

This work is built upon previous work of Barzilay et al. [11], using social media to study the diversity of developers' perception regarding example usage. This paper seeks to make the following contributions. First, by highlighting the concerns of software developers with respect to example usage, we identify additional reuse barriers and augment the existing body of knowledge in software reuse. Second, we demonstrate how human concerns dominate developers' behavior – in this respect, our case study of example usage can be used as a test bed to examine a larger array of activities and their associated human concerns. In addition, we extend prejudice theory to the domain of software engineering, and show that developers' perceptions regarding a professional activity extend beyond the activity itself, and that they attribute characteristics to other professionals that do not share their views. Our work also demonstrates how the views of an online community can be effectively surveyed and analyzed, and we share our methodology for doing so.

In the next section, we present our chosen theoretical lens of prejudice theory, and discuss relevant literature on software reuse. We then present our methodology for the study, which was carried out using a virtual focus group in a social networking community. In the fourth section, we explain our data analysis, and in the fifth

section present our findings using the lens of prejudice theory. Next, we integrate our findings regarding the human aspects involved with example usage with the literature. Finally, we discuss the limitations of this study, and draw some conclusions and implications about the reuse of software examples in software development communities.

2. Related work

In this section, we first discuss theories of prejudice and why they may be helpful for our research problem. We then discuss literature on software reuse.

2.1. Theories of prejudice

Prejudice literature is used in social sciences to study racism, sexism and discrimination. Why then might it be relevant to software example reuse? We noticed that the developers in our study had strong and entrenched opinions about reuse, and their beliefs are extended toward other software practitioners who hold counter opinions. Software construction is a human-intensive activity [108], as such, many aspects of it may be dealt from a human perspective. Using prejudice theory [86,6,23] we explain opposition to the use of code examples as *preconceived opinions originated in a narrow implicit context, in which they first encountered*. Chauvinism, for example, may result from growing in a culture in which women work only in certain jobs. Racism can build when a certain group is encountered only in a specific negative context. Even when behaviors are undeniably caused by situational factors, people will sometimes favor dispositional explanations – a misjudgment known as the “fundamental attribution error” [90]. The outline of this overview is adapted from Plous [86].

The majority of social scientists agree that “prejudice” involves a prejudgment, usually negative, about a group or its members [29,49,79], even if their precise definitions vary somewhat. We should also stress here that prejudice is not merely a statement of opinion or belief, but an attitude that can include feelings such as contempt, dislike, or loathing [29]. Below we outline some key dimensions of prejudice theory.

2.1.1. Authoritarian personality

Adorno et al. [3] concluded that the key to prejudice lay in what they call an “authoritarian personality.” They described authoritarians as rigid thinkers who obeyed authority, saw the world as black and white, and enforced strict adherence to social rules and hierarchies. Authoritarians harbor many double standards and hypocrisies, without realizing it [7]. Adult authoritarians travel in tight circles of like-minded people; they often think their views are commonly held in society, that they are the “Moral Majority” or the “Silent Majority” [7].

Authoritarian people are more likely than others to harbor prejudices against low-status groups [3,86]. Furthermore, Social Dominance Theory states [96] that society can be viewed as “*group-based hierarchies. In competition for scarce resources such as housing or employment, dominant groups create prejudiced “legitimizing myths” to provide moral and intellectual justification for their dominant position over other groups and validate their claim over the limited resources*” [96].

2.1.2. Categorical thinking

Allport [6] suggests that prejudice is partly an outgrowth of normal human functioning, and relates it to categorical thinking:

The human mind must think with the aid of categories...Once formed, categories are the basis for normal prejudgment. We cannot possibly avoid this process. Orderly living depends upon it.

[[6] p. 20]

2.1.2.1. Stereotypes. The term ‘stereotype’ is also integral to our understanding of prejudice, as it is often on the basis of stereotypes that prejudices are formed. The term was first coined in 1798 by the French printer Didot, and originally referred to the process of creating reproductions [8]. Lippmann [71] likened stereotypes to “pictures in the head,” or mental reproductions of reality. Over time the term has gradually come to mean generalizations, or over-generalizations, about members of a group. These generalizations can at times be positive (e.g., women are nurturing, Japanese excel at math), but for the most part, they tend to be negative and resistant to change.

Stereotypes are learned not only from the mass media, but from direct experience as well. Although some stereotypes are grounded in truth (e.g., it is true that men are, on average, more physically aggressive than women), many are distortions that arise from otherwise adaptive modes of thought. Once stereotypes are learned they sometimes take on a life of their own and become “self-perpetuating stereotypes” [99].

2.1.2.2. Outgroup homogeneity. In social psychology, an “ingroup” is defined as a group to which someone belongs, and an “outgroup” is a group to which the person does not belong [86] (One person’s ingroup may be another person’s outgroup, and vice versa). We tend to see ‘outgroup’ members as much less variable than ourselves – for instance, people might say of politicians, ‘they are all the same’. Research on ‘the outgroup homogeneity effect’ suggests that, when it comes to attitudes, values, personality traits, and other characteristics, people tend to see outgroup members as more alike than ingroup members [76]. This ‘sameness’ of the outgroup can have some negative consequences. For instance, outgroup members are at risk of being seen as interchangeable or expendable, and are more likely to be stereotyped [76]. This perception of sameness holds true, regardless of whether the outgroup is another race, religion, nationality, college major, or other naturally occurring group [69]. So, the implication for the exercise of prejudice is that differences within groups will tend to be minimized, and differences between groups will tend to be exaggerated. If these differences are also consistent with well-known stereotypes, the distortion in perception may be highly resistant to change [69].

A number of factors might produce the outgroup homogeneity effect [42]. Lack of exposure to the outgroup is a key factor [42,70]. People are also more motivated to make individual distinctions about ingroup members with whom they will have future contact [69], and tend to organize and recall information about ingroups in terms of persons, rather than abstract characteristics [83,84].

2.1.3. In group bias

Brewer [23] defines ingroups as “*bounded communities of mutual trust and obligation that delimit mutual interdependence and cooperation*”. An important aspect of this mutual trust is that it is depersonalized, extended to any member of the ingroup whether personally related or not [23].

It has been suggested that biases such as racism, contrary to expectation, are less a product of negative feelings toward another group, and more about loyalty and favoritism towards one’s ingroup.

As Brewer [23] (p. 438) suggests, “Ultimately, many forms of discrimination and bias may develop not because outgroups are hated, but because positive emotions such as admiration, sympathy, and trust are reserved for the ingroup.” The tendency of people to favor their own group, known as “ingroup bias,” has been found in cultures around the world [2,23,21].

Ingroup bias is easily triggered [105,106]. For instance, one study found that people are more likely to cooperate with another person when they learn that the person shares their birthday [73].

Tajfel [106] hypothesized that ingroup biases arise from similar dynamics concerning the need for self-esteem. By their Social Identity Theory, people maintain their self-esteem in part by identifying with groups and believing that the groups they belong to are better than other groups [107].

The optimal distinctiveness model of social identity [22] holds that “group identification is a product of opposing needs to inclusion (assimilation) and differentiation from others. One implication of the theory is that ingroup loyalty, and its concomitant depersonalized trust and cooperation, is most effectively engaged by distinctive groups or social categories” [23].

Brewer [23] also shows that ingroup bias can be used as a platform for outgroup hate. Maintaining ingroup integrity may lead to expressions of moral superiority, perceived threat by outgroup members, common goals, common values and social comparison.

2.1.4. Causal attributions

Causal attributions – in short, the way ingroup and outgroup members explain each other’s behavior – can be both a symptom and source of prejudice [86]. For example, if a single mother’s homelessness is attributed to dispositional factors such as personal laziness, poor character, or lack of ability, this could contribute to prejudice about single mothers. If however, her homelessness is attributed to situational factors such as job layoffs or domestic partner violence, people may feel less prejudice toward single mothers or these prejudices may not come into play [86]. People often make uncharitable attributions for the behavior of outgroup members. At the time of writing, it is claimed that the UK government are systematically ‘demonising’ the poor by deliberately caricaturing those on welfare benefits as ‘scroungers’ [38]. From the literature, we can see that those uncharitable attributions can happen three ways:

- **Just-World Attributions in an Unjust World.** In many situations, causal attributions implicitly follow a “just world” ideology that assumes people get what they deserve and deserve what they get [66,74].
- **The Fundamental Attribution Error.** In addition to just-world beliefs, people have a more general tendency to attribute behavior to dispositional causes, i.e. a person’s character. Even when behaviors are undeniably caused by situational factors, people will sometimes favor dispositional explanations – a misjudgement known as the “fundamental attribution error” [90].
- **The Ultimate Attribution Error.** Pettigrew suggests that ingroup members attribute negative outgroup behavior to dispositional factors, and attribute positive outgroup behavior to luck or situational factors [85].

2.2. Software reuse

Much of the literature about software reuse [60,44], deals with (re)using components that were written in order to be reused. The reuse of code examples found on the Internet expands on this notion, by considering forms of reuse that can be seen to be more pragmatic and opportunistic. This section reviews literature of covering both types of reuse.

Lim [68] reports on the metrics collected in two reuse programs at Hewlett–Packard that demonstrate improved quality, increased productivity, and reduced time to market. The results of economic cost-benefit analyses indicate that reuse can provide a substantial return on investment. Morisio et al. [75] identify some of the key factors in adopting or running a company-wide software reuse program. Three main causes of failure were: (a) not introducing reuse-specific processes, (b) not modifying non-reuse processes, and (c) not considering human factors [75]. Conversely, successes were achieved when, given the potential for reuse due to commonality

among applications, management committed to introducing reuse processes, modifying non-reuse processes, and addressing human factors [75].

Slyngstad et al. [100] describe the results of a study on developers' views on software reuse. Their results show that reuse benefits, from the developers' viewpoint, include lower costs, shorter development time, higher quality of reusable components and a standardized architecture. However, they found no relation between reuse and increased rework [100]. It is also interesting to note that, although applications using reusable components were trusted by the developers in this study, this trust did not extend to the reusable components themselves [100].

Land et al. [61] investigate reuse of software components from a practice perspective. They present qualitative results from an industrial survey on current practices and preferences, highlighting differences and similarities between development with reusable components, development without reusable components, and development of components for reuse [61]. Although component reuse is observed to occur, they report that their findings are still partly disappointing, since many potential benefits are currently not exploited [61]. Although the focus of this paper is example usage rather than component reuse, we find the two have a lot in common. It is also important to note that any serious attempt at software reuse must permeate the organization and allow existing processes and practices to be modified [53], and it is interesting to consider how this might play out in example usage.

