
Greg Jordan

Follow this and additional works at: https://digitalcommons.memphis.edu/etd

Recommended Citation
https://digitalcommons.memphis.edu/etd/3419

This Dissertation is brought to you for free and open access by University of Memphis Digital Commons. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of University of Memphis Digital Commons. For more information, please contact khggerty@memphis.edu.
Tutorial Documentation for Software Developers:
Evaluating the Impact of a Supporting Technology Documentation on Learning New Technology

by

Greg Jordan

A Dissertation
Submitted in Partial Fulfilment of the Requirements for the Degree of Doctor of Philosophy

Major: English

The University of Memphis
March 2022
Dedication

To my wife, Rachel, and our sons, Gregory, Samuel, and Andrew.

To my parents, Don and Linda.

To my in-laws, Richard and Clara.
Acknowledgements

This work could not have been possible without the feedback and guidance from my dissertation committee members - Drs. Michael Albers, Elizabeth Lane, and Emily Thrush. I will always be grateful for their kindness and collaboration during this time. I am thankful for their commitment to me as well as their personal contributions and to spurring others’ contributions in our field of study. I am especially indebted to Dr. Loel Kim for all the encouragement, candor, and effort as my advisor and dissertation chair. Your advice and support were essential to the improvement of my writing and helped ensure I made it through the most difficult stages of my work. I would also like to thank Drs. Mary Battle, Loel Kim, Reginald Martin, and Emily Thrush. Much of my professional successes resulted from instruction in your classrooms and counsel during office hours. I consider myself very lucky to have been taught by all of you. Finally, I would like to recognize my siblings (Elizabeth, Don, and Lynn), my extended family, and friends for their love, support, and encouragement. I am fortunate to have such a wonderful family and generous friends.
Abstract


Software has become central to every part of modern life and supports everything from managing personal schedules to managing global supply chain systems. In creating new or updating existing software, software developers often need to incorporate new technology. When software developers learn new technology using tutorials, their experience with secondary technologies—the technologies that accompany or complement the new technology—can have an impact on the comprehension of the new technology being covered. Based on this understanding, I designed a mixed methods study focused on a group of software learning a new technology to examine the following questions:

1. Does prior knowledge of a supporting technology affect learning a new technology?

2. Does providing external code examples of a secondary, supporting technology in a tutorial help make learning a new technology more effective?

Nine participants were drawn from the researcher's professional network of programmers, each of whom reported at least some Java programming experience (secondary technology) but lacked experience with the primary technology being introduced. During individual, remote usability sessions, each participant read a tutorial and completed tasks based on the tutorial. The study's data was the result of the usability sessions, surveys, and feedback after each session.

The sample size for this study resulted in rich data collected from sessions was used to discover if problems existed with the tutorial's approach, as well as to help.
Overall, the analysis supports the limited prior research in that participants experienced no significant usability problems with the tutorial’s design approach, which accounted for learners’ knowledge of a secondary program language, Java, by providing Java code examples in the tutorial. It also suggests that this approach did not introduce problems based on prior experience with the supporting technology. However, a larger sample group in a longer-term study should be conducted to expand the range and types of data being collected and analyzed as well as provide statistical validity. In addition, new survey questions could further quantify and classify "prior knowledge of the supporting technology" as well as open comparisons with participants who do not have prior experience the supporting technology to those who have experience.
# Table of Contents

List of Tables viii

List of Tables ix

Chapter | Page
---|---
1 | Introduction 1
   | Statement of Purpose 1
   | Research Rationale 3
   | Significance of Research 11
   | Statement of Problem 13
   | Conceptual Framework 14
   | Research Questions 16

2 | Literature Review 18
   | Introduction 18
   | Challenges Across Documentation for Software Developers 19
   | Subject Matter Knowledge and Writing Skills 20
   | Process and Technological Challenges 25
   | Structure, Standards & Guidelines 28
   | Creating Code Examples 29
   | Creating Concise Examples 30
   | Highlighting Important Elements 32
   | Building Upon Prior Examples 34
   | Providing Additional Resources 34
   | Scaffolding and Tutorial Effectiveness 36
   | Summary 37

3 | Methodology 38
   | Introduction 38
   | Participants 40
   | Data Collection 42
   | Survey & Usability Testing 42
   | Study Materials 43
   | Post Usability Survey & Follow-up 43
   | Data Capture for Analysis 44

4 | Results and Analysis 45
   | Introduction 45
   | Participant & Survey Data 45
   | Tutorial Data 48
   | Task Data 49
   | Post Usability Survey Data 52
   | Qualitative & Observational Data 54
## 5 Conclusions and Recommendations

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>57</td>
</tr>
<tr>
<td>Research Questions</td>
<td>58</td>
</tr>
<tr>
<td>Conclusions</td>
<td>59</td>
</tr>
<tr>
<td>Recommendations</td>
<td>60</td>
</tr>
</tbody>
</table>

### References

62

### Appendices

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Programmer Survey</td>
<td>71</td>
</tr>
<tr>
<td>B. Session Materials</td>
<td>72</td>
</tr>
<tr>
<td>C. Post-usability Survey</td>
<td>79</td>
</tr>
<tr>
<td>D. IRB Approval</td>
<td>80</td>
</tr>
</tbody>
</table>
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1. Software Categories and Subcategories</td>
<td>5</td>
</tr>
<tr>
<td>Table 2. Software SubCategory: Databases</td>
<td>5</td>
</tr>
<tr>
<td>Table 3. Professional and Java programming experience by participant</td>
<td>45</td>
</tr>
<tr>
<td>Table 4. Professional programming experience</td>
<td>46</td>
</tr>
<tr>
<td>Table 5. Java programming experience</td>
<td>46</td>
</tr>
<tr>
<td>Table 6. Java self-rating and skill-rating for tutorial reader</td>
<td>47</td>
</tr>
<tr>
<td>Table 7. Java self-rating</td>
<td>48</td>
</tr>
<tr>
<td>Table 8. Skill-rating for tutorial reader</td>
<td>48</td>
</tr>
<tr>
<td>Table 9. Reading times for tutorial</td>
<td>49</td>
</tr>
<tr>
<td>Table 10. Time on tasks by participant</td>
<td>50</td>
</tr>
<tr>
<td>Table 11. Mean time on tasks by professional programming experience</td>
<td>50</td>
</tr>
<tr>
<td>Table 12. Mean time on tasks by Java programming experience</td>
<td>51</td>
</tr>
<tr>
<td>Table 13. Mean time on tasks by Java self-rating</td>
<td>51</td>
</tr>
<tr>
<td>Table 14. Tutorial effectiveness ranking - understanding topic &amp; task completion by participant</td>
<td>52</td>
</tr>
<tr>
<td>Table 15. Tutorial effectiveness ranking - completion by professional programming experience</td>
<td>52</td>
</tr>
<tr>
<td>Table 16. Tutorial effectiveness - understanding topic &amp; task completion by Java programming experience</td>
<td>53</td>
</tr>
<tr>
<td>Table 17. Effectiveness of the tutorial - understanding and task completion by Java self-rating</td>
<td>53</td>
</tr>
<tr>
<td>Table 18. Tutorial usage by participant</td>
<td>54</td>
</tr>
<tr>
<td>Table 19. Tutorial usage by Professional programming experience</td>
<td>55</td>
</tr>
<tr>
<td>Table 20. Tutorial usage by Java programming experience</td>
<td>55</td>
</tr>
<tr>
<td>Table 21. Tutorial usage by Java self-rating</td>
<td>56</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Retrieving and Updating a Node</td>
<td>2</td>
</tr>
<tr>
<td>2. Xcode method with comments</td>
<td>8</td>
</tr>
<tr>
<td>4. Overview (Java Platform SE 7)</td>
<td>11</td>
</tr>
<tr>
<td>5. Table of Contents from Practical Neo4j</td>
<td>21</td>
</tr>
<tr>
<td>6. API Authentication function</td>
<td>31</td>
</tr>
<tr>
<td>7. Save method for UserInterface class</td>
<td>32</td>
</tr>
<tr>
<td>8. Removing a Node from Practical Neo4j</td>
<td>33</td>
</tr>
<tr>
<td>9. Matching on a Label</td>
<td>34</td>
</tr>
<tr>
<td>10. Updating a node property</td>
<td>34</td>
</tr>
<tr>
<td>11. Python driver code</td>
<td>35</td>
</tr>
<tr>
<td>12. API Authentication function showing programming language options</td>
<td>36</td>
</tr>
<tr>
<td>13. Research Triangulation for Usability Study</td>
<td>38</td>
</tr>
<tr>
<td>14. Quesenbery’s Five Dimensions of Usability</td>
<td>40</td>
</tr>
<tr>
<td>15. Number of Test Users</td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

Statement of Purpose

When software developers learn new technology using documentation, such as software programming tutorials, the understanding of supporting technologies can have an impact on the comprehension of the primary technology being covered. Supporting technologies—which might also be called secondary technologies—form a type of dependent relationship in software. A software developer will need to understand and utilize the secondary technology in order to make use of the primary technology. My dissertation study will examine the impact of the comprehension of a primary technology covered in a given piece of software documentation where documentation is also provided with a specific, secondary technology.

While many types of software documentation exist, most documentation studied by technical communication research is the type intended for end-users of software products. However, one type of documentation targets software developers of software products as the intended audience. This technical documentation “describes information about the design, code, interfaces and functionality of software to support developers in their tasks” as opposed to user documentation which “explains to end-users how they should use the software application” (Aghajani, “Software documentation issues unveiled” 1199). Developer documentation is specifically for software developers and provides guidelines and examples for the developer working in a specific area of the computer code, perhaps, how a particular part of the software programming should function or what features a specific library or function should be included so the entire program will run as efficiently as possible (Robillard et al. 479). While programming
documentation can be embedded within the programming code itself, the type of documentation this study focuses on is not embedded documentation but, instead, on manuals and books that provide a tutorial for software programmers, such as the example shown in Figure 1.

```java
public class SDNServiceClass {

    @Autowired
    public Neo4jOperations neo4jTemplate;

    public void updateNode() {
        // get the node
        Node node = neo4jTemplate.getNode(10);

        // update node properties
        node.setProperty("firstname", "Greg");
        node.setProperty("lastname", "Jordan");

        neo4jTemplate.save(node);
    }
}
```

Figure 1. Retrieving and Updating a Node from *Practical Neo4j*. Author's screenshot.

