# Patterns of Knowledge in API Reference Documentation

Walid Maalej and Martin P. Robillard

**Abstract**—Reading reference documentation is an important part of programming with APIs. Reference documentation complements the API by providing information not obvious from the syntax of the API. To improve the quality of reference documentation and the efficiency with which the relevant information it contains can be accessed, we must first understand its content. We report on a study of the nature and organization of knowledge contained in the reference documentation of the hundreds of APIs provided as part of two major technology platforms: Java SDK 6 and .NET 4.0. Our study involved the development of a taxonomy of knowledge types based on grounded methods and independent empirical validation. Seventeen trained coders used the taxonomy to rate a total of 5574 randomly-sampled documentation units to assess the knowledge they contain. Our results provide a comprehensive perspective on the *patterns of knowledge* in API documentation: observations about the types of knowledge it contains, and how this knowledge is distributed throughout the documentation. The taxonomy and patterns of knowledge we present in this paper can be used to help practitioners evaluate the content of their API documentation, better organize their documentation, and limit the amount of low-value content. They also provides a vocabulary that can help structure and facilitate discussions about the content of APIs.

## 1 Introduction

Application Programming Interfaces (APIs) enable the reuse of libraries and frameworks in software development. In essence, an API is a contract between the component providing a functionality and the component using that functionality (the client). The syntactic information is, in all but the most trivial cases, insufficient to allow a developer to correctly use the API in a programming task. First, interfaces abstract complex behavior, knowledge of which may be necessary to understand a feature. Second, even if the behavior of a component could be completely specified by its interface, developers often need ancillary knowledge about that element: how it relates to domain terms, how to combine it with other elements, etc. [27]. This knowledge is generally provided by documentation, in particular, by the API's *reference documentation*.

We define API reference documentation as a set of documents indexed by API element name, where each document specifically provides information about an element (class, method, etc.). For example, the API documentation of the Java Development Toolkit (JDK) is a set of web pages, one for each package or type in the API. Although many forms of API documentation exist, there is usually a clear distinction between reference documentation and other forms of documentation with a more pedagogical intent (e.g., tutorials, books, and FAQs).

- *W. Maalej is with the Institute of Informatics, University of Hamburg, Germany*
  *E-mail: maalejw@cs.tum.edu*
- *M.P. Robillard is with the School of Computer Science, McGill University, Montréal, QC, Canada*
  *E-mail: martin@cs.mcgill.ca*

Reference documentation is a necessary and significant part of a framework. For example, the reference documentation of the JDK 6 (SE and EE) totals over three million words, or six times the length of Tolstoy's epic novel, "War and Peace". Reference documentation also plays a crucial role in how developers learn and use an API, and developers can have high expectations about the information they should find therein [13], [27]. Empirical studies have described how developers have numerous and varied questions about the use of APIs (see Section 8). Efficient representation and access of knowledge in API reference documentation is therefore a likely factor for improving software development productivity.

Most technology platforms exposing APIs provide a documentation system with a uniform structure and look-and-feel for presenting and organizing the API documentation. For example, Java APIs are documented through Javadocs, documentation for Python modules can be generated with the pydoc utility, and Microsoft technologies, whose documentation is available through the MSDN website, follow the same look-and-feel. Unfortunately, no standard and a few conventions exist regarding the *content* of reference documentation. For example, an early article explains the rationale behind Javadocs and gives a set of conventions for what should and should not be part of Javadocs [18]. In practice, however, these conventions are not generally adhered to. A cursory look at an API reference documentation page will typically reveal a mixed bag of information items, including definitions, code snippets, specifications, how-to guides, implementation notes, references to other documents, etc. An extreme example is offered by the package overview pages of the JDK, which range from one

sentence about the functionality of the package to a detailed specification of the package's elements.

To reason about the quality and value of API reference documentation, we must first know about *what* knowledge it contains, and *how* this knowledge is organized. We refer to observations about these properties jointly as *patterns of knowledge* in API documentation. Without a basis for studying documentation content, we have no hope of measuring and improving it. To provide such a baseline, we undertook a systematic empirical investigation of the patterns of knowledge in the reference documentation of two major technology platforms: JDK 6 and .NET 4.0.

The methodological cornerstone of our study is the use of *content analysis* techniques [23], which involves the systematic review of a document sample by human *coders*, who rate documents for various types of contents according to a strict *coding guide*. Overall our study involved 17 coders who independently coded 5574 randomly-sampled documentation units totaling 431 136 words. For each documentation unit, *two* coders independently assessed whether the unit contained a certain knowledge type, such as functionality, implementation directives, or usage patterns. Determining a *reliable* taxonomy of knowledge types for API documentation was an important undertaking of this research project.

Our study provides the first comprehensive perspective on the patterns of knowledge in API reference documentation, i.e., **about the different types of knowledge it contains and how this knowledge is distributed among documents**. Among others, we made the following observations:

- Documentation for the JDK and .NET differ in that JDK documentation contains more conceptual knowledge whereas .NET contains more information about the structure of the API and its usage patterns (see Section 7.1).
- There is no major difference in the type of documentation content attached to classes vs. interfaces (see Section 6.1).
- 43.3% of documentation units attached to API class members in the JDK and 51.0% in .NET contain information of little or no value (see Section 6.1).

### *Overview and Contributions*

Section 2 formalizes our research questions and provides an overview of the research methods.

Section 3 presents our taxonomy of knowledge types, and describes the methods employed to elaborate it. **The taxonomy of knowledge types for API documentation forms the first contribution of this paper**.

Section 4 describes how we designed the quantitative part of the study following the principles of content analysis, and the tools developed for this purpose.