Rotenberger et al. [91] consolidate an array of existing software reuse success factors into six dimensions, based on co-occurrences of reuse practices in an empirical data set. The dimensions describe the characteristics of software reuse settings, ranging from ad-hoc reuse to systematic reuse with high management support. They conclude that while an improvement of software quality can be achieved without an emphasis on the reuse process, an organization will only obtain the full benefit of reuse if a formal reuse program is employed, and subjected to quality control through formal planning and continuous improvement. They also found Object Technologies to be insignificant in explaining reuse success [91].

2.2.1. Pragmatic reuse

In contrast to the reuse literature discussed above, using code examples is more pragmatic and less disciplined practice. While pragmatic reuse tasks (also called code scavenging) have been shown to be effective [60], little follow-up work has been done, with the exception of a study by Holmes and Walker [39] that develops a tool for pragmatic reuse tasks. One related piece of literature discusses an example extensive software development method is the Opportunistic Software Systems Development (OSSD) [78]. It is an approach in which developers meld together software pieces that they have found, which of course can also be seen as pragmatic reuse. Most often, they find unrelated software components and systems that were not designed to work together but that provide the functionality they want to include in a new system. Typically, in opportunistic development, developers spend less effort developing software functionality to meet particular requirements and more time developing "glue code" and using other techniques for integrating the various software pieces [78]. OSSD has emerged to meet the market demands of delivering software quickly and with more functionality, however methodology-wise it poses unique challenges to developers, development processes, requirements engineering, system architecture, maintenance, and evolution [78].

Obrenović et al. report in [80] that using opportunistic software development principles in software engineering education encourages students to be creative, innovative, and to develop solutions that cross the boundaries of different technologies. Hartmann et al. investigate opportunistic design [37] through an interview

study of 14 professional and hobbyist "mashers" from three design disciplines: Web 2.0, hardware, and ubiquitous computing. In their work, they discover the mashups' epistemic, pragmatic, and intrinsic values for creators [37].

Two studies of opportunistic programming are described in [20], in which Brandt et al. study the interleaving Web foraging, learning, and writing Code. In a lab experiment, they found that programmers leverage online resources with a range of intentions: they engage in just-in-time learning of new skills and approaches, clarify and extend their existing knowledge, and remind themselves of details deemed not worth remembering [20].

3. Research methodology

This study is part of a comprehensive research on the use of code examples by professional programmers [9]. As part of this study, we conducted an online survey in which we asked programmers about their examples usage practices and techniques. To encourage developers to participate in the survey, we used social network sites [19] and posted invitation on the LinkedIn ".NET People" discussion group. We received 134 comments in this discussion thread, written by 67 separate subjects, and it became a virtual focus group [56] on the developers' perceptions of example usage.

This emergent use of socio-professional media for research purposes was surprising to us (the original post was only meant to solicit developers' participation in our online survey). Nevertheless, we found that by using the services offered by LinkedIn¹ we could extract additional data points in the analysis process. Specifically, we used the discussion participants' background such as years of experience, size of company and role to examine whether their views regarding example usage correlate to these factors.

LinkedIn.com is a business-oriented social networking website, hosting more than 100 million professionals worldwide, and rapidly growing at roughly one million new LinkedIn members every week (as of March 2011²). The network is global, 56% of the users are outside of the United States. High tech is the second largest sector represented in LinkedIn with 9% of the members (largest sector is Service with 20%), and 5 out of the 6 most represented companies in LinkedIn are technological.

The volume of LinkedIn, as well as its rich engagement mechanisms, makes it a suitable tool for data gathering regarding professional communities. In this paper, we investigate professional developers' views regarding example usage by analyzing LinkedIn group discussions.

3.1. Virtual focus groups

LinkedIn allows members to form and join groups based on professional or shared interest. Users may take active parts in discussions and quickly discover the most popular discussions in their area of interest. Members have an active part in determining the top discussions by liking and commenting, and may find interesting discussions by seeing who liked a discussion and how many people commented.

More than 17 million LinkedIn users are members in groups, there are more than 1.5 million new group memberships per week, and more than 1.2 million weekly posts and comments (including likes, jobs, news, and shares).

LinkedIn group discussions may form an online virtual focus group [56]. They have some interesting properties that make them suitable to be used for research purposes, as is explained in what follows.

¹ <http://www.linkedin.com/>.

² <http://blog.linkedin.com/2011/03/22/linkedin-100-million/>.

3.1.1. Subject details

LinkedIn users have an online profile. This profile includes items such as their employment history, experience, education, social network, recommendations. Since some users also use this profile for job hunting the data there is detailed and accurate. This fact has some important implications for researchers:

- Researchers are able to cross-reference between participants' answers and their background (educational or professional) and find interesting correlations.
- It lowers the barrier to research participation (compared to an online survey) as subjects are not required to fill their details before answering the survey questions.
- Subjects are traceable: the researchers may contact a specific subject, by email or via the site's internal inbox for follow-up purposes.

3.1.2. Reaching potential subjects

The following properties assist researchers in reaching potential subjects:

- There are specific LinkedIn groups in many fields and interests that allow researchers to address a segmented community, which is not only the most relevant for the research purposes, but which may also be interested in participating in the discussion.
- Using social platforms might provide the participants with an incentive to take part in a discussion to increase their Web presence in some areas of interest. Increasing Web presence of a practitioner may put him or her in a professional light in this field, and might also impress potential employers.
- LinkedIn promotes new topics and hot topics to the group's front page, which assists researchers in reaching a larger audience if the community finds interest in their topic.

3.1.3. Discussion visibility

In discussion groups, as opposed to online surveys, the discussion is visible both during and after the research period. In the following, we address the implications of this fact on three different groups: researchers, participants and the rest of the community.

The *participants* can see each other's answers, which further promotes the discussion. Participants can relate to statements of others and can adapt their arguments accordingly. Participants may change their minds over time, and may be influenced by peer pressure. This factor improves the *qualitative* data that can be obtained from this research tool, however it may distort some *quantitative* data as, for example, someone whose opinion was already expressed by another participant, will refrain from repeating the argument, which may create a silent majority. However, all these advantages and disadvantages are common in regular focus groups as well.

Other *researchers* could subsequently review the discussion and be able to re-validate the conclusions by themselves. This process is made easy with such a tool, as it requires no further effort on behalf of the original researchers (such as editing and removing identifications). One exception to the visibility issue is when a subject chooses to remove his or her previous comment. In such a case, this comment will be available only to those who archived it before it was removed.

It should be noted that using such data for research purposes, even if that data is in the public domain, is subject to consent of the participants. Indeed, in the case study described in this paper, our original post indicated explicitly that it was part of an academic study, stating the purpose of the research, and the affiliation of the posting researcher (the first author of this paper).

The discussion is visible to *other members of the community*: hence, a practitioner in a particular field would contribute to a discussion in that field in order to increase his or her Web visibility in that area. This might be useful in building an online persona and could be used to impress potential future employers. It raises the dilemma of whether the assertions made by the participants are genuine, or reflect their perception of what is expected of them.

3.2. Discussion Participants

We used the online profile of the discussion participants to extract relevant data about their background. We went over the professional experience section of their LinkedIn profile and checked the *size* of the companies they work for, the *role* they fulfilled in these companies, and the total length of their relevant *employment period*. This data is summarized in [Tables 3.1–3.3](#), which can be found in [Appendix A](#).

Most of the discussion participants served in multiple roles and in multiple companies; only one of them had no relevant background (an HR person who used the discussion thread for head hunting). The most common positions were software developer, senior developer and analyst. Sixty-four percent of the participants have more than seven years' experience (41% have more than 11 years' experience), and the average experience period was ten years.

Working in large companies is sometimes associated with the awareness of rigorous methods and inner protocols. For example, public companies often have a strict policy regarding using open source code, in order to protect their own intellectual property. Thirty-nine percent of the participants have worked for large companies (more than 10,000 employees).

4. Data analysis

The data analysis methods employed herein are qualitative in nature. Qualitative research methods in the field of empirical software engineering have gained popularity because of the 'rich diverse data,' 'triangulation,' and 'greater interpretive results' [93] and are considered to be useful for studies where there is an involvement of social or soft factors in empirical software engineering and information systems [93,64].

For the purpose of analyzing the data, this study uses an adapted version of grounded theory procedures as proposed by Glaser and Strauss [34], Charmaz [25] and Glaser [33]. It is ideal for use in situations where no previous theory exists [81], so the method suits the investigation of example usage where there is little existing literature. Our aim was not so much to develop a theory or core category, rather to generate some initial concepts in this under researched area. Grounded theory methods, for example, been used successfully in the software engineering literature, to investigate software process improvement factors [104]. Practitioners of grounded theory also suggest that, once new concepts have been found, it is important to integrate those concepts with the existing literature [110].

In the following section, we provide further details regarding our analysis process in order to increase the transparency of the analysis process, and mitigate concerns of researchers' bias and in turn increase the trustworthiness of the research [58]. While our study is interpretive in nature [111], we incline to a view of 'weak constructivism' [82] that holds that triangulation is both useful and desirable. In a strong constructivist study, where all interpretations are held to be socially constructed, triangulation is said to be impossible as there is no one point of reality on which to triangulate [82]. Because of our interpretive position, while we have taken every opportunity to describe possible sources of bias

and our steps taken to mitigate that bias, it is also important to acknowledge that coding of data is an inherently subjective process.

4.1. Context

This study originated as part of the doctoral research of the first author [9]. In his dissertation, he intended to study tools and techniques to incorporate example usage in professional software development. In that dissertation, it was taken for granted that using examples is beneficial, and the original focus was on developers practices.

In order to study these practices he conducted an online survey, and solicited participants via posting invitations on various Web-based forums and discussion groups. One of these invitations was posted on the .NET People discussion group of LinkedIn, which is the focus of this paper. Although this post only meant to refer its readers to the online survey, six people chose to leave comments on the discussion page. Most of them expressed a suspicious and selective approach toward extensive example usage. As researchers, who investigate practices and techniques for effective example usage, we were surprised by these comments. In fact, for a similar post in the “IT Developer Network” group on LinkedIn, we got results fundamentally different from those received in the .NET People discussion group: out of 40 comments made by 37 participants, all but one reported extensive and habitual use of examples and were enthusiastic about it. The only exception was a senior software architect and former associate professor who although supporting example usage, disapproved copy and paste as it introduces code clones.