The necessity of tutorial documentation lay in its potential to positively influence both a program’s development process and the quality of the resulting software (Zhi 176) underscores the need for technical communication research to expand into this area. Therefore, this dissertation focuses on tutorial documentation, specifically the type offered in a longer form, external to the programming code, i.e., found in a book or manual, and which provide greater depth and context of a programming topic than either embedded documentation or article-length tutorials can offer. Article-length tutorials
more often aim to provide localized coverage of a single, narrow task or topic for a specific-use case related to the programming topic, whereas a book or manual can cover many tasks and a wide range of use cases for a programming topic.

In addition, the purpose of long-form documentation is to provide software developers with comprehensive coverage for an existing software language or product so that it might be subsequently used to create new or to update existing software. Since books and manuals cover a topic broadly over hundreds of pages, they are often tied to a point in time, and often that point in time is connected to a version number and release date of the software. This format also carries with it certain expectations and guidelines for the organization, content, and design of the software program. For example, a book or reference manual often includes an introduction that establishes an origin or history of the software which is followed by a review of use cases and tasks that will be covered. Each chapter or section introduces a new topic and concludes with summaries of the material for that chapter or section.

In my dissertation, the term “software developers” will be used to refer to computer programming professionals who learn and use programming languages, such as Java or Python, to create or maintain commercial, enterprise-level software, which is most commonly used in professional organizations, like a college or university, or businesses. In addition, the term “tutorial”–unless otherwise noted–will mean long-form format, such as a book or manual.

**Research Rationale**

Software has become central to every part of modern life. We rely upon software for everything from managing personal schedules to managing global supply chain
systems. The value users place on any type of software generally has a positive
correlation to its quality (Jones and Bonsignour 4). As defined by ISO 25010, the quality
standard for software includes the following characteristics (“ISO 25010”):

- **Functional Suitability.** Does the software solve a specific problem or set of problems?
- **Reliability.** Does it function in a consistent manner?
- **Operability.** Is it relatively attractive and easy to use?
- **Performance efficiency.** Does execution of the software require a relatively inordinate amount of time or computing resources?
- **Security.** Does it provide integrity and compliance as well as verify user authenticity and auditing where necessary?
- **Compatibility.** Does it make information exchange possible with other systems?
- **Maintainability.** Can it be modified relatively efficiently without sacrificing current levels of quality?
- **Transferability.** What is the relative degree the software can be transferred to another underlying software or hardware platform?

One of the most important tools for creating quality software is tutorial documentation
(Zhi et al. 176), which is typically grouped by the different categories of software into
which they belong, such as programming languages, operating systems, application
development frameworks, and databases. The documentation can be further categorized
within a subcategory and then further into specific instances of a subcategory. For example, databases could include relational databases, column databases, key-value
databases, document databases, or graph databases as shown in Table 1. In the last row of Table 1, graph databases could be further categorized into specific graph databases, such as Neo4j, DES, Giraph, or JanusGraph. Examples of the subcategorization are shown in Table 2. Tutorial documentation is found among all of these types and levels of programming.
Table 1. Software Categories and Subcategories

<table>
<thead>
<tr>
<th>Category</th>
<th>Purpose</th>
<th>Subcategories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming Languages</td>
<td>Writing individual software programs</td>
<td>Java, C#, Python, Ruby, PHP</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>Environment for running multiple software programs together</td>
<td>Windows, Android, iOS, Mac, and Linux</td>
</tr>
<tr>
<td>Application Framework</td>
<td>Provides consistency in construction of software programs</td>
<td>Spring, Flask, Ruby on Rails, Laravel</td>
</tr>
<tr>
<td>Databases</td>
<td>Information storage, retrieval, and management</td>
<td>Relational, Key-Value, Column-family, Document</td>
</tr>
</tbody>
</table>

Table 2. Software SubCategory: Databases

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Use Case</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>Well known data structure with simple relationships between the data</td>
<td>Oracle Relational Database</td>
</tr>
<tr>
<td>Key-Value</td>
<td>High volume of simple data structures</td>
<td>Amazon Dynamo</td>
</tr>
<tr>
<td>Column-family</td>
<td>High volume of single-type, large data</td>
<td>Apache Cassandra</td>
</tr>
<tr>
<td>Document</td>
<td>Varying data structure</td>
<td>MongoDB</td>
</tr>
<tr>
<td>Graph</td>
<td>Densely connected data with varying structure</td>
<td>Neo4j</td>
</tr>
</tbody>
</table>

Programmers who have expertise in a specific subcategory of database or even experience with a specific instance in a specific subcategory are often asked to contribute new documentation or to update existing documentation for their subcategory—even when a programmer might lack training in editing, documentation design, or technical communication. Like many trades and professions, expertise is often earned over years of focused practice and experience, and when choosing to hire between a formally trained
writer or a professional with a high level of practical programming experience, the latter almost always outweighs the former. Rarely are both skills–especially at a high level–found within the same person. As Val Swisher, CEO of Content Rules, points out technical writing candidates “with truly deep understanding of current software technology are very rare.” (“How to Find Good Programmer Writers”). The focus on the need for technical expertise points towards studying how programmers interact with tutorials.

The disciplinary preference for technical over writing expertise may appear as problematic in the tutorials: Professional software developers with a specific expertise often write technical tutorials, but the lack of writing expertise can result in less effective documentation, which can make the work of programming ensue less smoothly and effectively than desirable. Developers who lack training can overlook or leave out elementary and necessary steps in the document design process, such as performing an adequate audience analysis. As Garousi et al. point out in *Usage and Usefulness of Technical Software Documentation*, “making decisions about the amount and technical depth of documentation remains a major challenge for many practitioners” (665). For example, subject matter experts who write documentation for the Java programming language often provide example code, but do not consider whether the code might be too complex for some audiences or too basic for other audiences.

When considering diverse audiences, programmer-authors can be made more aware of skill-level frameworks, e.g., novice, intermediate, advanced, expert, etc., to segment their audience when designing user documentation for delivery of content and examples. Instead of a systematic or informed audience analysis, the author’s
understanding of significant audience differences is often subjective, resulting in a document that may be applicable to a narrower audience than they wish, or in the worst case, to the wrong audience than the one they wished to target.

Technical communication researchers have developed a robust body of knowledge in many aspects of software documentation, including audience analysis, with the research typically focusing on three areas of software documentation, including:

• Software Maintenance Documentation
• Software Project Documentation
• API Documentation

The next sections will focus on these areas of software documentation in order to provide context as compared to tutorial software as well as how to illustrate how each area fulfills a unique aspect of software documentation. While the focus of the dissertation is on tutorial documentation, it is important to define these other types of software documentation as they provide context for understanding tutorial documentation within the literature review.

Software Maintenance Documentation

The goal of maintenance documentation is to help software developers quickly comprehend the intent of a specific section of code in relation to certain program features or as it functions within the entire software product. Maintenance documentation can be informal or follow a stylized specification included in the code, such as developer comments appearing within the executable code. The comments in figure 2 provide a description of the function named *isThisFileNewerThanThatFile*, list the parameters that
can be passed to the function, and describe the value that is returned when the function is called.

```c
/**
 * Check whether a file at a given URL has a newer timestamp than a given file.
 * Example usage:
 * @code
 * NSURL *url1, *url2;
 * BOOL isNewer = [FileUtils
 *                 isThisFileNewerThanThatFile:url1 thatURL:url2];
 * @endcode
 * @see http://www.dadabeatnik.com for more information.
 * @param thisURL
 *     The URL of the source file.
 * @param thatURL
 *     The URL of the target file to check.
 * @return YES if the timestamp of @c thatURL is newer than the timestamp of @c thisURL, 
 *         otherwise NO.
 * @endcode
 * + (BOOL)isThisFileNewerThanThatFile:(NSURL *)thisURL thatURL:(NSURL *)thatURL;
 */
```


**Software Project Documentation**

On the other hand, sometimes more technical documentation will be more fully developed and in a more formal format, as shown in figure 3, a set of guidelines that describes the software architecture.
1. Introduction
1.1 Purpose
The main objective of this document is to illustrate the requirements of the project Library Management System. The document gives the detailed description of the both functional and non-functional requirements proposed by the client. The purpose of this project is to provide a friendly environment to maintain the details of books and library members. The main purpose of this project is to maintain easy circulation system using computers and to provide different reports. This project describes the hardware and software interface requirements using ER diagrams and UML diagrams.

1.2 Document Conventions
- Entire document should be justified.
- Convention for Main title
  - Font face: Times New Roman
  - Font style: Bold
  - Font Size: 14
- Convention for Sub title
  - Font face: Times New Roman
  - Font style: Bold
  - Font Size: 12
- Convention for body
  - Font face: Times New Roman
  - Font Size: 12

1.3 Scope of Development Project
Library Management System is basically updating the manual library system into an internet-based application so that the users can know the details of their accounts, availability of books, and maximum limit for borrowing.
The project is specifically designed for the use of librarians and library users. The product will work as a complete user interface for library management process and library usage from ordinary users. Library Management System can be used by any existing or new library to manage its books and book borrowing, insertion and monitoring. It is especially useful for any educational institute where modifications in the content can be done easily according to requirements.
The project can be easily implemented under various situations. We can add new features as and when we require, making reusability possible as there is flexibility in all the modules.
The language used for developing the project is Java as it is quite advantageous than other languages in terms of performance, tools available, cross platform compatibility, libraries, cost (freely available), and development process.

1.4 Definitions, Acronyms and Abbreviations
- JAVA -> platform independence
- SQL -> Structured Query Language
- ER -> Entity Relationship
- UML -> Unified Modeling Language
- IDE -> Integrated Development Environment
- SRS -> Software Requirement Specification

Department of Information Technology

Figure 3. Software Requirements Specification of Library Management System.

For example, in *How Software Engineers Use Documentation: The State of the Practice*, the authors “conducted three studies using several data-gathering approaches to elucidate the patterns by which software engineers (SEs) use and update documentation” (Lethbridge et al. 35). The research and survey data lead to an analysis of current
practices and attitudes towards maintenance documentation. However, like Zhi’s *Cost, Benefits and Quality of Software Development Documentation*, the focus remains on documentation in which the goal is creating new software programs or enhancing existing software programs—not learning new technologies, such as a programming language.

