Example Embedding:
On the Diversity of Example Usage in Professional Software Development

Thesis submitted for the degree of Doctor of Philosophy

By

Ohad Barzilay

This work was carried out under the supervision of

Prof. Amiram Yehudai and Prof. Orit Hazzan

Submitted to the Senate of Tel Aviv University

April 2011
To my parents
without whom this would not have been possible

To my wife and daughters
without whom this would have been possible much sooner
ACKNOWLEDGEMENTS

I wish to thank my advisors Amiram Yehudai and Orit Hazzan. Supervising a doctoral research is, in the spirit of this thesis, diverse. I am grateful for their kind of guidance. In addition to being academic advisors, Orit and Amiram became my role models for parenting. They encouraged me to take my own path and to experiment. They guided me through the process of learning from my mistakes and taking responsibility, and, most importantly, they taught me how to be independent, grow and develop my research. Their absolute support along the course of this research is not taken for granted. With the help of Amiram and Orit I was able to find my own voice in this study. For that I am indebted.

The observation phase of this study was made possible thanks to the assistance of several people who trusted me and welcomed me into their offices. In particular, I wish to thank Yehuda Afek and Hilik Stein, and their amazing team at Cisco. Yosi Harel, Michal Geva and Sigal Duek, and their wonderful team at Sun. Amir Kirsh, Gaby Kozakov, Pini Faran and the great staff at Comverse. The honesty, openness and cooperation of the software engineers, as well as the managers, of these three companies contributed significantly to this research and its outcomes.

I also wish to thank the reviewers of this thesis for their constructive and helpful comments, and to David Raveh for editing the final English version of this dissertation.

Finally, I want to thank my fellow graduate students, office and hallway neighbors, and my colleagues in teaching and in research for their help, advice and friendship.

Thank you.
ABSTRACT

In this paper we investigate example usage among professional software developers. By ‘examples’ we refer mostly to code examples – already existing code snippets that are used in a new context. Our definition for example is broad: existing code can be used in various scenarios and for various purposes including: learning purposes, code comprehension, feature implementation, solving problems and many others. Indeed, many benefits reside in systematic and habitual example usage both for software companies as well as for individuals. These benefits include increased productivity, improvement of code quality, consistency of both design and coding standards’ enforcement, and the establishment of an effective knowledge transfer mechanism both within and outside the organization.

Though these benefits may be appealing, we find large diversity in example usage ‘in the wild’. There is no consensus among developers regarding its benefits, the software development community does not address the barriers and challenges that accompany extensive example usage, and there is no explicit, widely accepted set of best practices for using examples either methodologically or systematically.

This paper focuses on the diversity of example usage.

This empirically driven software engineering research employed qualitative research methodology to build a field-grounded theory. The research was built bottom-up: we started from fine-grained activities observed in the field and, following the course of our study, the diversity became apparent. We used various tools for data gathering that included field observations, interviews, surveys, reflective questionnaires, focus groups and virtual focus groups. Each of these tools highlighted different aspects of example usage diversity.

We find that the developers’ approach to example usage is dominated by human rather than engineering factors. In addition, we find that developers are not aware of the existence of code examples. They are not aware of their methodical use, or are inattentive to their use in various contexts. In general, the software development community perceives example usage merely as a programming technique rather than acknowledging it as a fundamental software activity and as an expression of the software reuse principle.

We conclude by proposing that there should be a comprehensive approach towards example usage with respect to the software development ecosystem. We name this methodological example usage: Example Embedding.
CONTENTS

CHAPTER 1  INTRODUCTION ................................................................. 15
  1.1  Background ............................................................................. 16
  1.2  Motivation .............................................................................. 17
  1.3  Example Embedding ............................................................... 18
  1.4  Contributions ......................................................................... 20
  1.5  Thesis Outline ......................................................................... 21

CHAPTER 2  RELATED WORKS ............................................................... 25
  2.1  Developer Behavior Studies .................................................... 26
  2.2  Developer Tools ....................................................................... 26
  2.3  Code Search ........................................................................... 28
  2.4  Copy & Paste .......................................................................... 28
  2.5  Reuse .................................................................................... 29
  2.6  Pragmatic Reuse .................................................................... 30

CHAPTER 3  METHODOLOGY ................................................................. 33
  3.1  Empirical Software Engineering Research ............................. 34
  3.2  Qualitative Research ............................................................... 34
  3.3  Grounded Theory ................................................................... 35
  3.4  Research Course ..................................................................... 36
  3.5  Research Tools ........................................................................ 40
  3.6  Qualitative Analysis ............................................................... 48
CHAPTER 4  HUMAN ASPECTS OF EXAMPLE USAGE ..........................51

4.1  Introduction ............................................................................................................. 52
4.2  Methodology ............................................................................................................ 54
4.3  Vertical Data Analysis .............................................................................................. 58
4.4  Horizontal Data Analysis ......................................................................................... 69
4.5  Addressing Human Factors in StackOverflw.com .................................................. 80
4.6  Implications ............................................................................................................. 81
4.7  Summary .................................................................................................................. 82

CHAPTER 5  DEVELOPERS ATTENTIVENESS TO EXAMPLE USAGE ...83

5.1  Introduction ............................................................................................................. 84
5.2  Example Driven Development Observed ................................................................ 84
5.3  Attentiveness ........................................................................................................... 87
5.4  Focus Group .......................................................................................................... 93
5.5  Summary ................................................................................................................. 97

CHAPTER 6  WHY PROFESSIONAL DEVELOPERS USE EXAMPLES? ...99

6.1  Introduction ............................................................................................................. 100
6.2  Methodology ........................................................................................................... 101
6.3  Example Usage Purposes ....................................................................................... 104
6.4  Example Usage vs. Code Search ............................................................................ 112
6.5  Using Examples for Multiple Purposes ................................................................. 113
6.6  Implications ............................................................................................................ 118
6.7  Summary ................................................................................................................. 118
CHAPTER 7  EXAMPLE EMBEDDING ECOSYSTEM ...................... 119

7.1  Introduction ........................................................................................................................ 120

7.2  Software Activity ................................................................................................................... 121

7.3  Example Embedding Software Activity ............................................................................... 122

7.4  Ecosystem ............................................................................................................................ 131

7.5  The Case of Referring Previous Academic Work................................................................. 133

7.6  Summary And Future Work ................................................................................................. 134

CHAPTER 8  SUMMARY ............................................................................................................. 135

REFERENCES ............................................................................................................................ 137
LIST OF FIGURES

Figure 3.1: Research Course........................................................................................................... 37
Figure 3.2: Observation report taken during phase I........................................................................ 42
Figure 3.3: Screen capture of the LinkedIn discussion group.......................................................... 45
Figure 3.4: Screen capture of the online survey............................................................................... 46
Figure 3.5: The reflection questionnaire given to the case study subjects....................................... 47
Figure 3.6: A part of a preliminary categorization reverse lookup table that was circulated among the researchers ............................................................................................................. 50
Figure 4.1: A follow up post in "NET People" LinkedIn discussion group .................................... 52
Figure 4.2: Sizes of companies in which participants work............................................................. 56
Figure 4.3: Professional experience of participants (job title)........................................................ 57
Figure 4.4: Professional experience of participants (in years)....................................................... 57
Figure 4.5: Dividing participants into groups according to their approach to example usage and to their job title. Participants who work in multiple roles appear multiple times........ 58
Figure 4.6: Dividing participants into groups according to their approach to example usage and to their professional experience period. ................................................................. 59
Figure 4.7: Dividing participants into groups according to their approach to example usage and to their company size. Participants who work in multiple companies of different size appear multiple times.................................................................................................................. 60
Figure 5.1: High-level programming activities observed in Stylos and Myers study................. 91
Figure 6.1: Years of experience as developer.................................................................................. 102
Figure 6.2: Example of participant’s answer................................................................................... 104
Figure 6.3: Number of the participants, who considered multiple items of each category mentioned in Section 6.3........................................................................................................ 114
Figure 6.4: Distribution of participants by the number of task properties they mention. Categories are taken from Table 6.2.......................................................................................... 115
Figure 6.5: Distribution of participants by the number of development activities they mention. Categories are taken from Table 6.3.................................................................................. 116
Figure 7.1: Test driven development practice................................................................................. 124
Figure 7.2: Refactoring driven development practice...................................................................... 125
Figure 7.3: Software activity ecosystem........................................................................................ 132
“... as long as there were no machines, programming was no problem at all”

(The Humble Programmer, Edsger W. Dijkstra)

Chapter 1

INTRODUCTION

We find that example usage in professional software development is diverse. Developers do not consciously appreciate example usage benefits, the software development community does not address the barriers and challenges that accompany extensive example usage, and there is no explicit, widely accepted set of best practices for using examples methodically or systematically. In what follows we describe our empirically driven software engineering research that led us to these findings.
1.1 Background

Many developers use examples in their work. This fact should not come as a surprise to the reader. Example usage is a common practice for knowledge workers, as well as for others, and assists in learning and in performing various tasks. A good example can make an abstract idea clear by providing relevant context that helps the reader to understand and internalize the principles on which the example was based [34]. Moreover, examples also serve as templates [122]: when reading a good example it is easy to extract the repetitive structure from a specific context, and to reuse the repetitive part for new tasks [112][62].

Examples play an important role in teaching and learning programming. Students and teachers alike cite examples as the most helpful resource for learning to program [92], and programming by example systems [40] [136] [100] create generalized programs from examples provided by the user.

In software construction, examples could be enhanced even further. The source code of a software system is intangible [50]. Once created, it can be duplicated without any additional cost (apart from copyright issues). This means that the source code included in an example can be used not only for learning purposes, but also as a starting point for implementing a new functionality.

In this thesis we investigate example usage among professional software developers. By ‘examples’ we refer mostly to code examples – already existing code snippets that are used in a new context. Our definition for example is broad; examples can be used in various scenarios and for various purposes including: learning purposes, code comprehension, feature implementation, and problem solving. Indeed, many benefits reside in systematic and habitual example usage both for software companies as well as for individuals. These benefits include: enhancing software reuse [72][80], increased productivity [121], improvement of code quality [101], enforcement of design consistency [55][109] and of coding standards [22], and the establishment of an effective knowledge transfer mechanism both within and outside the organization [126].

Despite these benefits we find that example usage is diverse. There is no consensus among developers regarding its benefits, the software development community does not address the barriers and challenges that accompany extensive example usage, and there is no explicit, widely accepted set of best practices for using examples either methodically or systematically.

This paper focuses on the diversity of example usage.
1.2 Motivation

Software engineering research is all about better software development. There are many approaches to address this issue; however, we are focusing on the practices that make individuals and organizations more effective. Our research is empirically driven: we started by conducting field observations – we watched professional software engineers, employed by three global software companies at work, hoping to discover new things, new activities, of which we were not previously aware. This hope stems from two reasons: the Web revolution and the already acknowledged activity of refactoring, as we explain next.

1.2.1 The Web Revolution

The Web has revolutionized many aspects of our lives: the dating scene, and the way in which we consume news or listen to music are but a few examples of the sea change that the Web had brought into our daily lives. But, what about actual software development? Has the Web revolutionized it as well? Does the shoemaker's son always go barefoot? Recent conferences have addressed this issue, e.g. [5].

The immediate suspect for Web influence on software development was free code online. The open source community, as well as numerous technical blogs and community Websites, put a vast amount of free source code online, ranging from mere snippets to full-blown products. This code embodies domain knowledge of the software development community. Like the Internet, this knowledge is derived from sources of variable reliability, and is thus inherently chaotic, incomplete, and inconsistent.

We did not, however, embark on this research with online free code in mind. Instead, we conducted field observations in software companies for several weeks in the hope of identifying innovative patterns and behaviors upon which to focus our research in later stages, and ended up investigating example usage. Ironically, although online code is one form of example, it was the usage of examples locally, within the team, that ultimately motivated our research path.

1.2.2 Refactoring

Another motivation for our research path was the already familiar activity of refactoring. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [49]. Although refactoring code has been performed informally for years, William Opdyke's 1993 Ph.D. dissertation [114] is the first known resource to specifically examine refactoring.
Identifying refactoring as an activity and naming it promoted the following important processes: Firstly, it laid the foundations for others to build a catalogue describing different examples of such refactorings and discuss their subtleties [49], some of which are also available online [48]. Secondly, it enabled the development of software tools that would systematically apply various refactorings on existing code and ensure their correctness. Thirdly, it influenced the way in which programmers write code, using, for example, test-driven development techniques [17]. And fourthly, it affected the software construction lifecycle by allowing the design phase to be incorporated into the coding phase, as manifested in agile methodologies [19].

These advances would not have been possible had the refactoring activity not been extracted from the various activities that constitute the coding phase. We started this research hoping to identify another such activity, which is not currently well appreciated as a standalone technique, and thus establish the foundation for further advancements.

1.3 Example Embedding

We define Example Embedding (EE) as using already existing code in a new context. Why do we define a new term, rather than just say ‘using examples’? The reason stems, again, from the refactoring case. We argue that EE, like refactoring, is more than merely a programming technique. EE is a fundamental software construction activity and is an expression of both community knowledge accumulation and the software reuse principle. Habitual and methodical example usage is an expression of awareness of the existing body of knowledge and allows the developer to stand on the shoulders of giants.

For the same reason that ‘refactoring’ benefits from distinguishing it from ‘just rewriting’, so does ‘example embedding’ benefit from distinguishing it from ‘just using examples’. Moreover, refactoring and EE are things that developers do implicitly and inevitably. However, doing them in a non-methodical manner could introduce bugs. The conceptualization of refactoring, and, hopefully, of example embedding, provide the software engineering vocabulary with new abstractions, and promote the formulation of new process practices and tools.

Gabel and Su [52] discuss the possibility of a “singularity” in software engineering’s future: a point of convergence at which all necessary software will have been produced. While they believe that this question is of intrinsic academic interest, there are several practical applications and consequences including Automation of Programming, Development Tool Research and Code Reuse.

---

Consider what a programming course would be like after singularity has been achieved. What if we reached a point at which not all possible software had been produced, but only 50% of it, what would be our strategy then?

Java guru Joshua Bloch describes the centrality of examples in software development and the importance of writing good examples: “Code lives on as examples... [API] examples tend to get emulated heavily. If you get them right you’ve seeded the market with good uses of your API. If you get them wrong, conversely, you’ve ensured that there will be broken programs floating around the Web for years. Example programs should be exemplary. [...] My rule of thumb is you should spend 10 times as much time on example code as you do production code”.

However, not only is writing examples a complicated task, but also using them. Example usage is a composite task, involving multiple activities and skills – finding the suitable example or multiple examples, reviewing them, understanding them, testing, refactoring and changing them according to specific needs, etc. This complex process deserves a name of its own in order to raise the developers’ attentiveness to the various building blocks.

1.3.1 REUSE
EE is about using already existing code in a new context. This is also the essence of software reuse. However, as opposed to most of the work done in the area of software reuse, we do not limit the discussion to (re)using components that were written in order to be reused. Example code is not necessarily packaged nor used as a black box. We argue that reuse should not be limited to the use of code as is, but rather should be applied more freely, as long as the resulting unit is tested thoroughly. The difference between our approach to reuse and the conventional black box reuse is analogous to the difference between refactoring and thorough design. Specifically, as refactoring liberated us from the need to go through an elaborate design phase, EE aims at liberating us from the belief that reuse is limited to using the constructs offered by programming language such as functions, objects, and inheritance mechanisms, code reuse could also be performed more freely, explicitly by the developer.

1.3.2 PATTERNS
One of the alternative approaches for some EE motivations is the use of patterns [56]. A pattern, as opposed to an example, is a skeleton, a template, of a required behavior or functionality. It is more abstract, more general, than a specific use case, and in order to use a pattern one needs to instantiate it with the specific

http://www.youtube.com/watch?v=aAb7hSCtvGw
context. In contrast, an example is already an instance of something. An example is not *more general* than the required functionality; it is merely *different*. In order to use it in a new context it must be cloned and then rewritten.

An interesting analogy would be comparing the Java programming language with JavaScript. The inheritance mechanisms in the two languages are different. Java has a class-based (‘classical’) inheritance mechanism; JavaScript’s is based on cloning prototypes (‘prototypical’). When a new functionality is required in Java, the programmer has to extend the corresponding class (the template, the pattern) and then instantiate it, whereas in JavaScript the developer just clones a similar existing object and adapts it to his or her needs. The existing objects serve both as units of functionality and as prototypes for future extension.

One might ask which approach is better – examples or patterns? One argument in favor of examples would be that they do not require the programmer to abstract the task at hand [142].

1.4 **Contributions**

Example usage in software construction has been studied before. The literature describes work done in the area of developing tools for enhancing example-centric development, code search and recommendation systems [4].

The contribution of the research described in this dissertation stems from its perspective and settings. We study example usage ‘in the wild’ – our research subjects are professional developers working in the software industry. This fact provides us with the opportunity to portray the *diversity* in example usage among these developers in various dimensions, and by doing so to highlight the strengths and pitfalls of the example-centric approach.

Moreover, many of the previous studies investigating developers’ behavior (e.g. [35] [36] [37] [153]) have used controlled experiments and have relied on analyses that were based on heavily abstracted characterizations of developer behavior. Although this strategy allowed the investigators to study a larger number of subjects, potentially increasing confidence in the results, it also limited the scope of the results. By contrast, our observation phase took place at the development sites of large global software companies, observing professional developers, performing actual development tasks. We also used additional data gathering tools, such as surveys and focus groups, in order to compensate for the limited number of research subjects being observed.
Specifically, this dissertation offers contributions in four areas:

1. **Human factors of example usage** – We identify nine human factors that dominate developers’ approach to example usage: conforming to organizational goals, personal development, acknowledging example dexterity, ego, community identity, ownership, trust, role definition and analytical skills.

2. **Developers’ attentiveness to example usage** - We identify three types of inattentiveness that causes professional developers to use examples in certain contexts but not in others: inattentiveness in using examples involving different technologies, inattentiveness to examples of different scales, and inattentiveness to the variety of purposes examples may serve.

3. **Developers’ motivations for using examples** – We identify fifteen motivations for example use by professional developers, and divide them into three categories – motivations that are associated with 1) task specific property 2) development mode or activity, and 3) the software engineering aspect.

4. **A Comprehensive approach to example usage** – In order to address developers’ diversity with respect to example usage we propose to take a comprehensive approach that would conceptualize EE and take into account the multiple aspects of the software engineering ecosystem.

**1.5 Thesis Outline**

In chapter 2 we review related work. Investigating example usage among professional developers has much in common with other research fields in the area of software engineering research. Example-centric development was investigated by the code searching tools community, and is closely related to the opportunistic software system development approach. Substantial work has been done on the areas of code search and code clones, both involving extensive example usage. Research on software reuse, although often focusing on component reuse, is also relevant to the EE case.

In chapter 3 we describe the methodology used for this research. This empirically driven software engineering research employed qualitative research methodology to build a field-grounded theory. The research was built bottom-up: we started from fine-grained activities observed in the field and, following the course of our study, the diversity property became apparent. During the research we used various tools for data gathering including: field observations, interviews,
surveys, reflection questionnaires, focus groups and virtual focus groups; each of these tools provides additional evidence of this diversity. In this chapter we elaborate on the research methodology, the research course and theory establishment, and of each of the research tools.

In chapters 4-6 we examine developers’ diversity with respect to example usage; in each of them it is investigated from different perspective. We show that developers’ diversity is affected by human factors, it is manifested in the differences of attentiveness of developers, and it is evident in the variance of developers’ motivations for using examples.

In Chapter 4 we present a virtual focus group case study that examines .NET developers' perception regarding example usage. We show that in addition to the developers who support extensive and habitual example usage (35%) there are other developers who oppose example usage altogether (14%) and a majority (51%) that support limited example usage only. These third group developers restrict their example usage to learning purposes only, avoid using example code, and insist on writing it themselves.

Furthermore, we suggest that despite the engineering challenges involved in extensive example usage, the diversity in developers’ approach is caused by human, rather than engineering, factors. The developers’ approach to example usage is affected by their sense of professional and community identity, ego considerations, ownership and trust issues.

Some of the results described in Chapter 4 also appear in the paper:


In Chapter 5 we show how developers’ diversity with respect to example usage is manifested in their awareness of and attentiveness to example usage. We observed professional developers at work, and identify three types of inattentiveness that cause professional developers to use examples in certain contexts but not in others: inattentiveness to examples’ use involving different technologies, inattentiveness to examples of various scales, and inattentiveness to the variety of purposes examples may serve.

We also present a focus group case study in which we raised the awareness level of a group of developers to the potential benefit of the habitual and systematic use
of examples, and to the various purposes that examples may serve. We conducted a guided reflection process both immediately after a session on example usage and after 3 months, and found that merely discussing example usage influenced some of the participants to consider using examples more frequently than they did before.

**Chapter 5 is largely based on the paper:**


In Chapter 6 we study the primary motivations and the centrality of example usage in professional programming. In a Web survey we conducted, we obtained 192 responses from professional developers, who were asked about their example usage purposes. The four most frequent answers were: a) learning purposes, b) implementation purposes, c) when working with unfamiliar technology or library, and d) for specific tasks. We identified three axes that affect example usage motivation. Examples are used based on the nature of the task (unfamiliar, specific, complex), the development activity (learning, implementation, solving problems, code comprehension) and the motivation of the developer to address software engineering aspects (speed, reuse).