He therefore posted another comment in this discussion thread, in order to obtain more developer feedback on this issue. We received 134 comments in this discussion thread, written by 67 separate subjects (some subjects commented more than once), and it became a virtual focus group on the developers’ perceptions of example usage.

4.2. Iterative analysis process

While this study is interpretive in nature [111], we still felt it was useful to surface our assumptions and biases during coding in order to make our coding procedure as robust as possible. For instance, our assumption about the field of research – namely taking for granted that example usage is consensual – required us to be very careful in our analysis. It should be mentioned that some of the bias may be a result of the first author’s academic background of programming languages and software engineering tools, as well as some industry experience as a software engineer.

The main challenge we dealt with was the instinct to be judgmental towards assertions made by the discussion thread participants (it should be stated that the authors did not take part in the discussion apart from the two posts mentioned above). It was important, we felt, to keep an open mind [111] about the concepts we encountered in the data. In order to mitigate that challenge we conducted an iterative, bottom-up analysis process, which was adapted from [102]. We followed Charmaz’s [102] stages of open coding, focused coding and theoretical coding, with some modifications when we made distinctions in the data [27]. Each iteration reveals concepts at a higher level of abstraction, and allows us to focus on key concepts of interest, as we further explain and demonstrate in the following sections.

4.2.1. First iteration – open coding

All 134 posts (including 12 that were later deleted by their authors) were read and copied into an editable document. Then all posts were coded – that is, each idea was summarized in a

single tag or a sentence, and added to the text as a comment.³ Using comments allows the other researchers to examine the summary sentences in their original context, and increases the objectivity of the analysis process. It also allowed us to familiarize ourselves with the dataset.⁴

This open coding phase [102] resulted with hundreds of items grouped by numerous concepts, such as: patterns and techniques for examples usage, reasons for- and against using examples, statements about the nature of programming in large, and properties of good and bad examples.

In order to make sense of this overwhelming amount of diverse data, we looked for ways to organize the categories. Recall that our original motivation for the research was developers’ practices, and that we were surprised by the presence of developers who oppose example usage. Hence, we started by reviewing the complete discussion thread and determining for each participant whether he or she is using examples or not. Doing so, we were able to associate the concepts mentioned by each participant with a larger group. We also maintained a reverse index document, in which each concept was mapped to all its occurrences in the original text (using comment ids).

This systematic classification of the participants revealed another interesting fact – there is not a complete dichotomy between the two groups (users and non users of examples) – there was in fact a range of use. There is a whole spectrum of users who limit their example usage only to a specific context. In fact, most of the discussion group members (51%) reported a selective or limited use of examples: only examples of certain size, only for learning purposes, only from specific sources, etc. In some cases the selective use was explicit (e.g. when a participant asserts that he or she *never* uses the example code, or that he or she uses examples from a trust worthy source *only*) but sometimes the limited use was implied in the answer (e.g. when a participant reports using examples when starting something new but ignores other opportunities in which code examples could have been used as well). Therefore, we then classified each participant into one of three groups: (1) habitual example users, (2) those who oppose using examples altogether due to their inherent pitfalls and (3) those who use examples in a limited context only.

Some of the people we included in the third group are keen supporters of example usage; however, they were included in the third group because they mentioned using examples in a narrow or limited context only. Motivation for doing so is also partly explained from previous results by Barzilay et al. suggesting that even developers who favor example usage are not attentive to using them in multiple contexts [13].

In order to reduce researcher subjectivity, and the effect of personal opinion on the use of examples, the open coding was later reviewed by two other researchers, and disagreements were discussed and resolved.

Fig. 4.1 depicts 2 examples of the open coding discussed above.

4.2.2. Second iteration – focused coding

In the first iteration, we divided the discussion participants into three groups, according to their approach to example usage and summarized their arguments. However, we felt that our analysis had not moved beyond the descriptive level, a problem with coding that is described in [109]. We aspired to gain new insights using more analytical labels for concepts in the data. We did so in two steps. First, we conducted ‘focused coding’ defined by Charmaz [25] a more directed and conceptual stage of coding using the most significant or frequent codes. We looked for common issues

³ We used Microsoft Word Comments feature (available as part of the Track Changes toolbar).

⁴ The full text is 20,511 words long.

<p>"I use examples to tackle a <u>new technology</u>. I <u>cut and paste</u> my <u>refactored</u> code from <u>other projects</u>. I use snippets (from <u>myself</u> or <u>others</u>). Why <u>reinvent the wheel</u> every time you start a <u>new project</u>?"</p>
<p>Context: new_technology, new_project Technique: cut_and_paste, refactoring, full_reuse Source: myself, others, other_projects Motivation: reinvent_the_wheel</p>
<p>Group: Selective (3) – limited context</p>

<p>"with s/w apps becoming so std. Most of the common code is up there. Often it <u>saves times</u>, but I thoroughly <u>test</u> it before making it live. I also try to tweak for better <u>performance</u>, bc most of the code is NOT.. I develop apps using .net, jQuery, ajax.. and the websites I trust are <u>msdn.com, jquery.com, codeproject</u>."</p>
<p>Motivation: time Technique: testing, performance_tuning Source: msdn.com, jquery.com, codeproject</p>
<p>Group: Habitual (1)</p>

Fig. 4.1. Two examples of open coding. Each example is built of three parts: at the top is the text written by the discussion participants. In the middle are concepts which were located in the text (underlined), and in the bottom is the group this participant was associated with.

mentioned by participants of multiple groups – crosscutting codes (e.g. issues that were mentioned by both habitual example users and example antagonists). We found two such crosscutting codes, namely *time (speed)* and *copy-and-paste*.

To demonstrate how this coding stage assisted us in exposing additional concepts, consider, for example the following two quotes mentioning the time dimension with respect to example usage: "Anyone who is not busy looking for code for reuse is reinventing the wheel and wasting time". On the other hand: "... Yes, it may be reinventing the wheel or a little time consuming but I am not pressed for time". This latter quote referred us to the concern that developers, who do not use examples extensively, see time saving as incidental – a bonus they receive when copying and pasting the example code. They do not acknowledge its importance to the organization.

In the second stage of this analysis, we applied focused coding [25] to revisit the whole discussion thread using our two initial cross cutting themes by way of analytic direction. We ended up this iteration with 9 new themes, which are summarized in Table 4.1. Namely: (1) The way developers conform to organizational goals, (2) How developers perceive their personal development, (3) Whether the developer acknowledges the dexterity

required for effective example usage, (4) Ego, (5) Community identity, (6) Ownership, (7) Trust, (8) How developers perceive their role definition, and (9) the analytical skills that developers consider relevant to their job.

Reflecting on the research process thus far, and considering our initial bias towards tools and techniques, these human aspects were a major discovery for us. These initial focused codes suggested that current engineering practices are dominated by the developers' approach towards example usage which, in turn, is dominated by their personality.

Fig. 4.2 illustrates the focused coding of two posts, both mentioning copy and paste. Also, note that one of the posts in Fig. 4.1 also mentions the trust concept – this concept was not captured at the first iteration, but only at the focused coding phase.

4.2.3. Third iteration – applying the theoretical lens of prejudice theory to the coding

We finished the second iteration suggesting that example usage is dominated by 9 concepts within the two overarching issues of copy and paste, and time and speed. From a practical perspective, this insight introduces a new dimension that should be addressed in the software construction ecosystem. As the availability of existing code increases, those aspects are of interest to software organizations, training programs, tool builders and potentially others.

However, from a theoretical perspective, those 9 aspects did not add up to a single cohesive concept. In order to break this loop we reviewed the discussion thread again, this time ignoring our coding, and a new observation emerged. We noticed that some of the participants were not only referring to the mere activity of example usage, nor to themselves, but also made comments regarding other developers based on their *attitude* towards example usage. These references were coded in previous iteration as "ego" and "community identity". However, while reviewing these comments again it occurred to us that the hierarchical and judgmental tone, and the blanket statements and sometimes offensive language used, go beyond mere ego. We decided to pursue this direction further.

We referred to the literature to learn about theories of prejudice (presented in Section 2.1), and used the main concepts as to help us integrate and organize the findings. Putting this 'theoretical code' [33] as a frame over our findings helped us abstract and make sense of our findings.

We note that overlaying theoretical codes from the prejudice literature on top of existing codes was beneficial, and strengthened our confidence in the chosen theoretical framework and in the potential for theoretical integration. In many cases, the theoretical

Table 4.1

Summary of the themes from which developers' approach to example usage is derived.

Key issues	Theme	Example supporters	Example opponents
Time and speed	Conforming to organizational goals	Saving time is an explicit organizational goal	Saving time is accidental
	Personal development	Achieved via mastering effective development techniques	Achieved via mastering additional technologies
	Acknowledging example dexterity	Searching techniques could be learned	It is faster to write the code
Copy and Paste	Ego	Egoless programming	Disrespect developers outside of the organization
	Community identity	Example writers are no different than us	Do not consider themselves as example writers
	Ownership	Develop practices for taking ownership of code written by others	Avoid taking ownership of code written by others
	Trust	Develop mechanisms for gaining trust in people and in code	Do not trust developers outside of the organization
	Software engineer role definition	Deliver functionality	Write code
Analytical skills	Find, read, understand, and alter existing code	Write code	
		Incremental design	Design systems

"Yes definitely. Examples are a way to understand the concept better. I always analyse other's coding patterns to see if there is any scope for <u>improvement in my coding practices</u> . Copy-Paste not advised as it will never <u>help a programmer to improve</u> , but it will help in improving the <u>analytical skills</u> . Thanks all those great guys who take their time off to provide lot of examples to <u>support fellow programmers</u> ".
personal_development, analytical_skills, community_identity
"I <u>describe myself</u> as a Problem Solver as opposed to a Solution Repository! I may agree that the reverse <u>would produce</u> better experts but, I think, with <u>less creativity</u> ... I admire and positively commend CodeProject - I fact I have there <u>my own articles</u> ..."
role_definition, personal_development, analytical_skills, community_identity

Fig. 4.2. Two examples of focused coding. At the top of each example is the text written by the discussion participants. At the bottom are the human aspects which were located in the text (underlined).

codes corresponded to our selective codes, for example occurrences of the selective "community identity" code could easily be interpreted as "ingroup bias", which is manifested in "trust" towards other members of the ingroup. Furthermore, identifying most of the participants as "selective example users" at the first iteration can be examine through the lens of the optimal distinctiveness model of social identity [22] mentioned earlier in the context of prejudice – these developers' self-perception is a product of opposing needs to inclusion (assimilation) and differentiation from others.