**API Documentation**

Similarly, research in application programming interface (API) documentation has been conducted and although it overlaps somewhat with tutorial documentation, it is not the same. At its core, an API is a set of well-defined conventions and instructions for building software. For an API, the term “application” could represent a web site, an operating system, a database, or a software library. For example, figure 4 is excerpted from the Java Platform API, which lists some of the packages within the Java software library for version seven (need citation from oracle website).
The conventions of and instructions for API documentation often come in the form of code examples with a basic and very concise instruction for usage. The goal of API documentation is to help software developers understand what an API can do and how to use it. In *Asking and Answering Questions about Unfamiliar APIs*, the authors “focused on the questions the participants want answered about the use of an API” (Duala-Ekoko and Robillard 268). This study gets closer to addressing the ways in which supporting secondary technology might affect the developer’s ability to learn the primary technology; however, it does not examine factors frequently included in research on documentation intended for the software end-user, such as reading level.

**Significance of Research**

As Michael Albers points out in *Multidimensional Audience Analysis for Dynamic Information*, much of the economic and technical limitations of a “one size fits all”
approach to documentation has been overcome by online documentation but, more importantly, can be tailored to different skill levels. Traditional audience analysis considered aspects such as reader motivation, education level, experience, and organizational role. With the incorporation of online documentation, Albers suggests a multidimensional approach that includes, in addition to the traditional audience analysis, the following dimensions:

- the reader’s domain knowledge
- the level of subject-matter detail to include
- reader’s cognitive ability
- social and cultural dimensions shaping the reading

While Albers provides a more robust framework for audience analysis compared to one that focuses on a single or primary dimension of audience characteristics, the aspect of the expanded analytical dimensions only considers the audience’s knowledge of primary subject matter, but excludes interrelated and critical, secondary subject matter. As mentioned in the purpose statement, software documentation often focuses on the audience’s primary subject matter. My scope is expanded to the area of the domain knowledge dimension the programmer possesses to study discrete but connected topics within the multidimensional approach of audience analysis. For example, one audience might have an expertise in the Java programming language but no experience with graph databases. Conversely, another programmer audience might be well versed in graph database concepts but have limited Java experience.

As will be shown in the literature review conducted for this study, the research has infrequently focused on the long-form, tutorial-style documentation used and needed
by the programmers. Therefore, one aim of this study is to further research of audience analysis by going beyond application of the reader’s domain knowledge and include the impact of analyzing the reader’s attitudes and expectations of effective, tutorial-style instructions in such aspects as audience skill level, documenting advancements in the software, documentation structure, and code examples.

**Statement of Problem**

As established by Albers, documentation can be more targeted and effective by considering more than just traditional audience analysis, e.g., the level of the reader’s domain or subcategory knowledge. In addition, I argue that when developing software documentation, the varying levels of users’ knowledge of discrete, connected areas of subcategories should not only be considered, but accommodated in the documentation. For example, programmers who work with databases might possess advanced-level skill in one subcategory, such as relational databases, but possess only novice-level skill in another subcategory, such as graph databases. In another example, programmers who create web-based software could possess advanced-level skills in the subcategory of Java programming but possess only novice-level skills in the subcategory of Python programming.

As I experienced when writing *Practical Neo4j*, tutorial documentation for the graph database Neo4j, the blending of multiple subcategories creates a challenge of how much knowledge the writer can assume of the reader’s knowledge of the any given subcategory. If I created a tutorial that focused on using Neo4j with Java as the supporting language, then it would exclude Python programmers. To cover multiple subcategories by language, I decided to create distinct chapters that covered using Neo4j
with Java, Python, Ruby, Spring, and C# programming languages. However, this still does not address audience learning needs in terms of the level of knowledge they possess within each subcategory. Ultimately, this is important because a variance in prior knowledge can affect the effectiveness of software documentation.

**Conceptual Framework**

Enterprice software, which is computer software created primarily to meet the needs of an organization rather than individual consumers, is typically developed either within an organization or created by software companies for organizations, such as hospitals, auto supply companies, or universities. As part of my professional work in enterprise software development, I have heavily utilized documentation that demonstrates how to work efficiently and effectively when creating and maintaining software over its lifetime. Zhi et al. supports this point in that “software documentation is an integral part of any software development process” and that 72% of research in their study focused on the quality of software documentation (176). While cost and benefits are important factors of documentation, high-quality documentation provides both a foundation for utilizing a specific, primary software as well as supporting the speed at which the programmer understands how to use it (McBurney 1).

In 2014, I wrote *Practical Neo4j (Apress)* to document the operational use (the interaction with other software systems or end users) and implementation (the task of setting up and creating code to interact with the target system) of the Neo4j graph database. Graph databases are the most popular software for storing and analyzing relationships between objects, e.g., person, location, or subject. Social networks use graph databases to efficiently manage and analyze relationships between users within the
network, regardless of the size of the network. This information is ultimately utilized by the organization or business to improve the network or to persuade companies for advertising dollars. In this specific domain, Neo4j is often attributed as the most popular graph database based on the number of times it has been downloaded.

I focused on providing quality examples for implementing Neo4j in conjunction with six different programming languages and frameworks. For each language, the chapter’s text and code examples were placed through a peer review process to provide a baseline for certifying quality. Even with this peer review process that focused on quality, the aspect of audience, specifically the level of an individual’s skills, was a difficult area to identify appropriately.

For this book targeting programming professionals, I assumed a reader with at least a basic level of understanding of programming, e.g., one year of professional experience in creating software. However, when the author must address multiple functional areas—such as the specific computer language, data structures, or system administration—the programmer’s level of knowledge and experience can vary widely. For example, some programmers may be at an advanced level in their respective programming language experience but might be at a novice level in terms of data structure experience. Based on this understanding, I designed a study with the variable, skill level of users for a single programming language, and implemented examples of technology knowledge external to the new technology being learned.
Research Questions

The research questions driving my study are:

R1: Does prior knowledge of a supporting technology affect the process of learning a new technology?

R2: Does providing external code examples of a secondary, supporting technology in a tutorial help make learning a new technology more effective?

By effectiveness, I am examining the “completeness and accuracy with which users achieve their goals” (Quesenbery 3). While learning the new technology is the goal of the tutorial, a primary goal for the programmer is to be able to utilize the technology in future projects. As noted in the problem statement, a challenge software tutorial writers encounter is the providing adequate examples necessary to support the goal of learning a new technology. As shown in Figure 1, code examples are included as part of tutorials to demonstrate how secondary technology, such as Java, can work with the primary technology being covered, such as a database. These in-line examples are composed expressly for the tutorial and allow the reader to review condensed code which can help illustrate a specific concept or feature. To prevent the tutorial from including superfluous examples, some effort is expended to select code examples that highlight only the relevant code being introduced.

However, what is superfluous for some readers might be critical for other readers to understand the concept being covered. For instance, a code example might reference newer version of Java of which a reader has not yet used. Therefore, it might be necessary to provide the reader with more in-depth examples or even code from external tutorials, such a link to the core Java documentation. Considering that the depth for code
examples is relative, software tutorials could benefit from better understanding the impact of providing external code examples of a secondary, supporting technology on learning a new technology.
Chapter 2: Literature Review

Introduction

This review examines literature and studies which address issues impacting documentation made for software developers—regardless of type of software documentation—and will help provide a foundation for the subsequent study performed for this dissertation. As stated earlier, research on software documentation for developers has more frequently focused on API, software project documentation and software maintenance documentation—not the tutorial-style documentation used and needed by the programmers. The review aims to utilize the studies performed on these formats of software developer documentation and help to demonstrate the different considerations between as well as highlight shared concepts. This background from research on API, software project documentation and software maintenance documentation should also provide support in the construction of the elements of the dissertation’s usability study as well as offer comparisons on how the topic has been addressed in other studies. Since the code examples for supporting technology are a primary component of the study, it will also make sense to review studies that examined computer code across several software documentation formats and how other studies considered audience as a primary concern.

Good software documentation for developers is a critical tool for creating and maintaining good software. Not only does research show that documentation is ranked high by software developers in importance for both the creation and maintenance of software (Aghajani, “Software Documentation: The Practitioners’ Perspective” 590), but the quality of the documentation has an impact on the quality of the software (Zhi 176). Conversely, several studies identified by Chen and Huang show that low quality software
documentation was ranked near the top for causes of software errors and maintenance problems as well as lack of quality documentation resulting in “documentation being untrustworthy” in the eyes of software developers (982). While this research shows links between the quality of software and the quality of the documentation, the bulk of software documentation research focuses on the type of documentation that is aimed toward the end-user of software. In addition, there is an even more limited amount of research which specifically focuses on long-form tutorials such as books and manuals. As Dagenais and Robillard note, “Various types of documents are available to help developers learn about frameworks ranging from Application Programming Interface documentation to tutorials and reference manuals. We find though that very little is known generally about the creation and maintenance of developer documentation” (127).

Further, very limited research exists that explores the understanding of how software developers learn new programming languages or expand their knowledge of technologies in which they utilize for their work. Prat and others point out that, “remarkably little research has investigated the cognitive basis of what it takes to learn programming languages. The implications of such knowledge are wide reaching, both in terms of cultural barriers to pursuing computer sciences and for educational practices. Central to both are commonly held ideas about what it takes to be a ‘good’ programmer, many of which have not been empirically instantiated.” (3817).

**Challenges Across Documentation for Software Developers**

While the goals of software tutorial documentation, such as books and reference manuals, are different when compared to software documentation for API, project, and maintenance, they share many of the same challenges. For example, each of these types
of software documentation require the author to have not only have knowledge of the software being documented but to have effective writing skills so that the documentation is relatively easy to comprehend. The different types of software documentation also require varied skill sets and planning considerations regarding the specific documentation platform, distribution channels as well as any standards and guidelines that are followed based on the software documentation type. In the next sections, I will compare and contrast various topics which crossover between the different types of documentation, such as software and programming knowledge, writing skills, platforms choices, and standards as well as illustrate how these challenges have an impact on the overall quality for long-form tutorials versus other software developer documentation.