Furthermore, we counted the number of motivations mentioned by each participant. Developers who associate example usage with a single motivation may only be using examples in this context (e.g. only for unfamiliar technologies, or only for code comprehension). We argue that example usage purposes are diverse; for each axis, most of the participants relate the usage of the example with, at most, a single motivation. We also discuss the limitations of this analysis (one may argue that the diversity is not in the motivations themselves but rather in the explicitness of example usage for these tasks).

Furthermore we compare our findings to previous results by Sim et al. [132] of archetypal source code searches and draw potential implications of our work on tool builders for code search.

**Chapter 6 is largely based on the paper:**

*Why professional developers use examples*, Ohad Barzilay, Orit Hazzan, and Amiram Yehudai (in review), 2011

In Chapter 7 we propose that there should be a comprehensive approach to example usage to address its diversity. We coin a new term – *example embedding*
- to highlight the methodological and disciplined use of examples in software construction. We suggest that EE is a fundamental software engineering activity (and not merely a programming technique), and consider a comprehensive approach that would leverage its full potential. We examine two case studies, one of them being the advances in software engineering due to the conceptualization of the refactoring activity, and the other being the mechanisms used by the academic community in order to standardize the citations of previous work in academic publications.

The approach described in this chapter is motivated by the empirical research reported in this dissertation so far. However, as opposed to the previous chapters, this one is more argumentative in nature, and portrays different paths for future work.

Chapter 7 is largely based on the paper:


*And on the Essay:*


In Chapter 8 we summarize the thesis.
"Setting an example is not the main means of influencing another, it is the only means."

A. Einstein

Chapter 2

Related Works

Investigating example usage among professional developers has much in common with other research fields in the area of software engineering research. Example-centric development was investigated by the code searching tools’ community, and is closely related to the opportunistic software system development approach. Substantial work has been done on the areas of code search and code clones, both of which are involved in extensive example usage. Research on software reuse, although often focusing on component reuse, is also relevant to the sample embedding case. In what follows, we summarize the main contributions in each of these research areas, with respect to the research presented in this dissertation.
In the following, we situate our work with respect to software engineering research.

2.1 Developer Behavior Studies
Investigating software developers' behavior in general, and with respect to example usage in particular is not new. Many of the previous studies, investigating developers' behavior (e.g. [35] [36] [37] [153]) have used controlled experiments, and have relied on analyses that were based on heavily abstracted characterizations of developer behavior. Although this strategy allowed the investigators to study a greater number of subjects, potentially increasing confidence in the results, it also limited the scope of the said results.

Exploratory studies, on the other hand, allow the researcher to identify new behaviors. In the area of software engineering research, such studies, performed in lab settings, are typically carried out in order to explore how developers perform maintenance tasks, and understand code (e.g. [87] [94] [124]).

In contrast, our observation phase took place at the development sites of large global software companies, while observing professional developers doing actual development tasks. We also used additional data gathering tools, such as surveys and focus groups, in order to compensate for the limited number of research subjects being observed.

2.2 Developer Tools
Some of the empirical studies in software engineering resulted in the development of a software tool that would address the aspects revealed in the investigation. Two families of such tools, namely recommendation systems [4] and code search engines [3][8], rely heavily on providing the developer with code examples [9]. The Strathcona system [63], for example, finds and recommends source code examples based on an IDE's current context.

In our research we focus on examples using the "Google paradigm" [74], meaning with no other supportive tool besides the plain vanilla Google Web search engine. Although the developer would surely benefit from having a designated software tool to find relevant code examples, the use of such a tool should be incorporated in a wider software development context. For instance, such a tool should fit into the development process and practices. The findings of this research could be used to design such a tool that would be part of an EE ecosystem.

The following work was accompanied by empirical research both to motivate the creation of the tool, as well as to empirically evaluate its use.
Edwards [44] provides an IDE that illuminates code with examples. In order to implement a new functionality, such as a method, the programmer writes some choice examples that invoke the method, and an automated tool, called EG, writes the code of the required method interactively together with the programmer. Edwards lists some interesting implications of this approach, such as example-centric debugging and testing, teaching with examples and example driven development.

In [68], Hummel et al. use a slightly different approach. They present a tool that automatically finds and presents suitable reusable software components to developers, where the programmer specifies the functionality needed using an interface-like syntax.

Brandt et al. propose [25] that embedding a task-specific search engine in the development environment can significantly reduce the cost of finding information and thus allow programmers to write better code more easily. They developed Blueprint, a Web search interface integrated into the Adobe Flex Builder development environment that helps users locate example code.

Stylos and Myers [140] observed programmers using Web search in order to learn software libraries and have revealed problems and inefficiencies in their use. They identified six high-level programming activities involved in API learning – two of which involved finding examples and integrating them. Based on these observations they developed a prototype search tool, called Mica, which, on top of the Google search engine, helps programmers find examples when they already know which methods to use. Our field observation described in Section 0 supports Stylos and Myers model of high-level programming activities.

Redmiles proposes [121] an example-based design process in which the EXPLAINER tool is one component. Examples residing in a catalog repository must be retrieved by users. Retrieval requires users to articulate some specification of requirements for the current task in order that an appropriate example be delivered. Once an example is retrieved, users must be supported in the potential modification of the example to adapt it to their new task. Working with an example can lead to additional ideas about the current task, possibly leading to the refinement of the original requirements specification and the retrieval of additional examples. In empirical evaluation, Redmiles claims to reduce the variability of programmers’ performance through the methodological use of the example-based design process and tool.
2.3 Code Search
Code search is one of the activities involved in example usage. Several previous
research works have proposed a classification of source code search methods.

Sim et al. [132] conducted a survey to generate archetypes of source code
searching used by programmers during maintenance tasks. Respondents were
asked about their source code searching habits: what tools they used, why they
searched, and what they searched for. The four most common search targets
were: function definitions, different uses of a function, variable definitions, and
different uses of a variable. The most common search motivations were defect
repair, code reuse, program comprehension, feature addition, and impact
analysis. Umarji and Lopes [146] categorized these results along two orthogonal
dimensions: motivation (reuse vs. reference example) and size of search target. In
Section 6.4 we discuss [132] with respect to example search.

Janjic et al. [74] portray code search as a spectrum that ranges from somewhat
speculative searches to fully specified definitive searches, such as those that are
described by Hummel [67]. They provide a list of twelve archetype usage
scenarios for software search engines and position them against the life cycle of
the software.

Tool builders for code search (see e.g. [68] for an overview) often study
developers’ behaviors and motivations in order to build them the right tool. This
thesis also contributes to this effort by providing a context for such searches –EE
activities.

2.4 Copy & Paste
Using examples sometimes involved code cloning (copy and paste), and research
work done in this area may also, in some cases, be relevant for learning example
embedding.

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 [79], 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 [71].

Kim et al. [84] 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. As in our research, data gathering was performed by closely observing programmers as they programmed and by taking notes during the observation. In our framework, however, instead of investigating the phenomena observed, we sought to interpret the data at higher levels of abstraction, namely during an EE software activity [11]. Though C&P sometimes implies EE, using example abstraction enables the researcher to appreciate additional implications as well: organizational, cognitive, and practice- and process-oriented.

LaToza et al. [95] also addresses C&P example code by presenting both qualitative data from interviews and quantitative data from two surveys. Their research motivation was different from ours; they wished to know specifically for what types of activities developers used development environments. They also wanted to know whether developers chose different tools for different activities. 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.

2.5 REUSE

EE is about using an already existing code in a new context. This is also the essence of software reuse [90][72]. Although the majority of the work done in the area of software reuse deals with (re)using components that were written in order to be reused, most of their results are also relevant to the EE case.

Lim reports in [101] about 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. [108] aim at identifying some of the key factors in adopting or running a company-wide software reuse program. They analyzed twenty-four projects of European companies using structured interviews. Three main causes of failure were: (a) not introducing reuse-specific processes, (b) not modifying non-reuse processes, and (c) not considering human factors. The root cause was a lack of commitment by top management, or non-awareness of the importance of those factors, often coupled with the belief that using the object-oriented approach or setting up a repository seamlessly is sufficient to achieve success in reuse. Conversely, successes were achieved when, given a potential for reuse because of
commonality among applications, management committed to introducing reuse processes, modifying non-reuse processes, and addressing human factors.

Slyngstad et al. [135] describe the results of a survey followed by semi-structured interviews in order to characterize 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. Component understanding was generally sufficient, but documentation could be improved. Finally, quality attribute specifications were trusted for the applications using reusable components in new development, but not for the reusable components themselves. These results conform to our findings: in Section 6.3.3 we discuss similar developers’ views, and in Section 4.4 we discuss trust issues and time considerations.

Land et al. [93] investigate reuse of software components. 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. Although component reuse does occur, they report that their findings are still partly disappointing, since many potential benefits are currently not exploited. Although in our research we examine example usage rather than component reuse, we find the two have a lot in common. Any serious attempt at software reuse must permeate the organization and allow existing processes and practices to be modified [80].

Rotenberger et al. [125] 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.

2.6 Pragmatic Reuse

In contrast to the reuse literature discussed above, using code examples is much more pragmatic and undemanding. While pragmatic reuse tasks (also called code scavenging) have been shown to be effective [90], little follow-up work has been done. One such work by Holmes and Walker [64] reviews the recent work in this
area and describes a lightweight tool that supports the investigation and planning of pragmatic reuse tasks. They also conducted a survey among twelve industrial developers to evaluate pragmatic reuse.

One prominent example for an example extensive software development method is the Opportunistic Software Systems Development (OSSD) [110]. It is an approach in which developers meld together software pieces that they have found. Most often they find unrelated software components and systems that weren’t 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. 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 [110].

Obrenović et al. report in [113] 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 [60] 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.

Two studies of opportunistic programming are described in [26], 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. Traits that suggest the Web is being used for learning and reminding were also observed by analyzing a month of queries to an online programming portal.

In our study we did not find evidence of opportunistic development. Furthermore, the diversity in example usage we found raises several concerns that are not currently addressed by the OSSD community. In Chapter 7 we propose that there should be a comprehensive approach to address these concerns.
“...methods are more important than facts”

(Are Toy Problems Useful, D. Knuth)

Chapter 3

**Methodology**

This empirically driven software engineering research employed qualitative research methodology to build a field-grounded theory. The research was built bottom-up: we started from fine-grained activities observed in the field and following the course of our study, the diversity of example usage became apparent. During the research we used various tools for data gathering including: field observations, interviews, surveys, reflection questionnaires, focus groups and virtual focus groups. Each of these tools provides additional evidence for this diversity. In the following we elaborate on the research methodology, the research course and theory establishment, and of each of the research tools.
3.1 **Empirical Software Engineering Research**

Software engineering is concerned with developing large applications; it addresses the development, operation and maintenance of software in a systematic and controlled manner according to clearly defined and documented methodology, plans, and procedures [2]. *Empirical software engineering research* [21] aspires to establish software engineering as an engineering field based on scientific knowledge, and is based on both quantitative and qualitative paradigms for data gathering and data analysis [98][66][128]. Empirical research can be used to examine various aspects of software engineering, including development practices, processes and tools. Empirical methods are gaining increasing popularity in software engineering research; they can be used to substantiate existing theories, models or tools, or as a means to motivate further research [134][85][131].

Reports on the use of empirical methods in software engineering research have existed since the 1970s, mainly in the context of evaluating technologies in software engineering, such as integrated development environments, in conjunction with the software industry [15][21].

3.2 **Qualitative Research**

Among the various methods involved in empirical software engineering research, we focus in this research on the qualitative research paradigm [41], which is defined as an inquiry process of understanding a social or human problem. The method is based on building a complex, holistic, verbal picture that is conducted in a natural setting and reports the detailed views and attitudes of informants [38]. It often aims at providing a grounded theory in which theories emerge bottom-up from the data rather than top-down from existing theory [32].

In [117] Perry *et al.* describe software engineering field research aimed at higher granularity than the perspective taken by us. They logged thirteen developers on a large software project every hour for a year to quantify which of 13 activities they were engaged in. The categories distinguished different life cycle activities such as estimation, requirements, high level & low level design, test planning, coding, inspections, and high level & low level testing. This PhD research, on the other hand, focuses on finer grained activities that are currently assimilated within some larger activities.

Singer *et al.* present in [133] work practice data of the daily activities of software engineers gathered from observing an individual software engineer. They discuss the advantages of considering work practices in designing tools for software engineers, and include some requirements for a tool they developed based on the
results of their studies. Our approach, on the other hand, is not tools’ oriented, but rather holistic in nature. Although we see great importance for tool support, we also value other components of the software engineering ecosystem. Our approach, therefore, is driven by activity characterization, one of whose implications is the recommendation of a proper tool that would address a more abstract notion rather than the observed phenomena.

Seaman [128] describes qualitative methods in empirical studies of software engineering. She presents several qualitative methods for data collection and analysis and describes them in terms of how they might be incorporated into empirical studies of software engineering and, in particular, how they might be combined with quantitative methods. In our research we followed much of Seaman's advice, with some adjustments. The focus of the case studies described in [128] was to identify and characterize the software development processes customarily used in software organizations, such as [115].

Robillard et al. describe in [124] a study of five developers performing a change task on a medium-size open source system. They isolated the factors related to effective program investigation behavior by performing a detailed qualitative analysis of the program investigation behavior of both successful and unsuccessful developers, and report the evidence of the phenomenon of inattention blindness by developers skimming source code. Although Robillard et al. used a controlled experiment while our observations were carried out ‘in the field’ during authentic software development tasks, we use similar methods.

3.3 Grounded Theory

This research aims at providing a field-grounded theory [139][57][32], which is based on the informants’ perspective and explanations. To establish such theory the researchers gather data by observing, talking, and listening to the subjects (or informants) in their natural environments. In this kind of research, it is not intended to manipulate certain variables or factors, and the research arena is not the laboratory but rather the natural environment in which the activity ordinarily takes place.

The qualitative approach was a natural choice for this research since we wished to identify, extract, and formulate activities that, in part, only take place during commercial software development processes (the natural environment). The rationale behind these activities is implicit and the researchers extract it from the software developers (the informants) and make it explicit. Moreover, software construction is a human-intensive activity [144], and the qualitative paradigm provides the researcher with tools to properly address this property.
In our research, however, the natural settings were only maintained in the first phase of the research, the initial observation-based part of the experiments (described in Section 3.4.1 below). The later experiments, which constitute a major part of the thesis, include a questionnaire filled in after the informants had heard my lecture on example-usage, and on-line questionnaires and surveys specifically focused on example usage, where I had included my comments, and attempted to encourage discussion relating to this particular issue. This happened due to the major diversity that was discovered in example usage among the developers.

Instead of focusing on example usage as manifested in a specific context i.e. the few software companies observed during the research, we elected to examine the full spectrum of example usage by including the perspectives of as many developers as possible. However, this large number comes with a cost – the surveys, questionnaires and some of the interviews focused on example usage, and the data was not extracted by examining professional developers in their natural settings. Nevertheless, the data gathered in these later phases of the research is also based on talking, and listening to the subjects discussing aspects of example usage as manifested in their work as professional software developers (rather than laboratory experiments).

One limitation of this kind of data may be that the opinions reflected in the surveys and questionnaires may not conform to the actual practices of these developers. For example, they might not be aware of the actual frequency of their example usage, or of their dependence on Web search. One may argue that the diversity in example usage described in this paper does not reside in employing example usage practices, but rather in the attentiveness to example usage, and that the real thing being investigated here is how explicit example usage is in the development process. The issue is further discussed in Section 6.5.4.

Nonetheless, this research does not propose a new activity pattern proceeding from analytical or theoretical research; rather it presents a comprehensive description of an activity as a field-grounded theory, i.e. a field study that is accompanied by supplementary data gathering tools such as surveys, virtual focus groups and reflection questionnaires that help explain the (theoretical) context of the investigated phenomenon (activity pattern).

### 3.4 Research Course

The rationale for this research approach was derived from the working assumption that many of the best practices of software engineering reside in the minds of experienced programmers [137] and in the internal protocols of software companies. In some cases, this knowledge is implicit; even an
experienced and effective programmer might not be aware that some of his or her daily practices are not common knowledge. Thus, an observer might be able to extract some of this knowledge and make it explicit.

The data-analysis process is iterative. It accompanies and guides the data-gathering phase. The intermediate results obtained during data-gathering rounds gradually form a theory that is verified in later rounds of data gathering in the field or from the literature. During the first iterations, we named the categories using terms taken from the informants' and observers' vocabulary. In later phases, we aspired to turn those names into conceptual terms that better reflect abstract notions, constitute a theoretical interpretation of the relationship between the components of the phenomena in question, and provide a comprehensive description of the context in which they should be investigated.

The research process can be divided into three phases, built bottom-up, as depicted in Figure 3.1. In phase I, which served as a pilot, we used observations only. We considered a few fine-grained activities observed in the field, and chose to focus our attention on example usage. In phase II, having example usage in mind, we incorporated additional field observations, this time augmented with interviews. When we analyzed the data we had gathered in the first two phases [41], we noticed that example usage was neither applied systematically nor methodically. Hence, in phase III, we focused on example usage barriers, and used additional data gathering tools such as surveys, reflection questionnaires and focus groups to gain additional support for this observation, and to establish a theory.

Following is an elaboration of the three phases.

Figure 3.1: Research Course
3.4 Research Course

3.4.1 Phase I: Pilot

We conducted observations for a month at the development sites of two major, global software companies. We introduced ourselves to the teams at the beginning of that month, told them that we were interested in researching "patterns in software developments", and answered any questions. Ten developers agreed to take part in the research. Each day we joined a different developer for a programming session lasting two to three hours, for a total of fourteen sessions. The choice of which programmer to pair with was made ad-hoc, based on the programmers' availability and willingness, and the type of task they were engaged in (e.g. we preferred programming sessions over design meetings). We elaborate on the observation research tool in Section 3.5.1.

As part of these observations, we also spent time at the development site between the sessions. We joined programmers during their coffee breaks and lunches; we were present at some of their meetings and internal training sessions. These research settings were much inspired by Gideon Kunda's ethnographic research of the engineering division of a large US high-tech corporation described in [91].

We conducted this observation phase with an open mind. We did not know upfront what we are looking for, besides 'innovative patterns' or 'things with no name'. After the first phase of field observations, the example usage abstraction emerged: in some programming sessions it was obvious and explicit, however when we reviewed the data that we gathered a second time, we found additional interesting aspects that went unnoticed when we first analyzed this data without example usage in mind.

3.4.2 Phase II: Examples

With example usage in mind, we next conducted additional rounds of observations in a third major global software company. We joined a team of seven developers, one of whom was the team leader, and conducted two rounds of observations. Between the two rounds we shared our newly emerging research focus of example usage with the developers.

During this phase we also conducted some interviews with the developers regarding issues that arose during the observations, and regarding their views and input with respect to example usage. Having example usage in mind, we were more attentive, in real time, to interpreting the developers' activities in the prism of example usage. This time, we examined not only the techniques that the developers used but also asked ourselves whether some problems could have been avoided had example usage been considered.
We also conducted an interview with the chief programming officer of the organization regarding the company's overall efforts to enhance developers' productivity through example usage.

Augmenting our observations with interviews and follow up conversations enabled the emergence of a new theme: barriers to example usage. We noticed that many encounters with example usage were in the context of a problem or a challenge. Only then did we realize that we were biased towards example-centric development [44][25] and opportunistic software system development [110], i.e. software developers who acknowledge the benefits of extensive example usage, and proactively try to use examples wherever they can. The developers we met were minded to getting the job done and, if using examples was too cumbersome, too risky or not handy, they simply managed without. Our research subjects did not use examples as extensively as reported by recent example-centric and opportunistic development studies [24]: in some cases they were inattentive to using examples; in other cases they preferred to write the code by themselves, or found themselves struggling against the inherent challenges associated with example usage.

3.4.3 Phase III: Example Barriers

The third round was aimed at establishing a theory. Although we had succeeded in gathering rich qualitative data regarding example usage in the two previous phases, this data was somewhat arbitrary. To expand our data sources and to collect additional feedback on our preliminary findings, we implemented complementary data-gathering mechanisms:

- **Online Survey** – We conducted a comprehensive online example survey with over 300 participants. We asked the participants about the techniques they use when searching for examples; we further asked them to provide tips for novice developers and also asked them about the challenges they face and how they overcome them. We also asked them about why they use examples, about the advanced techniques they have developed, and about their overall reflections on example usage. We elaborate on the surveys in Section 3.5.3.

- **Focus Groups** – We ran two virtual focus groups: one with 40 entries and the other with 122 entries. These virtual focus groups emerged spontaneously in social network discussion groups, which we used to invite developers to take part in our online survey. We elaborate on the virtual focus group research tool in Section 3.5.2. We also assembled a third focus group (a 'real' one) comprising four experienced developers and two managers of a very large
global software company in order to examine how EE could be implemented in their organization.

- **Reflection Questionnaires** – We conducted a case study in which we discussed example usage with a group of twenty software practitioners, and then asked them to reflect about their own example usage practices. We also conducted a follow up after three months. We elaborate on the reflection questionnaires in Section 3.5.4.