Table 4.2 maps between the theoretical codes (drawn from the literature) and the selective codes deduced in the previous iteration. It can be seen that conforming to organizational goals (ORG) is mapped to only a single concept and it does not fit the chosen framework (in fact it even goes against our theory – we argue that prejudice towards example usage is contrary to the interest of the organization). Personal development and role definition (PER and ROL) are also mapped to only a single concept, however, they are dominant in our theory.

When we discussed our findings with practitioners, we found support for our conclusion – prejudice towards example users exists. For instance, one senior R&D manager confirmed that he hardly ever hired to his team developers who used code examples in the qualification exams performed by his company. When we asked him whether they are rejected based on their performance, or merely because of using code examples, he asserted that using code examples is a signal of developers not proficient enough to write original code, and that the vast majority of them do not understand the code they use.

5. Findings

In this section, we present our findings using a prejudice lens, and demonstrate how our findings map onto the prejudice concepts used in Table 4.2.

Table 4.2 Mapping selective codes to theoretical concepts.

Prejudice theory concept	Theoretical codes	ORG	PER	DEX	EGO	COM	OWN	TRU	ROL	ANA
Authoritarian personality	Rigid thinkers, obey authority, black and white, hierarchies, double standards, self-deception, like-minded, "Moral Majority", "legitimizing myths", limited resources	X			X	X	X			X
Categorical thinking	Stereotypes, generalizations, "pictures in the head", self-perpetuating			X	X					X
Outgroup homogeneity	No contact, alienation, expendables, stereotyped, values,				X	X	X	X		
In group bias	Common goals, shared values, social comparison, Power politics, Perceived threat, trust, cooperation, self-esteem, social identity,		X		X	X	X	X	X	X
Causal attributions	Just world, fundamental attribution error, ultimate attribution error			X	X					

Using the following abbreviations: Conforming to organizational goals – ORG. Personal development – PER. Acknowledging example dexterity – DEX. Ego – EGO. Community identity – COM. Ownership – OWN. Trust – TRU. Software engineer role definition – ROL. Analytical skills – ANA.

Similar to prejudice on other domains, the magnitude and the expression of prejudice differs considerably from one person to another. In some cases, it is explicit, in other cases we are able to identify dynamics that may lead, in the future, to prejudice on some level. For each concept, we also demonstrate the wide range of possibilities using direct quotes of the discussion participants.

In its explicit form, example usage, and in particular the act of copy-and-paste, was associated with carelessness, bad practice, and poor-quality work: "Hmmm, copy-paste code is bad programming; in fact, it is not programming at all". Some of the terms used were: "mindlessly copy-paste", "blindly copy-pasting" and "cut-and-paste monkey". One participant even described example writers as: "guys in their bedroom who most of the time have no idea".

However, our interest focuses on the more subtle cases. Below we examine some of the dynamics that may lead to prejudice, and in particular examine how prejudice might result in developers whose identity and sometimes even self-esteem are derived from perceiving themselves as code writers. As suggested by Bewer [23] this may result in the creation of ingroup bias, and can be in turn used as a platform for hate of the outgroup – in this domain, example users.

5.1. Ingroup bias

Consider the following excerpt:

"[...] the ultimate goal should be to "code". That means code most of the code yourself and figure it out....push yourself. Don't go into forums asking questions and getting the answer every time or looking for code snippets that do all the work for you. Start coding, struggle, and LEARN".

This developer manifests what, he believes, are the common goals and values of any developer, which he drives from his own "social identity". He perceives himself as a code writer, and feels strongly about it. It is reasonable to believe that writing code was the main practice he was trained to do (considering that the availability of high quality open source on the Internet is relatively a recent phenomenon). And now, facing an alternative set of skills (the use of code examples), he is looking for others sharing his views to form an ingroup.

For the members of this (virtual) group new code is the unit of progress, a sign of productivity (however misleading it may sometimes be). Copying, on the other hand, is perceived as a devalued shortcut – an imitation rather than a creation. In most university courses, the students are not allowed to share their work with fellow students, but are expected to write their own code.

In this context, it is understandable that developers feel that they are expected to write new code: "A job of a programmer is to program, not play with Lego blocks where every block fits everywhere (nearly)". One programmer even reported that writing code is the expectation of some managers: "The problem I have is when the team leader says 'Always code it from scratch', even if you are

essentially going to code 70% of it the same and add some custom code to it". Another developer took a more pragmatic and less subtle approach toward example usage: "Some people will try to justify their existence by thinking that [they] have to code every line of every module they are working on, but that is just silly and a waste of time".

As discussed earlier, maintaining ingroup integrity may lead to expressions of moral superiority and social comparison. In other cases, example usage opponents explicitly expressed their moral superiority using images and blanket statements ("cut-and-paste monkey"), and considering them as unprofessional: "I would never respect a copy/paste or even a drag/drop programmer. They lack understanding or even the passion to creating code". And this one: "blindly copy-pasting the code from the examples is a very bad practice. A lot of people do that in IT industry. But, I don't consider them as coders..." These speakers use social comparison to distinguish between themselves and the outgroup of "copy and pasters", and attribute them degrading characteristics (have no passion to creating code, not considered coders). These quotes also exemplify the case of ego, which may be associated with prejudice [36,41].

5.1.1. Group identification

Another aspect of group association is related to the large community of developers worldwide – programmers who oppose the use of examples do not consider themselves example *writers*. The few participants who reported writing examples were also keen supporters of example usage. Example writers see themselves as part of a larger community, and acknowledge the social aspects of software development: "I personally look at forums more often than not because posts are usually written by real programmers or at least answered by real programmers. I also answer questions on forums so that karma, if you will, does not catch up with me. I actually rather enjoy answering questions about things I know".

Example users acknowledge that code examples were written by developers much like themselves or their teammates, and they respect their competence. One developer even compared avoiding example usage to not being open to the opinions of others: "I equate some of these statements similarly to saying...I don't listen to others' opinions because they are often wrong". Another developer suggests that opposing to using examples "is directly related to and is the result of egoistical and therefore overconfident developers [...] our profession needs teamwork, not an ego contest". The following participant does not see any fundamental difference between his teammates and the people on the Web: "Maybe it's my bad experience, but I've worked with as many bad coders per capita as I find on the internet as well".

Another indicator to group identification via the lens of example usage is the fact that many of the participants confessed using code examples for certain purposes, for example only for learning purposes, only when starting a new project, etc. By that, they demonstrate the distinctiveness model of social identity [22], preserving self-identity without excluding themselves from the larger group of example opponents. Doing so could also be seen as an indicator of causal attribution error (discussed below) – avoiding certain practices legitimizes holding position against extensive example users.

5.1.2. Trust and ownership

Having ingroup bias often limits boundaries of trust and cooperation [46], which may explain why some developers avoid copy and paste at all cost. They do not *trust* other programmers enough to take responsibility for and ownership of their code. These programmers find it difficult to understand existing code, they feel that they cannot identify fallacies in someone else's code nor test it thoroughly. They prefer to write the code by themselves and take responsibility for it rather than trust others, and perhaps lose control over their code.

Other programmers report that they only use trusted sources such as Microsoft's built in tools or trusted Websites. One developer says: "the Websites I trust are msdn.com, jquery.com, codeproject". Note that he said the Websites I *trust* and not the Website I *use*. In several cases developers report using their own stash of examples, in one case a developer reports that although he does not usually copy example code, when he starts a new project "...I grab a ton of code from old projects. But it is generally my code (or my team's code)". He trusts himself, and trusts his teammates. He trusts the familiar.

Somewhat related issue is developer's *ownership* of the code. Taking ownership of something is a delicate and sensitive process by nature and involves taking *responsibility* for possible failures. By copying and pasting code into one's own code base, the programmer takes ownership as well as responsibility over someone else's code. This issue dominated some of the comments presented above. In two cases, developers mentioned the words 'respected' and 'reputable' to emphasize this point.

Some of the activities that developers perform after pasting an example could be interpreted as steps in the process of taking ownership of this code. Among these activities, we find applying performance and security considerations, refactoring, and commenting. One developer said: "I go through the code line by line, and write comments appropriate to the situation". Several programmers reported that they like to convert the style of the example (naming conventions, indentation, etc.) to their own style; one said that this process is assisted by the development environment. Several programmers reported that they end up rewriting the entire example: "I always end up rewriting them according to my personal style, and this gives me a further understanding of what I am implementing".

We see these post-paste activities, and style change in particular, as some kind of a 'taking ownership ceremony' performed by developers. As taking ownership is a delicate process it is for to these developers to cling to some kind of routine (the ceremony) in familiar stages, to reassure themselves that they have taken appropriate measures to minimize the risk of incorporating the new code, and in addition, the style change causes the code to look familiar and less intimidating. This ceremony is, of course, not only a symbolic act – adhering to strict coding conventions across the code base of the organization increases the code readability and improves collaborative work. Other post-paste activities are also accorded their technical justification – however, the ceremony itself makes it easier for programmers to accept code written by others.

5.2. Outgroup homogeneity

Some developers share their bad experiences with bad example users, which affected their approach towards example usage in general: "I was speaking from my own experience. I've been hired many times to fix code that had been stapled together using examples from code cookbooks." One way in which outgroup (perceived) homogeneity is achieved is when there is only little contact between the outgroup- and ingroup members [70,42], as this developer emphasized: "Personally, I've never met a copy and paste person (obviously that's not programming) and in reality it won't get you very far when life confronts you with a problem".

5.3. Authoritarian personality

An early research stream about prejudice [3] argued that it is associated with an authoritarian personality highlighting strict distinctions and hierarchies.

A few participants emphasized that they never use example code in production: "But I never use 'examples' as a part of the *real code*" or "I'd never use code samples in any commercial product"

and “But I never use them at work”. All these programmers claim that they use copy-and-paste examples for learning purposes only.

These developers distinguish between example code and their product code (written by themselves or by their teammates). This distinction is hierarchical and judgmental – production code is perceived to be superior while examples are inferior. They insist on pointing that the examples never find their way into production, the ‘real work’, and suggest that its purity is not compromised by the ‘inferior’ example code. Some of these developers not only look down on example code, they also look down on the developers using it: “I would never respect a copy/paste or even a drag/drop programmer. They lack understanding or even the passion to creating code”.