Section 5 describes how we evaluated the quality of our data through systematic analysis of inter-coder agreement. This analysis validated the taxonomy and enabled us to clean the data from which we derive our results. **Our tools and methods for conducting content analysis studies of software engineering artifacts is the second contribution of this paper**.

Section 6 analyzes the distribution of knowledge types in Java and .NET along different dimensions. **The quantitative characterization of the distribution of knowledge types across API reference documentation (a set of knowledge patterns) is the third contribution of this paper**. This characterization supports the definition of the concept of *documentation style* as a collection of knowledge patterns applicable to a cohesive subset of documents.

Section 7 discusses the main outcomes of the study, their implications for the software development practice related to API documentation, and the future research they motivate.

Section 8 reviews the related work **and compares the categorizations of knowledge types in software engineering, which forms the fourth contribution of this paper.**

### *Implications*

The taxonomy and patterns of knowledge we present in this paper can be used to help practitioners evaluate the content of their API documentation, better organize their documentation, and limit the amount of low-value content. They also provides a vocabulary that can help structure and facilitate discussions about the content of APIs. A more detailed discussion of these implications can be found in Section 7.2.

## 2 Study Design

We introduce our research questions and the process we followed to answer them.

### 2.1 Research Questions

Our fundamental goal was to discover *what* is described in API reference documentation, and *how* this content is organized, independently of official organizational templates imposed by documentation systems. Our specific research questions were:

1) What are the *different* types of knowledge captured in API reference documentation?
2) Can we reliably identify these knowledge types in free-form text?
3) How are these knowledge types distributed across different groups of API elements (e.g., classes, interfaces, fields, and methods)?
4) How are different knowledge types combined?
5) Are there differences in the distribution of knowledge types between different technology platforms?

Our research questions involve important concepts, such as *knowledge* and *knowledge types*. Providing a rigorous definition for such terms is in itself a non-trivial problem. For an overview of the research questions, the intuitive definition of the above terms should be sufficient. The next section provides precise, empirically-derived definitions.

In addition to knowledge types, our research questions rely on two other important concepts: documentation page, and documentation unit. We consider a *documentation page* to be a single web page or equivalent document as supplied by the documentation system for an API. In contrast, a *documentation unit* defines the documentation specifically associated with an API element (a class, method, field, etc.). How documentation units relate to pages depends on the documentation system. For example, in .NET most units are on their own page, whereas in the JDK documentation units of members are found directly on the documentation page of their declaring type. Our study is concerned with documentation units.

Finally, we note the overloading of the term *type*. We distinguish between *knowledge types* as defined in Section 3, and *programming language types* (i.e., classes, interfaces, enums, etc.). We avoid ambiguity by using the complete expression "knowledge type" when the meaning is unclear from the context.

### 2.2 Overview of the Methodology

Our research is based on content analysis, a methodology for studying the content of recorded human communications, and is inspired by Neuendorf [23]. Figure 1 summarizes the process we followed to answer our research questions. The complete process includes a mixture of qualitative and quantitative methods organized in four sequential phases.

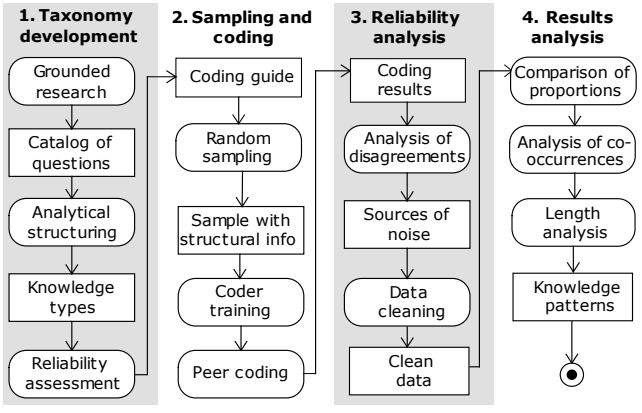


Fig. 1. Overview of the research process

Phase 1 addressed the **first research question** using a combination of grounded and analytical methods to derive a taxonomy of knowledge types.

In Phase 2 we used the taxonomy as a coding guide and had 17 trained coders review a random sample of documentation units to assess whether each unit contained knowledge of the different types in our taxonomy. Each knowledge type became a *variable* that had to be *rated* with the value True (if it is present in the unit) or False (if not). We use the expression *rating a unit* to mean rating all variables for the unit. In this phase each documentation unit was independently rated by two randomly-assigned coders. The result was a database of *ratings*, which also contained *disagreements* for some variables in some documentation units (e.g., for the documentation unit of method $m$ coder 1 rated the presence of knowledge type $T$ as True and coder 2 coded it as False).

In Phase 3 we systematically analyzed the disagreements to *a)* evaluate the work of the coders, *b)* evaluate the quality of the guide, and *c)* design a scheme to resolve disagreements. This analysis allowed us to answer our **second research question**. After applying the data cleaning scheme, each rated variable in a unit was reconciled into a single value: True or False.

In Phase 4 we conducted statistical analyses on the clean data to answer the **last three research questions**.

Each study phase required making decisions about the experimental design. These decisions involved tradeoffs that impact the **threats to the validity** of the results. We discuss our decisions and their impact on the validity throughout Sections 3–6.

## 3 A Taxonomy of Knowledge Types

The most challenging part of this research project consisted of describing the different knowledge types commonly found in API reference documentation. Many authors have discussed the different types of knowledge used in various software engineering contexts, and in some cases provided empirically-grounded taxonomies. Unfortunately, a careful review of previous work (see Section 8) showed that existing taxonomies are neither directly applicable to API documentation, nor sufficiently detailed to be directly used as knowledge types definitions for our purpose. We thus elaborated a taxonomy of knowledge types for API reference documentation through an iterative refinement process. Our goal was to produce a taxonomy that would be:

1) Reliable, in that different people consistently come to the same conclusion about the knowledge types contained in a documentation unit.
2) Meaningful, listing knowledge types relevant to the practice of software development.
3) Fined-grained, providing more than just a few high-level categories.