The data we gathered and analyzed during this phase highlights the diversity of example usage in the software industry. In Chapter 5 we discuss this diversity with respect to awareness and attentiveness issues – different developers are attentive to different opportunities for example usage. This issue was supported both by our focus group case study (see Section 5.4) as well as by the online survey participants (see further analysis in Sections 6.5.1 and 6.5.2).

We found that the developers’ approach to example usage is dominated by human factors rather than engineering ones, which contributes to the diversity in example usage, as different people have different sets of beliefs and opinions. Developers are not aware of the existence of code examples, nor to the methodical use of them, nor are they attentive to their use in different contexts. In general, the software development community perceives example usage as merely a programming technique rather than acknowledging it as a fundamental software activity, and as an expression of the software reuse principle.

This diversity is the organizing theory of this entire PhD research.

### 3.5 Research Tools

#### 3.5.1 Observations

The data-gathering techniques used during the first two phases were based on observation. We used observational techniques of the first degree [98] that require direct involvement of the subjects, in this case the software engineers, in the form of participant observation and think-aloud protocols [98]. Hence, by researching in the field, in the subject’s natural environment, the researcher is exposed to the valuable implicit knowledge that is the target of the entire research.

During the observation sessions, the researcher documents everything that transpires. In our research, we used a three-column observation report format. The observer writes down what he or she sees in the first column, and what he or she hears in the second column. Most of the data in this column is written in the
first person, and it usually contains things that the subject says (to him or herself or to the observer) while working. In the third column, the observer writes down what he or she *thinks*. This column is reserved for the observer’s notes and comments and also serves as a field diary. Some of the data in the third column is recorded in real time, during the observation, but some of it is added during later stages of data inspection and interpretation.

The third column of the observation report is the core of the data analysis. It provides a wider context and, subsequently, facilitates categorization of the activities observed in the field [43]. Figure 3.2 shows one page of our observation reports (originally handwritten in Hebrew).
**3.5 Research Tools**

**Observation Report**

Company: A major company for router software  
Subject: Adam. He has 8 years of experience. He speaks loud and clear and explains everything he does in simple words.  
Environment:  
- On the cube walls there are sketches of the development tree, current commit schedule view, and current stretch schedule view.  
- On his desk there are 2 apples and a bottle of mineral water.

<table>
<thead>
<tr>
<th>What the observer sees</th>
<th>What the observer hears</th>
<th>What the observer thinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>I am working on a module from within a script that reads a variable in the motherboard. Earlier today I received examples for the way it is done. Benny showed me how to translate and I need to figure out how to implement it. Now I compare between the script files to estimate the essence of the change needed.</td>
<td>Examples. What is Benny’s official role? Estimations.</td>
<td></td>
</tr>
<tr>
<td>Adam browses the code (only eyes). Sees an interesting line (path) and copies it into another file.</td>
<td>I am saving the path in a file. I will need it later. Making preparations. Sticky notes analogy. The todo-list grows along with the work.</td>
<td></td>
</tr>
<tr>
<td>Adam plays with the cursor up and down.</td>
<td>We will compare between the files and try to see the difference in logic. Reading code. How can a novice learn to read code? Is it like reading texts as learned in high school?</td>
<td></td>
</tr>
<tr>
<td>Opens Emacs</td>
<td>Now I save the original file and try to merge it. Usually we use BeyondCompare but the changes here are local, no comparison is needed, I know which changes interests me. It is more a development task than a merge task; this is why Emacs is better here. Choosing one's tools (in which other professions?). When is merge similar to development?</td>
<td></td>
</tr>
<tr>
<td>Changes his screen to vertical position and all elements on screen rearrange automatically accordingly</td>
<td>Our company is thoughtful… Later I intend to write and I need to see a lot of code Preparing the environment (like before a surgery).</td>
<td></td>
</tr>
<tr>
<td>The cellular phone rings. Adam checks the caller ID but then silences the device and returns to his work</td>
<td>I even disabled the email notification popups. I only check messages twice a day. Where were we?... this is exactly why don't I answer, but family, you know… Sterile environment (again - surgery).</td>
<td></td>
</tr>
<tr>
<td>Edits a code snippet and adds a comment above it</td>
<td>Here I pass a local variable and I commented that this is a local. This is meant for later on, when someone else will read the code Explaining to himself</td>
<td></td>
</tr>
<tr>
<td>Add another check (if)</td>
<td>The check if this is a ‘leaf’ is done only if it is installed Checking the code by reading it. Wasn't it better to run some scripts or test cases? Staring at the code. Is there a methodological way to read code?</td>
<td></td>
</tr>
<tr>
<td>Finishes to edit this code snippet and goes over it with his eyes (staring) to see if it is correct</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.2:** Observation report taken during phase I.
3.5.2 *Virtual Focus Groups*

One of the data gathering tools we used in this work was an online virtual focus group [88]. This virtual focus group was established using a group discussion on the LinkedIn Website, which has some interesting properties that make it suitable to be used as a virtual focus group research tool, as explained in what follows.

**Subjects 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 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.

**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 set 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 http://www.linkedin.com/
**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, removing identifications, etc.). 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.

- 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.
3.5 Research Tools

Figure 3.3: Screen capture of the LinkedIn discussion group.
3.5.3 Online Survey
We conducted a comprehensive online example survey with over 300 participants. We asked the participants about the techniques they use when they are searching for examples; we further asked them to provide tips for novice developers and also asked them about the challenges they face and how they overcome them. We also asked them about their example usage purposes, about the advanced techniques that they have developed, as well as about their overall reflections on example usage.

Using Examples in Software Development

We are conducting this survey as part of a research, held in Tel Aviv University, that investigates the use of examples in software development. By ‘using examples’ we mean any use of an already existing code in the development process. We are looking forward to hear your thoughts on that matter. We also believe that you may gain new insights regarding the way you work just from filling this survey.

Thank you for participating.

Your details

Privacy notice: Your identification (name and email) is optional, however it is required in order to be able to contact you in case a follow-up interview will be needed. Your details will be removed from this questionnaire during the data analysis process and in any future publication.

Name:

Email:

Gender:
- Male
- Female

Years of experience as a developer:
- Less than 1 year
- 1-3 years
- 3-6 years
- 6-10 years
- more than 10 years

Figure 3.4: Screen capture of the online survey.
3.5.4 Reflection Questionnaires

After discussing the benefits and barriers of example usage with a group of developers, we handed the participants a reflection questionnaire (Figure 3.5) in which we asked them (a) whether they use examples in their work, (b) whether they are in favor of using examples, (c) whether they were influenced by the session, and (d) how they estimate the session will affect their work in the future.

The use of reflection [61] guides professionals such as architects, managers, musicians and educators to rethink and examine their professional creations during and after the accomplishment of the creation process. We distinguish between feedback data based on reflection questionnaires and feedback data based on other questionnaires. The latter address the session, the speaker, the material and their quality, while the former focus on the practitioner perspective, such as what have they learned, and their insights and perceptions regarding their personal experience [12].

![Figure 3.5: The reflection questionnaire given to the case study subjects](image-url)
3.6 Qualitative Analysis

The qualitative literature (e.g., [128] [139]) includes numerous techniques to ensure that qualitative analysis remains objective. The specific data analysis techniques used in phase III of this research are also discussed in the context of the corresponding experiments (Sections 4.2.2, 0). In the following we address the challenges in maintaining objectivity in this research and how we address them.

3.6.1 Observation Analysis

The observation sessions were conducted by a single observer (the author of this thesis), for following reasons: firstly, actually being observed could be a stressful experience, and multiple observers might create an even more stressful experience. Secondly, there was a substantial administrative overhead in putting these observations together (such as signing NDA agreements, and obtaining managerial approval) and we wanted to remove as many obstacles as possible in order to proceed with the observations.

A single observer might challenge the objectivity of the research findings by introducing preconceptions, bias or agenda (which the observer might not even be aware of their existence). To address this concern we took the following measures [33]:

- The observation reports were written in real-time (during the observations), and before the focus of the research was determined. This way, the observer would not be tempted to interpret what he sees or hears in the wrong context.
- The observation reports were sent to the research supervisors, so they could interpret the data independently of the observer.
- The analysis of the results was conducted in iterations, in which the research supervisors reviewed the results, and stated their reservations. This process was kept until all disagreements were resolved. In order to resolve a dispute we reexamined the reports to look for further evidence supporting (or refuting) the issue in question.
- We used informant feedback [156] by discussing our findings with experienced developers who participated in later phases of the research (e.g., the focus group case study described in Section 5.4). In this thesis we only included findings that were found to be authentic.
3.6.2 *Survey Analysis*

In order to maintain objectivity while analyzing the data written by the informants (discussion group transcripts, surveys, reflection questionnaires), we took the following measures:

- The raw data was shared between the author of this thesis and his supervisors to enable them to interpret the data independently.
- The analysis followed the Constant Comparison Method with *postformed* codes as described in [128]. These codes, as well as the coded text, were circulated among the researchers, in order to render the coding rationale explicit.
- After a categorization was identified, a reverse lookup table was created, in which each category is back-linked to all places in the raw data from which it was derived (Figure 3.6). This tool was used for traceability purposes, and to make sure that the original context would not be lost during the analysis process.
- Further confidence in the results was achieved via Cross-Case Analysis [128], in which some of the categories emerge in the analysis of different data sets. For instance, most of the variability factors in example usage discussed in the online survey analysis (Section 6.3.2) also emerged at the virtual focus group case study analysis (Section 4.3.3).
Figure 3.6: A part of a preliminary categorization reverse lookup table that was circulated among the researchers. The different code colors represent meta categories.
“That's the thing about people who think they hate computers. What they really hate is lousy programmers.”

- Larry Niven

Chapter 4

**Human Aspects of Example Usage**

*We present a virtual focus group case study that examines .NET developers’ perception of example usage. We divide the developers into groups by their position with respect to example usage and analyze their considerations.*

*We find great diversity among developers, and suggest that it is a result of human, rather than engineering factors. Moreover we discuss the implications of these findings on the popularity and acceptance of pragmatic reuse approaches, such as the Opportunistic Software Systems Development (OSSD) approach. We also examine the design decisions of the popular Q&A developers’ Website, Stack Overflow, which address similar human issues.*
4.1 INTRODUCTION

We conducted an online survey in which we asked programmers about their examples’ usage practices and techniques (this survey is discussed in Chapter 6). To encourage developers to participate in the survey, we posted invitations on various Web-based forums and discussion groups. One of these invitations was posted on the .NET People discussion group of LinkedIn, a business-oriented social networking Website [1], see Figure 3.3.

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 than 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 (we addressed this issue in Section 0).

We therefore posted another comment in this discussion thread, in order to obtain more developer feedback on this issue (Figure 4.1).

![Figure 4.1: A follow up post in ".NET People" LinkedIn discussion group.](image)

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. See the full transcript of the thread, including our own posts, at [10].

We divided these developers into 3 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. We then summarized their arguments for usage or lack thereof:

- The main points raised by example supporters are that: (1) using examples is more efficient, and (2) examples promote pragmatic reuse.

- Example antagonists argued against using examples due to: (1) the poor quality of examples, (2) bad experiences with example users, and (3) lack of important properties: testability, understandability, and documentation.

- The third group of developers use examples in the following, limited contexts: (1) for learning purposes only, (2) avoidance of copying the example code (3) for specific kinds of development tasks only, (4) based on the example size, and (5) based on the example source.

In Section 4.3 we elaborate on the main points raised by example supporters and opponents, and on the variability in example usage among participants included in the third group with respect to various aspects.

We further focused on crosscutting issues that were mentioned by all three groups, and examined how each group justifies its point. We came up with a list of themes from which these explanations are derived. We reviewed the whole thread again with the themes in mind and looked for a holistic theoretical framework – a single notion that would encompass the themes and provide a wider interpretation for all explanations at a higher level of abstraction. Although the participants mention engineering considerations, as presented in the vertical analysis, we conclude that human aspects dominate developers' behavior with respect to example usage. In Section 4.4 we elaborate on this.

This research goes beyond a mere report on the opinions of a set of people. Most of the participants in the reported discussion are experienced developers (see Section 4.2.4), and they are affected by legitimate albeit human concerns. In Section 4.6 we highlight the potential that resides in overcoming these concerns as manifested by the OSSD approach. We also examine ways to tackle some of the human concerns by reviewing the design decisions used by the Stack Overflow Website.

4.1.1 RELATED WORK
Avoiding the usage of knowledge sources outside of a team or organization is not unique to example usage and could be seen as a special case of the "Not Invented Here" syndrome [81]. This term is used to describe a persistent social, corporate
or institutional culture that avoids using or buying already existing products, research or knowledge because of their external origins

From the organizational perspective, examples are encapsulation of professional knowledge. In organizational literature it is recognized [75] that professionals seek knowledge from their own personal networks—ego-centered networks—which extend beyond the formal organizational structures [30] [39], in response to the need for rapid *ad hoc* knowledge collaboration. The ego-centered networks consist of connections with whom the professionals (the egos) have had some prior professional contact. Sometimes, the network will consist of members of public and private organizations with multiple and often conflicting interests [53].

### 4.2 Methodology

#### 4.2.1 Virtual Focus Group

In this research we used an online virtual focus group [88] as our primary data-gathering tool. This virtual focus group was established using a group discussion on the LinkedIn Website, which has several interesting properties that make it suitable to be used as a virtual focus group research tool, as discussed earlier in Section 3.5.2.

#### 4.2.2 Qualitative Data Analysis

The data analysis process we used included the following steps, which were adapted from [139]:

1. All 134 posts (including 12 that were later deleted by their authors) were read and copied into an editable document.

2. All posts were conceptualized: that is, each post was summarized in a single sentence, and added to the text as a comment. Using comments allows the other researchers to examine the summary sentences in their original context, and increases the objectivity of the analysis process.

3. Vertical analysis: based on Stage 2, the following categorization of the participants into three groups emerged: those who are against example usage, those who support example usage, and those who support selective or limited example usage only. The main arguments of each group were also summarized.

4. Horizontal analysis: we focused on crosscutting issues mentioned by all groups, namely *time (speed)* and *copy and paste*, and examined how each group justified its position.
5. We reviewed the entire thread again with these themes in mind and looked for a holistic theoretical framework – a single notion that would encompass the themes and provide a wider interpretation for all explanations at a higher level of abstraction. We found that human aspects dominate developers’ approaches to example usage.

4.2.3 LIMITATIONS
One threat to the validity of this research stems from the small number of participants and their homogeneous technological interest (.NET), since they do not represent the entire developers’ community. This is true. However, we do not draw any quantitative conclusions from this research but, rather, are interested in the qualitative results. In Section 4.3 we indeed use percentages; however, they are only used to summarize this specific discussion thread.

Another threat is that since users are posting on a public forum there may be bias on the part of the volunteers - i.e., only those people who feel strongly about this topic (or employees from very small companies who are unrestricted) have commented. We address this concern by presenting the size of companies for which the participants work, and, again, we do not purport to represent the entire developers’ community. This limitation is also related to a more general limitation of focus groups, in which the dynamics of the discussion can be easily swayed by few dominant voices (See also Section 3.5.2).

Another threat related to the public aspect of the medium is participants who do not write what they really think, but what they think is expected of them (e.g. by future employers). If this is true, it does not make the results less surprising – it implies that most of the participants think that they should express opinions opposing rather than supporting example usage.

Some of the participants in this discussion associate extensive example usage with copy and paste (and sometimes blind C&P). One could claim that these participants do not object to example usage per se but rather to irresponsible code duplications. It implies that these developers are not aware of legitimate uses of examples, which conforms to our study regarding attentiveness and awareness discussed in Chapter 5.

Another threat is related to the licensing issue (though mentioned by only one participant) – in some large companies developers are not allowed to use open source code due to copyright laws. These developers would not admit to using open source code, even if they actually do so.
As we mentioned in the introduction, we found this discussion somewhat surprising, and dramatically different (definitely more intense) than the sporadic comments we were used to getting. However, this fact also made the discussion thread an excellent opportunity to investigate the more profound reasons dominating the developers’ attitude to example usage.

4.2.4 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 Figure 4.2-Figure 4.4.

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).

Figure 4.2: Size of companies in which participants work
4.3 Vertical Data Analysis

Figure 4.3: Professional experience of participants (job title)

Figure 4.4: Professional experience of participants (in years)
4.3 Vertical Data Analysis

Below we present a summary of the vertical data analysis process. Based on the participants' responses, we divided them into 3 major groups (the verticals) according to their approach to example usage: (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.

Thirty five percent of the participants reported they use examples habitually and for various purposes. Fourteen percent oppose example usage altogether, and 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.

As Figure 4.5 shows, similar proportions were maintained across different roles, with the following two exceptions: there were no managers who oppose example usage altogether, and there were more founders/owners who support extensive example usage than those who make limited use of examples. This could be explained by the fact that founders are more attentive to the business and organizational perspective of software engineering and therefore see the benefit of example usage in a wider context. We elaborate on this issue in Section 4.4.1 when we examine example usage from the organizational perspective.

This quantitative data is only intended to characterize the participants of this specific virtual focus group. Due to the inherent limitations of the virtual focus group research tool we do not purport to argue the partition of the software development community at large.

![Figure 4.5: Dividing participants into groups according to their approach to example usage and to their job title. Participants who work in multiple roles appear multiple times.](image-url)
In Figure 4.6 the participants are partitioned by their years of professional experience. Although not statistically significant, it can be seen that the 11-19 group is the only one in which there are more example supporters than selective example users. In addition, in this group the proportion of example opponents is smaller than the overall proportion of example opponents. If we ignore the participants with an experience period of less than 3 years or more than 20 years (less than 6 participants in each subgroup) we see that example supporters increase in proportion to the engineers’ experience. We propose a few explanations for this observation: firstly, since there are some challenges involved in extensive example usage, novice developers, who lack the tools to address these challenges, do not leverage the full potential of code examples. The more experienced a developer is – the more tools he or she acquires that could assist him or her in addressing these challenges, and uses examples in additional ways. This also conforms previous empirical findings [107] showing that reuse-related experience is one of the two major factors (together with self-efficacy) affecting software developers’ intention to reuse software assets. We elaborate on this issue in Section 4.4 when we discuss example usage in the context of personal development, example dexterity and the development of analytical skills.

Also, experienced developers might have been exposed to a programming style different to their own and developed their own example usage attentiveness. We found that example usage level of attentiveness could be easily increased – See our case study described in Section 5.4.

Another explanation is related to the fact that many experienced developers are promoted to managerial positions, or are at least assigned managerial responsibilities. As discussed earlier, such responsibilities make them consider example usage in a wider perspective.

![Figure 4.6: Dividing participants into groups according to their approach to example usage and to their professional experience period.](image-url)
In Figure 4.7 the participants are partitioned by the size of the company they work for. Selective example usage is dominant in most company sizes with the exception of self employed participants, companies with 201-1000 employees and companies with 5001-10000 employees. This result refutes our preliminary conjecture (and that of some of the reviewers of earlier versions of this research) that working in large companies would be associated with explicit example usage policy. This result shows that example usage is somewhat arbitrary. In Chapter 7 we discuss the implications of conceptualizing example usage on the software development process.

**Figure 4.7: Dividing participants into groups according to their approach to example usage and to their company size. Participants who work in multiple companies of different size appear multiple times.**

In Sections 4.3.1 and 4.3.2 we elaborate on the reasons given by example supporters and opponents, and provide representative quotes. In Section 4.3.3 we characterize the variability in example usage among participants included in the third group with respect to various aspects. Table 4.1 summarizes the reasons for and against example usage as expressed by the virtual discussion group participants:

**Table 4.1: Summary of example usage and avoidance reasons**

<table>
<thead>
<tr>
<th>Reasons in favor of using examples</th>
<th>Reasons against using examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>It saves time</td>
<td>Examples are of low quality</td>
</tr>
<tr>
<td>Examples provide better understanding</td>
<td>The developer had bad experiences with copied code</td>
</tr>
<tr>
<td>Pragmatic code reuse</td>
<td>Examples lack important properties: they are neither fully testable nor understandable</td>
</tr>
</tbody>
</table>
4.3.1 Habitual Example Users

Following is an illustration of example usage benefits as mentioned by habitual example users. The main arguments relate to efficiency and to pragmatic reuse.

Efficiency

The following representative quote highlights the aspects of time saving and better understanding: "It seems that I am in the minority – I love coding samples. I understand the concepts better when samples accompany the verbiage. I will go further by saying that I hardly ever encounter a good article (or blog) that does not use coding examples to illustrate a point. When this code fits my purpose I do not hesitate to use it. The sample code usually needs to be fortified, but it saves me all the typing".

Another developer described how technology providers can write examples in order to communicate their preferred style to the developers: "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'".

In the discussion thread, 47% of the developers stated that they never copy the code of the examples they find. In response, a senior software engineer comments on also using example code: "In support of copy and paste which everyone seems to hate, I have to say it's a lot easier to edit and refine that way then it is to type the entire sample a character at a time".

Pragmatic reuse

Some developers describe how extensive example usage allows them to incorporate the reuse principle and pragmatic approach in their work. The following participant situates example usage practices in the context of economic reality: "Being commercial programmers we often need to think about what is the pragmatic approach. If samples (and even cut and paste) can help with saving us time and money, when backed up by solid engineering and programming methodologies, isn't that a good thing?"