It is possible that the community studied might have some authoritarian characteristics to begin with, when viewed through the lens of prejudice theory. Our data set was compiled using a discussion thread of .net developers. .net is a proprietary Microsoft technology. Microsoft, as opposed to many other technology companies, creates its own proprietary ecosystem (database, development environment, operating system, web server, and more). Moreover, Microsoft also takes a centralistic approach in comprehensively documenting all its products.

It is arguable that a .net developer is acting in an “authoritarian world” dominated by the technology manufacturer, and imposing only limited room for the developers to express themselves using the tools given to them. As illustrated in this quote: “I read MSDN code samples to see how Microsoft does it. They developed .net...that makes their code samples a great tool for us, their library users, to optimize our syntax and best way to write “the .net way””.

It is also possible that .net developers may consider themselves as an ‘in-group’, in prejudice theory terms, and would interpret practices nurtured out of the community as inferior. In this context, prejudice against example users may also be justified on the basis of employment scarcity [96], explicitly addressed in this quote: “Any software developer/engineer doing this is going to [have to] find a new jobs real fast”.

5.4. Categorical thinking and stereotypes

As we mentioned above, some programmers stereotype habitual example users as unprofessional. We believe that this is, in part, due to the (stereotypical) bad reputation of copy and paste, as we further explain.

The *context* in which many software developers were trained was different from the one we have today (year 2013). Only few years ago, much of the free source code which is now available online, simply did not exist. Moreover, some of the tools and platforms which assist in the process of retrieving, assessing and incorporating this code were not mature enough and did not function well. In this “old” context, developers were educated that copying code is wrong. However, there was an implicit assumption in this commandment – that the duplicated code is their own code (or part of their organization code base), and by copying from themselves they act unprofessionally, introducing redundancy to the system and a potential future inconsistency. Even when code has started to appear online, incorporating it into one’s own code base was still error prone, and assessing the quality of arbitrary code snippets found online outweighed the benefit of the reuse process.

In the new context of code available on the Internet, and mature tools and platforms for code retrieval, which reduce the risk involved in such code reuse, the assumptions mentioned above do not hold anymore. However, because of the implicit nature of the original context, some of the practitioners may still perceive example usage as wrong and those who do that be stigmatized as unprofessional. For these developers Copy and Paste serves a *symptom* for malpractice and low quality, and developers using this

method are categorized as such in a stereotypical way. Moreover, copying by and large is stereotyped as wrong. In many cases copying is seen as cheating. For instance, at school, developers would have been taught as children that copying homework (or worse, during exams) is prohibited and subject to sanctions.

5.5. Causal attributions

Some of the participants attribute low quality coding to the fact that the code author was a “copy-and-paste-programmer”: “I was speaking from my own experience. I’ve been hired many times to fix code that had been stapled together using examples from code cookbooks.” And this one: “Unfortunately I got a call from a client to do support on a project that was built by a bunch of students from a local university. Clearly a cut and paste job, and was definitely a rough read” – again, low quality is attributed to the fact the developer copy and pasted this code. Mentioning that the developer were students from “local university” implies lack of competence and experience.

Examples of the ultimate attribution error [85] exist when considering the dexterity (and extra work) required using examples effectively. Some developers claim that using examples does not save time at all: “I find that I can spend less time writing my own free JavaScript plugins or just raw CSS tricks than I would spend trying to customize some obscure little feature the client is looking for in the RAD tools”, whether because of the time it takes to find the right example (“There’s nothing worse than cribbing a sample, spending a lot of time with it and then finding it doesn’t actually work...”) or the time it takes to understand or rewrite it (“when I am working with a programmer that productizes a sample, he/she is always tweaking and debugging the code until they end up rewriting [...] It is more cost effective to write your own code, rather than cutting and pasting someone else’s code”).

It is reasonable to assume that these developers lack the appropriate practices and skills needed to use examples effectively: they were trained to write code rather than to search and read code. They are used to thorough designing not frequent refactoring. Mastering effective example usage requires training, practice, and time; however, it is first necessary to acknowledge the fact that such training is essential.

On the other hand, some participants agree with the assertion that additional time should be invested in ‘fortifying the example’ – making the necessary changes in the code in order to address corner cases that were not addressed by the example writer; however they do not see that as a problem. On the contrary, they appreciate the reuse of the underlying ideas: “Even if I take half-baked code that is backed by a good idea or good design, I am better off starting from the sample than starting from scratch”. This individual implies he has a proper practice of taking an example and manipulating it. He explicitly acknowledges the kind of effort that example usage requires.

6. Discussion

Strauss [103] says it is essential to ‘grapple’ with the technical literature once a grounded analysis has been completed. We felt it is important, therefore, to engage also these ‘intermediary’ findings with domain specific literature, i.e. organizational studies, information systems and software engineering. Doing so, we obtain further confidence in our inductive coding process and are able to demonstrate additional aspects in which we extend the literature (other than introducing the prejudice lens discussed earlier).

Further, we identify where our concepts confirm and extend the literature. The discussion is summarized in the table below.

Theme	Confirmation in the literature	Findings where we extend the literature
<i>Time and speed</i>		
Conforming to organizational goals	Bosch [18]: speed trumps R&D	Methodological example usage as an instance of organizational knowledge management – see [51,45] and Lee and Kim [65]
Personal development	[40] and Pal ^a – developers want skill development, regardless of language used	Fuller and Unwin [5] suggest that an expansive approach to learning provides the basis for the integration of personal and organizational development. With respect to example usage, developers who use examples extensively integrate their personal development goals (becoming an effective developer) with the organizational ones (delivering software fast)
Acknowledging example dexterity	Tradeoffs between time and quality do exist with regard to example reuse [16]	Aligned with [53] and conversely to Slyngstad et al. [100], some developers perceive example usage, and consequently software reuse, as time consuming
<i>Copy and Paste</i>		
Ego	Ego was present in the developers approaches to example reuse, confirming Lechner [63] and Williams and Kessler [115]	The community studied is essentially a community of individuals, which suggests that in order to implement the ‘egoless programming’ approach [113] restructuring of the social environment is needed – such as by nurturing a community identity
Community identity	Some participants do not see themselves as part of the same community, confirming Glass [35]	Ferrán-Urdaneta [28] suggests that “a community is a more effective structure than a team for legitimizing knowledge”. In this case, the lack of a community identity would delegitimize knowledge of examples
Ownership	Developers tend to use code they own or gained some expertise with [26]	We suggest that a collective ownership practice [15] assists in removing the barriers for taking advantage of code written by others
Trust	Sharma et al. [95] identify trust as one of the concerns of using open source	In the data we find evidence for trust attributed to individuals, companies and websites We extend Serva et al. [94] work on boundary-spanning activities [4] to examples outside the organization or the team. Extending Holmes and Walker [39], we also found that developers gain trust via testing and understanding.
Software engineer role definition	Traditional perceptions of software engineers [87,59] were confirmed in the data	Our data also illustrated shifting self-perceptions of software engineering, as evidenced by the statement: “I describe myself as a Problem Solver as opposed to a Solution Repository!” This is similar to [78] but it is also noticeable that, in this public forum, participants managed their professional image for prospective employers
Analytical skills	Carr [24] suggests that search-oriented information retrieval behavior might have a negative effect on a person’s ability to stay focused for long. Similar opinions were voiced by two of the discussion group participants, who commented that extensive example usage interferes with learning	On the other hand, Obrenović et al. report in [80] that using opportunistic software development principles might be helpful for creative solutions – we suggest that extensive example usage may be such a case

^a <http://onlamp.com/pub/a/onlamp/2005/06/23/whatdevswant.html>.

6.1. Time and speed

Programming time is a fundamental issue in professional and commercial programming. In the professional software industry, a short time-to-market period provides the organization with a competitive edge and reduces costs, which, ultimately, pays the programmers’ salaries. Bosch [18] claims that increasing speed trumps any other improvement with which R&D can provide the company, and he quotes Jack Welch: “If you are not moving at the speed of the marketplace you’re already dead – you just haven’t stopped breathing yet”.

6.1.1. Conforming to organizational goals

We examine this issue further from the organizational behavior theory perspective. Specifically, we consider methodological example usage as a form of organizational knowledge management. As knowledge emerges as the primary strategic resource for firms in the 21st century, researchers and practitioners strive for clues on how to accumulate knowledge resources effectively and manage them for competitive advantage [65]. The final goal of knowledge management is to gain competitive advantage and sustain it by producing new products or services or enhancing organizational processes in terms of speed, quality and costs [51,45]. Lee and

Kim [65] propose an integrated management framework for building organizational capabilities of knowledge management. The framework comprises four major components of management: organizational knowledge, knowledge workers, knowledge management processes, and information technology.

Thus, a methodological example usage is actually an explicit organizational goal, and employees who neglect this aspect do not only ‘take their time’ but also interfere with achieving this goal.

6.1.2. Personal development

The Personal Software Process (PSP) [40] legitimizes this approach. PSP training focuses on the skills (rather than on technologies) required by individual software engineers to improve their personal performance. This approach is also supported by Pal. In his article “What Developers Want”⁵ he claims that “Irrespective of the language programmers choose for expressing solutions, their wants and needs are similar. They need to be productive and efficient, with technologies that do not get in the way but rather help them produce high-quality software”.

Fuller and Unwin [5] address the integration of organizational and personal development. They argue that learning environments that offer employees diverse forms of participation foster learning at work. They mention three participatory dimensions: (1) opportunities for engaging in multiple (and overlapping) communities of practice at and beyond the workplace; (2) access to a multidimensional approach to the acquisition of expertise through the organization of work and job design; and (3) the opportunity to pursue knowledge-based, work-related courses and qualifications. They suggest that the expansive approach to learning provides the basis for the integration of personal and organizational development. Indeed, with respect to example usage, developers who use examples extensively integrate their personal development goals (becoming an effective developer) with the organizational ones (delivering software fast).

6.1.3. Acknowledging example dexterity

In the agile community there is a discussion about possible tradeoffs between speed and quality. Robert Martin (‘Uncle Bob’) argues in his blog post [16] that ‘speed kills’ – an undisciplined programming rush is counterproductive and that ‘slow and steady wins the race’. He is referring to a previous post by Ron Jeffries [47] claiming there is no tradeoff between quality and speed, however it requires skill and judgment. Lack of such skill, or lack of example usage discipline, may imply a tradeoff between time and quality.