The outcome of this process was a detailed taxonomy of knowledge types usable as a *coding guide* for the quantitative analysis of the content of API documentation. Producing a coding guide that fulfilled the above goals required over six months of on-going experi-

mentation and revision to elaborate and is offered as one of the main contributions of this paper.[1]

Table 1 summarizes our final taxonomy, in which we differentiate between 12 types of knowledge. In the title of each knowledge type, the text in bold indicates the shorthand used throughout this paper.

We elaborated our taxonomy through numerous iterations grouped into the following three steps, each with its own methodological underpinnings.

### 3.1 Knowledge Type Identification

In the first step, we used a grounded approach to elicit a preliminary list of knowledge types present in API reference documentation. The idea of a grounded approach is that [23, p.102]:

> When existing theory or research cannot give a complete picture of the message pool, the researcher may need to immerse himself or herself in the world of the message pool and conduct a qualitative scrutiny of a representative subset of the content to be examined.

Each author independently selected sentences from the reference documentation of two different open-source systems: HttpComponents[2] and Jena.[3] These systems were selected for their mature and extensive reference documentation. To select sentences, we employed a process inspired by *theoretical sampling* [3]. This involves refining and adjusting the sampling procedure as data is collected. A theoretical sample is generally not representative of a population, but it ensures the data set captures as many aspects of the phenomenon of interest as possible. This was the goal in this study phase.

API documentation is expressed in natural language, and its overwhelming variety of style and content escapes any obvious classification. Our initial attempts at elaborating a classification for knowledge types only based on sentences led to unsatisfactory, ever-shifting, indistinct categories. To meet the classification challenge, we transformed each sentence into the main *question* it answered. The use of questions to model knowledge and information needs in software development has been successfully used by other researchers in the past to capture and reason about similar types of information independently of their context [8], [17], [30].

This step resulted in a collection of over 100 questions. We then reconciled similar questions, merged groups of questions differing only in small variants, and reworded the questions to achieve consistency in style.

### 3.2 Analytical Structuring

A limitation of the grounded approach is that it does not guarantee that all knowledge types will be uncovered. In a second step we refined our catalog with a detailed review of the literature (see Section 8) and through analytical reasoning, expanding all variation points for a question template. For instance if a question was "what is the meaning of a return value?", we added variations like "what is the meaning of a parameter?.

As part of this process we assessed the reliability of our catalog by independently coding individual sentences in randomly-selected sets of API elements in open-source APIs others that those distributed as part of the JDK 6.0 and .NET 4.0.[4] The goal of this evaluation was to determine if any obvious questions had been left out, and assess the ease of associating sentences with questions that model knowledge types. Each author tagged each sentence with the question(s) that best represented the knowledge it captured. During this phase, agreement between the two authors varied between 58% and 84%. Based on this experience and a study of the disagreements we encountered, we made three major changes to our taxonomy.

First, we gave up on the idea to associate knowledge types to individual sentences, and instead analyzed documentation units as a whole. Second, as most units contained several knowledge types, we decided to change our measurement scheme. We split the single question (which knowledge types are contained in this unit) into a set of individual knowledge type questions (how much knowledge of type T is contained in this unit). Third, because it is unreasonable to expect coders to code 48 variables for each documentation unit (and also because of the similarity and overlap between questions), we grouped the questions into 12 variables (roughly corresponding to the 12 final types described in Table 1).

### 3.3 Testing and Reliability Assessment

In the last step, we tested the reliability of our taxonomy by iteratively coding various random samples of 50 units, studying the disagreements, and making improvements based on the findings. As part of this process, we added an increasing number of clarifications to the coding guide about how to code different variables. The samples consisted of documentation units which were to be coded for about 12 variables that represent to what degree knowledge of different types was present. We experimented with different discrete scales, going from an initial 0–3 scale down to a binary one. We conducted three formal reliability tests, measuring inter-coder agreement for each variable. We measured the reliability using

1. We submitted the coding guide as attachment to this paper. We plan to publicly release the guide, the coding tool, and the samples.
2. http://hc.apache.org
3. http://jena.sourceforge.net
4. We used HttpComponents, Jena, SWT, and Hibernate.

TABLE 1
Taxonomy of Knowledge Types (Summary)

| Knowledge Type | Description (Excerpt) |
|---|---|
| **Functionality** and Behavior | Describes what the API does (or does not do) in terms of functionality or features. Describes what happens when the API is used (a field value is set, or a method is called). |
| **Concepts** | Explains the meaning of terms used to name or describe an API element, or describes design or domain concepts used or implemented by the API. |
| **Directives** | Specifies what users are allowed / not allowed to do with the API element. Directives are clear contracts. |
| **Purpose** and Rationale | Explains the purpose of providing an element or the rationale of a certain design decision. Typically, this is information that answers a "why" question: Why is this element provided by the API? Why is this designed this way? Why would we want to use this? |
| **Quality** Attributes and Internal Aspects | Describes quality attributes of the API, also known as non-functional requirements, for example, the performance implications. Also applies to information about the API's internal implementation that is only indirectly related to its observable behavior. |
| **Control**-Flow | Describes how the API (or the framework) manages the flow of control, for example by stating what events cause a certain callback to be triggered, or by listing the order in which API methods will be automatically called by the framework itself. |
| **Structure** | Describes the internal organization of a compound element (e.g. important classes, fields, or methods), information about type hierarchies, or how elements are related to each other. |
| **Patterns** | Describes how to accomplish specific outcomes with the API, for example, how to implement a certain scenario, how the behavior of an element can be customized, etc. |
| Code **Examples** | Provides code examples of how to use and combine elements to implement certain functionality or design outcomes. |
| **Environment** | Describes aspects related to the environment in which the API is used, but not the API directly, e.g., compatibility issues, differences between versions, or licensing information. |
| **References** | Includes any pointer to external documents, either in the form of hyperlinks, tagged "see also" reference, or mentions of other documents (such as standards or manuals). |
| **Non-information** | A section of documentation containing any complete sentence or self-contained fragment of text that provides only uninformative boilerplate text. |