**Subject Matter Knowledge and Writing Skills**

One of the biggest challenges in creating good software documentation for software developers is having both adequate knowledge of the software topic as well as effective writing skills that clearly convey how to make use of the software. For example, if a software developer is creating API documentation to demonstrate how the Java programming language would read a data file to process the contained information, then the minimum requirement is to write the code example within a Javadoc—the standard format for inline documentation in Java. The very narrow writing requirement of including only a code example is where the similarity ends when writing API or maintenance documentation versus tutorial documentation. Tutorial documentation needs an author that has deep subject-matter knowledge as well as write the accompanying prose and code examples but write in such a way that clearly conveys how the code examples work in multiple use cases. In the area of tutorial documentation,
especially books and reference manuals, the task of writing pages of, for example, how
the internals of a programming function or code library works and then mixing in code
examples is a common pattern, which will be repeated chapter after chapter for hundreds
of pages. As shown in figure 5, these tutorials often begin with an introductory chapter
which contains a high-level view of the technology. The remainder of the chapters will
then provide a focus on a specific topic, each of which will include multiple, relevant
examples and are, in turn, often broken down into even more narrow topics.

Figure 5. Table of Contents from Practical Neo4j. Author's screenshot.
In this example, the selected technology is the graph database called Neo4j. In long-form tutorials, the first tutorial often is a “quick start” chapter designed with elementary examples serving as foundation for more complex usage as well as underpinning how the technology can be useful in different cases and domains. The quick start will usually be followed by chapters grouped into logical parts and each chapter provides more details use cases for common software applications, e.g., how to model data with a graph database versus a relational database or how to use the database with various programming languages. The example highlights a key difference between API or maintenance documentation versus tutorial documentation, specifically showing the latter often demands the author have both broad and deep subject-matter knowledge related to the primary programming topic and have writing skills that tie together loosely related, complex topics with a consistent, concise, and clear voice.

A related challenge for all types of software documentation is not just combining subject matter knowledge with good writing, but the lack of formal training or professional courses for software documentation authors as well as the continued development and improvement of their writing skills. In *The Types, Roles, and Practices of Documentation in Data Analytics Open Source Software Libraries*, the authors state that "writing, contributing, and reviewing [software] documentation often requires not only these skills but also an additional set that are often not taught in traditional software engineering” (Geiger et al 785). Participants in the study understood that their documentation would be accessible and read by peers outside of their own organization,
and, therefore, noted that “storytelling and creative writing skills were highly important for documentation” (Geiger et al 786).

However, there is a burgeoning view that formal training—for both software development and software documentation—is growing less important for roles that fulfill this type of work. While it is not essential that software developers have undergraduate or even advanced degrees to become proficient and create good software, the organizations which need software developers are beginning to see formal training as less important to meet the demands of these roles. As the availability of software developers has begun to create supply-demand problem, companies—especially those technology-focused companies such as Google, Apple, and IBM—are modifying or completing removing their requirements in which applicants for software developer roles have a college degree (Connley).

Software developers’ views are also evolving regarding the need for formal training. A recent survey of more than 116,000 developers showed that 31.9% of developers at small companies—those with less than 50 employees—did not complete a bachelor’s degree program (“HackerRank Developer Skills Report”). The same survey found that 14.9% of developers at large companies—those with more than 1,000 and less than 10,000 employees—did not complete a bachelor’s degree program. A 2020 survey from StackOverflow—a popular code example website as well as frequently used resource in software developer related studies—revealed out of 48,465 respondents that 9.8% believed that a formal education was “critically important” and that 16.1% said it was “not important at all” (“Stack Overflow Developer Survey 2020”). In addition, the 2021 StackOverflow surveyed showed that software developers with an undergraduate or
higher degree fell by 5.3% (“Stack Overflow Developer Survey 2021”). Finally, the last 3 years of StackOverflow surveys found that over 60% of software developers that did have an undergraduate degree majored in a computer science related discipline.

These studies demonstrate that support for formal education–regardless of discipline–is waning among software developers and, even when formal training has been utilized, the majority have chosen a computer science degree. This view of formal education ultimately translates to software developers having been part of fewer college-level writing courses as well as fewer opportunities for college-level writing assignments. While formal training might be less prevalent for software developers, those that having writing responsibilities still view writing abilities as the key to creating good and useful documentation. For example, all the participants in Bottomley’s study, What Part Writer? What Part Programmer? A Survey of Practices and Knowledge Used in Programmer Writing, “indicated that a technical background or a strong interest in learning technical skills was important for their work” (Bottomley 810), but still “the majority of interviewees felt that good writing skills were more important than technical skills” (Bottomley 810).

Of course, software developers can develop writing skills outside of college coursework or other formal training, but as Gieger et al. found “many participants expressed that they lacked the correct skills to write good documentation for their own software” (785). Overall, creating good documentation–regardless of category of software documentation–is more than just having subject-matter knowledge and having strong writing skills, but prioritizing formal training to further build confidence and competency. As one study participant in Gieger et al.’s study shared, “I don’t know
many people who enjoy writing documentation. I think one of the reasons being it’s not a skill that we learn” (786).

**Process and Technological Challenges**

In addition to the technical and writing skills, there are often processes and technological considerations involved in creating different types of software documentation. For example, authors–regardless of documentation type–also need contend with high likelihood that errors will be introduced within the code examples. As a respondent in Bottomley’s study commented, “‘There are so many ways that errors can creep into documentation, either by not understanding something or getting the wrong version of source code.’” (804). Even though code errors find their way into software documentation, certain types of software documentation more frequently take on the form of “living documentation” or a constant state of revision and updates, such as API documentation.

As new features are added, updated, or removed from the API, the documentation will expand in some sections, change in others, and then be removed altogether in other sections. This constant state of revision provides a route to not only update the documentation itself but also fix errors in code example through the same means and often can be published within a matter of minutes. In addition, software tools and automatic processes can help find and fix errors within APIs. For example, Zhong and Su utilized an approach, called *DocRef*, which combines natural language processing and data parsing to detect possible errors within API documentation (804). When they applied DOCREF and evaluated its effectiveness on the latest documentations of five widely used API libraries, it found more than 1,000 new documentation errors.
However, as published work, tutorial documentation tends toward stasis, making an error-free tutorial difficult to achieve. Even the use of a process like DocRef will often not overcome the resistance to change quite as well as the pressure of collaborative reviews or new contracts for publishing. For example, the route to update a book or a reference manual might require the creation of an entirely new edition which requires discussions and further agreements with the book’s publisher and then rounds of revisions with reviewers and editors. With a potential lengthy revision process for a subsequent edition for a book, the time from the initial idea to release of the edition might be months or longer. This lengthy process is one the reasons supporting the rise of more direct-publish, online tutorial documentation. Even though the medium changes, it still retains the structure as well as carries with it many of the same processes and safeguards prior to publishing. Dagenais and Robillard point out that since “books about open-source projects are not always updated, their main advantage lies in the improvement of the quality of the official documentation and the time that the contributors take to reflect on their design decisions” (131).

Tutorial authors also must contend with the technology platforms used to create, edit, and publish the software documentation. The evolution of software development has included methods to create and store not just prior software versions, but snapshots in time based on a specific contributor. These snapshots can be split into parallel tracks of programming work, which form what’s called a code repository, and each contributor’s work creates what is known as a branch. The technology used for code repositories and branches creates another layer of knowledge required for software developers and adds skills and workflows that software documentation writers must also learn. As Geiger et
al. points out, API and maintenance documentation will often “store documentation in the repository they use to store code, requiring a working knowledge of version control and online code repositories like GitHub” (786). Even though the documentation platform may be intuitive, it still means that writers for software documentation will need to use the same steps as those required for creating and modifying the code within the repository. In the case of an API or maintenance documentation, it will typically require the following steps:

1. creating a new code branch or pulling an existing branch of the code from the repository onto the author’s computer
2. creating new, deleting, or updating existing documentation
3. review the changes against existing documentation
4. pushing the changes to the branch within the repository
5. requesting the changes to be merged into the primary branch (which is also called the master branch)
6. waiting for someone in the project to review the changes
7. responding to any concerns regarding the changes
8. making requested updates based on the review

These steps then must be repeated for subsequent addition, changes, or deletions for the documentation and any of the code that is contained within the documentation.

In the case of a tutorial, the process will be twice as involved as API or maintenance documentation. For example, if an author is creating a book about a new programming language, the book might be created and editing in a word processing file, the condensed code examples kept in both the word processing file and a code repository,
and then the completed working application is kept in yet another code repository. The tutorial author is not only left to keep track of documentation and code in multiple locations but must do so in multiple formats. In addition, the crossover of code from one format to another could introduce new errors as well as require additional time for the process of transforming the code from one format to another.

**Structure, Standards & Guidelines**

Compared to other forms of software documentation, software tutorials have limited standardized or formal guidelines that can be used in creating or updating documentation. While ISO 26514, *Requirements for designers and developers of user documentation*, could serve as a blueprint for other types of software documentation, this ISO explicitly defines its primary audience as users of software—not its creators. In addition, user documentation focuses on known features and repeatable steps within software operations that are static and limited to what has been coded. These boundaries of the end-user software will do provides a scope for the documentation. Conversely, software tutorials will have their scope determined by whatever the author chooses to convey and its results are only set by what the software developers can surmise from the documentation.

API documentation typically has a formalized structure that is provided by the programming language creators, software vendor, or other group of software maintainers. For example, the Java programming language uses *Javadoc* to generate API documentation (“How to Write Doc Comments for the Javadoc Tool”). This provides a built-in structure for software developers to follow when creating an API using the Java programming language. The structure for software tutorials might have some influence
based on the standards provided from a book publisher. However, without the guidance of an industry standard, the structure will differ from publisher to publisher.

In addition, API documentation consists of natural language sentences, blocks of code samples, and code terminology—like *methods, classes, and variables*. This specific format as well as conventions provided vendors allow for API documentation authors to have consistency and structure to aid in the success of the documentation. For example, Meng and others provided an example outcome in *Optimizing API Documentation: Some Guidelines and Effects*. In their study, one group had access to documentation which had been optimized following guidelines for API documentation design and the other group used non-optimized documentation. Their results showed that developers working with optimized documentation made fewer errors on the test tasks as well as were faster in planning and executing the tasks (Meng et al. 8).