6.2. Copy and Paste

Literature on the topic of code cloning often asserts that duplicating code within a software system is bad practice, that it causes harm to the system’s design, and should be avoided. However, in [52], Kasper et al. there is significant evidence that cloning is often used in a variety of ways as a principled engineering tool. They describe several patterns of cloning they have observed in their case studies and discuss the advantages and disadvantages associated with using them. Jackson and Kang advocate code duplication for ‘mixed-criticality’ systems [43].

Kim et al. [54] describe an ethnographic study of copy and paste (C&P) programming practices and present a taxonomy of C&P usage patterns. They propose a set of tools that can both reduce software maintenance problems incurred by C&P and better support the C&P scenarios used.

LaToza et al. [62] also address C&P example code by presenting both qualitative data from interviews and quantitative data from

two surveys. They identify six distinct forms of code duplication; each clone type is characterized by its creation mechanism, by whether developers are aware they are creating clones, by the refactoring challenges to remove the clones, and by the size of the clones. The most studied clone type, i.e., example clones, occurs when some code is copied and pasted, and modified. The authors expect that this usually involves a small amount of code.

6.2.1. Ego

We find support for the ego effect on software development in the literature as well. Lechner [63] addresses egocentric behavior when he describes a failing extreme programming project. Williams and Kessler [115] found ego to be a dominant factor in pair programming. They describe excess ego problems as well as “too little ego” ones, and discuss the way to deal with them. In contrast to these, in the case of example usage the egocentric behavior is not attributed to a team member, so it is harder to identify and to address.

The ‘egoless programming’ approach [113] suggests that the problem of ego must be overcome by a restructuring of the social environment and, by this means, a restructuring of the programmers’ value system in that environment [112].

6.2.2. Community identity

The community identity of developers was studied primarily in the open source development context. Madey et al. [72] analyze the open source software development phenomenon based on social network theory. Lewis [67] examines how to use online communities to drive commercial product development. Sharma et al. [95] propose a framework for creating hybrid-open source software communities to let traditional organizations implement and benefit from open source development practices.

In the literature, we find distinctions between the treatment of open source *software* and the treatment of open source *developers*. On the one hand, Raymond claims in [88] that the open source revolution has helped propel the collaborative approach to software development into the mainstream. On the other hand, however, when it comes to the open source developers Glass says [35]: “I don’t know who these crazy people are who want to write, read and even revise all that code without being paid anything for it at all”. By talking about ‘these people’ Glass is exemplifying the approach expressed by some of the discussion participants – they do not see themselves as part of the same community.

Let us investigate this issue further from a wider perspective, considering examples as units of knowledge. Ferrán-Urdaneta [28] investigated knowledge creation, knowledge legitimization, and knowledge sharing from the perspective of two group structures: teams and communities. He suggests that “a community is a more effective structure than a team for legitimizing knowledge”, hence in the example usage case, the lack of a community identity would delegitimize this knowledge – the example. It is suggested that inside large corporations, managers encourage the emergence of “communities of practice” [114].

6.2.3. Ownership

The issue of ownership was also addressed by the agile software development community. Two of the Extreme Programming practices, *shared code* [15] and *collective ownership* [14], may assist in disarming some of the developers concerns in taking ownership of code which was written by others. Beck and Andres [15] agree that if no one person is responsible for a piece of code, then everyone will act irresponsibly. Unless the team has developed a sense of collective responsibility, no one is responsible and quality will deteriorate. People will make changes without regard for the team-wide consequences. In order for collective ownership to succeed Beck [14] proposes to integrate the code frequently, to write and run tests, to pair program and to adhere to coding standards.

⁵ <http://onlamp.com/pub/a/onlamp/2005/06/23/whatdevswant.html>.

Jeffries et al. [48] also address this issue. They claim that developers are afraid to change code not owned by them, and quote a typical extreme programming developer as saying: “I’m not afraid to change my own code. And it’s all my own code”. Again, the collective ownership practice helps in removing the barriers for changing code written by others.

Fritz et al. [30] address the problem of developers’ varied levels of expertise in different parts of their code, and propose a degree-of-knowledge model to capture source code familiarity. As code ownership is associated with expertise in that code, which developers wish to avoid – because once titled an expert, software engineers find themselves being allocated to projects based on their past experiences, rather than those that may be more intellectually challenging and have room for learning [26]. Such a mechanism might make it easier for developers to take ownership of code, conditional on their not being considered experts in the domain of that code.

6.2.4. Trust

Individuals may generally *trust* members of their inter-organizational ego-centered networks in terms of member competence [46]. Sharma et al. [95] identify trust as one of the concerns of using open source, and propose that core developers work closely with one another and develop trust. Trust issues in software development were also investigated by Serva et al. [94] in the context of outsourcing. They find trust as encouraging boundary-spanning activities [4] (in the example usage case – examples outside the organization or the team).

In a survey by Holmes and Walker [39] they asked industrial developers about their approach to pragmatic reuse. One of the major reasons for reuse was to increase the reliability of their code. The developers wanted to “leverage existing testing”. Code was more desirable if tests existed for it as they increased the developers’ trust in the quality of that code. This is also a case of the emergence of a community of practice discussed earlier.

6.2.5. Challenging software engineer role definitions

Pour et al. [87] discuss the credentialization of the software engineering profession through education, accreditation, licensing and certification. These mechanisms contribute to the stabilization of the classic perception of the software engineer as a code writer, as they are backed by the ‘classic’ software engineering bodies of knowledge (e.g. [1]). In an article called “What Do You Mean I Can’t Call Myself a Software Engineer?” Speed [101] describes a licensing criterion specifically suited to software engineers, explains the legal issues involved and how they affect software engineers. Kruchten [59] investigates the engineering aspects of software engineering and lists five differentiating characteristics: absence of a fundamental theory, ease of change, rapid evolution of technologies, very low manufacturing costs, and no borders.

In Section 3.1 we noted that due to the visibility of the discussion to potential employers, participants might say things in order to satisfy them. If this is the case, it implies that most of the participants think that they ought to express opinions that oppose example usage rather than support it, which further strengthens the way they perceive their role.

Another derivative of the programmers’ perceptions of role definition is related to the *personal development* perception mentioned earlier: programmers, who perceive themselves as *code writers* will also tend to perceive their personal development as being able to write more code.

In addition to the perception of the individual role in the software community is the essential perception of the community itself. A creation process is often considered nobler than a

reformation process; some even consider computer programming to be an art [55]. In the same way that an architect enjoys more prestige than does a handyman, so does the classic programmer aspire to be held in high regard (or as one of the developers quoted in [13] said: “At the end of the day, we are all merely plumbers...”, suggesting that the essence of the software engineering job boils down to putting the components together and making small non-glorious fixes, similar to approach manifested by OSSD [78]). After all, some of them might have chosen programming as their profession because of its status.

6.2.6. Analytical skills

In his article “Is Google Making Us Stupid?” [24], Carr suggests that search-oriented information retrieval behavior might have a negative effect on a person’s ability to stay focused for long. He also quotes Wolf [116] who said: “We are not only *what* we read – we are *how* we read”. When we read online, she says, we tend to become “mere decoders of information.” Our ability to interpret text, to make the rich mental connections that form when we read deeply and without distraction, remains largely disengaged. Similar opinions were voiced by two of the discussion group participants, who commented that extensive example usage interferes with learning. On the other hand, Obrenović et al. report in [80] that using opportunistic software development principles in software engineering education encourages students to be creative and innovative, and to develop solutions that cross the boundaries of different technologies.

7. Limitations

While this study has followed a ‘weak constructivist’ approach [82] and is interpretive in nature, we know that some readers will be interested in the limitations of our coding approach from a positivist perspective. The weak constructivist approach acknowledges the subjectivity of coding and takes steps to triangulate, while asserting that reality is socially constructed. We took the following steps to ensure a robust and valid study.

1. Reducing subjectivity in the coding process
Researchers are human, and as such they have their own opinions and beliefs. Grounded theory is very clear that concepts emerging from the data should be considered in their own right rather than the researcher imposing an interpretation [32]. We approached the data with an ‘open mind rather than an empty head’ [111], not forgetting relevant literature but making sure that we looked at what the data was telling us first. We also took care to surface our own assumptions about the utility of example usage and acknowledge those assumptions. Another method used to limit the effect of subjective interpretation was having the coding member checked by two other researchers, and having the second author of this paper joining after the data have already been collected.
2. Considering the biases inherent in the data set
The data source we used – a public online discussion group, as described in Section 3, is subject to limitations. Limitations include the bias of volunteers, peer pressure, and priming the opinions of participants by others (these limitations exist also in offline focus groups but are enhanced in virtual medium). Another issue is self-censorship – the participants may say what they think is expected from them. However, if this is true, this underlines our findings, as it suggests that these participants think that the expectations of them are to code by themselves.

3. Generalizability

As an interpretive case study [111] we make no claims to generalize to a population, rather, we wish to generalize to theory, and we have been careful in this paper to integrate our findings with the literature. That said, when seeking triangulation from professional developers we have encountered few additional expressions of prejudice against eager example users, associating them with lack of competence and unprofessionalism.

8. Conclusion

This paper has examined reuse of code examples in a particular community. We explored the following questions.

- How do software developers perceive example usage in their work, and in particular a usage that involves copy and paste? What arguments do they use to justify their position? Which concerns do they consider?
- How can these perceptions be explained?

We would highlight the following key findings from our study:

- Using Copy & Paste in the context of looking for code examples on the web is very different from the use of Copy & Paste in one's own code. However due to the categorization process, some developers do not take the new context (of the Internet) into account as the prejudgment has already fixed. For these developers, writing code is part of their professional identity.

This study makes the following additional contributions: it introduces some new concepts, and extends the software engineering reuse literature. Specifically we discuss 9 aspects that dominate developers' perceptions with respect to example reuse (though these aspects may be found applicable for additional context as well), and discuss how reuse could be enhanced with respect to each of them. This research also makes a methodological contribution by demonstrating how an online community might be studied, and showing how posts were analyzed.

There are a few implications to this research. In a commercial context, revealing implicit prejudice and disarming it may allow developers to leverage further benefit of code reuse, and may improve the collaboration of individuals, teams and organizations. Moreover, prejudice may interfere with achieving organizational goals, or while conducting an organizational change. Having the prejudice lens in mind, one may incorporate methods which were proven effective in addressing prejudice in different context (racism, sexism, nationalism) as part of the software engineering management tools.