Cohen's Kappa metric, which accounts for chance agreement (and only measures agreement over and above chance) on a scale of 0–1. We note that there is no universal agreement about how to interpret this value. However, the measure is generally considered very conservative, with values 0.61–0.80 considered "substantial" and 0.81–1.00 "almost perfect" in one influential paper [19, p.165].

The per-variable kappa agreement values increased from an average of 0.31 in the first reliability test, to 0.567 in the second, to 0.664 in the third. In each iteration we studied disagreements and reorganized the guide correspondingly. For example, after the first iteration we merged the Purpose and Rationale knowledge types. After the second iteration we introduced the Non-information knowledge type; and after the third iteration we reduced the scale to a binary one (True and False).

The guide received its final structure with this last step, but underwent additional cosmetic and clarification improvements based on initial feedback from the coders (as part of the training phase only).

## 4 Sampling and Coding

We describe the sampling and coding procedures to collect a data set of knowledge types found in a representative sample of documentation units for our two target systems, JDK 6 and .NET 4.0. We chose to study two specific technology platforms to address the challenge of *representativeness*. Although it would be desirable to collect data that could represent API reference documentation in general, this goal is unachievable because the population of all APIs is unknown and unbounded. By studying a well-defined and understood population of API elements, we can use simple statistical tools to ensure that our observations are representative *within this population.* We chose JDK and .NET as study targets because they were very large fully-documented APIs, covering a broad spectrum of functionality, heavily-used, roughly equivalent, and yet offering interesting contrasts. Although each technology has its own particularities, the study of the documentation for the hundreds of APIs in these two major technology platforms already provides a strong basis for understanding the content and organization of API documentation for systems that benefited from major investments.

### 4.1 Unit of Sampling

The unit of sampling for our study is the *documentation unit*, which provides information associated with a specific API element. The format of a documentation unit varies depending on the nature of the API element it describes. We distinguish between three different types of documentation units, which map to three different levels of containments for API elements: modules (which corresponds to Java packages and .NET assemblies[5]), types (mostly classes and interfaces), and type members (fields and methods).

We exclude module-units from the study because they form a small and very heterogeneous population

5. In the .NET framework both *assemblies* and *namespaces* are high-level units of containment. We chose assemblies because they form a unit of both logical and physical code organization [10].

TABLE 2
Population and sample sizes

| | Java Development Kit | | | .NET Framework | | |
|---|---|---|---|---|---|---|
| | **Population** | **Calculated** | **Sample** | **Population** | **Calculated** | **Sample** |
| Number of Modules | 297 | | | 78 | | |
| Number of Types | 5664 | 1209 | 1243 | 7736 | 1282 | 1283 |
| Number of Members (Methods, Fields, etc.) | 52 871 | 1493 | 1549 | 57 216 | 1496 | 1499 |
| (Number of Types + Number of Members) | 58 535 | 2702 | 2792 | 64 952 | 2778 | 2782 |
| Average Number of Types per Module | 19.1 (*SD*=28.4) | | | 99.2 (*SD*=149.2) | | |
| Average Number of Members per Module | 178.0 (*SD*=398.5) | | | 733.5 (*SD*=1083.8) | | |

of documents. Both JDK and .NET modules show extreme variation in size (see the last two rows of Table 2). Additionally, .NET module overview pages typically provide no documentation and are rather used as navigation hubs. Our study therefore investigated type-level and member-level units.

### 4.2 Target populations

For the purpose of sampling, we consider that type-level and member-level documentation units form distinct populations. We can easily make hypotheses about why the content of documentation units for these two element types would be different, and indeed the quantitative results presented in Section 6 provide overwhelming evidence of significant differences between them. For this reason, it would be an experimental mistake to consider them equivalent for the purpose of sampling. Hence, conceptually, our study targets four distinct populations: Java types, Java members, .NET types, and .NET members. The Java populations include the documentation units for all the types and members defined in the 297 packages forming the union of the Java 6 SE and EE specifications. The .NET populations include the documentation units for the types and members defined in all 78 assemblies comprising the .NET 4.0 framework. Table 2 shows the size of the type and member populations for both Java and .NET (column Population). The last two rows describe the distribution of types and members across modules.

### 4.3 Sampling

Our primary concern was to produce samples that are as representative of their respective population as possible. We thus randomly sampled $N_p$ elements from each population. For each of the four populations $p \in \{\mathrm{JavaType}, \mathrm{JavaMember}, .\mathrm{NETType}, .\mathrm{NETMember}\}$, $N_p$ was chosen so that questions asked on the sample would be representative of the population within a 2.5% confidence interval and 95% confidence level. That is, we are 95% confident that our results are generalizable to the entire population within a 2.5 error rate [28, p.210]. Column "Calculated" in Table 2 shows the sample size calculated in that way for each population.