**Creating Code Examples**

Software developers rely upon code examples in their pursuit of many aspects of their roles, such as learning a new programming language, expanding their capabilities with a programming language with which they are already proficient, or figuring out how to utilize a new API documentation. For software documentation to be effective, it must provide good and clear code examples which can be used by developer to support fundamental steps, such as adequate testing of new code or improvements to existing code. The code examples created for tutorial software documentation will have some slight differences in how the examples are constructed as compared to API documentation, but there are also core elements that have similar traits. Regardless the type of software developer documentation, the studies and surveys across the literature
review consistently brought up four elements of good code examples including: conciseness, highlighting important elements, building upon prior examples, and providing additional resources.

**Creating Concise Examples**

Good code examples are essential to provide not only additional context for the accompanying tutorial but also to correctly represent how code should be implemented. Achieving this balance can be difficult: when creating code examples, a common challenge for all types of software documentation is to use examples that provide enough detail so that the learner can understand the topic or concept but avoid adding so much detail that it distracts from the specific topic. Developers need also need relatively generic examples that can illustrate how the code can be used in different use cases.

In *Application Programming Interface Documentation: What Do Software Developers Want?*, the developers in the study point out that “in order to benefit from a code example, brief and informative explanations are necessary that explain what is going on and why the code looks the way it does” and believe that “code examples are expected to be concise” (Meng et al. 323). The code provided within an example might use concepts, such as variable names, to enable both conciseness and clarity. As shown in figure 6, the code in the top right panel uses the variable named *requestOptions* to represent the variable name for the *RequestOptions* value. The author could have abbreviated the variable name or used any other arbitrary variable name to represent the value. However, the specific use of *requestOptions* as the variable name is both concise and clearly defines what function or method within the API is being utilized.
In addition, the prose shown in the left column provides the necessary context for the code in the right column. In both panels on the right, the snippets of code only provide the necessary details to cover implementation of the Authentication function and all other code is intentionally left out. As Ying and Robillard note that “the effectiveness or usability of a code example can generally be related to its intended usage, evidence is mounting that concise code examples are particularly desirable, especially for pedagogical purposes” (460).


Nashehi and others point out that another “way to make code concise is to leave unnecessary details out and show their absence with some place holders (such as comments or ellipses) which usually transforms the code to a solution skeleton.” (28). In figure 7, the intent is to show the read how just the “save” method should be implemented. The technique of placing ellipses before and after the “save” omits parts of
the code that, in this case, was covered in prior examples or, in some cases, might already be well understood by the audience.

```java
Listing 11-20. UserInterface

   public interface UserInterface {
       ...
       public GraphStory save(GraphStory graphStory) throws Exception;
       ...
   }
```

Figure 7. Save method for UserInterface class from Practical Neo4j. Author's screenshot.

**Highlighting Important Elements**

Another important characteristic of good examples is calling out or highlighting important elements within the code or about the code being executed. Ying and Robillard cited several studies that support the notion of software developers rely upon examples for everyday work (460). The most effective examples often showed an ability to surface the most important aspects of the code. As one study participant noted, "It’s tough to know the context of the example and yet it has to be very small, and only highlight exactly what the concept in the API is that you’re looking for" (Ying and Robillard 460).
Figure 8. Removing a Node from *Practical Neo4j*. Author's screenshot.

For example, code provided within the documentation typically illustrates a primary process that will be executed, such as removing a specific node from a database show in figure 8. However, the code in figure 8 also requires a previous process to run–specifically the noted by the comment //get the node. It might have been understood that the node must be selected before it can be deleted–which is a common process for managing data records. So, a pair of ellipses could have been used decrease the overall size of the code example but decreasing the code coverage–simply for the sake of brevity–would have left out critical component for the developer’s consideration. figure 8 also demonstrates another example of highlighting, specifically found in the Note box just above the code. Without this callout regarding the behavior of relationships impacting the removal of nodes, developers may have been confused as to why certain nodes could not be removed while other were removed without any issues.
Building Upon Prior Examples

A third concept that helps create good code examples is utilizing or building upon examples from other sections of the documentation. As Meng and others write it is helpful especially “in [the] case of more complex examples—be organized as a series of consecutive chunks” (“Application Programming Interface Documentation” 323). In figure 9, the code demonstrates how to retrieve data from the database using a specific label. Next, part of the code from figure 9 is re-used in figure 10 to introduce a new concept and the repetition also helps to reinforce the example learned in figure 9.

```cypher
MATCH (u:User {username: "greg"})
RETURN u.fullName, u.email, u.username
```

Figure 9. Matching on a Label from Practical Neo4j. Author's screenshot.

```cypher
MATCH (u:User {username: "greg"})
SET u.fullName = 'Greg Jordan'
RETURN u
```

Figure 10. Updating a node property from Practical Neo4j. Author's screenshot.

Providing Additional Resources

A final trait of creating good code examples is providing additional resources or context for the code examples. As Subramanian et al. note, external resources for code examples often provided additional context but “the official documentation does not link to the examples that could help developers, and the examples rarely link to the documentation.” (643). The study looked at ways to integrate code examples in different languages to reference to official vendor-created documentation. figure 11 shows a
similar technique where the navigation highlighted within the code examples provides a way for the reader to navigate from the tutorial to the specific details explaining components of the code.


Providing additional resources can also increase the code example coverage. In figure 12, the Stripe API Authentication allows the software developer to select the programming language that matches with their project or task. The dropdown is embedded within the code example, so the developer does not need to navigate away from the current section of the API. Figure 11 provides a similar navigational tool for languages applying to its API.
**Authentication**

The Stripe API uses API keys to authenticate requests. You can view and manage your API keys in the Stripe Dashboard.

Test mode secret keys have the prefix `sk_test_`, and live mode secret keys have the prefix `sk_live_`. Alternatively, you can use restricted API keys for granular permissions.

Your API keys carry many privileges, so be sure to keep them secure! Do not share your secret API keys in publicly accessible areas such as GitHub, client-side code, and so forth.

Use your API key by assigning it to `Stripe.apiKey`. The Java library will then automatically send this key in each request.

You can also set a per-request key with an option. This is often useful for Connect applications that use multiple API keys during the lifetime of a process.

All API requests must be made over HTTPS. Calls made over plain HTTP will fail. API requests without authentication will also fail.

Related video: Authentication.

Was this section helpful?  Yes  No

---


**Scaffolding and Tutorial Effectiveness**

One approach to bringing the research around software documentation challenges and good code examples together in an effective tutorial employs the use scaffolding. As Quintana and others point out, scaffolding works to address some need for an audience, first with the scaffold functionality created in a conceptual idea and then followed up with its implementation in a physical design. Following this process, we might identify the audience need as learning a new data design. If the scaffolding concept is a capturing the data design within a single step, then the physical implementation might be represented by one form and an accompanying table that captures the data. If the scaffolding concept is a multi-step form with validation at each step, then the physical implementation might be represented by process map outlining each step with its own table or interrelated tables (Quintana 82).
As noted by Linder and others, the scaffolding helps introduce complex topics by creating supports which build upon each other. “These supports may be social or cognitive: Properly managed group assignments are one example of social scaffolding, while tiered assignments that build up students’ understanding incrementally are an example of cognitive scaffolding” (Linder 239). Long-form software tutorials can incorporate scaffolding by creating a different version of “tiered assignments” which will ultimately help the reader build upon their knowledge as the move through the examples.

**Summary**

Good documentation is critical for creating as well as maintaining effective and reliable software. Even though good documentation leads to good software, the type studied by technical communication research to the greatest degree is the documentation intended for end-users of software product. Furthermore, there is an even more limited amount of research which specifically focuses long-form documentation aimed at software developers, such as books and manuals. However, there are common issues and challenges that exist across all types of software developer documentation and the available literature offers useful insight when comparing the various formats of software developer documentation as well as to help highlight shared concepts.
Chapter 3: Methodology

Introduction

The purpose of the research is to explore the impact of providing examples for a secondary, supporting technology in a tutorial that teaches a new technology. This section describes the participants, data collection process, and analysis. As suggested by Whitenton (“Triangulation: Get Better Research Results”) and shown in figure 13, the study will be supported by using a triangulation of the following methods:

1. Skill level: survey information on external program knowledge and self-reported skill levels.
2. Task metrics: time on task and task success rates.
3. Qualitative comments: users’ experience from remote observation & post-survey.

Figure 13. Research Triangulation for Usability Study. Author’s Figure.
As noted in *Characteristic-Based, Task-Based, and Results-Based: Three Value Systems for Assessing Professionally Produced Technical Communication Products*, these three systems can be a guide to assess the value of technical communication (Carliner 83). However, using the constraint of the task-based aspect allows the focus given the limited scope of the study.

In addition, it is important to focus on the aspects that align with the primary goals of the tutorial. Again, as noted in Quesenbery’s dimensions of usability and shown in figure 14, this provides a framework to emphasize the most important of dimension of the tutorial: whether it is effective (4). While the dimensions shown in the figure could suggest the same relative weight, Quesenbery’s dimensions are designed to allow each dimension its own weight and that the weight for each dimension can be set based on the context of use. As Quesenbery’s points out, “it would be convenient if each of the dimensions of usability were equally important in every product. They are not, just as the usability requirements for a product depend in part on the context of use” (4). While the four other dimensions obviously play a role, they are a much smaller percentage—10% each—of the entire usability score. In addition, the study will measure participants completion rate and time on task using a secondary technology—in this case the Java language—of which the participants will have some knowledge.
Participants

The participant pool started with software programmers pre-screened for the following characteristics:

- each participant has some Java programming language experience in a work role
- each participant possesses no to limited experience using graph database technology as a primary role

Java was selected as the language for the participants’ experience and the study based on Java’s popularity and widespread usage in software applications over the past twenty years (‘Java | TIOBE - The Software Quality Company’). The pool of participants was drawn from my professional network, including co-workers of the author, and from programmer interest groups with which I am a member. In the group of co-worker participants, none of the participants had subordinate or supervisory role with the author. Since the participants must have some programming experience with Java, I completed a
stratified sampling of the participants. Given the scope of the study and using guidance from Nielsen (“How Many Test Users in a Usability Study?”), the ideal, target size of the participants included between 5 and 10 programmers from this initial pool. As shown in figure 15, Nielsen Norman Group evaluated the usability findings in 83 case studies and had many usability findings using 10 or fewer participants. While 10 or fewer participants does not establish statistical validity, it does support the primary goal of the usability tests which is to discover issues or problems that might surface during the testing sessions. As Macefield stresses, there is no “one size fits all” but suggests the usability research community sees 5 to 10 participants as ideal “for studies related to problem discovery” (38). Overall, the study’s aims as well as the context and complexity should be factors in determining a study’s size.