Another developer addresses the fundamental reuse principle in programming saying that: "Anyone that isn't looking for code for reuse is reinventing the wheel and wasting time". Pragmatic reuse is also mentioned by what the following software analyst is saying: "...in Development there is a mantra 'Work Smart not Work Hard'. If you want to do something and someone has already done that particular work, then I think it is better to analyze the existing code and adapt it to your needs instead of starting from scratch. I think there is no harm in using and trying existing examples and codes for your own purpose."
4.3.2 Opposing Example Usage

Fourteen percent of the participants oppose example usage altogether. In contrast to other participants who are aware of the pitfalls of example usage and suggest ways to tackle them, the practitioners in this group find examples dangerous and harmful. We shall be analyzing the position of these participants and highlight some commonalities among them.

Example quality

The second comment on this discussion thread came from an experienced startup developer and owner addressing the quality of examples on the Web: "[I] Never [use examples], Most samples on the internet are written by guys in their bedrooms who most of the time have no idea. The quality of the code is usually sub-standard and is usually a copy from someone else’s blog. There are too many copiers and pasters out there calling themselves programmers". A managing director agrees with these statements and also states that examples might be misleading: "I agree with some of the people here; there are a lot of people here who offer advice that is both misleading, and often entirely wrong. The examples and answers offer no value in many cases, and may even lead the inexperienced reader to thinking that a particular style or approach is correct when it is not. There are few people out there who actually know what they are talking about, but even they sometimes offer examples that are incorrect, or have bad style, design, and so on."

Bad reputation

Some of the participants are disrespectful of people who use examples regularly, challenge their expertise and even make derogatory remarks. They say that example usage, and in particular copy and paste, is not appropriate in professional software development: "If someone finds a few lines of code in a blog or Website or book, likes how it looks and just copy-pastes it into their own apps thinking it to be a third party component - then they should be devalued by everybody - not just snobs :)". Another developer says: "I usually find it's quite a bit quicker to write original code than to be a cut-and-paste monkey". Another one agrees: "Looking at examples to understand the concepts is not wrong. However, 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..."). A similar approach is expressed by this participant, although he has not yet encountered a copy and paste individual in person: "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". The following one implies that extensive example usage might cost the developer his or her job:
"Any software developer/engineer doing this [using examples extensively] is going to [have to] find a new job real fast".

In order to support their claims some participants use exaggerations, images and blanket statements ("cut-and-paste monkey", "blindly copy-pasting") and equate example usage with copy and paste, partly, perhaps, because of our follow-up post (Figure 4.1). Others 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". And this one: "I've seen lots of quick and dirty approaches to coding... and, I've made lots of money fixing coding 'short-cuts'".

**Example properties**

Some of the participants reject example usage because of certain properties that the example lacks. They claimed that examples could neither be thoroughly tested nor understood. Examples were referred as too simplistic, poorly documented and neither optimized nor styled to the developer's needs. Note that these properties were mentioned not as pitfalls that one should be aware of when he or she uses examples, but rather as general properties of examples – which renders example usage unseemly.

We illustrate the above characteristics quoting one of the discussion participants. This software architect implies that exhaustive example testing is impossible: "'Copy-paste-test-alt' is a very dangerous thing to do. If your test cases aren't exhaustive (which is quite impossible if you don't know how the code works), then you'll be passing a lot of tests only for the code to fail at runtime. I would never even consider blindly using sample code, not even from Microsoft".

In a later post he also claims that examples could not be fully understood: "Say you see a 50 line sample that you understand and use. What are the chances that you missed one tiny bit? Example: generating thumbnails with Silverlight. Most sample code won't consider the large memory requirements caused by write buffer’s Render() holding onto the Image’s reference. And for one or two images, it's fine. You'll even understand the code. But when you get an out of memory, you won't know what hit you".

**4.3.3 Limited/Selective Example Usage**

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). Some of the people we included in the third group are keen supporters of example usage; however, they were included in the third group but not in the first one because they mentioned using examples in a narrow or limited context only.

In Chapter 5 we investigate example usage with respect to developers’ attentiveness. We show there that developers miss opportunities to use examples because they are inattentive enough to search for them or to fully utilize them once they are found. The members of the third group in this discussion thread are somewhat different. In contrast to the subject described in Chapter 5, who was not aware that further opportunities exist, here, in some cases the participants explicitly mention that they avoid using examples in certain contexts for various reasons. However, we believe that in some cases attentiveness does play a role here as well – a participant can explicitly avoid using examples in certain contexts and still miss opportunities to use examples in others due to his or her obliviousness.

We elaborate in the following on the variability in example usage among participants included in the third group with respect to various aspects.

**Learning**
Sixty five percent of the participants approved of using examples for learning purposes. However, delving deeper into what learning means to each of them reveals that even here examples are used in a limited and selective manner with respect to aspects such as: the environment, the centrality of examples in the learning process, and the things demonstrated by the examples.

- **Learning Mode** – Learning is exclusively associated by some developers with teaching environments, or when they are in ‘learning mode’: "I only use sample when I’m learning something new. But never use them at work". An IT manager supports using examples for learning purposes in a teaching environment: "I believe that the use of sample code is an excellent teaching tool. I would unhesitatingly support its use in a teaching environment".

- **Centrality of Examples in Learning** – Examples could serve a key role in the learning process, or could only be used as a supplementary tool for understanding, as demonstrated by these two quotes: "Once you start understanding a concept, code samples often aid in that understanding", and on the other hand the following participant accords examples a more central role in learning and understanding: "I would take samples over
documentation anyway [...] a quick proof of concept built from a few samples can quickly give you the understanding to build it for real”.

- **Learning What?** – Examples assist the participants in learning different things: One participant said that examples are good for learning a new technique or object interface. Another said: "I use examples all the time in order to learn new methodologies or technologies". Another software engineer stated that examples should be used *exclusively* for learning the technology and never for algorithmic purposes: "Code samples should never influence a logical solution". The following quote shows that examples assist in learning not only by demonstrating a good style but also by demonstrating a bad one: "...even when you get something that is a royal mess, you can normally learn from it, even if it only is how not to do it”.

- **Interferes with Learning** – Two participants, however, highlighted negative aspects of example usage with respect to learning. One participant reported that cutting and pasting was counterproductive for one of his students when he was a teaching assistant in college: "This person had been with me in class for 4 years and had spent so much time cutting and pasting that they missed out on their education". The second participant said: "If all you know is how to cut and paste, how do you know when and how to modify the sample into your complete answer??"

**Copy and paste**

The issue of using the code of the example by copying and pasting was addressed by several participants. While several participants insist on *not* using example code and claim to only read the example, learn it and write the desired functionality by themselves ("I never copy and paste code that is going to be used for production. I merely use it only to achieve an understanding of what I was missing then I develop it internally"), other participants found that copying and pasting example code saves them time ("When this code fits my purpose, I do not hesitate to use it. The sample code usually needs to be fortified, but it saves me all the typing"). A few others took a more selective approach and reported only using copy and paste code from a trusted source or when the code is small enough. We further elaborate on copy and paste in Section 4.4.2, in which we identify underlying themes dominating the participants’ approach to this issue.

**State of Mind (development mode)**

Some developers associate example usage with a specific development activity or a certain state of mind. This issue is also partially addressed in Chapter 5, in which we show how the attentiveness of a developer to example usage is affected by the type of cognitive task and the abstraction level.
This diversity in example usage is the organizing framework of this dissertation. All the variability factors below also reoccur in the analysis of the survey described in Chapter 6, in which we identify three axes for example usage motivations: (1) task properties, (2) type of development activity, and (3) software engineering considerations. The first two items below, 'Hello World' and 'Level of Familiarity', are considered task properties. The last three items are development activity types. The third item, 'Default vs. Last Resort' is addressed in both axes (task property = specific task and development activity = problem solving).

- **Hello World** – The concept 'Hello World' describes a minimal program demonstrating a new programming language or a new technology. The following developer uses 'Hello World' examples: "I use examples to tackle a new technology. I cut and paste my refactored code from other projects. I use snippets (from myself or others). Why reinvent the wheel every time you start a new project?". This approach, however, was not supported by all participants: "I have to say that I always start my developments from scratch. [...] It is hard to believe that some people think you can start an application from a sample, I certainly never considered that".

- **Level of Familiarity** – In addition to using examples to perform tasks with which the developer is unfamiliar, some developers report also using examples for tasks with which they are already somewhat acquainted. The following .NET developer associates example usage with doubts: "As a developer, when I have doubts about how to handle different situations I 'Google' what I need and I do some kind of research reading in many Web pages and sources". This one used examples for tasks he was already familiar with, but did not remember exactly how to do them: "When I was a baby programmer, I used samples to help me remember how to do something. The more I use something, it goes into my brain, and then the sample is no longer needed". Another created an example repository of his own to use for tasks he is already familiar with: "I don't know how many times I've forgotten how to use and stream reader or writer and had to go back to my 'examples' to find something out".

- **Default vs. 'Last Resort'** – A developer reported using examples "all the time". However he stressed that he only uses them as a last resort: "When I am stuck on a problem or working in an area I am not that familiar with [I] have no problem referencing a working model to learn from". While for other developers using examples is the default and common practice: "I'm most likely to cut-and-paste when I'm doing something for the first time that is specific to non-core libraries".
• **Syntax** – In addition to the above states, which are related to writing new code, one participant mentioned that code samples could also be useful for debugging: "Any framework as big as .NET framework takes years (even for seasoned programmers) to master – During these years it would be almost impossible NOT to find yourself one day (or lots of days) in need of a code sample to better understand a specific feature or sometimes to simply resolve a compile error”.

• **Code Comprehension** – A developer also mentioned using examples in code comprehension and problem decomposition processes: "Code samples is great to get an understanding of a certain part of a bigger picture”.

• **Periodic example reviewing.** The following participant read examples with no specific task to hand: "It is also good to review samples from time to time, just to see other programmers’ views on how to implement something, or just reviewing their styles. At times you may find something that you can adapt or say next time I should look at implementing that way”.

**Size**

Some of the participants reported that they avoid using examples larger than a certain size because they find long snippets hard to understand and to test. They said that when the code is long it is more likely to miss a corner case or an undesired behavior. One developer reported that he uses very small examples: "For me most sample code is a one liner or less”. Another developer was more liberal about the size: "When you copy and paste code, typically it’s very, very small pieces of logic, only a couple lines...or one type of method call”. However other developers found exceptions to this rule of thumb, such as if the example’s source is trusted: "In very rare cases, some more code is copy pasted when it comes from a trusted source (like Rick Strahl giving a black box 50 lines of code). But overall, the copy pasting is limited to a very little portion of the code and that too, only to reduce typing time”.

In one case a developer reported successfully using a whole class as an example: "Under very rare circumstances I have taken advantage of a whole class. The only example that springs to mind here was the need to create my own "diff” command. I presumed this wasn’t supported in the base libraries and that someone must have done it already so why invent the wheel. It probably would have taken me around three days to code from scratch and I may have made mistakes. As it was I had something useable in 3 hours, including my own improved custom interface”.

Another somewhat related issue deals with the proportion between the total amount of code that was taken from examples and the 'hand written' code in the
final software system. In addition to participants that reported never using examples or always using them, we found an interesting comment – a software engineer who was trying to defend occasional copy and paste said: "we do code from scratch the MAJORITY of the time but not ALL of the time for EVERY line of code". In a later post of his, he justifies his position both by the expectation of him and what he sees as the ultimate goal of programming: "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".

**Example source**

In some cases developers state that they are selective regarding the examples they use based on the example source. Few developers mention Microsoft and its official Websites as a trustworthy source (the discussion was hosted in a .NET LinkedIn group): "I think it's highly dependent on the source of the examples. I use a lot of code samples from www.asp.net and the MSDN sites, while I treat other sources with more caution". Other developers also mention other well known developers' Websites such as CodeProject and JQuery.com. One developer states that although he does not use examples currently, he would consider using them if they were backed up by a reputable organization: "One thing that I could see work some day is a free code library endorsed by a reputable organization". Two other developers mention "respected open source projects" as another good source to borrow code from.

In contrast to the above, one developer stated that the example source does not play a role in his example usage, and that poor quality is compensated by the iterative process, which he uses when looking for code snippets: "I always look for samples on Google - it even does not matter the source - I just browse rapidly the text and the code [...] Lot of times I discover my answer in somebody else's question on obscure forums; good or bad answers to those questions usually redirect me to other sites or other key words to search on Google".

Although most of the participants mentioned the Web as their primary example reference, and some of them simply referenced 'Google', two participants mention that they also use books, magazines and written documentation: "I use books and magazines as well to reference the objects, properties, and methods of code that I need to write. Documentation is perfect when you know how to read it and use it. And many times the documentation provides great examples as well". Several participants mentioned the Visual Studio built-in templates as kind of examples starter kit.
Table 4.2 summarizes the variability in example usage among participants included in the third group with respect to various aspects.

**Table 4.2: Summary of the variability in example usage among participants included in the third group**

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Kind of limited use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning</td>
<td>Only in ‘learning mode’ vs. using examples for ad-hoc learning</td>
</tr>
<tr>
<td></td>
<td>Only as a supplementary tool for documentation vs. using examples instead of reading documentation</td>
</tr>
<tr>
<td></td>
<td>Only for learning unfamiliar technology vs. using examples to implement the business logic</td>
</tr>
<tr>
<td>Utilizing the example code</td>
<td>Only read and understand the example code vs. copy the example code</td>
</tr>
<tr>
<td>State of mind</td>
<td>When starting a new project vs. starting a new project from scratch</td>
</tr>
<tr>
<td></td>
<td>For unfamiliar tasks vs. for already encountered tasks</td>
</tr>
<tr>
<td></td>
<td>Start with an example vs. refer to example after getting stuck on a problem</td>
</tr>
<tr>
<td></td>
<td>For feature implementation vs. for debugging vs. for code comprehension</td>
</tr>
<tr>
<td></td>
<td>Using examples ad-hoc vs. review examples periodically with no specific task at hand</td>
</tr>
<tr>
<td>Example size</td>
<td>Using only small examples</td>
</tr>
<tr>
<td>Example source</td>
<td>Using only examples from trusted source</td>
</tr>
</tbody>
</table>

**4.4 Horizontal Data Analysis**

In the previous Section we divided the discussion participants into three groups according to their approach to example usage and summarized their arguments. It is apparent that the participants are highly diverse in their views and behaviors. Why is that? Why isn’t example usage as consensual as some other issues in software engineering? For example, in the software community it is widely agreed that `goto` is considered harmful, or that function body should be kept short. What dominates developers approach to example usage?

We looked for issues mentioned by participants of multiple groups – crosscutting themes (e.g. issues that were mentioned by both habitual example users and example antagonists). This would provide a common ground for comparing apples and oranges. We found two such crosscutting concerns, namely time *(speed)* and *copy-and-paste*. When we revisit the participants’ references with respect to these themes we find that although some of the participants justify their position using engineering considerations, some human underlying themes dominate their approach.
Table 4.3 summarizes the themes from which the developers’ approach to example usage is derived, and compares the various perspectives of example supporters and antagonists with respect to these themes.

**Table 4.3: Summary of the themes from which developers’ approach to example usage is derived**

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

Following, we review each of the two crosscutting concerns and elaborate on its underlying themes.

### 4.4.1 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 [23] 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”.

Conforming to Organizational Goals
Software engineers, who are attentive to the needs of the organizations they serve, also tend to be attentive to the time that programming tasks take and acquire practices that assist them in minimizing the time, as stated by one participant: "Anyone who is not busy looking for code for reuse is reinventing the wheel and wasting time".

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: "Yes, it may be reinventing the wheel or a little time consuming but I am not pressed for time".

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 [97]. 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 [78] [73]. Lee and Kim [97] 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.

Personal development
The way that developers perceive their personal development affects the way in which they use their time. In some cases, developers' desire to master the underlying technology (so as to add it to their professional toolbox), might prevent them from taking a faster, more pragmatic approach. Such developers tend to spend excessive time learning, even at the expense of their product schedule. The following developer, for example, describes what he thinks to be the 'ultimate goal of programming': "Start coding, struggle, and LEARN".

Habitual example users, on the other hand, see the personal development in becoming effective and efficient programmers. Taking the pragmatic approach
time after time improves their code retrieval and code comprehension skills, and makes them faster and better programmers.

The Personal Software Process (PSP) [69] 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”\(^4\) 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 [7] 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).

**Acknowledging Example Dexterity**

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.

In the agile community there is a discussion about possible tradeoffs between speed and quality. Robert Martin ('Uncle Bob') argues in his blog post [20] 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 [76] 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 lead to the conclusion arrived at by the following programmer implying a tradeoff between time and quality (as opposed to [76]) in the following parable: "To sweeten cookies you could use sugar. If the client had diabetes, you'd definitely need artificial sweeteners. You wouldn't want your client's sugar [level] to go up simply coz you saved two minutes (and didn't pay enough attention)".

### 4.4.2 Copy and Paste

Using the code of the example by copying and pasting it was one of the most controversial issues in the discussion thread. Forty seven percent insisted that they do not use example code and claimed to only read the example, learn it, and then write the desired functionality by themselves: "I never copy and paste code that is going to be used for production. I merely use it to achieve an understanding of what I was missing, and then I develop it internally". We also note that in several commercial companies, as mentioned by one of the participants, copying code from the Internet is forbidden due to copyright and licensing issues. Thirty-five percent of the participants reported that copying and pasting example code saves them time: "When the code fits my purpose, I do not hesitate to use it. The sample code usually needs to be fortified, but it saves me all the typing". The other eighteen percent took a more selective approach and
reported only copying and pasting code from trusted sources, or when the code was relatively short.

How do programmers explain their attitude to copy-and-paste? Those who use it said that it saves them time, and makes it easier for them to edit the code further. Those who oppose copy-and-paste, however, did not justify their approach explicitly; hence the analysis in this case was more challenging. We briefly present some of the themes that dominated participants' attitudes toward copying and pasting.

**Ego**

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".

Copy-and-paste was repeatedly 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".

We find support for the ego effect on software development in the literature as well. Lechner [96] addresses egocentric behavior when he describes a failing extreme programming project. Williams and Kessler [154] 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 [149] 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 [148].

**Community identity**

Developers 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 guys 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".

The community identity of developers was studied primarily in the open source development context. Madey et al. [102] analyze the open source software development phenomenon based on social network theory. Lewis [99] examines how to use online communities to drive commercial product development. Sharma et al. [130] 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 [120] 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 [58]: “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 [46] 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 identify would delegitimize this knowledge – the example. It is suggested that inside large corporations, managers can encourage the emergence of “communities of practice” [151].

Ownership
Another underlying theme that affects the developers' attitudes towards copy-and-paste is the 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".

The issue of ownership was also addressed by the agile software development community. Two of the Extreme Programming practices, shared code [18] and collective ownership [16], may assist in disarming some of the developers concerns in taking ownership of code which was written by others. Beck and Andres [18] 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 [16] proposes to integrate the code
frequently, to write and run tests, to pair program and to adhere to coding standards.

Jeffries et al. [77] 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. [51] 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 [42]), 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.

**Trust**

Some developers, as we have shown, 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.

In the posts of many programmers, we find characteristics of taking the ownership of code. The most dominant is the understanding – programmers feel that if they fully understand the code, they may take ownership of it. It conforms to the explanation that we are afraid of things we don't understand or don't know well enough and that we trust the things we know well. Some developers
reported studying the example in conjunction with testing it, and others only use testing.

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.

Individuals may generally trust members of their inter-organizational ego-centered networks in terms of member competence [75]. Sharma et al. [130] 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. [129] in the context of outsourcing. They find trust as encouraging boundary-spanning activities [6] (in the example usage case – examples outside the organization or the team).

In a survey by Holmes and Walker [64] 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.

**Software Engineer Role Definition**

Copy-and-paste has become a much easier and more frequent activity due to graphical user interfaces and modern editing programs that make it readily available. In addition, online documentation and the Web make much more code accessible. Software engineers, however, are trained primarily to write code. 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)". Writing new code was the foundation of their programming training; one programmer even reported that it 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".

In Section 3.5.2 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 [86]. 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 Section 5.4.2 said: "At the end of the day, we are all merely plumbers..."). After all, some of them might have chosen programming as their profession because of its status.

Pour et al. [119] examine ways to make software engineering respectable including 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. [2]). Ensmenger [45] explores the many diverse attitudes and opinions on what professionalism meant in the 1950s and 1960s. In an article called "What Do You Mean I Can’t Call Myself a Software Engineer?" Speed [138] describes a licensing criterion specifically suited to software engineers, explains the legal issues involved and how they affect software engineers. Kruchten [89] 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.

A pragmatic approach, which challenges the classic definition of a software engineer, is also manifested by OSSD [110]. It is an approach in which developers meld together software pieces that they have found. They mostly find unrelated software components and systems that weren't 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.

**Analytical Skills**

Some of the positions for and against copy-and-paste stem from the speakers' perceptions of the analytical skills the action involves; some practitioners might have chosen programming as an occupation because they were good at its core tasks, and might now be concerned about their possible inability to perform as well at the new tasks at hand. While on the one hand, copy-and-paste antagonists see it as a grayish task, copy-and-paste fans recognize the challenge: "Copy-paste [is] not advised, as it will never help a programmer improve, but it will help improve his analytical skills". A keen example supporter said: "I describe myself as a Problem Solver as opposed to a Solution Repository! I may agree that the reverse would produce better experts but, I think, with less creativity..." This quote also illustrates nicely the role definition aspect.