For a study of reference documentation, the representativeness of a uniform random sample is threatened by the distribution of types and members among modules (Java packages and .NET assemblies). Under the reasonable assumption that modules cluster API elements that support similar functionality or domain tasks, we can expect that the documentation for the elements in a module can be influenced by the module in which they are defined. Because in both JDK and .NET, the number of types and members within modules shows extreme variability (see the last two rows of Table 2), a uniform random sampling strategy has the risk that small modules with only a few types will not be represented at all. To mitigate this threat, we adopted a *stratified random* sampling strategy [23, p.85]. Within each module we randomly sampled units from the types and members populations in proportion to the number of types and members in the module, respectively. This ensured that the sample included units from all modules, while respecting the overall distribution among modules. If the calculated fraction of units for a module, after rounding to the closest integer, resulted in a zero value, we systematically increased this value to 1 to ensure that all modules are represented. This procedure slightly increased the size of the samples. Columns "Sample" in Table 2 show the final sample sizes, taking into account the adjustment for the stratification.

### 4.4 Coding Procedure

Our sample consists of 5574 documentation units. With the redundancy required to measure inter-coder agreement, we needed to collect a total of $5574 \times 2 = 11\,148$ *ratings* of individual documentation units. The data collection task consisted of reading a documentation unit and, for each knowledge type described in Table 1, indicating whether this knowledge type was present or not in the unit. With an estimate of 90 seconds per rating, this task would require an effort of 279 person-hours, or about 36 days of full-time, highly-focused, and specialized work. Collecting the data required a large team of coders.

#### 4.4.1 Coder Recruitment and Training

Seventeen coders participated in the data collection effort: the two authors, 9 M.Sc. students and part-time developers from the Technische Universität München (TUM), 5 researchers who are pursuing their Ph.Ds from the same institution, as well as one M.Sc. student from McGill University. All coders had at least two years of programming experience and were familiar with the use of APIs and the reading of their documentation. All coders intensively used the development environments Eclipse and/or Visual Studio for at least one year.

To train the coders, we provided them with the coding guide and an exercise consisting of 30 documentation units to rate. The training set was composed of documentation units that were not in the sample and combined class, interface, method, and field documentation units. After the coders completed the exercise, we sent them an answer key. The coders were then asked to compare their answers with the answer key and clarify any misunderstanding by either studying the differences or asking questions. Finally, we held a 75-minute plenary training session with all the coders present either in person or by video-conference. During the session we clarified the issues encountered during the exercise. We then integrated these clarifications in the official coding guide used for the study. The training session was video-recorded and made available to coders.

All coders were compensated for their work. Six of them were hired specially to perform the coding. The rest were already employed as research assistants in one of the authors' institution. As an additional incentive to do the task carefully, we promised a 200 EUR gift card for an on-line store to the coder with the highest overall agreement (see Section 5).

#### 4.4.2 Coding Tool and Data Collection

We integrated all aspects of sampling and data collection in a tool called CADo (Content Analysis for API Documentation), developed as part of this research project. CADo is a distributed application built on top of a database. The database stores the entire population of documentation units from which it can generate random samples. The database also stores the list of coders. CADo can then automatically generate random assignments of documentation units to coders. Finally, CADo includes a rich client that allows coders to log in and view their assigned units one-by-one, enter their ratings, and store them in the database. The client also presents the description of all knowledge types, to facilitate coding. Figure 2 shows a screenshot of the CADo client used by coders. The single coding window includes a view of the documentation unit (A), containment and structural information about the associated element (B), 12 checkboxes corresponding to the 12 knowledge type variables (C), and a tool-tip window showing the description of a knowledge type extracted from the coding guide (D).

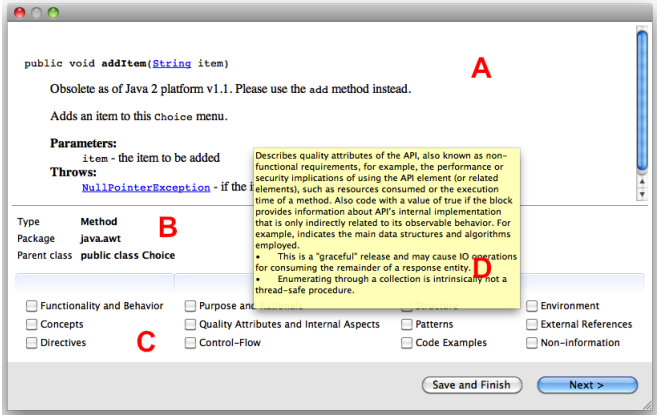


Fig. 2. A screenshot of our Content Analysis for API Documentation tool (CADo) used by a coder

Each unit in the database was randomly assigned to two coders while balancing the overall number of units per coder, and keeping the proportion of Java and .NET units per coder roughly equal. Each coder received 628–790 units to code.[6] Once all coders were informed of their assignment, we collected the data over a period of approximately three months. We note that this does not mean that each coder took three months to complete their assignment. Rather, coders completed their task according their availability in a total time-span of three months. During this time, all preparatory and reference material for their task remained available.

## 5 Coder Agreement

Before turning to a detailed analysis of the patterns of knowledge in reference documentation, we first provide an analysis of the agreement between coders. Inter-coder agreement measures the extent to which different coders provided the same rating (True or False) for a given knowledge type. An analysis of inter-coder agreement (or agreement, for short) provides us with the answer to three important questions. First, *agreement per coder* (Section 5.1) helps us assess the quality of the work done by each coder, and ensures that all coders performed their task in earnest. *Agreement per variable* (Section 5.2) assesses to what extent the presence of different knowledge types as described in our coding guide can be reliably and independently assessed by humans. Finally, we use agreement data to draw conclusions about potential *causes of disagreements* between coders through a qualitative analysis (Section 5.3), and to derive measures to clean the data (Section 5.4).