Finally, this study may also be considered in the broader context of the changing software engineering landscape. The recent availability of information over the Web, and in our context – availability of source code, is challenging the way software is produced. Some of the main abstractions used in the software domain, namely *development* and *construction*, do not adequately describe the emerging practices involving pragmatic and opportunistic reuse. These practices favor composing over constructing and finding over developing. In this context, prejudice can be perceived as a reaction to change and an act resulting from fear of the new and unknown.

Acknowledgments

The authors wish to thank Ms. Tamar Shavit for her insightful comments on an earlier version of this manuscript. Thanks are also due to the editors and the anonymous reviewers of this paper for their constructive comments.

Appendix A

See Tables 3.1–3.3

Table 3.1

Professional experience of participants (years of experience).

Years of experience	No. of participants
Less than 1 year	5
1–3 years	5
4–6 years	14
7–10 years	15
11–19 years	20
20 years or more	7

Table 3.2

Professional experience of participants (job title).

Position	No. of Participants
Tester	2
Support	2
Assistant/intern	2
Developer/programmer	49
Senior developer/team leader	26
Manager	7
Product manager	1
Architect/analyst	25
Consultant/freelance	17
Founder/Owner	11

Table 3.3

Size of companies in which participants work (by number of employees).

Company size	No. of participants
Self	11
2–10	8
11–50	22
51–200	23
201–1000	16
1001–5000	20
5001–10,000	6
10000+	26

References

- [1] Software Engineering Body of Knowledge. <<http://www.swebok.org/index.htm>>.
- [2] C.L. Aberson, M. Healy, V. Romero, Ingroup bias and self-esteem: a meta-analysis, *Personal. Soc. Psychol. Rev.* 4 (2) (2000) 157–173.
- [3] T.W. Adorno, E. Frenkel-Brunswick, D.J. Levinson, R.N. Sanford, *The Authoritarian Personality*, 1950.
- [4] H. Aldrich, D. Herker, *Boundary spanning roles and organization structure*, *Acad. Manage. Rev.* 2 (2) (1977) 217–230.
- [5] F. Alison, U. Lorna, *Workplace learning in context, Expansive Learning Environments: Integrating Personal and Organisational Development*, Routledge, 2004. Chapter, pp. 126–144.
- [6] G.W. Allport, *The Nature of Prejudice*, Basic Books, 1979.
- [7] B. Altemeyer, *The other authoritarian personality, Advances in Experimental Social Psychology*, vol. 30, Academic Press, 1998, pp. 47–92.
- [8] R.D. Ashmore, F.K. Del Boca, *Conceptual approaches to stereotypes and stereotyping, Cogn. Process. Stereotyping Intergroup Behav.* 1 (1981) 35.
- [9] O. Barzilay, *Example Embedding: On the Diversity of Example Usage in Professional Software Development*, PhD thesis, Tel Aviv University, 2012.
- [10] O. Barzilay, O. Hazzan, A. Yehudai, *Characterizing example embedding as a software activity*, in: SUITE 2009: Proceedings of the 1st international workshop on Search-Driven Development – Users, Infrastructure, Tools and Evaluation at ICSE '09, IEEE Computer Society, 2009, pp. 9–12.
- [11] O. Barzilay, O. Hazzan, A. Yehudai, *Using social media to study the diversity of example usage among professional developers*, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, SIGSOFT/FSE '11, ACM, New York, NY, USA, 2011, pp. 472–475.

- [12] O. Barzilay, C. Treude, A. Zagalsky, Facilitating crowd sourced software engineering via stack overflow, in: S.E. Sim, R.E. Gallardo-Valencia (Eds.), *Finding Source Code on the Web for Remix and Reuse*, Springer, New York, 2013, pp. 289–308.
- [13] O. Barzilay, A. Yehudai, O. Hazzan, Developers attentiveness to example usage, in: *Human Aspects of Software Engineering, HAoSE '10*, ACM, New York, NY, USA, 2010, pp. 1–8.
- [14] K. Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley Professional, 1999.
- [15] K. Beck, C. Andres, *Extreme Programming Explained: Embrace Change, second ed.*, Addison-Wesley Professional, 2004.
- [16] U. Bob, Speed Kills (blog post), February 2009. <<http://blog.objectmentor.com/articles/2009/02/03/speed>>.
- [17] B. Bokowski, Coffeestrainer: statically-checked constraints on the definition and use of types in java, in: *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-7*, Springer-Verlag, London, UK, 1999, pp. 355–374.
- [18] J. Bosch, Keynote address: toward compositional software engineering, in: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10*, ACM, New York, NY, USA, 2010, pp. 1–2.
- [19] d.m. boyd, N.B. Ellison, Social network sites: definition, history, and scholarship, *J. Comput.-Mediated Commun.* 13 (1) (2007) 210–230.
- [20] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, S.R. Klemmer, Two studies of opportunistic programming: interleaving web foraging, learning, and writing code, in: *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI '09*, ACM, New York, NY, USA, 2009, pp. 1589–1598.
- [21] M.B. Brewer, In-group bias in the minimal intergroup situation: a cognitive-motivational analysis, *Psychol. Bull.* 86 (2) (1979) 307–324.
- [22] M.B. Brewer, The social self: on being the same and different at the same time, *Pers. Soc. Psychol. Bull.* 17 (5) (1991) 475–482.
- [23] M.B. Brewer, The psychology of prejudice: ingroup love and outgroup hate?, *J. Soc. Issues* 55 (3) (1999) 429–444.
- [24] N. Carr, Is Google Making Us Stupid?, *The Atlantic* (2008)
- [25] K. Charmaz, *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis (Introducing Qualitative Methods series)*, first ed., Sage Publications Ltd., 2006.
- [26] K.C. Desouza, Barriers to effective use of knowledge management systems in software engineering, *Commun. ACM* 46 (2003) 99–101.
- [27] I. Dey, *Qualitative Data Analysis: A User-Friendly Guide*, Routledge, London, 1993.
- [28] C. Ferrán-Urdaneta, Teams or communities? Organizational structures for knowledge management, in: *Proceedings of the 1999 ACM SIGCPR Conference on Computer Personnel Research, SIGCPR '99*, ACM, New York, NY, USA, 1999, pp. 128–134.
- [29] S.T. Fiske, Stereotyping, prejudice, and discrimination, in: Daniel T. Gilbert, Susan T. Fiske and Gardner Lindzey (Hg.), *Handbook of Social Psychology*, McGraw-Hill, Boston, 1998, S. 357–411.
- [30] T. Fritz, J. Ou, G.C. Murphy, E. Murphy-Hill, A degree-of-knowledge model to capture source code familiarity, in: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, vol. 1, ICSE '10*, ACM, New York, NY, USA, 2010, pp. 385–394.
- [31] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: abstraction and reuse of object-oriented design, in: O. Nierstrasz (Ed), *ECOOP '93 – Object-Oriented Programming, Lecture Notes in Computer Science*, vol. 707, Springer, Berlin/Heidelberg, 1993, pp. 406–431. doi:10.1007/3-540-47910-4_21.
- [32] B.G. Glaser, *Emergence vs Forcing: Basics of Grounded Theory Analysis*, Sociology Press, 1992.
- [33] B.G. Glaser, *The Grounded Theory Perspective III: Theoretical Coding*, Sociology Press, 2005.
- [34] B.G. Glaser, A.L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine Transaction, 1967.
- [35] R.L. Glass, Of open source, linux ... and hype, *IEEE Softw.* 16 (128) (1999) 126–127.
- [36] O. Govorun, B.K. Payne, Ego—depletion and prejudice: separating automatic and controlled components, *Soc. Cogn.* 24 (2) (2006) 111–136.
- [37] B. Hartmann, S. Doorley, S. Klemmer, Hacking, mashing, gluing: understanding opportunistic design, *Pervasive Comput., IEEE* 7 (3) (2008) 46–54.
- [38] T. Helm, Demonisation of the poor is taking place ... horrible things will happen, *The Guardian*, 2012. <<http://www.theguardian.com/politics/2012/nov/17/demonisation-poor>>.
- [39] R. Holmes, R.J. Walker, Supporting the investigation and planning of pragmatic reuse tasks, in: *Proceedings of the 29th International Conference on Software Engineering, ICSE '07*, IEEE Computer Society, Washington, DC, USA, 2007, pp. 447–457.
- [40] W. Humphrey, *PSP(sm): A Self-improvement Process for Software Engineers*, first ed., Addison-Wesley Professional, 2005.
- [41] M. Inzlicht, L. McKay, J. Aronson, Stigma as ego depletion how being the target of prejudice affects self-control, *Psychol. Sci.* 17 (3) (2006) 262–269.
- [42] M.R. Islam, M. Hewstone, Dimensions of contact as predictors of intergroup anxiety, perceived out-group variability, and out-group attitude: an integrative model, *Pers. Soc. Psychol. Bull.* 19 (6) (1993) 700–710.
- [43] D. Jackson, E. Kang, Separation of concerns for dependable software design, in: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER '10*, ACM, New York, NY, USA, 2010, pp. 173–176.
- [44] I. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.
- [45] P.A. James Brian Quinn, S. Finkelstein, Leveraging intellect, *Acad. Manage. Exec.* 10 (3) (1996) 7–27.
- [46] S.L. Jarvenpaa, A. Majchrzak, Knowledge collaboration among professionals protecting national security: role of transactive memories in ego-centered knowledge networks, *Organ. Sci.* 19 (2) (2008) 260–276.
- [47] R. Jeffries, Quality-speed Tradeoff – You're Kidding Yourself, February 2009. <<http://xprogramming.com/blog/quality-speed-tradeoff-youre-kidding-yourself>>.
- [48] R.E. Jeffries, A. Anderson, C. Hendrickson, *Extreme Programming Installed*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.
- [49] J.M. Jones, *Prejudice and Racism*, second ed., McGraw Hill, NY, 1997.
- [50] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, Do code clones matter?, in: *Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society*, 2009, pp. 485–495.
- [51] B. Junnarkar, Leveraging collective intellect by building organizational capabilities, *Expert Syst. Appl.* 13 (1) (1997) 29–40.
- [52] C. Kasper, M.W. Godfrey, “cloning considered harmful” considered harmful, in: *WCSE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, IEEE Computer Society, 2006, pp. 19–28.
- [53] E.A. Karlsson, *Software Reuse: A Holistic Approach*, John Wiley & Sons, Chichester, 1998.
- [54] M. Kim, L. Bergman, T. Lau, D. Notkin, An ethnographic study of copy and paste programming practices in OOP, in: *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, IEEE Computer Society, 2004, pp. 83–92.
- [55] D.E. Knuth, *The art of Computer Programming*, Addison-Wiley, 1968.
- [56] J. Kontio, J. Bragge, L. Lehtola, The focus group method as an empirical tool in software engineering, in: F. Shull, J. Singer, D.I.K. Sjberg (Eds.), *Guide to Advanced Empirical Software Engineering*, Springer, London, 2008, pp. 93–116.
- [57] R. Koschke, Survey of research on software clones, *Dupl. Redund. Simil. Softw.* (2007) 6301.
- [58] L. Krefting, Rigor in qualitative research: the assessment of trustworthiness, *Am. J. Occup. Ther.* 45 (3) (1991) 214–222.
- [59] P. Kruchten, Putting the “engineering” into “software engineering”, in: *Software Engineering Conference, Proceedings*, Australian, 2004, pp. 2–8.
- [60] C.W. Krueger, *Software reuse*, *ACM Comput. Surv.* 24 (June 1992) 131–183.
- [61] R. Land, D. Sundmark, F. Lüders, I. Krasteva, A. Causevic, Reuse with software components – a survey of industrial state of practice, in: *ICSR '09: Proceedings of the 11th International Conference on Software Reuse*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 150–159.
- [62] T.D. LaToza, G. Venolia, R. DeLine, Maintaining mental models: a study of developer work habits, in: *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, ACM, 2006, pp. 492–501.
- [63] M. Lechner, Xp team psychology – an inside view, *PPIG 2008: Psychology of Programming Interest, Group*, 2008.
- [64] A. Lee, J. Liebenau, J.I. DeGross, *Information Systems and Qualitative Research*, Chapman & Hall, 1997.
- [65] J.-H. Lee, Y.-G. Kim, A stage model of organizational knowledge management: a latent content analysis, *Expert Syst. Appl.* 20 (4) (2001) 299–311.
- [66] M.J. Lerner, *The Belief in a Just World: A Fundamental Delusion*, Plenum Press, New York, 1980.
- [67] S. Lewis, Using online communities to drive commercial product development, in: *CHI '08 Extended Abstracts on Human Factors in Computing Systems, CHI EA '08*, ACM, New York, NY, USA, 2008, pp. 2039–2044.
- [68] W. Lim, Effects of reuse on quality, productivity, and economics, *IEEE Softw.* 11 (5) (1994) 23–30.
- [69] P.W. Linville, The heterogeneity of homogeneity, *Attrib. Soc. Interact.: The Legacy of Edward E. Jones* (1998) 423–462.
- [70] P.W. Linville, G.W. Fischer, Exemplar and abstraction models of perceived group variability and stereotypicality, *Soc. Cogn.* 11 (1) (1993) 92–125.
- [71] W. Lippmann, *Public opinion*, Transaction Pub, 1997.
- [72] V. Madey, Gregory; Freeh, R. Tynan, The open source software development phenomenon: An analysis based on social network theory, in: *AMCIS 2002 Proceedings*, 2002.
- [73] D.T. Miller, J.S. Downs, D.A. Prentice, Minimal conditions for the creation of a unit relationship: the social bond between birthmates, *Eur. J. Soc. Psychol.* 28 (3) (1998) 475–481.
- [74] L. Montada, M.J. Lerner, Responses to Victimization and Belief in a Just World, Springer, 1998.
- [75] M. Morisio, M. Ezran, C. Tully, Success and failure factors in software reuse, *IEEE Trans. Softw. Eng.* 28 (4) (2002) 340–357.
- [76] B. Mullen, L.-T. Hu, Perceptions of ingroup and outgroup variability: a meta-analytic integration, *Basic Appl. Soc. Psychol.* 10 (3) (1989) 233–252.
- [77] M. Nanard, J. Nanard, P. Kahn, Pushing reuse in hypermedia design: golden rules, design patterns and constructive templates, in: *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space—Structure in Hypermedia Systems: Links, Objects, Time and*