In his article "Is Google Making Us Stupid?" [31], 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 [155] 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 [113] 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.

**4.5 Addressing Human Factors in StackOverflw.com**

Although developers are diverse in their approach to example usage, those who oppose example usage do not do so arbitrarily. They have legitimate, albeit
human, concerns. Below, we examine how some of these concerns were addressed on a related domain – Websites for questions and answers (Q&A) for professional developers. We examine the design decisions of one of these Websites, namely Stack Overflow, and interpret them in the context of the human concerns described earlier.

Stack Overflow\(^5\) is a popular Q&A Website that facilitates the exchange of knowledge between programmers connected via the Internet. In the three years since its foundation in 2008, more than 1 million questions have been asked on Stack Overflow, and more than 2.5 million answers have been provided [145].

Stack Overflow is centered around nine design decisions\(^6\): Voting is used as a mechanism to distinguish good answers from bad ones. Users can up-vote answers they like, and down-vote answers they dislike. In addition, the user asking a question can accept one answer as the official answer. Tags are used to organize questions. Users must attach at least one tag and can attach up to five tags when asking a question. Editing of both questions and answers allows users to improve the quality and to turn the Q&A exchanges into wiki-like structures. Badges are awarded to users to reward them for their contributions once they reach certain thresholds. This form of karma is used to encourage contributions. Pre-Search helps avoid duplicate questions by showing similar entries as soon as a user has finished typing the title of a question. Stack Overflow was designed to be used so that Google is UI. Web pages on stackoverflow.com are optimized towards search engines and performance. To ensure critical mass, several programmers were explicitly asked to contribute in the early stages of Stack Overflow.

We see that many of the design decisions address the human factors that emerged in our study, and have transformed Stack Overflow into a community. The badges and karma give the users sense of belonging – of being part of a large developers’ community. The voting mechanism allows the community to rank both users and answers, and tackle the trust issue. Taking ownership of a code snippet is easier after it has received community approval, and ego is confronted with community feedback and the transparency of the ranking mechanism.

4.6 Implications

One may take an organizational perspective in order to address the human concerns of example usage. In our analysis we mention several opportunities in which the organizational perspective could prove useful, such as by making

\(^5\) http://stackoverflow.com/
\(^6\) http://www.youtube.com/watch?v=NWHfY_IvKlQ
example usage an explicit organizational goal, by nurturing a global community identity and by providing a means for the integration of personal and organizational development. A similar approach was adopted by Procter & Gamble, as reported in [70]: “We needed to move the company's attitude from resistance to innovations ‘not invented here’ to enthusiasm for those ‘proudly found elsewhere.’ And we needed to change how we defined, and perceived, our R&D organization—from 7,500 people inside to 7,500 plus 1.5 million outside, with a permeable boundary between them.”

Furthermore, examining the design decisions taken by the Stack Overflow team might inspire one to use similar design decisions in order to build a corresponding Web community for code examples, which could address some of the concerns of the example usage opponents.

### 4.7 Summary

In this chapter we presented a virtual focus group case study that examines .NET developers’ perception regarding example usage. We divided the developers into groups by their position with respect to example usage and analyzed their considerations. We found great diversity among the developers, and suggest that it is a result of the following human factors: conforming to organizational goals, personal development, acknowledging example dexterity, ego, community identity, ownership, trust, role definition and analytical skills.

One might wonder about the validity of this assertion, since technical and engineering aspects of example usage do indeed exist. These include the need to test and ensure that the example code does not violate any property of the system, as well as security and performance considerations. In response, we argue that all these technical challenges also exist for habitual example users; however, they acknowledge them as being part of the job. A bug may indeed be hiding in the example code, but taking explicit ownership of the code (including testing) will discover it. It takes time to find the right example and rewrite it to your needs, but acknowledging the skill required to do so (and receiving proper training) is the first step toward getting better at it.
Chapter 5

**DEVELOPERS ATTENTIVENESS TO EXAMPLE USAGE**

In this chapter we focus on the developer’s awareness of and attentiveness to example usage. We identify three types of inattentiveness that causes professional developers to only use examples in certain contexts but not in others: inattentiveness in using examples involving different technologies, inattentiveness to examples of different scales, and inattentiveness to the variety of purposes examples may serve. We present three programming sessions, in which we observed professional software developers at work; the third session illustrates the three types of inattentiveness. We also describe a focus group case study that further investigates developers’ attentiveness to example usage, and discuss ways to address attentiveness issues.
5.1 INTRODUCTION
We describe several cases in which developers only use examples in certain contexts but not in others, and present examples that are only used in a limited way and whose full potential is not exploited.

Specifically, in this chapter we describe three of the programming sessions we observed in the course of our research. The first observation illustrates an effective example driven development practice; the second observation illustrates one of the pitfalls of using examples, namely examples that are scattered across several modules; and in the third observation we focus on the developer’s inattentiveness to example usage.

We also present a focus group case study in which we raised the awareness level of a group of developers to the potential benefits of the habitual and systematic use of examples and to the various purposes that examples may serve. We conducted a guided reflection process both immediately after a session on example usage and after 3 months, and found that merely discussing example usage influenced some of the participants to consider using examples more frequently than they did before.

In the Discussion, we examine ways of addressing issues of awareness and attentiveness. We suggest that example usage should be treated not only as a programming technique, but should be appreciated as a fundamental activity of software construction. As such, it is not related to a specific domain or context and is not overlooked when context changes.

5.2 EXAMPLE DRIVEN DEVELOPMENT OBSERVED
The first time we encountered an explicit example usage was during an observation session with an experienced developer, Adam, who was performing example driven development (without calling it that). His team was working on embedded software for specific hardware, received from an overseas’ office. No one on the team knew how to work with this specific hardware, therefore the team assigned one team member, Dan, to learn how to perform its basic operations. This team member wrote a set of 10 short example programs, each of them demonstrating another feature of the hardware, and emailed them to all the other team members.

We joined Adam for a programming session in which he was trying to map to memory a new piece of hardware recently added to their existing large and complex machine. The code running on the machine is written in C. At the beginning of the session Adam opens his email account and finds a message containing several code examples. He explains: "Dan sent some examples for
several developers in the team that deal with this stuff... after I met him and explained to him what I need he sent me some more examples”. From that point on, he was very determined: he went over the example names, chose an example and opened it. He said: "this is the 'Hello World' of base boards".

He opens the code in which he should incorporate the new functionality side by side with the example code, and thinks aloud: "...but in this part, as opposed to the example, the reader is the base board and the writer is the daughter card which says 'I'm up'...". Over the next 50 minutes, Adam was able to rewrite the example to his needs and to complete his programming task.

The literature has already found think-aloud behavior to be useful when using examples. Michelene et al. [34] found that comprehension is achieved through self-explanation. While reading examples good students generate many explanations that refine and expand the conditions for the action parts of the example solutions, and relate these actions to principles in the text. These self-explanations are guided by accurate monitoring of their own understanding and misunderstanding.

During the observation itself Adam’s use of examples this did not seem to be innovative in any way: just another case of a simple common sense. Common sense, however, is not that common. When we subsequently reviewed the observation report of this session we noticed three interesting things in retrospect:

- Firstly, the work was directed. Adam did not waste time pondering, he was focused and determined. Although he did not know in advance how to perform the task, he knew exactly what would get him there: he opened his email targeted at finding the closest example to his specific task and took it from there. Using examples as templates for work among students was studied in [122].
- Secondly, the team used examples as means of collective learning and spreading knowledge to the other team members.
- Thirdly, the example usage itself is not atomic, but rather a composition of several activities: browsing the examples, choosing the most relevant one, copying the example code and adapting it to a specific context. These various activities, and possibly others, serve as the building blocks of the example usage activity. They require different skills, their best practices should be better understood, and could perhaps be helped by designated software tools.

Other researchers also observed example-centric programming. Stylos and Myers [140] identified the high-level activities involved in API learning while observing
computer science graduate students perform three small programming projects in Java. Brandt et al. [25] conducted a laboratory study in which graduate students used Blueprint, a Web search interface that helps users locate example code. In [25] they describe an example-centric behavior of Jenny, one of the participants.

5.2.1 *Barriers to Example Driven Development Observed*

We augmented our later observations with interviews and follow up conversations, which facilitated the emergence of a new theme: *example usage barriers*. We often notice that when encountering an example usage it is in the context of a problem or of a challenge.

The developers we met were committed to getting the job done, and if using examples was too cumbersome, too risky or not handy, they just made do without. Our research subjects did not use examples as extensively as reported in recent and opportunistic development studies [24]. In some cases they were inattentive to using examples, in other cases they preferred to write the code by themselves, or found themselves struggling against the inherent challenges that accompany example usage.

One such occasion occurred when we observed two developers trying to add a new option to a drop-down menu. This menu was part of the Web front end managing a complicated Telco system. Initially, adding a new option to an already existing component seemed an easy task; the already existing options could serve as examples for how it could be accomplished. However, a simple copy & paste did not yield the anticipated results. Although the new option appeared in the Web interface, selecting it did not trigger the correct functionality. Over the next seventy-five minutes, the two developers tried to pin down the problem. They went through several technologies, configuration files and sub systems. They tried to reach the developers in charge of each of the sub systems involved with this feature but with no success (two were absent; one was unable to assist). They even migrated to another deployment environment (itself a complicated task involving tedious server configuration and resource allocation within the organization) but to no avail. The observation session ended without resolving this problem.

We avoided asking the developer questions either during this session or after it; when one is trying to fix something that is not working the last thing he or she wants is to be asked niggling questions by an outside observer (being observed could, in itself, be a stressful experience). Moreover, we suspected that this unfinished task might be perceived by the two developers as their failure, and that they would not feel comfortable discussing it, as it was still fresh.
However, when we visited one of the developers again, a few weeks later, we asked him about what went wrong in this particular case. The developer told us that the problem was copying the XSLT code exclusively and neglecting other layers of the system in which the new feature should be updated. The snippet they copied and pasted ("the example") was only one part of the complete feature. When a single example is scattered across multiple modules with no references between them it is hard for it to be reused in full.

This session, as well as others, motivated us to explore example usage from the barriers' perspective: which factors affect example usage, which properties of example usage could be addressed in order to make it more useful, more effective and more frequent. In the next Section we focus on one such factor: the attentiveness of the developer to example usage.

5.3 ATTENTIVENESS

To illustrate the three aspects of example attentiveness, we describe and analyze three situations that occurred during one session that we observed with the same software engineer. One may argue that in some of the situations, attentiveness was not the primary factor affecting the programmer's decisions, even if that was true for some or even all of the situations involving that particular software engineer, all three situations were found by our focus group study (see Section 5.4) to be genuine and authentic.

5.3.1 ONLY ON PYTHON

The Situation

We joined Ben for a programming session in which he needed to convert some files from one format to another. This task required some file system operations in the Python programming language. Ben had only recently started to work with Python, and the session began with Ben opening a browser on the Google search page and typing: "list files in directory Python". Ben then browsed through the results on the first page, selected the official documentation site (http://docs.python.org) and indeed, found a description of the required function: os.listdir. Ben added the function call to the editor by typing it in himself, and ran it. The code broke due to an indentation problem (missing tab), which Ben fixed quickly, tested the code, and found that it worked.

Chris, a fellow programmer working in the next room, came into Ben's room and interrupted Ben's work. He needed Ben to send him the new version of some files as soon as possible (Chris was stuck with collisions in the middle of a commit). After Chris left, Ben tried to send him an entire directory using the SCP utility (a command line software tool used to securely transfer computer files between
local and remote hosts). The command failed due to an incorrect parameter format. Ben asked the researcher sitting beside him (me!) whether he knew the correct format, which he did not. Then, Ben tried typing man scp in the command line (man is a common help utility in UNIX systems). The documentation of the command loaded and Ben searched for the terms "directory" and later "dir", but did not find what he was looking for. Finally, he called Chris and asked him. Chris solved the problem: Ben should have added "-r" for recursive copy.

**Analysis**

In this short scenario we saw the same person applying two different strategies to find answers within a very short time frame. The first strategy, a Web search for code in Python, seemed to work just fine. However, in order to find the SCP format, the software engineer first asked a person in his immediate vicinity, then consulted local documentation, and finally asked a remote person. Indeed, we can discern a pattern of expanding search circles. But would a Web search not be applicable here as well?

We find this to be an example of the attentiveness issue. Ben is not aware that Web search and example usage are general techniques that are applicable to various development contexts. Ben, who is new to Python, is accustomed to working in Python in short search-write cycles: he articulates the next step to himself, searches the Internet on how to implement it in Python, writes the code down, and so on. However, it seems that to him that this way of working relates specifically to Python.

There are several reasons why Ben might think this: he possibly thinks Python is unique in its Web presence due to its large open-source community; perhaps he just emulates the way the person who taught him Python works, or perhaps he thinks that SCP is too esoteric to be documented on the Web. Whatever the reason was, it seems that simply drawing Ben’s attention to the issue would encourage him to use Web search and online examples in new contexts.

When we discuss this issue with developers, we find support for our observation. In one anecdotal case, a developer declared that he only searches for examples in the C programming language but never in the CShell scripting language, whereas a fellow developer mentioned that CShell is like a “heaven for example searchers” because “CShell developers like to brag about their cryptic scripts by putting them online”.

5.3 Attentiveness

5.3.2 Read Only

The Situation

After Ben sent the required files to Chris, he returned to his tasks and after figuring out how to list the files in a given directory, he was ready to move on. Ben opened Google and typed "create new file in python". He examined the result on one of the sites, but was not satisfied, so he went back to the search page and refined his query with the string: "dive into python for experienced programmers". The first result now appeared - diveintopython.org/

Ben examined the resulting Webpage, which describes the function open and gives several examples for using it. He read the page carefully and then typed f=open("filename") in the interactive shell, which seemed to work. Ben next tried f=open("filename", "rw"), but this time it did not work. Finally, he typed f=open("filename", "w"), which worked as well. At this point, Ben left the interactive shell and updated the script he was writing in the editor with the open command. Execution of the code, however, failed and careful examination revealed several issues including a redundant quotation mark, an illegal directory structure, and a non-existent file. After correcting these issues, the program ran smoothly and we took a coffee break.

Analysis

In this scenario, Ben knew what he was looking for, and he even knew where to look for it (he added the Website slogan to the search query). However, when he found a code snippet that matched exactly what he needed, he did not copy and paste it, but rather typed it in again manually. This also happened in the previous situation. For Ben, the example serves as a learning aid, and a reference; he is inattentive to any other usages of it (such as using the code itself). Note, that some of the problems that caused the execution to fail at first (including the missing tab in the previous situation) would not have happen had Ben copied and pasted the example as is.

This situation demonstrates that code reuse via example usage can be achieved in many ways. However, when a code example is read but not used, then in fact only the design behind the code is reused and the reuse may not be taken advantage of fully. This is somewhat related to Steve McConnell's idea that "the most powerful form of reuse is full reuse" [106]. This discussion is, of course, context dependant - code reuse is not possible or beneficial in all situations.

Inattentiveness in using example code might be the result of various possible reasons: Ben might have encountered code examples in his programming courses in university in which examples are used for learning purposes to stimulate
effective inductive and deductive learning. Encountering examples in this context only might have fixated Ben's cognizance of example usage: only to learn from, but not to (re)use. Ben became accustomed to associating examples with learning.

There are two, somewhat related, issues here: context and fixation. After using the same method in the same way for long enough, one becomes accustomed to that usage and no longer seeks new ways of using it. In this sense, programming techniques are no different from other crafts [105].

When we discussed this scenario later on with other programmers, some speculation was voiced regarding a possible reason, other than inattentiveness, for not using C&P. Some programmers spoke of a conscious choice, reporting that they do the same thing, especially if the code example is large. Typing the code themselves provides them with the opportunity to review it once again and improve their understanding of it. Some programmers said that it helps them remember it for future use and to identify possible hidden problems that did not arise when they first read the code. Other programmers claimed that remembering things for the future is useless, explaining that should a developer ever need this piece of functionality again, it would still be faster for the developer to repeat the search process than to rely on his or her own memory. One of the programmers phrased this thought beautifully and said: "I describe myself as a Problem Solver as opposed to a Solution Repository!" [10].

Another possible explanation for not using copy and paste but for re-typing the example instead is the developer's attempt to avoid code duplication. Copy and paste has earned a bad reputation as a means of introducing duplication into the source code, which is considered the root of all evil. However, re-typing a code snippet is probably worse, since a consistently cloned code fragment is as good as the source and can potentially be identified and extracted by an automatic clone detection tool. On the other hand, a re-typed fragment might introduce new bugs due to the inconsistency of the duplication, fail to preserve important documentation notes in-lined in the example source, and is less likely to be identified by automatic tools as a clone. Moreover, when an example is copied, the chances of maintaining the code's original integrity and consistency increase.

The above benefits of copy and paste relate to re-typing the code. We note that copying code in order to use it unaltered may introduce problems, and one could think of other, better, techniques, which may involve, among other things, rewriting and refactoring the original code.
5.3 Attentiveness

5.3.3 THINK BIG

The Situation
When we returned from our coffee break, the script Ben had written was still running, but Ben suspected it was stuck. He interrupted the program and looked for reference files that could be compared with the files converted by his script. This task required the creation of some folders and the updating of other files. Ben switched to the browser window, where the results of his previous search still appeared. He checked to see whether the word "write" appeared on the page, but it did not. Therefore, he typed "file python write" in the search textbox and selected the Webpage diveintopython.org (which he also used before).

Analysis
Looking at the three sessions as a whole, it is now easy to observe that Ben was performing search operations on related tasks: list the files in a directory, open file, write to file. These tasks are not just related; they encompass a common flow for performing file system operations. When searching the Web, however, Ben did not look for the entire flow but rather broke it down (in his head) into sub-tasks and addressed each of them independently – this is one aspect of what functional decomposition is about: divide and conquer.

This behavior conforms to the learning steps observed by Stylos and Myers in API learning [140], depicted in Figure 5.1. In their study each of the programmers started with an initial idea of what their application was to do (A). Only after getting an overview of the structure of the APIs they would use (B) did the programmers begin to design how to implement their application (C). These initial design ideas would sometimes require further high-level API understanding (B). Once they had a high-level design, they looked for specific methods that could accomplish their task (D). Once they found the name of a method in an API, they searched to find out exactly how to use it, using documentation and example code (E).

![Figure 5.1: High-level programming activities observed in Stylos and Myers study](image-url)
Let us, then, consider a different strategy. What if Ben had looked for an example of the entire scenario on the Web? Queries such as: "Python write each file in a directory" or "Python example writing to all files in directory" would yield many promising results, including books, tutorials and blogs. For example, when using the former query, the third result Google gives is the "Tutorial on File and Directory Access in Python", which demonstrates how to list a directory, read its files, and write them. Google’s sixth option is a blog page called: "Loop through files in directory and modifying them on the fly", which demonstrates, as its name implies, how to use the Python walk utility to open, read and write files (file listing is implied in the walk utility). The fifth result of a search for the latter query reveals a book chapter by O’Reilly entitled "Learning Python - Sample chapter 9: Common Tasks in Python", which, again, presents the required flow.

Some of our research subjects with whom we discussed this issue reported that they are attentive to using examples only up to a certain size. This size may vary considerably from one developer to another, and ranges from a single character (one developer sought an example using the Linux shell redirection sign – ‘>’) to an entire product (for example, XULRunner allows the embedding of a full Mozilla Web browser in other projects and products). Some developers reported that they do not use examples of other scales because they are not aware of their existence or because they are inattentive to using them since it is not part of their routine. A classification of source pieces by size is also suggested in [54] [146] and is also discussed in Section 4.3.3.

So, why did Ben not look for the entire flow in the first place? There are several possible reasons; some of which are related to Ben’s awareness and attentiveness to example usage (or lack thereof). Ben started to code only after he had already decomposed the problem in his mind. He was locked into a mindset of implementing each part by means of a quick search-write cycle and then compiling the parts together. At this stage, he was inattentive to any other course of action. When thinking at a certain level of abstraction, or granularity, one is only attentive to the possibilities that exist at that level. However, if Ben had been considering example usage while he was thinking at the higher level of abstraction (during the problem composition phase), he might have not missed the opportunity to find a comprehensive solution to his problem.

Two possible explanations for Ben’s inattentiveness that were discussed earlier apply here as well: context and fixation. Ben may associate example usage with a certain level of abstraction or with a specific problem size because he saw it applied effectively by a senior developer (or even by himself) in a task of that
particular size. Such size may, however, be arbitrary. Failing to extract the abstract notion of example usage from the specific programming technique, and then repeating the technique again and again in consistently similar settings, may “anchor” the technique in its original context and settings and prevent the practitioner from applying it differently.

Another issue related to example size involves example retrieval techniques and tools. I still recall my astonishment when a fellow programmer showed me that he could ask Google whatever question he liked, no matter how technical or esoteric. Before this revelation, I used to simplify the search query so that Google returned a Website in which some further inquiry was needed. This, however, is often unnecessary. In this case, the awareness issue referred to the capabilities of the search engine, and in the more general case – the capabilities of the code retrieval tools (whether IDE search capabilities or others). This issue is also addressed by one of our focus group subjects in the next Section.