![Figure 15. Number of Test Users from: “How Many Test Users in a Usability Study?”](https://www.nngroup.com/articles/how-many-test-users/)

With the understanding that some subjects would have availability conflicts between the request to participate and the actual participation date, the schedule offered alternate dates and time for participation. The first 10 to respond that they were able participate were chosen as the participant. If a participant had a conflict or was unable to make the originally schedule session, the schedule allowed for up to 4 weeks from the original participation date to reschedule.

Data Collection

The research data was collected using a survey of the test subjects followed by a usability test of a software tutorial and concluded with post-usability evaluation. The surveys and usability test were completed remotely using Zoom, and I was the proctor for the survey and usability test. In preparation for the study, I performed initial tests to ensure the forms and necessary software would work as expected. I selected two programmers with Java experience and conducted pilot tests with both the surveys and usability test. The primary goals of the pilot test were to ensure the written task instructions were clear, to verify the design is efficient, and remove any discovered problems. During the pilot tests, the surveys were completed as expected. However, first coding task displayed some code that should have been removed before the pilot test. It was corrected and re-tested during a subsequent pilot test.

Survey & Usability Testing

The programmer survey, shown in Appendix B, was provided prior to starting the usability test and was created and collected using the Qualtrics online survey software. The first two questions of the survey gathered the participants’ self-report of their programming background and the third and final asked participants what they considered
the necessary skill level of secondary technology of reader when learning a new technology. The participants were provided the link to the programmer survey via a message window at the start of the Zoom session.

**Study Materials**

Each participant in the usability session were asked to read tutorial documentation and then accomplish core tasks based on the material in the tutorial. Appendix B provides both the tutorial documentation as well as the tasks. The tutorial provided examples using the version 8 release of the Java programming language, and the task environment provided instructions to complete the tasks using the same version of Java as the tutorial documentation. The expectation was that the tutorial documentation would take approximately 10 minutes to read, and the participant would then be given two different tasks to perform with a total time limit of 30 minutes. Each of the tasks included the same instructions for each participant, and the participants would be provided with as many attempts as necessary to complete tasks within the allotted time. Once the participant had completed the first task, they could move onto the second task. For the Java and database example code, the participants were shown code as it relates directly to the primary technology and excludes code not necessary to demonstrate the primary technology. The tutorial also provided a link back to the tutorial documentation.

**Post Usability Survey & Follow-up**

The post-task survey of the usability session, shown in Appendix C, was created and collected using the Qualtrics online survey software. This survey allowed the participants to rate the effectiveness of the tutorial based on their perceived satisfaction in their task completion and rating of tutorial effectiveness. Following the post-task survey,
I had the opportunity to ask open-ended questions that about actions I observed during their interaction with the tutorial and tasks. For example, if a participant paused during a portion of the tutorial, then, as part of a mixed methods approach, I could draw out feedback directly from the participant which might have indicated, for example, confusion or misunderstanding. On the other hand, a behavior that looks like the participant was confused might be a misinterpretation that is explained by a reason unrelated to the tutorial, such as a participant simply pausing to reflect on an example.

**Data Capture for Analysis**

The primary data points resulted from the participants’ task completion information—both the time on task and whether they successfully completed the task. The tutorial reading time and time on tasks were collected for each participant as well as the mean time for each task and a mean overall time on tasks. I then grouped each task data point based on the participants’ self-reported Java experience and expertise ratings. The task completion resulted in either a zero (did not complete task) or one (completed task) point assignment and was collected for each participant, calculated the mean result for each task, a mean result for an overall score, and then for each group of participants based on their self-selected Java experience and expertise ratings. The participants’ post-survey responses were also used to shed light on observations made during the usability session and gather more information that might help explain their behaviors and performance during the usability session.
Chapter 4: Results and Analysis

Introduction

The study took place over an 8-week span and initially included 10 participants. Of the 10 total participants who were scheduled for a usability session, only 9 were able to attend the scheduled or rescheduled usability session. The participant who was unable to make their initial scheduled usability session was also unable to reschedule within the 4-week window allotted for rescheduling.

For the data provided in the following sections, time-based activity is displayed by showing minutes and seconds in this format: m: ss. In addition, each section provides the individual participant scores and then is followed by, where applicable, the participant’s specific grouping, e.g., programming experience and self-selecting programing rating.

Participant & Survey Data

The initial survey, shown in Appendix B, provided to each participant asked 4 questions. The first two questions and participants response are shown in Table 3.

Table 3. Overall Professional programming and Java programming experience by Participant

<table>
<thead>
<tr>
<th>Participant #</th>
<th>Professional Programming Experience</th>
<th>Java Programming Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Less than 3 years</td>
<td>Between 3 to 5 years</td>
</tr>
<tr>
<td>2</td>
<td>Between 3 to 5 years</td>
<td>Less than 3 years</td>
</tr>
<tr>
<td>3</td>
<td>More than 10 years</td>
<td>More than 10 years</td>
</tr>
<tr>
<td>4</td>
<td>Between 5 to 10 years</td>
<td>Less than 3 years</td>
</tr>
<tr>
<td>5</td>
<td>Between 5 to 10 years</td>
<td>Between 5 to 10 years</td>
</tr>
<tr>
<td>6</td>
<td>More than 10 years</td>
<td>Between 5 to 10 years</td>
</tr>
<tr>
<td>7</td>
<td>Less than 3 years</td>
<td>Between 3 to 5 years</td>
</tr>
<tr>
<td>8</td>
<td>Less than 3 years</td>
<td>Less than 3 years</td>
</tr>
<tr>
<td>9</td>
<td>More than 10 years</td>
<td>Between 5 to 10 years</td>
</tr>
</tbody>
</table>
These two questions were focused on group participants by experience level in both overall professional programming experience as well as Java programming experience. Table 4 and Table 5 show the aggregate count for the participant by the four experience level groupings. The experience level grouping would further allow the results to shown mean or aggregate count and help support findings discussed in chapter 5.

Table 4. Professional programming experience

<table>
<thead>
<tr>
<th>Professional Programming Experience</th>
<th># Of Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>3</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>1</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>2</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 5. Java programming experience

<table>
<thead>
<tr>
<th>Java Programming Experience</th>
<th># Of Participants (out of 9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>3</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>2</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>3</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>1</td>
</tr>
</tbody>
</table>

The initial survey also asked the participants to rate their Java programming skill on a scale of 1 to 10, where 10 would be considered an expert level. Finally, the survey asked the participants what they would expect the skill rating of a reader for a secondary technology when learning of new technology, e.g., learning how to connect to a database.
(new technology) using Java (secondary technology). Each participant score for both the overall programming experience and secondary programming experience are provided in Table 6 along with the mean scores for each. The mean for overall programming experience was 5.44 and Java programming experience was 5.78.

Table 6. Java self-rating and skill-rating for tutorial reader

<table>
<thead>
<tr>
<th>Participant #</th>
<th>Self-rating of Java programming skill</th>
<th>Skill-rating for reader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>8</td>
</tr>
</tbody>
</table>

To round out the initial survey data for the self-rating and reader skill-rating, I have compiled the aggregate counts as shown in tables 7 and 8.
Table 7. Java self-rating

<table>
<thead>
<tr>
<th>Self-rating of Java programming skill</th>
<th># Of Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Skill-rating for tutorial reader

<table>
<thead>
<tr>
<th>Skill-rating for reader</th>
<th># Of Participants</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

**Tutorial Data**

After completing the initial survey, the participants were given a link to the tutorial documentation, which is shown in Appendix B. The start time and end time for
each participant was tracked and is provided in Table 9. The expectation was the participants would be able to complete the tutorial in under 10 minutes. The mean reading time for the participant group was 6 minutes and 19 seconds.

Table 9. Reading times for tutorial

<table>
<thead>
<tr>
<th>Participant</th>
<th>Tutorial reading time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5:11</td>
</tr>
<tr>
<td>2</td>
<td>5:13</td>
</tr>
<tr>
<td>3</td>
<td>7:35</td>
</tr>
<tr>
<td>4</td>
<td>7:39</td>
</tr>
<tr>
<td>5</td>
<td>4:28</td>
</tr>
<tr>
<td>6</td>
<td>4:14</td>
</tr>
<tr>
<td>7</td>
<td>10:28</td>
</tr>
<tr>
<td>8</td>
<td>4:14</td>
</tr>
<tr>
<td>9</td>
<td>7:35</td>
</tr>
</tbody>
</table>

Task Data

After completing the tutorial, the participants were given two tasks to complete based on the tutorial they just completed. The first task, as shown in Appendix B, instructed the participants to add additional code to the existing code block which would connect to the database, create a new “Person” object, and return the object from the database. As noted in chapter 3, the participants were reminded that the code they were adding would be compared to the correct version of the code. Each of the participants time on task as well as their score for each task is shown in Table 10.
Table 10. Time on tasks by participant

<table>
<thead>
<tr>
<th>Participant #</th>
<th>Task #1: time</th>
<th>Task #1: score</th>
<th>Task #2: time</th>
<th>Task #2: score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4:09</td>
<td>1</td>
<td>0:35</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3:35</td>
<td>1</td>
<td>0:58</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1:16</td>
<td>1</td>
<td>0:53</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2:46</td>
<td>1</td>
<td>2:48</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1:59</td>
<td>1</td>
<td>1:26</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1:07</td>
<td>1</td>
<td>0:43</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3:38</td>
<td>1</td>
<td>1:19</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2:02</td>
<td>1</td>
<td>1:02</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1:16</td>
<td>1</td>
<td>0:53</td>
<td>1</td>
</tr>
</tbody>
</table>

As shown in table 10, each of the participants were able to successfully complete both tasks with a mean time of 2:25 and 1:11, respectively. Other than participants #1 and #2 requiring two attempts on task one, all participants were able to complete the tasks on their initial attempt.