6. The differences were caused by meeting all coding constraints, and since a small number of corrupted data had to be re-coded.

TABLE 3
Agreement by Coder

| Variable | Min | Median | Max | Worst | Best |
|---|---|---|---|---|---|
| Functionality | 0.618 | 0.701 | 0.749 | $C_{10}$ | $C_5$ |
| Concepts | 0.658 | 0.794 | 0.844 | $C_8$ | $C_2$ |
| Directives | 0.753 | 0.830 | 0.864 | $C_6$ | $C_2$ |
| Purpose | 0.598 | 0.764 | 0.813 | $C_4$ | $C_{10}$ |
| Quality | 0.770 | 0.909 | 0.944 | $C_{14}$ | $C_4$ |
| Control | 0.816 | 0.887 | 0.904 | $C_8$ | $C_{13}$ |
| Structure | 0.624 | 0.703 | 0.781 | $C_8$ | $C_{17}$ |
| Patterns | 0.802 | 0.877 | 0.915 | $C_8$ | $C_4$ |
| Examples | 0.956 | 0.969 | 0.981 | $C_6$ | $C_{15}$ |
| Environment | 0.450 | 0.923 | 0.952 | $C_4$ | $C_{16}$ |
| References | 0.309 | 0.826 | 0.890 | $C_4$ | $C_{17}$ |
| Non-info | 0.595 | 0.712 | 0.770 | $C_1$ | $C_{16}$ |
| **Overall** | 0.723 | 0.820 | 0.840 | $C_4$ | $C_{16}$ |

TABLE 4
Agreement by Variable

| Variable | Frequency (f) | Chance | Agreement |
|---|---|---|---|
| Functionality | 0.681 | 0.566 | 0.702 |
| Concepts | 0.179 | 0.706 | 0.762 |
| Directives | 0.186 | 0.698 | 0.821 |
| Purpose | 0.217 | 0.660 | 0.749 |
| Quality | 0.070 | 0.869 | 0.898 |
| Control | 0.102 | 0.817 | 0.878 |
| Structure | 0.289 | 0.589 | 0.703 |
| Patterns | 0.097 | 0.825 | 0.876 |
| Examples | 0.132 | 0.770 | 0.969 |
| Environment | 0.069 | 0.871 | 0.897 |
| References | 0.183 | 0.701 | 0.796 |
| Non-info | 0.268 | 0.608 | 0.706 |

### 5.1 Agreement by Coder

Any documentation unit coded by one coder was also coded by a second, randomly-selected co-coder. The assignment randomization was done per unit. That is, a coder $C_i$ could in theory have up to 16 different co-coders for his assigned units. For a given variable, we calculated the agreement score for $C_i$ by dividing the number of ratings which agree with the second coder by the total number of units coded by $C_i$. For example, if $C_1$ coded 500 units, and 400 of their Functionality ratings were the same as the ratings of the co-coder, $C_1$'s agreement score for Functionality would be $400/500 = 0.80$. We calculated the overall agreement by summing all agreements over all variables and dividing by the number of units times 12 (the number of variables).

Table 3 reports on the coders' performance. The first three numeric columns report on the minimum, median, and maximum agreement observed for all coders for a variable. The last two columns show the identification number of the coder with the worst (minimum) and best (maximum) agreement.

Overall, we conclude that the performance of coders is adequate for the purpose of the study and that all coders performed their task in earnest. In particular, the median agreement is above 0.7 for all variables and the overall median agreement is above 0.8. Although some coders did not perform very well, no single coder systematically under-performed across all variables. From this analysis, the only problem we detect is that coder 4 did not correctly interpret the guide for variables Environment and References. Except for this one outlier (representing about 1.9% of the ratings),[7] all coders had agreement above 0.87 for Environment and 0.63 for Reference.

### 5.2 Agreement by Variable

Agreement by variable measures to what extent two independent coders will agree on the value of a knowledge type variable for a given unit. This measure is important because it is a direct assessment of the reliability of the coding guide. If two independent coders usually agree on the presence or absence of a given knowledge type, then we can be confident that this type, as described, corresponds to a cohesive concept easily understood by documentation readers.

We measure agreement for a variable by dividing the number of units for which the two coders agreed (i.e., both rated True or both rated False), over the total number of units. This measure of *raw agreement* is one among many alternatives for measuring agreement by variable [23]. The advantage of the raw agreement measure is that it is simple to interpret. The disadvantage is that it does not take into account that coders might agree by chance. With binary variables the probability of chance agreement is 0.5. Other measures of agreement, such as Cohen's Kappa [23], take the distribution of ratings by coders into account. Although calculating the Kappa metric for two coders is simple, it is unclear how this can be done for multiple coders. Independently of the strategy chosen, the interpretation of the metric is tenuous [23].

We therefore report raw agreements, along with a conservative estimate of chance agreement that is sensitive to the distribution of True values in the sample. Indeed, a baseline of 0.5 for chance agreement implies that two coders would choose blindly one of two values with probability $p = 0.5$. In practice, if coders are aware that certain knowledge types are either very pervasive or very rare, they can increase their chances by erring on the side of the more popular value (True or False). If coders can "guess" the relative frequency of a certain knowledge type in the sample, they can rate that variable as True with the probability that corresponds to this frequency. Given this behavior model, the chance agreement increases above 0.5. For example, if both coders correctly guess that Control is only present 10% of the time and rate this variable randomly as True with $p = 0.10$, the chance of random agreement becomes $p^2 + (1-p)^2 = 0.82$.