5.4 Focus Group

5.4.1 Settings
Twenty software professionals participated in a session on example usage in software construction. This group holds regular bi-monthly meetings on software development and technology, in which guest speakers talk about their fields of expertise (the sessions are free of charge). We volunteered to conduct one such session, provided we could subsequently conduct some research. We published the session agenda on the Internet in advance.

During the session, we first took 45 minutes to describe our research on example usage so far. We discussed (1) the potential benefits of the systematic use of examples; (2) the barriers that we identified as preventing example usage from being applied more extensively; and (3) several techniques that we observed being applied by individuals and organization that are related to example usage.

Next, we handed the participants a reflection questionnaire in which we asked them (a) whether they use examples in their work, (b) whether they are in favor of using examples, (c) whether they were influenced by the session, and (d) how they estimate the session will affect their work in the future. All of the questions were open ended. We also requested the subjects’ consent to take part in a follow-up interview to be held several weeks later. Fifteen reflection questionnaires were completed and submitted and ten of the programmers were willing to participate in a follow-up interview.
Three months later we sent the participants a follow-up email, in which we asked the following questions: (1) did the talk (or completing the questionnaire) affect your awareness of reuse and example usage? If so, in what way? (2) Do you feel any change in the way you work with respect to code reuse and example usage since the talk? (3) Have you incorporated any new techniques or practices in your work with respect to example usage? If so – please describe them? (4) Have you gained any new insights regarding example usage since the talk? If so – please describe them? Three participants responded to the follow-up email.

5.4.2 RESULTS

Measuring attentiveness and awareness is a tricky task. Asking your subjects about their awareness of something falls into the trap of Heisenberg’s uncertainty principle – merely asking the question affects the answer. Therefore, our reflection questionnaire does not address attentiveness directly. In fact, when we conducted the guided reflection process we did not have attentiveness in mind. Attentiveness is a theme that we only identified after analyzing the questionnaires.

Seven of the fifteen subjects stated in their answers that the session increased their level of awareness of new opportunities for example usage. Three used the term awareness, and others used the terms attention, motivation, and importance. Four of the eight other subjects mentioned that, following the session, they had some new ideas about example usage that they considered using in their work. Four subjects mentioned no new gain from the session. One of them, a developer with 10 years of experience, said that his way of programming is already too structured to change it now. He also mentioned that he always uses examples (so he does not object to the concept of using examples but rather to changing the way he works).

All subjects reported they already use examples frequently and extensively. Many of them stated that examples are an inherent part of their work.

Among the three responses to the follow-up survey, we received two responses from the four subjects who, in the first questionnaire, did not mention any new benefits. One of them stated that the talk helped him formalize some ideas that he already had and that it also help him legitimize "the notion that I’m not the only one doing that :)". The second subject shared some of his example usage patterns, but stated that he had been practicing them even before the talk. The third subject who responded to the follow-up survey mentioned that the talk brought example usage to his attention and that: "I should use it more. As I say, at the end of the day, we are all merely plumbers...."
Following is a summary of the example usage attentiveness aspect raised by our subjects in their reflection questionnaires and during the follow up correspondence. The data analysis process revealed two main themes: (1) attentiveness to new opportunities in which examples can be used, and (2) acknowledging example usage as a fundamental software activity. We now elaborate on these two aspects.

**New Opportunities**

The focus group subjects stated explicitly that they gained awareness of new opportunities in which examples can be used. The opportunities include new contexts in which the subjects were inattentive to searching for examples such as additional technologies for which examples may be beneficial, new scales of examples, and full utilization of the examples found (code reuse). These results highlight the key role that awareness and attentiveness play in extensive and effective examples’ usage. Merely discussing issues regarding effective example usage with developers provides them with new opportunities to exploit such usage in their daily work. These results also contribute to the validity of the observation analysis presented in Section 5.3. In what follows, we illustrate cases in which developers gained awareness of new example usage opportunities:

Several software engineers reported that being exposed to these ideas would promote their example usage in the future. One programmer says: "I was using examples before [the session], but from now on I will certainly be more conscious about them and use them more intelligently". Another programmer wrote: "Raising the level of awareness [of example usage] will increase example usage".

One programmer reported that he was more aware of the existence of better examples and of examples for more complex tasks than previously (similar to the analysis of Ben’s behavior in Section 0): "[I learned that] breaking down a programming task into steps and looking for an example for each step might be less effective than looking for an example for the entire task". Another programmer related to example retrieval, saying that he now knows that example searching techniques can be learned. It is reasonable to assume that this new insight will encourage him to look for examples that were previously out of his reach.

Another kind of new opportunity relates to full utilization of the example and was demonstrated by an experienced programmer who reported that following the talk he started using the example code as well, and used examples not only "as an inspiration".
Programmers reported that their example attentiveness was affected by the level of familiarity and understanding of the programming task. We observed however that the level of understanding affects programmers differently. Some programmers reported that they only use examples for tasks they are already familiar with "just to refresh their memory" [10]. Other programmers report the opposite: if they understand the task, they write the code and only use examples when they reach a deadlock or are at a loss as to how to tackle the task. Following is an illustrative excerpt from one reflection questionnaire: "I use examples frequently, however it depends on the nature of the task. Usually, I use examples for tasks that I don't know how to perform... I was particularly impressed by the idea of using examples for tasks that I know how to tackle".

In Section 4.3.3 we examine this issue further, and highlight the large variability existing on the various occasions in which examples are used. The main variables are: code reuse, the developer's state of mind (development mode), the example size, the example source and the learning factor. Indeed, the first three factors correspond to the three attentiveness aspects presented above: example context, scale and utilization.

**Fundamental Activity**

Two subjects referred to the conceptualization of example usage as a stand-alone activity: "Example usage requires a name of its own". Another developer said: "I was familiar with the examples and the explanations before. The interesting part was formulizing this activity [example usage] similarly to refactoring". This subject is referring to an analogy we made in the session in which we reviewed how conceptualizing the term 'refactoring' and acknowledging it as a fundamental activity in software construction influenced the software engineering ecosystem: tools, practices, software lifecycle, and training (see Chapter 7). A third subject only arrived at a similar conclusion during the follow-up survey 3 months later: "The talk helped [me] formalize some ideas I already had. I had been a learner-by-example for years but, as you know, putting a name to something makes it much more real and relevant."

Although most of the session focused on using code examples, some assertions related to other aspects of software construction as well. One subject said that from now on he would also use examples for documentation purposes and in client training. Two participants reported that they would establish an example-aware development process. Some subjects noted that the connection between effective example usage and other factors had become clear to them thanks to the session. Among the factors mentioned by the subjects were developer productivity, development speed, and code quality.
The above assertions can be explained as resulting from an increase in the abstraction level of the concept of example usage. As long as example usage was an implicit programming technique, these implications were not considered. However, considering example usage to be a fundamental software activity places it on the same abstraction level as other fundamental issues of software construction and facilitates and examination of the implications of example usage on issues such as productivity, speed, and quality.

One subject reported that the reflection questionnaire changed his opinion on example usage even more than the preceding session did. "The questionnaire highlights the importance of this issue", he said.

5.5 Summary

In this chapter we demonstrated how developers’ diversity with respect to example usage is manifested in their attentiveness to example usage. We characterized three types of lack of awareness or attentiveness related to context: technology, scale, and utilization. In the focus group case study we showed that a discussion and guided reflection process encourages developers, even experienced ones, to use examples more extensively and consciously, and increases their awareness of new opportunities in which examples can be used.

These results highlight the fragility of example usage; the behavior of our subjects, with respect to example usage, is diverse, implicit and context sensitive. It comes to mind that if enhancing example usage is desired, this fragility should be explicitly addressed through conceptualization and abstraction.
“Fewer things are harder to put up with than the annoyance of a good example.”

- Mark Twain

Chapter 6

**WHY PROFESSIONAL DEVELOPERS USE EXAMPLES?**

We conducted a Web survey to study the primary motivations and the centrality of example usage in professional programming. We obtained 192 responses from professional developers who were asked about their example usage objectives. The four most frequent answers were: a) learning purposes, b) implementation purposes, c) when working with an unfamiliar technology or library, and d) for specific tasks. We identified three axes that affect example usage motivation: a) the properties of the task, b) the development activity, and c) software engineering considerations. Below we discuss example usage motivation diversity, and the centrality of example usage to the development process of the survey participants. We conclude by situating our findings in the arena of software tools for code search.
6.1 INTRODUCTION

In previous chapters we illustrated several cases in which developers only use examples in certain contexts, but fail to apply them to others; additionally, some developers appear, even when employing the examples, only to do so in a limited way, thus failing in exploiting the full potential of these very examples.

In this chapter, we wish to examine this issue further by employing quantitative data as well. We focus on the following research questions:

- RQ1: Why do professional developers use examples? What are their motivations? What are the characteristics of the tasks for which examples are used? Is example usage central to software development?

In order to answer these questions, we have conducted an online survey in which we asked various questions regarding example usage. Among 311 survey submissions, 192 participants answered the question: For what purposes do you use examples? Could you think of other purposes for using examples? Most of these participants were experienced programmers (61% had more than 6 years of programming experience), and reported that they use examples in their work on a daily basis.

During the qualitative data analysis, three categories (axes) have emerged: examples are used based on the properties of the task (e.g. unfamiliar, specific, complex), the development activity (e.g. learning, implementation, solving problems, code comprehension) and the motivation of the developer to address software engineering aspects (e.g. speed, reuse).

In the following, we count how many motivations were mentioned by each participant. Developers who only appear to associate example usage with a single motivation may only be using examples in this context (e.g. only for unfamiliar technologies, or only for code comprehension). We argue that example usage purposes are diverse; for each axis, most of the participants relate the usage of the example with, at most, a single motivation.

These results support our thesis regarding the diversity of example usage. We find additional evidence that developers associate example usage with a specific development activity, which conforms to our previous results regarding the variability of example usage as described in Section 4.3.3. We also find that most of the developers who participated in this survey (69%) do not associate example usage with global software engineering concerns, but rather with the immediate context of the task at hand, which conforms to our previous results regarding the fragility of example usage (Section Error! Reference source not found.). Being...
able to show similar results by using independent data sets strengthen our confidence in these results.

Below we compare our findings to previous results by Sim et al. [132] of archetypal source code searches. The differences between our findings and those presented in [132] imply that although code search and example usage have many characteristics in common, developers associate examples with learning more than they associate code search with learning. We conclude this chapter by drawing potential implications of our work on tool builders for code search.

6.2 Methodology

The primary data-gathering tool in this research was an online survey [118]. The survey (available online⁷) contained no mandatory questions. Each subject was asked some personal details: name, email, gender and years of experience as a developer. We defined example usage as any use of an already existing code in the development process. We asked each developer how often he or she looks for examples in his or her work, and proposed several options for answering.

The remaining survey questions were open-ended. For each question, an unlimited text area was provided. We asked the developers about the techniques they use when searching for examples; we further asked them to provide tips for novice developers and also asked them about the challenges they face and how they overcome them. We also asked them about their example usage purposes, about the advanced techniques they have developed, as well as about their overall reflections on example usage.

Subjects that stated in the beginning of the survey that they do not use examples in their work were asked for both their reasons and what should be changed, to make them inclined to use examples more frequently.

6.2.1 Research Settings

The survey was published online using Google Forms technology⁸. We have solicited professional developers’ participation by posting invitations on social and professional networks, such as LinkedIn, Facebook, and DZone. Over a period of four months, we had a few thousand entries to the online survey; however, probably due to the in-depth survey, which involved mostly open-ended questions, only 311 surveys were actually submitted.

---

⁷ http://tinyurl.com/usingExamples

⁸ http://www.youtube.com/watch?v=IzgaUOW6GIs
In 280 surveys at least one question was answered (31 filled their personal details only). In 51 of them, only the question regarding the frequency in which the example usage is used was answered (all other questions required typing). This leaves 229 questionnaires with meaningful qualitative data (total word count: 35,000 words).

In this chapter, we focus on the answers to the questions: For what purposes do you use examples? Could you think of other purposes for using examples? These questions were answered by 192 participants (word count: 4000 words).

6.2.2 RESEARCH PARTICIPANTS
Of the 192 participants, 93% were male and 6% female (not all of the participants answered all the questions). About half of the survey participants have 10 years of experience or more (See Figure 6.1).

![Figure 6.1: Years of experience as developer](image)

Most of the participants, as may be observed in Table 6.1, use examples at least once a day. Seven developers stated that they do not use examples in their work; however, the question regarding their example usage purposes was not presented to them, and they are therefore excluded from this data set.
Table 6.1: Frequency of example usage among research participants

<table>
<thead>
<tr>
<th>Frequency</th>
<th>No. of participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>All the time, every few minutes</td>
<td>24</td>
</tr>
<tr>
<td>Once an hour</td>
<td>34</td>
</tr>
<tr>
<td>Once a day</td>
<td>69</td>
</tr>
<tr>
<td>Once a week</td>
<td>28</td>
</tr>
<tr>
<td>Less than once a week</td>
<td>16</td>
</tr>
<tr>
<td>Other</td>
<td>21</td>
</tr>
</tbody>
</table>

### 6.2.3 DATA ANALYSIS
The analysis process of the qualitative data we used in this research includes the following steps, which were adapted from [139]:

1. All the statements in the survey were conceptualized, that is, each idea that appears in the text was summarized in one sentence, and added to a themes’ document, with an attached reference to its source. Related ideas were associated by using a common parent term, thus forming a tree structure.

2. After phase 1 was over, we revisited all the answers that were given to the question about example usage purposes. Three categories (axes) became apparent: task properties, activity types and software engineering considerations.

3. We went over all 192 answers that were given to the question regarding the purposes of example usage, and tagged each answer according to the three axes (coding). In one case, tasks that were considered specific, we used data from step 1 of the analysis process (most specific examples were given as answers to the question regarding advanced techniques).

4. In order to count tag occurrences, we incorporated all the data in a spreadsheet: each row represents a participant, and each column represents a tag.
6.3 **Example Usage Purposes**

Developers use examples for various purposes and in a variety of ways. It is sufficient to examine the verbs that are used by the survey participants in order to appreciate the wide spectrum of activities and motivations that example usage entails: learn, do, solve, use, test, implement, understand, discover, compare, explore, write, find, find out, familiarize, clarify, figure out, improve, see, get, read, make, avoid, search, adapt, study, refactor and more.

Each answer was examined to discover whether it is related to: (1) the properties of the task at hand: whether the task is considered to be simple, new, complex etc; (2) the type of the activity at hand: whether it is a task of implementation, design, code comprehension, etc, and (3) software engineering aspects of example usage: speed, quality, reuse, etc. We did not start with these categories *a-priori*, but rather they became apparent during the data analysis process. Nevertheless, these categories conform to our previous findings (see below) and support our thesis regarding the diversity of example usage. We note that most of the developers in our survey (74%) associate example usage with an activity type or a state of mind (in Section 6.5.2 we show that most of them associate it with a single one). These developers use examples based on immediate context. By contrast, we find that only 31% of the developers mention example usage in a wider context of the software development process. This also supports our previous conclusion, described in Chapter 5, that attentiveness is a dominant factor in example usage, and that example usage is context sensitive.

In their answers, some subjects related to more than one category, or to more than one item in a particular category. Figure 6.2 is a typical survey answer. It was coded with: speed, reuse, common and unfamiliar.

```
Whenever I have a non-trivial task that smells like something other people must have encountered before. When I need to get into unfamiliar technology quickly.
```

*Figure 6.2: Example of participant's answer*

Furthermore we elaborate on each of the axes and their corresponding tags. For each axis we mention in parenthesis the number of participants who addressed it.

### 6.3.1 Task Properties (94)

In the following, we classify developers' motivations by virtue of the task properties for which the examples are used, as perceived from the developer's perspective. We demonstrate each property using direct quotes of the survey.
participants. Table 6.2 summarizes the number of participants addressing each task property.

**Table 6.2: Task properties associated with example usage**

<table>
<thead>
<tr>
<th>Task property</th>
<th>No. of participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unfamiliar</td>
<td>50</td>
</tr>
<tr>
<td>Specific</td>
<td>50</td>
</tr>
<tr>
<td>Complex</td>
<td>16</td>
</tr>
<tr>
<td>Common</td>
<td>7</td>
</tr>
<tr>
<td>Hello World</td>
<td>5</td>
</tr>
</tbody>
</table>

**Unfamiliar, new** – Example usage is associated with tasks that are new or unfamiliar to the developer, such as the use of a new API or a new technology. Among the unfamiliar things that the developers mention, the following seem most prominent: programming languages, tools, methodologies and legacy code.

Developers report that they familiarize themselves with the object in question by dint of exploring examples, and performing pre-studies. Familiarization is different from learning (See Section 6.3.2); the aim of this activity is not to learn or to understand, but rather to obtain an overview, to come to grips with the best practices or to just get a sense of it, as explained by the following participant: “[I use examples] mostly to get an initial familiarity with a broad API I don’t know yet. Given 300 functions in a flat API, it’s sometimes hard to focus on the main/important ones. A good example helps identify those, and lets you save the rest for later study”. Some developers report that by browsing multiple examples in this process, they are exposed to new ideas, which assist them in identifying best practices as well as common pitfalls to avoid.

Examples are known to be a useful tool when working with an unfamiliar API [9][140]. Indeed, in the literature, we find that examples are particularly useful for learning unfamiliar tasks based on cognitive principles of learning from examples and problem solving by analogy [121].

**Specific** – Some developers said that they use examples in order to perform specific tasks, such as XML parsing, IO operations or list manipulations. They have associated these tasks with example usage and accordingly, every time that they need to perform these kinds of tasks, they refer to their examples. Among the
forty-nine examples of specific tasks, we find language specific tasks, framework specific tasks and operation specific tasks.

We also noted this phenomenon (of associating example usage with a specific task) during our observations, described in Chapter 5 – a developer who was using examples extensively in a specific context (Python programming), avoided using examples altogether in another task (tool configuration), in which examples could be just as useful.

In the context of example Web search, looking for a specific example could be considered a special case of the navigational Web search proposed by Broder [27]. He came up with a trichotomy of Web search “types”: navigational, informational, and transactional. Navigational searches are those that are intended to find a specific Website the user has in mind.

We note that there is hardly any intersection between ‘unfamiliar tasks’ discussed above and ‘specific tasks’, which are typically tasks that the developer performs multiple times (thus, they are no longer unfamiliar). It seems that both kinds of tasks are popular among developers for using examples, however, in Section 6.5 we show that not many developers use examples in both cases – another example of the diversity of example usage.

**Complex** – Developers use examples for tasks that they consider to be complex, complicated, advanced, and abounding with details: “Examples are these small bites of code you will never remember in full. For example usage of HTTP request in PHP or usage of any module of any CMS system” (this quote was also coded as ’specific’). Especially if the task involves many technologies and languages in which the developer is not versed. Examples were also reported to assist with forgotten details in [25].

**Common** – Developers look for examples for tasks that they consider simple, small, basic or common, as they think that these cases are more likely to have proper examples and be easier to find: “[I use examples for] implementing something general that ‘someone must have implemented already’”. These examples are short, containing only few lines, so that it is easier to understand them in relation to, and to combine them with, the existing code.

Looking for examples that are perceived to be common or basic could be explained from the perspective of the developer’s awareness of the example’s existence. Even if the developer has not encountered this specific example before, he or she is confident that it appears in a tutorial or online documentation site.
We note that many of the common tasks are considered to be simple, while many of the complex tasks discussed earlier are considered to be complicated. Although the dichotomy is not as clear as with the unfamiliar-specific case (there are tasks that are considered both complicated and common) we find it to be another example of the diversity of example usage.

**Hello World** – Developers use examples to have a working prototype – "Hello World". When starting a new project, the developer wants to be sure that there is no problem with either the infrastructure or the configuration of the platform, so that when some logic is implemented, it can be run error-free: “[it is] hard to remember [the] last time I started a project from scratch”. Hello World applications take care of details that developers tend to forget, due to the fact that bootstrapping a project is rare. In some cases, the framework or the development environment provides the developer with basic "Hello World" applications. This kind of example usage was also identified in a previous analysis of the variability factors of example usage in Section 4.3.3.

The use of ‘Hello World’ examples is implicitly encouraged by the way programming is being taught. The "Hello, World" program has been a staple of introductory computer programming classes ever since Kernighan and Ritchie's C book [82] was initially published in 1978 [152]. Exposing novices to "Hello World” programs early in their training encourages using rapid prototyping, the learning from sample code—or just using it [103].

6.3.2 **Activity Type (142)**

In the following, we outline the purposes for example usage, according to the activities performed by professional developers during their work: implementation, design, learning, understanding and self-improvement. This list digresses from the previous one (Section 6.3.1), as it focuses on the cognitive skills of the developer rather than on the specific properties of the task. This might suggest that developers associate example usage with a certain state of mind, which conforms to our previous findings (Section 4.3.3) and was also noted in our field observations (Section 0). Some of these activities may be associated with software life-cycle phases, even if the outlining is incomplete. Only one participant reported that he uses examples in the process of requirement analysis and client negotiation; therefore, we did not include it in the table. Testing was also mentioned by a single participant only.