Table 11. Mean time on tasks by professional programming experience

<table>
<thead>
<tr>
<th>Professional Programming Experience</th>
<th>Task #1</th>
<th>Task #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>3:16</td>
<td>0:59</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>3:35</td>
<td>0:58</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>2:22</td>
<td>2:07</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>1:13</td>
<td>0:50</td>
</tr>
</tbody>
</table>
Table 12. Mean time on tasks by java programming experience

<table>
<thead>
<tr>
<th>Java Programming Experience</th>
<th>Task #1</th>
<th>Task #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>2:48</td>
<td>1:36</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>3:54</td>
<td>0:57</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>1:27</td>
<td>1:01</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>1:16</td>
<td>0:53</td>
</tr>
</tbody>
</table>

For individual participants, the time to complete tasks went down between tasks #1 and #2 with the exception of participant #4. In addition, the mean time to complete tasks for all groups went down between tasks #1 and #2 with the exception of the Java self-rating “2”.

Table 13. Mean time on tasks by Java self-rating

<table>
<thead>
<tr>
<th>Java Self-Rating</th>
<th>Task #1</th>
<th>Task #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2:46</td>
<td>2:48</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3:35</td>
<td>0:58</td>
</tr>
<tr>
<td>5</td>
<td>3:38</td>
<td>1:19</td>
</tr>
<tr>
<td>6</td>
<td>2:29</td>
<td>0:50</td>
</tr>
<tr>
<td>7</td>
<td>1:07</td>
<td>0:43</td>
</tr>
<tr>
<td>8</td>
<td>1:37</td>
<td>1:10</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Post Usability Survey Data

After completing the tasks, the participants were given a follow-up survey in which they were asked to rate the effectiveness of the tutorial on (1) whether it enhanced their understanding of the main topic (graph databases) and (2) completing the tasks. Both questions were on a scale of 1 to 10, where 10 would be considered the most effective.

Table 14. Tutorial effectiveness ranking - understanding topic & task completion by participant

<table>
<thead>
<tr>
<th>Participant</th>
<th>Understanding topic</th>
<th>Task completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 15. Tutorial effectiveness ranking - understanding topic & task completion by professional programming experience

<table>
<thead>
<tr>
<th>Professional Programming Experience</th>
<th>Understanding topic</th>
<th>Task completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Across all participants, the mean score for effectiveness towards enhanced understanding was an 8 and the mean score for effectiveness towards completing the tasks was a 10. While table 14 shows the individual score for each of the participants, tables 15, 16 and 17 show the scores based on the various groupings.

Table 16. Tutorial effectiveness - understanding topic & task completion by Java programming experience

<table>
<thead>
<tr>
<th>Java Programming Experience</th>
<th>Understanding topic</th>
<th>Task completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 17. Effectiveness of the tutorial - understanding and task completion by Java self-rating

<table>
<thead>
<tr>
<th>Java Self-Rating</th>
<th>Understanding topic</th>
<th>Task completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Qualitative & Observational Data

All the participants conducted the tasks using Google’s Chrome browser. Seven of the attendees used Apple’s operating system while the two used Microsoft Windows as their operating system. Since the participants were allowed to use the tutorial documentation during their task sessions, I also observed how they interacted with the tutorial. Specifically, I wanted to see if the used the tutorial in completing one or both tasks. I also wanted to see if the participants used the same tab that contained the tutorial—assuming they kept it open—or used the link on the task page, which would create a new tab with the tutorial. Finally, I wanted to see if any of the participants would copy the text from the tutorial and paste it into the task form. Table 18 shows the usage of the tutorial during the tasks for each participant based on each type of interaction.

Table 18. Tutorial usage by participant

<table>
<thead>
<tr>
<th>Participant #</th>
<th>Used tutorial</th>
<th>Used tutorial in both tasks</th>
<th>Opened a new window or used tab</th>
<th>Copy/pasted from tutorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>9</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Tables 19, 20 and 21 show the usage of the tutorial during the tasks for each participant group based on each type of interaction. It should be noted that the grouping here is done as a total count for each grouping rather than a mean value.

Table 19. Tutorial usage by Professional programming experience

<table>
<thead>
<tr>
<th>Professional Programming Experience</th>
<th>Used tutorial</th>
<th>Used tutorial in both tasks</th>
<th>New window or used tab</th>
<th>Copy/pasted from tutorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 20. Tutorial usage by Java programming experience

<table>
<thead>
<tr>
<th>Java Programming Experience</th>
<th>Used tutorial</th>
<th>Used tutorial in both tasks</th>
<th>New window or used tab</th>
<th>Copy/pasted from tutorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than 3 years</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Between 3 and 5 years</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Between 5 and 10 years</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>More than 10 years</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Finally, participants were asked in open-ended discussion whether they had noticed that the tutorial documentation excluded some Java code, specifically “Did you notice imports and class were missing in the examples?” All participants said that they had noticed it was not included. One participant remarked that it “was not important to understanding the example” and another said, “If there’s enough documentation and enough examples, I can follow along.”
Chapter 5: Discussion

Introduction

This focus of this chapter is to interpret the usability results in conjunction with the two research questions, review suggested conclusions which were formed from the data from the usability sessions, and then discuss the recommendations found for this study and that might be included in further studies. The research questions sought to examine the impact of having prior knowledge of a secondary technology towards learning a new technology and how examples of the secondary technology influence learning a new technology. As discussed in chapter 4, the study began with a participant survey, which was conducted to understand the participants level of programming experience through self-evaluations and would help to establish how audiences would be classified for the programming tutorial usability sessions. In confluence with the participant self-classifications, their interaction with the tutorial and working through subsequent tasks helped to demonstrate an approach for gauging the impact of prior knowledge of a secondary technology towards understanding the primary technology as well as seeing how programmers utilize code examples when followed by simulated coding tasks. The research questions and conclusion sections of this chapter also aim to incorporate the usability sessions’ quantitative and qualitative data to address other discoveries made in the study. I will begin the chapter by addressing the research questions based on what the outcomes suggest as answers and then compare against what was learned from the usability sessions.
Research Questions

The research questions driving my study are:

**R1: Does prior knowledge of a supporting technology affect learning a new technology?** Based on the results of the tasks, the mean time data does at least suggest that software developers with more experience are more likely to complete programming tasks more quickly than those with less experience when learning a new technology. In addition, participants with more experience were less likely to use the tutorial in the tasks. However, each of the participants—regardless of their level of experience—were able to complete both tasks, which could also simply suggest that there are no significant usability problems with the tutorial. Further, almost all the participants utilized the documentation during the first task assignment, which might only imply that some similarities in the tasks also had an impact on learning effectiveness.

**R2: Does providing external code examples of a secondary, supporting technology in a tutorial help make learning a new technology more effective?** With limiting the tutorial code examples and code tasks to just two items—along with statistical viability of the participant sample size—the study cannot conclusively show that examples of secondary technology make learning a new technology more effective. However, the data and feedback can at least suggest that there are no significant usability problems with the approach used in the tutorial, especially concerning the formatting and depth of the secondary technology in the code examples. Even though the study only used two examples, one participant shared that “if there’s enough documentation and enough examples, I can follow along.” In support of this notion, we can also look at how each participant was asked “Did you notice imports and class were missing in the examples?”
and responded “yes”. As noted by another participant, the removing of this part of the Java code “was not important to understanding the example”.

Conclusions

The data collected from sessions was used to discover if problems existed with the tutorial's approach as well as help address the study’s broad research questions. While the participant size for this study does not provide statistical validity, the results, in terms of usability, indicate no significant problems concerning the approach used in the tutorial. As each of the participants were able to complete the tasks based on the information provided within the documentation, the outcomes from the sessions suggest that using a known secondary technology to “scaffold” programmer learning of an unknown primary technology was a sound approach (Quintana 82).

Further, at least at this beginning foray into the learning situation for programmers, the data suggests that including a known, supporting technology to learn an unknown, primary technology did not introduce problems. The results showed that a mean time on the second task at a participant level, experience level and self-rating level were lower versus the first task, which helps support the scaffolding approach. With only one outlier, the overall task time on both tasks went down as the experience level went up for either overall professional programming or Java programming as well as the self-rating level for Java expertise.

Finally, the data can suggest that there are no significant usability problems with the approach used in the tutorial and tasks as related to providing external code examples. Again, the mean time on the second task at a participant level as well as a group level for experience and self-rating versus the first task. With only one outlier, the overall task
time on both tasks trended downward as the experience level increased upward in either overall professional programming or Java programming as well as the self-rating level for Java expertise. Ideally, this study would help provide additional context toward understanding how tutorials can apply both experience and domain knowledge in software tutorials, especially in relation to audience skill level and depth of code examples.

**Recommendations**

The study provided some further insight into the usage of tutorials for software programmers, specifically scaffolding the learner’s access to the primary technology via a secondary technology they already know—and by including and formatting code examples it may make learning a new technology more effective.

However, there are additions and changes to the study that would bolster the data’s validity in its current approach. First, the sample group should be expanded to include programmers that lack experience with the primary technology as well as the supporting technology. If one of the aims is to understand how code examples and structure impact learning effectiveness, then using participants with different bands of experience levels for the secondary technology is a good beginning. However, the control group (those with experience in the secondary technology) should have a more clearly differentiated variable group (those without experience in the secondary technology but with some programming experience). Finally, the tasks should be created and tested so that the score reflects more than just matching a participant’s code against the baseline code. The formatting of code in a real-world scenario has few restraints, so a better test score would also match against results from executing the code.
Another change would be providing a task environment that was isolated from the tutorial or even split the participants between those who had tutorial access and those who had no access to the tutorial. It is important to understand how the participants interact with the tutorial during tasks, but it provided a new depth of comparison to compare to those who complete task without access. This scenario would closely mimic some real-world scenarios where access to documentation might be constrained by a paywall or firewall.