Table 4 reports on the agreement by variable. The

7. Coder 4 rated $637/5574$ units $\times 2/12$ problem variables $= 1.9\%$ of all ratings.

TABLE 5
Disagreement rates: JDK vs. .NET and members vs. types

| Variable | JDK | .NET | p | $\|\phi\|$ | Members | Types | p | $\|\phi\|$ |
|---|---|---|---|---|---|---|---|---|
| Functionality | 0.271 | **0.324** | 0.00002 | 0.057 | 0.278 | **0.321** | 0.00055 | 0.047 |
| Concepts | 0.229 | 0.246 | 0.14 | | 0.205 | **0.277** | 0.00000 | 0.085 |
| Directives | 0.186 | 0.171 | 0.18 | | 0.150 | **0.213** | 0.00000 | 0.081 |
| Purpose | 0.248 | 0.254 | 0.63 | | 0.175 | **0.343** | 0.00000 | 0.193 |
| Quality | 0.093 | 0.111 | 0.030 | | 0.076 | **0.134** | 0.00000 | 0.095 |
| Control | 0.119 | 0.124 | 0.62 | | 0.081 | **0.171** | 0.00000 | 0.136 |
| Structure | 0.268 | **0.326** | 0.00000 | 0.063 | 0.232 | **0.375** | 0.00000 | 0.155 |
| Patterns | 0.105 | **0.143** | 0.00002 | 0.058 | 0.072 | **0.187** | 0.00000 | 0.174 |
| Examples | 0.023 | **0.038** | 0.00174 | 0.043 | 0.022 | **0.041** | 0.00005 | 0.055 |
| Environment | 0.079 | **0.128** | 0.00000 | 0.081 | 0.067 | **0.147** | 0.00000 | 0.131 |
| References | 0.183 | **0.224** | 0.00015 | 0.051 | 0.174 | **0.239** | 0.00000 | 0.080 |
| Non-info | 0.264 | **0.325** | 0.00000 | 0.067 | **0.360** | 0.215 | 0.00000 | 0.159 |

first numeric column reports the frequency $f$ of the knowledge type (number of True values for that variable divided by the number of ratings). The second numeric column shows the distribution-aware probability of chance agreement ($f^2 + (1-f)^2$). The last column shows the actual observed agreement.[8]

The results show that all raw agreements are well above the basic threshold of 0.5 (all values are above 0.7). Moreover, all agreement values are above the much more conservative threshold of distribution-aware chance agreement. We conclude that coders were able to reliably assess the presence or absence of different knowledge types in API documentation.

### 5.3 Analysis of Disagreements

Disagreements between coders are interesting because they can reveal the difficulty of rating a variable for a documentation unit. This can be caused by either properties of the coding guide (e.g., unclear, ambiguous), or properties of the units themselves (unclear, badly organized, etc.).

We investigated, for each variable, the frequency of disagreement between coders and whether there exists a bias in the rate of disagreement in JDK vs. .NET and types vs. members.

Table 5 reports on the analysis. The first two numerical columns show, respectively, the frequency of disagreement in the JDK and .NET populations. For example, for the Functionality knowledge type, coders disagreed over 27.1% of the elements in the JDK and over 32.4% of the elements in .NET. We estimated the significance of the difference between proportions using a $\chi^2$ (chi-squared) test of independence. Our test is applied to a 2-by-2 contingency matrix for the variables Platform { JDK, .NET } and Disagreement {Yes, No}. The $\chi^2$ test of independence computes the probability of obtaining a given distribution of values in a contingency table if the two variables are truly independent. The third column reports the p-value of the test. For example, if the variable "Platform" was independent from the variable "Disagreement", the probability of observing the proportions 0.271 and 0.324 would be 0.00002. Following the usual tradition for interpreting p-values, we consider that differences with $p < 0.01$ are statistically significant [28]. For statistically significant differences, we highlight the greatest ratio in **bold**.

For all statistically significant results, we also report the effect size in terms of the $\phi$ (phi) coefficient. The $\phi$-coefficient is a special case of Pearson's R product moment coefficient for two dichotomous variables. The $\phi$-coefficient ranges between -1 and +1 indicating inverse or direct perfect association, respectively. In our case, a $\phi$ value of 1 means that there is a disagreement for all JDK elements, and no disagreement for any .NET element (and vice-versa for a value of -1). A $\phi$-coefficient of 0 means that exactly the same proportion of disagreements is observed for JDK and .NET. The meaning of intermediate values is open to interpretation, but generally 0.1 is considered a small effect, 0.3 a medium effect, and $> 0.5$ a strong effect [9, p. 474]. For ease of interpretation, we provide the absolute value of the $\phi$-coefficient. The direction of the difference is obvious from the magnitude of the proportions.

We observe that .NET units consistently involve more disagreement than JDK units. The effect size is small, but the fact that there are significant differences for seven variables that consistently show more disagreement for .NET, supports the hypothesis that .NET units are more difficult to rate. In Section **??** we comment on differences that may be a factor of documentation clarity. In the case of types vs. members, we observe that types are consistently more difficult to rate than members with the exception of Non-information. This observation can be explained by the fact that type-level documentation is usually longer and much richer than member-level. For Non-information, this knowledge type is usually more present in members (where, e.g., the name of a method is simply repeated in the documentation), so

8. The agreement numbers in Table 4 represent a *single aggregated* value, whereas the numbers in Table 3 represent a *distribution* of agreement values across coders.