Table 6.3 summarizes the activities that are associated with example usage. In the table we visually distinguish between software engineering related activities (implementation, problem solving and design) and activities associated with the individual (learning, comprehension and self improvement). This dichotomy
somewhat corresponds to the distinction between the organizational goals and the personal development discussed in Section 4.4.1. We also note that four of the six items on the table (not including ‘implementation’ and ‘design’) correspond to the variability factors of example usage discussed in Section 4.3.3.

One might find this division to be arbitrary. The distinctions between some of the activities are extremely fine. In using examples to solve problems, isn’t the programmer learning something? In comprehending, he/she is also learning. Doesn’t self-improvement also involve learning and improving problem-solving skills? (Otherwise, what is being improved?).

However, the value of these distinctions is in manifesting how developers explicitly describe their perception of example usage. A developer who chooses to mention using examples for self improvement only (e.g. regularly, one hour a week), might not use examples to learn the underlying technology involved in the task on which he or she is currently working at work, or vice versa. Although in both cases examples might be used in the same way, this developer may associate this course of action with a specific activity and avoid using examples in others. Although some of these activities overlap, mentioning only one of them implies inattentiveness to the others. We further address this issue in the limitations section below (Section 6.5.4).

<table>
<thead>
<tr>
<th>Activity type</th>
<th>Activity</th>
<th>No. of participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software engineering activity</td>
<td>Implementation</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Problem solving</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>Design</td>
<td>33</td>
</tr>
<tr>
<td>Individual activity</td>
<td>Learning</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>Comprehension</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>Self improvement</td>
<td>6</td>
</tr>
</tbody>
</table>

**Learning** - Examples are used to learn many different things: language syntax, technologies and tools, programming methodologies, techniques, and concepts. Learning is associated with new things. Learning is achieved when the developer is in a 'learning mode', within the ambience of studying, but also ad-hoc – in the context of a specific task.
6.3 Example Usage Purposes

One interesting observation is that, for many learning tasks, some of the developers mentioned the performance of the tasks themselves, while neglecting the learning aspect in their report. For example, some developers wrote: "to solve a problem I have", while other developers stated: "to learn how to solve a problem I have". Although it seems to us that, in several cases, the two versions were used interchangeably, the learning dimension of performing a specific task calls for a further investigation.

On the same note, some developers stated that they work in many technologies simultaneously. They said that they are using examples in order to 'avoid the need to learn' the specific syntax of each technology.

Learning from examples was investigated substantially in the literature. Renkle [123], for example, found that successful learners tend to employ more principle-based explanations, more explication of operator-goal combinations, and more anticipative reasoning. In addition, he identified two types of effective learners, labeled anticipative reasoners and principle-based explainers. Since example usage encourages self-explanation [34], it is not surprising that learning was the most popular example usage motivation.

**Implementation** – Developers use examples when they code. They look for already existing examples of the required functionality, or for examples that use the technology, language or API in which they work. During data analysis we felt that the implementation activity was implied in additional cases, but we used it exclusively on explicit ones, such as this one: “I use them when I write new code”.

Using examples for implementation purposes is the essence of several pragmatic reuse approaches. Pragmatic reuse tasks (also called code scavenging) have been shown to be effective [90] [64]. One prominent example for an example extensive software development method is the OSSD [110]. It is an approach in which developers meld together software pieces (examples!) that they have found.

**Problem solving** – Examples are used when problems arise. Developers associate them with difficulties, errors, challenges, fixing things and resolving issues: “I use them for two reasons, to see how other people solve some kind of new problem that arises in front of me, or to compare other ways of solving problems I already had”. For some developers, example search is not a default behavior (rather, they would usually search for documentation or start coding), so that when a problem occurs, they are already out of their ‘comfort zone’ and are not acting automatically. Many developers report that finding an example is often quicker than referring to documentation.
In the literature, examples were found to be very useful in problem solving [34] and there were reports that the use of worked examples could substitute for problem solving in learning algebra [141]. It is therefore, no wonder that many developers use them as a tool when needing to solve problems in their work; they have already encountered examples in the context of problem solving.

**Design** – Examples are used to assist in design decisions - to provide ideas and inspiration with a view to structuring the code, applying design patterns or imposing a specific architecture. In many cases, these design decisions involve exploring multiple examples and choosing between alternatives: “Examples of other developers gives a different point of view on the problem I’m working on. Also you can learn other techniques of development. This helps you to create your own style - by filtering mostly interesting ideas and techniques. Examples also give you opportunity to get more experience - you see other projects, other problems, other strategy for design and different solutions of same problem”. Design activities include the remodeling of the existing design (refactoring), or changing the original approach that the developer had in mind before reading the example.

Examples were found to be useful in the process of design comprehension [65]; they assist the developer in building a mental model of the system [147][127], since they also play an important part in the processes of reading and writing design documents. This fact explains the popularity of using examples for design purposes as may be seen in Table 6.3

**Comprehension** – Examples are used for software comprehension [147]; to understand how things work; to understand code, syntax, API usage, tools, theory and methodology; to understand a design, the requirements for the task in hand and the documentation: “I use examples to try and understand the overall design of a system and build a mental map between expressions I do not understand to expressions I do understand”.

Comprehension is different from learning (though in our survey some of the participants used the terms interchangeably): Comprehension is usually used in the context of a specific code snippet – what the code does or how it works. Learning is related to something more general. Some of the survey participants use examples in order to learn new things. Even if they have a specific programming task at hand, they would not use the example before they have understood what it does, but also not before mastering the general principle demonstrated by the example.
Michelene et al. [34] found that comprehension is achieved through self-explanation. When reading examples good students generate many explanations that refine and expand the conditions for the action parts of the example solutions, and relate these actions to principles in the text. This self-explanation is guided by an accurate monitoring of their own understanding and misunderstanding.

**Self improvement** – Some developers read examples habitually with no specific task at hand; they only use it in order to train themselves and to improve their skills in terms of reading code and learning new things. In this way, they keep themselves up to date with the latest technology and common language idioms: “Mostly learning, and improving my problem solving mentality. Examples can provide different perspectives than those you already have, so they can give you an additional insight into your problem space and your solution space”.

Robert Martin tells novice developers in his book [104]: “What kind of work will you be doing? You’ll be reading code. Lots of code. And you will be challenged to think about what’s right about that code, and what’s wrong with it.” Martin addresses the same issue addressed by our survey participants – reading code written by others (examples!) is a key tool for professional self-improvement.

### 6.3.3 SOFTWARE ENGINEERING ASPECTS (60)

Twenty seven percent of the participants answered that they use examples due to software engineering concerns, such as: speeding up development time, enhancing code reuse and increasing code quality. Table 6.4 summarizes the distribution of these answers.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>No. of participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse</td>
<td>32</td>
</tr>
<tr>
<td>Speed</td>
<td>28</td>
</tr>
<tr>
<td>Quality</td>
<td>8</td>
</tr>
<tr>
<td>Pragmatism</td>
<td>7</td>
</tr>
</tbody>
</table>

**Reuse** – By using examples, there is, in fact, a reuse both of others’ code and design. Examples serve as a community-wide knowledge manifestation, through the publicity of the examples, the standards and the best practices of the industry.
It assists in maintaining standardization in coding conventions and design. Using examples is perceived as “standing on the shoulders of giants”, as learning from past successes and errors, and as “avoiding reinventing the wheel”.

**Speed** – Using examples correctly may speed up development time and learning time. Many developers find that it is faster to see an example than to read the documentation: “to speed up coding, to spend less time reading and understanding documentation, and copy pasting existing code to see if it works”. Examples appear to accelerate the rather technical bootstrapping process at the beginning of a new project. Development speed was also one of the themes in the virtual discussion group case study described in Section 4.4.1.

**Quality** – Some of the examples are of codes of high quality and a substantial record of proof. Some of them are taken from production code, which was thoroughly tested and actually used in live systems. Public examples on the Web were reviewed by the community and their stumbling-blocks are corrected or documented. Notable is the Stack Overflow Website⁹, which provides a ranking mechanism both for developers and for developers’ answers.

**Pragmatism** – Some developers use examples to improve their own productivity as engineers, and highlight the pragmatic aspect of it (sometimes out of laziness). By using examples habitually they are able to deliver quickly, and better serve their company, or as summarized by one of the participants “to get things done”. In contrast to the ‘self improvement’ category mentioned in Section 6.3.2, we specify ‘pragmatic’ example usage performed *ad hoc* in the context of a specific task.

In a survey by Holmes and Walker [64] they asked industrial developers about their approach to pragmatic reuse. The most popular reason for reusing source code was so the developers could save themselves time. This was backed up repeatedly in the written questions with comments such as “reusing code is quicker and easier than starting from scratch”. The next major reason for these reuse activities was to increase the reliability of their code. The developers wanted to ”leverage existing testing”. Code was more desirable if tests existed for it since they increased the developers’ trust in the quality of that code.

### 6.4 Example Usage vs. Code Search

In Table 6.3 we list the software activities that are involved in example usage. We want to compare these results to the motivation for searching, as suggested by Sim *et al.* in [132]. In their survey analysis (69 participants, 111 scenarios are

---

provided by the participants and 94 search motivations are identified), they found that the most frequent motivations for code search were (in descending order): (a) defect repair, (b) code reuse, (c) program understanding, (d) impact analysis, (e) maintenance, and (f) feature addition. Moreover, the program understanding and maintenance motivations were used as little as possible because "it could be argued that all searches were performed for that purpose". When we analyzed our surveys, we felt particularly that the implementation motivation was implicit in the participants' answers.

The motivations of learning, design and self improvement are absent from their list. The implementation motivation, which corresponds to the feature addition, is located at the top of our list and the bottom of theirs.

Sim et al. also asked their participants "How useful is it to search source code when..." along with a list of ten activities from the software development cycle, and asked respondents to give a rating on a scale of one (low) to five (high). It was found that the tasks in which searching was most useful (median rating 5) revolved around the repairing of bugs or defects, the understanding of old code, and the addition of a new feature to the old software. All three activities were found at the top of our activities' list as well (i.e., after learning).

Some factors may have affected the variations in results mentioned above: firstly, the two surveys asked different questions: we asked about example usage, while Sim et al. asked about code search; secondly, it is apparent that the context was biased: most participants in [132] referred to local code search, whereas most of our participants were referring to Web search (117 participants had mentioned that explicitly) [54]; thirdly, in the multiple choice questions in [132], there are only 'classic' software life-cycle activities, which may also have influenced the participants to mention these activities only.

These differences may suggest that although code search and example usage have many characteristics in common, developers associate examples with learning more than they associate code search with learning.

6.5 Using Examples for Multiple Purposes

Below, we count the number of participants who relate to multiple items in the same category in their answer, e.g. use examples for both implementation tasks and design decisions, or while taking into account both quality aspects and time saving aspects. If a developer uses examples for multiple purposes, one may assume that examples are inherent in his or her work. Such a developer is explicitly aware of the multiple uses and benefits embodied in extensive and
habitual example usage, and might actually perform example-centric development.

In previous chapters we described cases in which example usage is context sensitive, and developers who use examples in certain situations avoid using them in others. In this analysis, we seek to obtain quantitative results to support or disprove our qualitative observations. We use this analysis to study both on the *centrality* of examples in the development process (i.e. developers who use examples for multiple purposes), as well as the *diversity* among developers in their example usage (i.e. having no dominant usage purpose mentioned by most of the developers).

Figure 6.3 summarizes the number of the participants, who considered multiple items of each category mentioned in Section 0.

![Figure 6.3: number of the participants, who considered multiple items in each category mentioned in Section 0.](image)

6.5.1 **Multiple Kinds of Tasks**

Developers who appreciate example usage as a fundamental software activity would take a proactive approach in order to use examples for many kinds of tasks: whether the task involves an unfamiliar technology (so they would look for relevant examples using this technology) or a familiar one (so they refer to their own previous usage, or to examples they have previously encountered).
Furthermore, habitual example users would use examples for both simple and complex tasks, for bootstrapping as well as for intermediate tasks.

Figure 6.3 shows that 64 participants (68% of the participants that addressed task properties) mentioned that they use examples for a specific type of task. Most of the survey participants (101 participants, 53%) do not associate example usage with any specific type of programming task. Only thirty participants (16%) use it with respect to multiple task types (See Figure 6.4).

![Figure 6.4: distribution of participants by the number of task properties they mention. Categories are taken from Table 6.2.](image)

These results suggest that:

1. Example usage is **not central** to the development process of most of our subjects, or at least they are **not explicitly aware** of the multiple purposes it serves.
2. Our research subjects are **diverse** with respect to their example usage purposes – different developers find examples useful for different non-overlapping purposes. The most dominant type of tasks that developers use examples for are unfamiliar tasks and specific ones (which are inherently
6.5 Using Examples for Multiple Purposes

diverse), each of them are mentioned by 50 developers (Table 6.2), who constitute 26% of the total number of participants.

6.5.2 Multiple Activities

Similar argument to the above could be made with respect to developer activities. Developers who acknowledge that example usage is fundamental to software development, would have mentioned it with respect to various kinds of activities: both ad-hoc, such as implementing, designing or solving a specific problem, as well as general, long term purposes, such as learning and self-improvement.

An interesting data point is that more developers mentioned example usage with respect to any software activity (142) than developers who ignored activities altogether (50). This was not the case for the task property criteria. This suggests that developers do associate examples with some kind of software activity.

It can be seen in Figure 6.3 that most of the participants who considered software development activities (85 of 142) associate example usage with a specific one. Thirty percent of the developers reported that they use examples in more than one type of activity. Figure 6.5 reveals that for some developers, example usage is merely perceived as a learning technique (13%), while others use it *ad-hoc* for implementation tasks (17%).

![Figure 6.5: distribution of participants by the number of development activities they mention. Categories are taken from Table 6.3.](image-url)
6.5.3 Software Engineering Aspects

Some developers use examples because it is pragmatic: the examples deliver the required functionality and save them time. Eleven (11) participants actually used the phrase "reinventing the wheel" to describe their approach. However, an extensive example usage is sometimes perceived as not legitimate and the developers who subscribe to such an approach are devalued as we discussed in Section 4.4; ill-recognition which, in turn, sends a mixed message to the pragmatic programmers, and can give rise to self-doubt and to the question whether what they are doing is on the mark or not.

Figure 6.3 shows that sixty-nine percent did not address software engineering aspects (such as speed, quality, reuse) at all. Seventy-five percent of the developers, who mentioned software engineering aspects, only addressed one aspect. These results suggest that most of our participants perceive example usage as a technique and do not consider (at least explicitly) its implications on software engineering as a whole.

6.5.4 Limitations of Analysis

In the above analysis we suggest that example usage is more diverse than, and not as central as, one would expect from experienced developers that reported using examples frequently (30% of them reported that they use examples at least once an hour and 13% every few minutes). However, we did not actually observe each of these developers to see whether they might use examples for additional purposes, other than the ones reported in the survey.

One may argue that the diversity argument does not lay in example usage purposes, but rather in attentiveness to example usage purposes, and that the real matter being investigated here is how explicit example usage is in the development process. Another threat to the validity of these results is the verbose nature of this survey (participants might have been too lazy to mention multiple purposes).

We wish to address some of these arguments:

- The wording of the question - “For what purposes do you use examples? Could you think of other purposes for using examples?“ - was meant to encourage the survey participants to address multiple purposes: We asked about purposes (in plural) and suggested that there might be additional purposes than the immediate ones that come to mind.

- The answers to this question were verbose – total word count of 4000 words (21 words per answer on average), and for this analysis we only considered
participants who chose to answer this question (there were another 37 participants who left this question unanswered).

- We surveyed a substantial number of experienced developers (though not as many as recently reported by Gorschek et al. [59]).

6.6 IMPLICATIONS

Another contribution of this study is in the area of code search. Tool builders for code search [3][8], who are focused on finding the best code snippets, might neglect the context in which this search is performed [74]. In this research, we provide such a context – of the task or the activity at hand. Different kinds of task or different kinds of activities may benefit from different features of the search tool.

For example, to address the fact that many of the tasks involving example usage were already encountered by the developer before (Table 6.2) – an IDE bookmarking mechanism might be considered. Furthermore, as common tasks and Hello World tasks are associated with example usage, tool builders might consider augmenting their tools with a catalog of common tasks that would trade search with browse, and a set of Hello World prototypes.

In addition, from the activity type perspective, most code search tool builders focus on finding code for implementation purposes. While this is indeed a primary purpose, other purposes are neglected. For example, a code search tool for design decision purposes, might seek to provide tools for code navigation and dependency browsing. Learning purposes could be addressed by a code annotations’ mechanism, in addition to – and independently of the example code (inside the IDE or the Web browser).

6.7 SUMMARY

In this chapter, we have investigated the primary motivations of example usage as seen in professional programming, by means of an online survey. We identified three axes that affect the developer’s motivation and discussed these results with respect to code search research.

We further examined the diversity in example usage motivations and the centrality of it to the development process. We suggested that developers who address only a single item within an axis might also actually use examples in a narrow way, or in a limited context, or at least may not be attentive to additional example usage scenarios.
“Good programmers know what to write. Great ones know what to rewrite and reuse”

(The cathedral and the bazaar, E. Raymond)

Chapter 7

**EXAMPLE EMBEDDING ECOSYSTEM**

In this chapter, we coin a new term – example embedding (EE) – to highlight the methodological and disciplined use of examples in software construction. We suggest that EE is a fundamental software engineering activity (and not merely a programming technique), and propose to consider a comprehensive approach that would leverage its full potential, and address its current diversity. We examine two case studies, one of them is the advances in software engineering due to the conceptualization of the refactoring activity, and the other is the mechanisms used by the academic community in order to standardize the citations of previous work in academic publications to enhance the referencing of related work.

The approach described in this chapter is motivated by the empirical research reported in this dissertation so far. However, as opposed to the previous chapters, this one is more argumentative in nature, and portrays different paths for future work.
7.1 Introduction

In the previous chapters we investigated the diversity of example usage in professional software development. We found that the developers do not consensually appreciate example usage benefits, the software development community does not address the human challenges that accompany extensive example usage, and there is no explicit, widely accepted set of best practices for using examples methodologically or systematically.

Many benefits reside in systematic and habitual example usage for both software companies and individuals. These benefits include: enhancing software reuse [72][80], increased productivity [121], improvement of code quality [101], enforcement of consistency of design [55][109] and of coding standards [22], and the establishment of an effective knowledge transfer mechanism both inside and outside the organization [126].

Recent pragmatic reuse approaches such as OSSD [110] aim at giving code examples a central place in the development process. However, despite reports of the increasing popularity of these approaches [24], we demonstrated that they still need to address several social and human concerns in order to be consensual. Furthermore, we examine ways to address these issues in order to leverage example usage’s full potential in software development.

In Chapter 4 we examined how the Q&A Website Stack Overflow addresses developers’ human concerns, and we suggested taking a similar approach in the examples arena too. In Chapter 5 we described a case study in which we raised the level of developers’ attentiveness to example usage, and received encouraging results. In Chapter 6 we identified developers’ motivations for example usage and proposed addressing them in software tools for code search.

Our three suggestions adopt various tactics to tackle different aspects of the same problem – the diversity of example usage. Each of these tactics by itself would not be enough. It appears that a comprehensive approach is needed. In what follows we portray such an approach.

In order to leverage the full potential of example usage we examine leveraging the refactoring software activity from a retrospect of 18 years, since it was conceptualized in William Opdyke’s Ph.D. dissertation in 1993 [114]. Refactoring is a methodological and disciplined technique for restructuring code; similarly, we wish to define Example Embedding – a methodological and disciplined technique for using examples in software construction.
The approach described in this chapter is motivated by the empirical research reported in this dissertation so far. However, as opposed to the previous chapters, this one is more argumentative in nature, and portrays different paths for future work.

### 7.2 Software Activity

We define software activity to be a collection of fine-grained techniques, which together assemble an abstract key notion in software development. The perspective of software activity allows us to leverage notions and abstractions from evident fine-grained techniques in software development work to form new practices and tools. Some of the already acknowledged software activities, such as coding or debugging, are composed of other finer software activities. For example, refactoring is one of the software activities that compose the coding software activity.

Refactoring is defined [47] as a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. We argue, however, that refactoring is much more profound than merely 'a technique': we consider refactoring to be a software activity. Refactoring is fundamental to software engineering; it is a conceptualization of the key principle that software is not only written but also rewritten. This is an inherent part of the work, and emerges in various forms and scales, from renaming local variables, to system wide architectural change. Although refactoring code has been performed informally for years, William Opdyke’s 1993 Ph.D. dissertation [114] is the first known resource that specifically examines refactoring.

The mere identification of refactoring as an activity, in addition to its naming and definition, promoted the following important processes:

- Firstly, it laid the foundations for others to build a catalogue of various examples of refactoring [49], some of which are also available online [48], prescribing how to apply them properly and discussing their subtleties. These catalogues also contributed to raising the level of abstraction in software design. The various refactorings serve as design building blocks, and allow a designer, for example, to refer to 'extract method' without needing to explain it.
• Secondly, it enabled the development of software tools that would systematically apply various refactoring activities on existing code and ensure their correctness\(^{10}\).

• Thirdly, it influenced coding practices, the way in which programmers write code, using, for example, test-driven development techniques [17].