- Space–Structure in Hypermedia Systems, *HYPERTEXT '98*, ACM, New York, NY, USA, 1998, pp. 11–20.
- [78] C. Ncube, P. Oberndorf, A.W. Kark, Opportunistic software systems development: making systems from what's available, *IEEE Softw.* 25 (6) (2008) 38–41.
- [79] T.D. Nelson, *The Psychology of Prejudice*, Pearson Allyn and Bacon, 2006.
- [80] Z. Obrenovic, D. Gasevic, A. Eliens, Stimulating creativity through opportunistic software development, *IEEE Softw.* 25 (6) (2008) 64–70.
- [81] W.J. Orlikowski, Case tools as organizational change: investigating incremental and radical changes in systems development, *MIS Quart.* 17 (3) (1993) 309–340.
- [82] W.J. Orlikowski, J.J. Baroudi, Studying information technology in organizations: research approaches and assumptions, *Inform. Syst. Res.* 2 (1) (1991) 1–28.
- [83] T.M. Ostrom, S.L. Carpenter, C. Sedikides, F. Li, Differential processing of in-group and out-group information, *J. Pers. Soc. Psychol.* 64 (1993) 21–21.
- [84] B. Park, C.M. Judd, Measures and models of perceived group variability, *J. Pers. Soc. Psychol.* 59 (2) (1990) 173–191.
- [85] T.F. Pettigrew, The ultimate attribution error: extending allport's cognitive analysis of prejudice, *Pers. Soc. Psychol. Bull.* 5 (4) (1979) 461–476.
- [86] S. Plous, The psychology of prejudice, stereotyping, and discrimination: an overview, *Understand. Prejudice Discrim.* (2003) 3–48.
- [87] G. Pour, M. Griss, M. Lutz, The push to make software engineering respectable, *Computer* 33 (5) (2000) 35–43.
- [88] E.S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly & Associates Inc., Sebastopol, CA, USA, 2001.
- [89] D.F. Redmiles, Reducing the variability of programmers' performance through explained examples, in: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, ACM, New York, NY, USA, 1993, pp. 67–73.
- [90] L. Ross, The intuitive psychologist and his shortcomings: distortions in the attribution process, *Adv. Exp. Soc. Psychol.* 10 (1977) 173–220.
- [91] M.A. Rothenberger, K.J. Dooley, U.R. Kulkarni, N. Nada, Strategies for software reuse: a principal component analysis of reuse practices, *IEEE Trans. Software Eng.* 29 (2003) 825–837.
- [92] I. Rus, M. Lindvall, Guest editors' introduction: knowledge management in software engineering, *IEEE Softw.* 19 (2002) 26–38.
- [93] C.B. Seaman, Qualitative methods in empirical studies of software engineering, *Softw. Eng.* 25 (4) (1999) 557–572.
- [94] M.A. Serva, M.A. Fuller, R.C. Mayer, Trust in systems development: a model of management and developer interaction research in progress, in: *Proceedings of the 2000 ACM SIGCPR Conference on Computer Personnel Research*, SIGCPR '00, ACM, New York, NY, USA, 2000, pp. 188–191.
- [95] S. Sharma, V. Sugumaran, B. Rajagopalan, A framework for creating hybrid-open source software communities, *Inform. Syst. J.* 12 (1) (2002) 7–25.
- [96] J. Sidanius, F. Pratto, L. Bobo, Racism, conservatism, affirmative action, and intellectual sophistication: a matter of principled conservatism or group dominance?***, *J. Pers. Soc. Psychol.* 70 (1996) 476–490.
- [97] S.E. Sim, C.L.A. Clarke, R.C. Holt, Archetypal source code searches: a survey of software developers and maintainers, in: *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, IEEE Computer Society, Washington, DC, USA, 1998, page 180.
- [98] S.E. Sim, R.E. Gallardo-Valencia (Eds.), *Finding Source Code on the Web for Remix and Reuse*, Springer, 2013.
- [99] B.J. Skrypnek, M. Snyder, On the self-perpetuating nature of stereotypes about women and men, *J. Exp. Soc. Psychol.* 18 (3) (1982) 277–291.
- [100] O.P.N. Slyngstad, A. Gupta, R. Conradi, P. Mohagheghi, H. Rønneberg, E. Landre, An empirical study of developers views on software reuse in statoil asa, in: *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, ACM, New York, NY, USA, 2006, pp. 242–251.
- [101] J.R. Speed, What do you mean I can't call myself a software engineer?, *IEEE Softw.* 16 (1999) 45–50.
- [102] A. Strauss, J.M. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, second ed., SAGE Publications, 1990.
- [103] A.L. Strauss, *Qualitative Analysis for Social Scientists*, Cambridge University Press, 1987.
- [104] M. Sulayman, C. Urquhart, E. Mendes, S. Seidel, Software process improvement success factors for small and medium web companies: a qualitative study, *Inf. Softw. Technol.* 54 (5) (2012) 479–500.
- [105] H. Tajfel, Experiments in intergroup discrimination, *Sci. Am.* 223 (5) (1970) 96–102.
- [106] H. Tajfel, *Human Groups and Social Categories: Studies in Social Psychology*, CUP Archive, 1981.
- [107] H. Tajfel, J.C. Turner, *The social identity theory of intergroup behavior*, 2004.
- [108] J. Tomayko, O. Hazzan, *Human Aspects of Software Engineering*, Charles River Media, 2004.
- [109] C. Urquhart, *Grounded Theory for Qualitative Research: A Practical Guide*, Sage, 2012.
- [110] C. Urquhart, H. Lehmann, M.D. Myers, Putting the 'theory' back into grounded theory: guidelines for grounded theory studies in information systems, *Inform. Syst. J.* 20 (4) (2010) 357–381.
- [111] G. Walsham, Interpretive case studies in is research: nature and method, *Eur. J. Inform. Syst.* 4 (2) (1995) 74–81.
- [112] G. Weinberg, Egoless programming, *IEEE Softw.* 16 (1) (1999) 118–120.
- [113] G.M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold Co., New York, NY, USA, 1988.
- [114] E. Wenger, *Communities of Practice. Learning, Meaning, and Identity*, Cambridge University Press, 1998.
- [115] L. Williams, R. Kessler, *Pair Programming Illuminated*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [116] M. Wolf, Proust and the Squid: The Story and Science of the Reading Brain, HarperCollins, NY, 2007.