A significant expansion of the study could include a much larger sample group rendering data that is appropriate for statistical analysis could make the preliminary findings reported here more rigorous. Future studies could also draw from across domain experience, such as balancing programmer-participants across retail, finance, healthcare, and other industries. Also, the tutorial for the study in this dissertation was limited to a chapter section in order to better manage the scope. To extend study in this area beyond the scope of this one, a future study that increases the in-depth coverage of the primary technology. A suitable protocol could be further developed from this study and then applied across more sections of the tutorial. In addition, future studies should expand the time frame to accommodate changing schedules for participants as well as include a mix of remote and on-site usability sessions. In addition, new participant survey questions could further quantify and classify "prior knowledge of the supporting technology" as well as open comparisons with participants who do not have prior experience the supporting technology with those who have experience.
References


https://www.slideshare.net/Chetan2608/software-requirements specification-of-library-management-system.


“Stack Overflow Developer Survey 2019.” *Stack Overflow*,


“Stack Overflow Developer Survey 2020.” *Stack Overflow*,


“Stack Overflow Developer Survey 2021.” *Stack Overflow*,


Appendix A: Programmer Survey

1. How many years of professional programming experience do you have?
   a. Less than 3 years
   b. Between 3 to 5 years
   c. Between 5 to 10 years
   d. More than 10 years

2. How much Java programming language experience do you have?
   a. Less than 3 years
   b. Between 3 to 5 years
   c. Between 5 to 10 years
   d. More than 10 years

3. What would you rate your Java programming language skill level? [1 to 10, 10 being expert]

4. When learning a new technology, such as a database system, what should be the skill level of the reader for the secondary technology, e.g., connecting to a database using Java? [1 to 10, 10 being expert]
Appendix B: Study Materials

Working with Graphs

What do NASA, Bank of America, and Comcast have in common with many academic and research projects? They all depend on graph databases as a strategic part of how they store and analyze information.

Why have such a wide range of industries and fields found a common relationship through graph databases? The short answer is graphs offer superior analytical performance over other types of databases while also maintaining change resiliency.

In this introductory chapter, we are going to cover the graph database called Neo4j and how to do some basic operations. Neo4j what is called an LP graph (labeled property graph) as opposed to an RDF graph (Resource Description Framework graph). If you are new to graph databases, then one way to view the system is by imagining a network of objects with each object connected to one another by one or more relationships. In graph terminology, the objects are called vertices and the relationships are called edges.

By understanding how these edges and vertices are connected, it allows companies like Facebook, Netflix, and Amazon to make better product and content recommendations, more quickly spot trends, and more easily recognize nodes with the most influence.

![Facebook, Netflix, Amazon](image)

*Figure 1-1. Examples of Graphs*
Like relational databases, graph databases can store data for both transactional and analytic purposes – also known as online transactional processing (OLTP) and online analytical processing (OLAP), respectively. However, graphs have an advantage over relational systems for certain types of operations, especially data analysis with complex and varied relationships.

Because of their specific design, graphs allow for queries to execute much faster and in a much more consistent manner. As evidence, it is why companies like Twitter and LinkedIn moved from relational databases to purpose-built graph databases to handle relationship data at a global scale.

**Nodes and Relationships**

When discussing graph databases, vertices are more commonly referred to as *nodes* and edges are more commonly referred to as *relationships* (Figure 1-2). Although these terms may be used interchangeably, this book follows the common usage uses the terms *nodes* and *relationships*.

![Figure 1-2. Two nodes connected by a relationship](image)

A node contains attributes or properties that typically fit the context of the type of node. The relationship typically has direction, as shown in Figure 1-2, to convey how the relationships is stored and read. Unlike the primary and foreign keys that connect rows within a relational database, relationships within a graph database can also have attributes or properties.
Managing Nodes and Relationships

Managing nodes and their properties will likely be the bulk of your application’s graph-related code. The maintenance of nodes is set in motion with the creation process. Creating a node begins with setting up a connection to the database, as shown in Listing 1.

Listing 1. Creating a Node

// import & class statements removed for brevity

private final Driver driver;

public static void main(String... args) throws Exception{
    try (Neo4jJavaExample nje = new Neo4jJavaExample("bolt://localhost:7687", "neo4j", "password") )
    {
        nje.createPerson("someName");
    }
}

public Neo4jJavaExample(String uri, String user, String password)
{
    driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password));
}

@Override
public void close() throws Exception{
    driver.close();
}

public void createPerson(final String name)
{
    try (Session session = driver.session())
    {
        String nje = session.writeTransaction(tx ->
        {
            Result result = tx.run("CREATE (:Person {name: $name} ) RETURN p.name + ', from node ' + id(a)"");
            return result.single().get(0).asString();
        });
        System.out.println(nje);
    }
}
Retrieving and Updating a Node

Once nodes have been added to the database, we will need a way to retrieve and modify them. Listing 2 shows the process for finding a node by its node id value and updating it.

Listing 2. Retrieving and Updating a Node

// import & class statements removed for brevity

```java
private final Driver driver;

public static void main(String... args) throws Exception{
    try ( Neo4jJavaExample nje = new Neo4jJavaExample( "bolt://localhost:7687", "neo4j", "password" ) ) {
        nje.updatePerson("Blue","Emil");
    }
}

public Neo4jJavaExample(String uri, String user, String password) {
    driver = GraphDatabase.driver( uri, AuthTokens.basic( user, password ) );
}

@Override
public void close() throws Exception{
    driver.close();
}

public void updatePerson(String favoriteColor, String name) {
    try ( Session session = driver.session() ) {
        String nje = session.writeTransaction( tx -> {
            Result result = tx.run("MATCH (p:Person {name: $name}) SET p.favoriteColor = $favoriteColor + 
                "RETURN p.name + ', from node ' + id(a)",
                parameters( "favoriteColor", favoriteColor, "name", name ) );
            return result.single().get( 0 ).asString();
        } );
        System.out.println(nje);
    }
}
```

Summary

At this point, you should be comfortable with the basics of connecting to Neo4j and then creating, matching on, and updating a node. The link below provides a quick exercise to cover what we learned here. Please click the link to start the exercise.
Labels

Neo4j uses the concept of labels and were introduced to group nodes. As the example in Figure 1-3 demonstrates, you can define a node as “Person” and then provide additional values for each property of the node as necessary. By grouping nodes in this way, we can query the graph to show common subsets of node types and how they might be related. Because labeling offers a way to categorize through an index, this process increases the speed at which data can be queried. Labeling also offers a way to enforce attribute constraints, e.g., all Person and Business nodes must have a name.

![Diagram of labeled nodes](image)

*Figure 1-3. Labels provide a way for nodes to be categorized*

In the next section, we will cover basic operations in Neo4j using the Java programming language.
Task #1

Follow these steps to proceed. You can refer to the documentation at any time.

1. Only modify code as requested. Don't add additional spaces or tabs.
2. Add code so it will create a person named Emil.
3. Finally, Your code is being compared to a version that compiles correctly. Review the code and make changes so that it will compile correctly.

```java
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
import org.neo4j.driver.Session;

import static org.neo4j.driver.Values.parameters;

public class Neo4jJavaExample implements AutoCloseable {
    private final Driver driver;

    public static void main(String... args) throws Exception {
        try (Neo4jJavaExample nje = new Neo4jJavaExample("bolt://localhost:7687", "neo4j", "password") ) {
            
        }
    }

    public Neo4jJavaExample(String uri, String user, String password) {
        driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password));
    }

    @Override
    public void close() throws Exception {
        driver.close();
    }

    public static Person createPerson(final String name) {
        try (Session session = driver.session()) {
            String nje = session.writeTransaction(tx -> {
                Result result = tx.run("CREATE (p: Person) SET p.name = \$name", parameters("name", name));
                return result.single().get("id").asString();
            });
            System.out.println(nje);
        }
    }
```
Task #2

Follow these steps to proceed. You can refer to the documentation at any time.

1. Only modify code as requested. Don’t add additional spaces or tabs.
2. Add code so it will update a person named Emil favorite color to Blue
3. Finally, Your code is being compared to a version that compiles correctly. Review the code and only make changes so that it will compile correctly

```java
import org.neo4j.driver.AuthTokens;
import org.neo4j.driver.Driver;
import org.neo4j.driver.GraphDatabase;
import org.neo4j.driver.Result;
import org.neo4j.driver.Session;
import static org.neo4j.driver.Values.parameters;

public class Neo4jJavaExample implements AutoCloseable {
    private final Driver driver;

    public Neo4jJavaExample(String uri, String user, String password) {
        driver = GraphDatabase.driver(uri, AuthTokens.basic(user, password));
    }

    @Override
    public void close() throws Exception {
        driver.close();
    }

    public void updatePerson(String favoriteColor, String name) {
        String nje = session.writeTransaction(tx -> {
            Result result = tx.run("MATCH (p:Person {name: $name}) SET p.favoriteColor = $favoriteColor"
                           + "RETURN p.name ",
                           parameters("$favoriteColor", favoriteColor, "name", name));
            return result.single().get(0).asString();
        });
        System.out.println(nje);
    }
}
```
Appendix C: Post Usability Survey

1. In terms of understanding the database technology, how would you rate the effectiveness of the tutorial? [1 to 10, 10 being most effective]

2. In terms of completing the tasks, how would you rate the effectiveness of the tutorial? [1 to 10, 10 being most effective]
Appendix D: IRB Approval

Institutional Review Board
Division of Research and Innovation
Office of Research Compliance
University of Memphis
315 Admin Bldg
Memphis, TN 38152-3370

May 26, 2021

PI Name: Gregory Jordan
Co-Investigators:
Advisor and/or Co-PI: Loel Kim
Submission Type: Initial
Title: Evaluating the Impact of a Supporting Technology Documentation on Learning New Technology
IRB ID: #PROC-FY2021-385
Exempt Approval: May 19, 2021

The University of Memphis Institutional Review Board, FWA00000815, has reviewed your submission in accordance with all applicable statuses and regulations as well as ethical principles.

Approval of this project is given with the following obligations:

1. When the project is finished a completion submission is required
2. Any changes to the approved protocol requires board approval prior to implementation
3. When necessary submit an incident/adverse events for board review
4. Human subjects training is required every 2 years and is to be kept current at citiprogram.org.

For any additional questions or concerns please contact us at irb@memphis.edu or 901.678.2705

Thank you,
James P. Whelan, Ph.D.
Institutional Review Board Chair
The University of Memphis.