• And fourthly, it affected the software construction lifecycle by allowing the design phase to be incorporated into the coding phase, as manifested in the agile methodologies [19], and by legitimizing time allocation for activities that improve the code without adding functionality.

These advances would not have been possible had the refactoring activity not been extracted from the various activities that constitute the coding phase.

We define the Example Embedding software activity to be the disciplined use of already written code in the process of writing new code. If such conceptualization would encourage similar advances in methodology and technology, they could together leverage the full potential of example usage in software construction.

### 7.3 Example Embedding Software Activity

Let us now examine the various aspects of software activities with respect to example usage.

#### 7.3.1 Attentiveness

The mere naming and definition of a new software activity in itself increases the level of abstraction in the software engineering discourse, and raises the developer’s level of awareness of the existence of a new concern.

We coined the term example embedding rather than simply using the intuitive example usage term, in order to highlight the methodology and discipline that should be borne in mind when using examples.

The definition and characterization of EE raises the programmers' level of awareness of code reuse; it is not only a programming language feature but should also be assimilated as common practice. We argue that reuse should not be limited to the use of code as is, but should rather be applied more freely, as long as the resulting unit is tested thoroughly. The difference between our approach to reuse and the conventional black box reuse [83] is analogous to the difference between refactoring and thorough design. Specifically, as refactoring liberated us

from the need to go through an elaborate design phase, EE aims at liberating us from the belief that we need to reuse the code as-is, packaged as methods or closed modules.

The motivation to explicitly address developers’ attentiveness stems from our case study described in Chapter 5. We showed that developers might miss opportunities to use examples not because they oppose using them, but rather because of unawareness of their existence or inattentiveness to using them. One possible reason for this inattentiveness is that reuse is being performed implicitly and is context dependant. By extracting this concern, and giving it a name, example usage may be enhanced and performed in new contexts. Attentiveness would be achieved because example usage would become explicit.

Moreover, the focus group case study described in Section 5.5 shows promising results about how example attentiveness could be increased; even a relatively short session with experienced developers contributed to raising the level of awareness of most of them to new opportunities for using examples (11 of 15).

7.3.2 Recipe Catalogue and Example Repositories
We use the term recipe catalogue to describe a detailed and comprehensive description of all known instances of some software activity. [56] presents a good example of a catalogue book regarding design patterns. [47] offers a comprehensive online catalogue for refactoring. As with cookbooks, just as different dishes require different levels of attentiveness, so do software activities.

We identify two major tasks in building such a recipe catalogue: the first task is to classify and categorize the differences between the various kinds of instances of the software activity; the second task is to describe the recipes – a comprehensive discussion of each, highlighting its subtleties, best practices and common variants.

More specifically, in Section 6.3.2 we discuss several types of example usage based on the activity type in which it is involved, such as implementation, problem solving, code comprehension and design. One could think of different principles that are applicable to using examples in each of these contexts. An example usage catalog may provide a categorization of the examples of the various types, and may also highlight relevant practices for each of these different scenarios.

Although we are not aware of example catalogues in the sense described above, there are several example repositories. Some of them deal with a specific technology and are centrally administered such as SWT\textsuperscript{11}, Google Chart Tools\textsuperscript{12}.

\textsuperscript{11} http://www.eclipse.org/swt/snippets/
and JavaScript\textsuperscript{13} while others are general, community example repositories\textsuperscript{14,15,16}. The community repositories leave much to be desired – their examples are neither structured nor classified. A proper classification, though, requires further investigation.

7.3.3 \textit{Practices}

We define a \textit{software practice} as \textit{sequence} of software activities. As opposed to software activity, which is declarative in nature, software practice is imperative.

For example: \textit{test driven development} \textsuperscript{[17]} is a software practice comprising testing, coding and refactoring software activities, orchestrated following the rationale that testing should precede coding to ensure test coverage, and that coding should be followed by refactoring to promote incremental design. Figure 7.1 depicts the test driven development practice in an automaton notation. Such notation emphasizes the "test first" principle – each Coding state is preceded by writing the corresponding tests. It also suggests that a development cycle is not completed unless refactoring was performed by putting Refactoring as the only accepting state. It should be noted that applying the tests is implicit in this diagram, and is performed as part of the Coding phase; coding is only completed when all tests pass.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{test-driven-development.png}
\caption{Test driven development practice.}
\end{figure}

On the other hand, one could think of another software practice, say \textit{refactoring driven development} (See Figure 7.2). In this case, the same three activities – testing, coding and refactoring – appear, but they are orchestrated differently.

\begin{itemize}
\item \textsuperscript{12} http://code.google.com/apis/visualization/documentation/examples.html
\item \textsuperscript{13} http://www.w3schools.com/js/js_examples.asp
\item \textsuperscript{14} http://www.refactory.org/
\item \textsuperscript{15} http://www.exampledepot.com/
\item \textsuperscript{16} http://snippets.dzone.com/
\end{itemize}
(testing here includes both writing the test and applying it). The rationale behind such orchestration is as follows: every code change should be preceded by a refactoring that would establish the proper conditions required for introducing the new code. In addition, every code change would be followed by a test to check its correctness.

![Diagram of development practice](image)

**Figure 7.2: Refactoring driven development practice.**

We borrow the term orchestration from the Web service domain terminology [116]. In this domain it is common to speak not only of orchestration but also of **choreography**, which is more collaborative and allows each involved party to describe its part in the interaction. Indeed, in both practices described above, the dichotomy between the phases is not complete. In practice – refactoring, coding and testing are interleaved even if the developer is following a particular development practice. However, raising the level of awareness of the developer that a specific practice is composed of some smaller building blocks promotes the rationale of this practice.

In order to find corresponding practices to EE, one needs to identify the building blocks of example usage, and sequence them. As part of the survey described in Chapter 6, the developers were asked about their example usage practices. A preliminary analysis of their answers reveals at least 12 distinct (though related) building blocks, which are presented below:

- **Browse**: browsing example code (e.g. when example search yields multiple results), looking for relevant snippets, and filtering non-relevant ones (e.g. matching programming language, compatible license, etc.).
- **Comment**: adding comments to the example found.
- **Learn**: using the example to learn a technology, programming language, syntax, API, programming techniques etc.
- **Modify**: modifying the example found to match the specific context. The modification may change the behavior of the code.
7.3 Example Embedding Software Activity

- **Refactor**: changing the structure of the example or of the code in which it is to be incorporated, without changing its behavior (e.g. design changes, applying coding conventions, etc.)
- **Run**: running the example may be used to make sure it is executable and may enhance developers’ understanding by providing immediate feedback. There are various activities involved in making the code run: compiling, deployment, build, etc.
- **Search**: searching for relevant examples (e.g. on the Internet, on a local code base, documentation, etc.)
- **Share**: sharing the knowledge gained by the example with the community (e.g. posting on blog or on a wiki.)
- **Test**: testing the example. This includes developing black box and white box tests, unit testing, acceptance tests, test driven development etc.
- **Understand**: understanding the example (design, behavior, algorithm). We distinguish between understanding and learning: understanding is related to a specific example (e.g. how does the function convert from one format to another), as opposed to learning which is of something more general than a specific example (learning Java, learning Android).
- **Wrap**: making the example suitable for future reuse (extract to method, put in a designated project, etc.).
- **Write**: adding the example code to the existing one (e.g. using copy & paste or typing manually).

However, building block orchestration and choreography are not enough, and further study should also be conducted in order to learn more about the individual items. Some of these building blocks were not recognized as part of software construction activity, some were only recently appreciated as such and others have changed significantly over recent years. For example: sharing knowledge or searching for code has changed dramatically over the last 15 years, and it still not obvious how to do it effectively in the context of software construction. This, however, is beyond the scope of this dissertation.

### 7.3.4 Development Process

Introducing examples to the development process needs to be addressed at the organizational level. Software organizations should encourage and legitimize example usage, and have a strict policy about its caveats (e.g. licensing and preserving intellectual property).

Examples could be considered at various parts of the development process, and in various granularities, including the coding phase, design, for documentation purposes, training, testing and possibly others.
More specifically, in this dissertation we examined several opportunities for software organizations to promote example usage; in chapter 4 we examined example usage from the human and social perspective. We showed that developers’ behavior is derived, in part, from what they perceive as the organization’s goal. An organization that would take a proactive approach such as using an integrated management framework for building organizational capabilities of knowledge management, as proposed by Lee and Kim [97], would be able to promote example usage along the development process.

In Section 6.3.2 we review several activity types that may benefit from example usage, some of them correspond to phases in the development process (such as design and maintenance). These activities imply that example usage could be incorporated into the development process in various parts, rather than manifested as a mere coding technique. Consider, for instance, an example oriented requirement analysis phase or a methodological process to design a system based on reviewing and reusing existing designs (examples of existing designs).

In Chapter 5 we suggested that example usage suffers from inattentiveness, and argued that in some cases reuse opportunities are consequently missed. However, methodically using examples along the development process, would make it an explicit concern in software development and assist in keeping the developers’ attention on it.

An interesting addition to the development process would be writing examples. We argue that each programmer should provide examples that illustrate how every code snippet he or she writes can be used. This should be anchored in the development process as with testing and documenting the code; since programmers are expected to document and test their code, they should be expected to write examples that use their code. A systematic enforcement of example writing would strive to create example coverage of the code. Example coverage promotes building blocks with inner strength and weak decoupling, and is therefore reusable.

Making example writing a habit, paves the way for public and general purpose example repositories, which can be integrated with documentation sources, maintaining a similar structure and receiving wide community support.

### 7.3.5 Tools

Software tools are useful for activity implementation in at least two ways. Firstly, after one characterizes a key activity, it is easier to identify repetitive tasks. In many cases, these tasks could be automated or at least supported by an automatic
software tool that performs them easily. Furthermore, in some cases the tool could consistently ensure and enforce correctness. A second use of a software tool is in providing a framework that captures the sub elements of the software activity and assists in streamlining them. Working with such a framework guides the developer through the major steps.

For example, the xUnit software tool [17] provides not only support for repetitive tasks introducing a large set of assertions, but also a framework that streamlines the setup, run tests, and tear down scenarios.

Moreover, in some cases, software tools do not only assist the development process, but also enable it. The advances of software tools to meet the concerns of certain activities, practices and processes, create new opportunities that would not have been possible otherwise. This is an iterative process: new activities pose new requirements for new tools, which in turn offer new possibilities for further activities. For example, large refactoring is a challenging task in complex software systems without the support of an automatic refactoring tool. Once such a tool was created, it inspired developers to incorporate refactoring into the test driven development basic cycle, and use it frequently and extensively not only for design tasks but also for programming ones.

Another important aspect regarding the adoption of a new software tool, and consequently a new practice, is tool integration. Modern Integrated Development Environments (IDEs) present new opportunities for augmenting existing development environments with new tools, deployed as plugins, and relieving the context-switch involved in using an external standalone tool.

In Section 7.3.3 we listed a few potential building blocks in the EE activity. Tool support should address each of these activities individually as well as their integration. Following are several possible such tool enhancements:

- **IDE-browser integration.** When a programmer conducts a Web search using a general purpose Web search engine, he or she needs to switch from the IDE to another application (the browser). The search may yield several results, which s/he opens in different tabs and, for examination purposes, copies and pastes each of them back into the IDE. We propose an integrated solution in which the Web search would be possible from within the IDE. The various possible solutions could be presented side-by-side next to the code in which they should be embedded. This enhancement preserves the programming context along the process of finding, evaluating, merging and embedding the example.
7.3 Example Embedding Software Activity

- **Extract example IDE support.** Modern IDEs support large number of automatic refactorings; we wish to have something similar for EE. One of which is Extract Example - Extract Method with respect to EE; when a programmer browses production code and spots a snippet that fits his or her needs, the IDE should offer assistance in this task as described in [84]. However, it is suggested that copy and paste would not be the preferred consumption of an example (to avoid code duplication) but rather an extraction of the example snippet and using it from both places. Tool assistance could also include *slicing*, removing unnecessary parts of the snippet [150] [143].

- **IDE-example repositories integration.** Such integration was suggested by [111]. Recent advances in IDE technology and useful heuristics such as [63] [25], as well as the development of recommender systems [4] could overcome the challenges of searching and browsing these repositories.

I am currently involved in building such a tool, with the above properties, that is a direct implication of the research described in this thesis. Together with another graduate student, we use StackOverflow API to find code examples that appear as part of *accepted answers*\(^\text{17}\) on the StackOverflow Website. Our tool collects these code examples and stores them in a designated repository. To every snippet it adds meta information such as the technology, version, community rank, number of views, and descriptive keywords taken from the text surrounding the code in its original context. The tool is embedded in the Eclipse IDE and provides the developer with code example search capabilities without needing to switch context.

This design of the tool explicitly addresses the following concerns/insights:

- Using the tool should be as similar as possible to the user experience of Web search, however without losing the development context (no special query language, no standalone tool).
- A developer can differentiate relevant examples from non-relevant ones; hence the tool should provide the means to *browse* examples effectively (and not only to search them).
- The developer should have an indication regarding the quality of the example (part of an accepted answer, community votes), in order to address human concerns.

---

\(^{17}\) Accepted answers are answers that the user who asked the question found to be suitable for what he or she needed.
7.3.6 Training

Our familiarity with many academic computer science and software engineering programs teaches us that they do not give modern development activities and practices the attention they deserve. Moreover, the fundamentals of software development are considered to be writing computer programs from scratch. Only in advanced courses, if such exist, does the student encounter other development scenarios such as maintenance, building on top of existing systems, agile processes and others.

One could think of another approach in which it is the other way around: writing a program from scratch is considered to be the advanced topic, while the fundamentals of software engineering would be, say, the agile development scenario. With this approach development activities and practices gain more attention than at present, and the students are exposed not only to the best practices of software design, but also to the best practices of programming activity. Consider how students learn a foreign language in school; at least in my school, reading comprehension questions and unseen-passage questions came before the writing of essays.

For example, consider the need to modify code in order to conform to a different context than in the source of the example. This difficulty is widely known, and is also identified in the qualitative analysis, but in the current developer training programs it is hardly addressed at all. The reason for this, as described by Buck and Stucki [28][29], is that most undergraduate curricula focus on developing program application and synthesis skills (i.e., writing code), primarily acquired through hands-on activities. One could think of a designated course that would focus on this problem highlighting the things that the developer should be taking into account, the strategies to do so, and the tools to use.

In Section 4.4 we identify several assertions of professional developers that may suggest that these developers were not trained to use examples effectively. In the analysis of the field observations (Section 5.3) we speculated that the avoidance of example usage was also related to the way that the specific technology was learned – learning Python from online resources contributed to using online resources later in Python related tasks.

Obrenović et al [113] report 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 various technologies. However, as discussed in Section 4.4.2, there are also other approaches suggesting that search-oriented information retrieval behavior might have a negative effect on a person’s ability to stay focused for long [31]. Indeed,
EE highlights the variability of the cognitive skills involved in software construction: those related to search and adaptation, in addition to concretizing algorithms, and design to programming language code.

In our research we encountered many developers who claim to use examples in their work. These developers, however, reported different techniques for doing so. A training process, highlighting the best practices and potential caveats of extensive example usage, may assist in improving their work. For example, training would be beneficial to teach the novice how to effectively apply the set of proposed practices listed in Section 7.3.3 above. Consider, for example, the following hypothetical rules of thumb: Any time a developer looks for an example on the Internet, she must find at least three variations of the required functionality – so that she can compare them and identify aspects she was not aware of; Any time she uses a code that she has not written herself, she must provide a reference to the original copy of that code (on the Web or in another project); Any time she writes a piece of functionality by herself, she must list (possibly as part of the code review process) all places she checked to see whether this functionality had not already been implemented elsewhere, and so can be reused.

In addition to formulating EE best practices, an educator would need to rely on a supportive infrastructure, such as appropriate software tools and a comprehensive set of predefined examples, to be able to deliver that message to the novice developers and let them experiment with finding and embedding examples on demand.

A comprehensive description of EE software activity along with some well-formulated practices and supportive tools enables educators to emphasize the use of examples in software development from the very beginning of computer science and software engineering education.

7.4 Ecosystem

In some cases, a software activity alone, without its supportive framework, is considered harmful and counterproductive. For example, one of the clichés of software engineering says: "If it ain't broke – don't fix it!" which is exactly opposite in its spirit to the rationale for refactoring. This contradiction is settled by what we call the software engineering ecosystem – symbiotic relations between the various components involved in software construction – which together compensate for the pitfalls of a certain activity.

Refactoring, for example, is supported by the ecosystem as follows: comprehensive refactoring catalogues describe a wide variety of possible
refactoring activities, provide useful information on how to apply them systematically and highlight their subtleties. Automatic refactoring tools reduce the probability that the code change introduces a bug into the system. Agile practices, such as test driven development, guides developers to refactor the code only after a new addition is introduced and tested, which aids in avoiding spending precious time on unnecessary cosmetic changes. A software process that promotes refactoring also promotes simple design to allow easier application of future refactorings as well as to save time, knowing that this time investment will save time in future stages during the system growth.

Moreover, once EE is woven into the ecosystem, explicit developers’ attentiveness is no longer an issue; example attentiveness is inherently woven in using by the tools or in following the development process.

Figure 7.3 depicts some of the relationships between the various components of the software activity ecosystem: the boxes represent the components, and the annotated arrows present the relations between those components.

![Figure 7.3: Software activity ecosystem.](image)

The EE ecosystem is analogous to the refactoring one. Proper EE practices would motivate the creation of new tools, which, in turn, enhance these practices and motivate new ones.
7.5 THE CASE OF REFERRING PREVIOUS ACADEMIC WORK

Let us revisit the ecosystem concept by investigating another case of building upon previous work, this time, outside of the software construction domain. This is the case of referring previous and related work in academic research. We find this discussion relevant to software construction as well.

Academic work is not performed in a void. Progress in research is achieved by the accumulation of knowledge. Standing on the shoulders of giants allows a single researcher to look further away than what she can see on her own two feet. Furthermore, referencing previous and related work situates the current work in context, and allows the community to appreciate its unique contribution. Referencing previous work is a fundamental part of the academic research.

Let us investigate how referencing previous academic work was woven into the academic ecosystem:

- **Catalogues and repositories** – Academic work is organized in libraries. The ACM digital library is one example of such a library in the domain of computer science and related areas. The work is organized through categories, keywords, and technical terms, so that it may be found and browsed easily.

- **Tools and technologies** – Extensive incorporation of references in large works may be tedious, exhaustive and error-prone. However, there are software tools that assist in managing this process. This paper, for example, uses the BibTeX format and tools to organize the references, order them, and put them in the proper format.

- **Process** – Referencing academic work could be incorporated in different ways to the process of research. While one research course could be previous work driven, another sequence could be starting with a clean slate and, after results have been found, the previous work is used to situate them in context, and within a theoretical framework.

- **Community** – The academic community enforces the usage of references in academic work, by not accepting articles for publication if they refrain from mentioning previous and related work. This is achieved by weaving the concern of previous work into the peer review mechanism.

- **Practices** – For some people, it is easier to write from scratch than to read and incorporate new contributions on top of existing ones. This requires the practitioner to develop appropriate skills and effective techniques to streamline the incorporation of previous work with the writing of the new one.

- **Training** – Many graduate schools offer scientific writing courses in which referencing issues are also discussed. Such courses often provide the novice
with practical advice regarding whom to reference, the dates of the references, their accessibility, ranking, and so on.

Although the analogy to software construction only works to a certain extent, the mechanism described above for incorporating previous work into the academic ecosystem may offer inspiration for integrating example usage into the software engineering ecosystem.

### 7.6 SUMMARY AND FUTURE WORK

In this chapter, we described a perspective for empirical software engineering research aimed at identifying and characterizing fine grained development activities that are currently assimilated in other activities. We examined possible implications of such identification and introduced our definitions to software activities, software practices and the software ecosystem. We also provided some illustrations taken from our experience applying this perspective in field research investigating the EE software activity. Although in this chapter we focused on *programming* practices, we believe that the discussion is also relevant for other software activities apart from programming such as design or analysis activities that could be investigated.

Firstly, it provides a conceptual system to be used in future similar research settings. Secondly, it provides a preliminary description of a specific activity, Example Embedding, for which a community effort is needed in order to encompass and explore its multi-faceted nature.
“I wanted a perfect ending. Now I’ve learned, the hard way, that some poems don’t rhyme, and some stories don’t have a clear beginning, middle and end. Life is about not knowing, having to change, taking the moment and making the best of it, without knowing what's going to happen next.”

- Gilda Radner

Chapter 8

Summary

We did not embark on this research with either example usage or its diversity in mind. Instead, we conducted field observations in software companies for several weeks in the hope of identifying innovative patterns and behaviors upon which to focus our research in later stages. We noticed that examples are being used in the course of the development, and focused our attention on this issue.

Our investigation revealed that despite the great potential in systematic and habitual example usage, it is (still?) diverse - its benefits are not consensually appreciated by the developers, the software development community does not address the social challenges that accompany extensive example usage, and there is no explicit, widely accepted set of best practices for using examples either methodically or systematically.

We conclude this journey in portraying a comprehensive approach that, by addressing the software engineering ecosystem, addresses many of the concerns that are currently preventing examples from being used extensively and effectively.

In fact, this journey is just beginning...
REFERENCES