TABLE 6
Causes of Disagreements (FP: % of false positives, FN: % of false negatives, A: % of Ambiguous)

| Variable | # | FP | FN | A | Problematic Coders |
|---|---|---|---|---|---|
| Functionality | 34 | 29 | 21 | 50 | {1,12}(3);{2,15}(2) |
| Concepts | 27 | 37 | 4 | 59 | {4,8}(2) |
| Directives | 20 | 35 | 20 | 45 | 11(4) |
| Purpose | 28 | 68 | 21 | 11 | 12(8);4(4);8(3);{10,11}(2) |
| Quality | 12 | 50 | 8 | 42 | 15(2) |
| Control | 14 | 43 | 5 | 21 | 1(3) |
| Structure | 33 | 24 | 45 | 30 | {1,9,11}(3);{3,8,17}(2) |
| Patterns | 14 | 7 | 71 | 21 | {2,4,8,9}(2) |
| Examples | 5 | 0 | 60 | 40 | |
| Environ. | 13 | 69 | 15 | 15 | 4(6);11(2) |
| References | 22 | 45 | 41 | 14 | 4(5);11(4);13(3);15(2) |
| Non-info | 33 | 21 | 58 | 21 | 16(5);11(4);{9,14}(3);{1,2,3}(2) |
| Total | 255 | 31 | 36 | 32 | |

coders have to make a greater number of decisions in these cases.

To gain further insight into the nature of disagreements, we manually inspected, for each variable, a random sample of 2% of all units with a disagreement (i.e., 2% of Functionality disagreements, 2% of Concepts disagreements, etc.). This calculation led to an overall sample of 255 documentation units. One of the authors manually inspected each unit in the sample and classified the disagreement as follows:

- **False negative:** One coder clearly made a mistake by not indicating the presence of a knowledge type when clear evidence of that type is present.
- **False positive:** One coder clearly made a mistake by indicating the presence of a knowledge type when no clear evidence of that type is present.
- **Ambiguous:** Both interpretations are possible. Either the guide did not cover the situation at hand, the language of the documentation was ambiguous or vague, or a combination of both.

In the case of a mistake, the inspector noted the responsible coder and potential sources of problems.

Table 6 summarizes our findings. For each variable, the table reports: the number of units in the sample of disagreements (#), the percentage of false positive (FP), false negative (FN), and ambiguous (A) units recorded, and a list of problematic coders. This list includes all coders who were responsible for at least two mistakes in the sample for one variable, together with their number of mistakes. For example, for Directives, we had 20 documentation units in the disagreement sample, 35% of these were caused by false positives, and Coder 11 was responsible for 4 of the 20 disagreements.

Our qualitative analysis of the 225 sampled documentation units associated wiht a coder disagreement provided us with a wealth of information. First, it gave *overall tendencies*, for example that coders tended to over-eagerly classify units as containing Purpose. Second, it confirmed our quantitative estimate of coder performance, for instance by pointing out that Coder 4 is responsible for a large number of disagreements for the Environment variable (which corroborates the data in Table 3). This kind of knowledge was invaluable in the design of our disagreement reconciliation scheme (see Section 5.4). Finally, the qualitative analysis surfaced specific writing issues that make documentation difficult to understand, and areas where the clarity of the coding guide can be improved. We provide one example of each.

In the .NET documentation, the sentence "This class cannot be inherited" appears in the documentation of `sealed` classes, which by definition cannot be inherited. It was not clear whether coders were expected to interpret this sentence as a Directive (see Table 1) or whether this was automatically-generated boilerplate text. This issue can easily be addressed with additional instructions in the coding guide.

In the JDK documentation we find the sentence "[Class] `AccessibleJToggleButton` provides an implementation of the Java Accessibility API appropriate to toggle button user-interface elements." Use of the term "appropriate" is confusing in this context. Does this describe a feature of the API (Functionality), a recommendation on when to use it (Purpose), or does it simply point to the inheritance relation (Structure)?

### 5.4 Disagreement Reconciliation

To investigate the distribution of knowledge types in API documentation based on our data set, we needed to reconcile all disagreements to one of the two binary values.[9] The observations collected in the analysis of disagreements provided us with the strategies necessary to reconcile them. For each knowledge type, we implemented a procedure that inspects each disagreement and selects the value most likely to be correct, using one of the following strategies in order:

1) If the disagreement is in our disagreement sample, select the value manually determined.
2) If the disagreement involves only one problematic coder for the knowledge type at hand (see Table 6), select the value of the other coder.
3) If the disagreement involves a variable that shows a general tendency toward either false positives or negatives, correct accordingly. We consider a knowledge type to have a general tendency toward false positives or negatives if, and only if, more than 50% of the disagreements are caused by false positives or false negatives.
4) If the disagreement involves two problematic coders, but one made more mistakes than the other, select the value of the less problematic coder.
5) If the agreement ratio of one coder is clearly superior to the agreement ratio of the other coder,

9. Other schemes are possible, such as averaging the values. However, the validity of these schemes is dubious in our case.

select the coder with the higher agreement ratio (see Table 3). As our definition of "superior", we tested that the difference between the ratios was greater than the standard deviation of all pairwise agreement ratios for the knowledge type.

6) The case is ambiguous. Select False.

Using this algorithm, we reconciled a total of 12 493 disagreements, i.e., (12 493/11 148 coded documentation units × 12 knowledge types =) 9.3% of all ratings in our data set (see Table 7).

TABLE 7
Conflict resolution results

| Conflict resolution strategy | # ratings |
|---|---|
| 1) Manual correction | 255 |
| 2) Only one coder is problematic | 5005 |
| 3) Strong tendency of the variable | 2663 |
| 4) One coder is more problematic than the other | 148 |
| 5) Default order of coders | 2121 |
| 6) Ambiguous (use 0 as default) | 2301 |
| Total | 12 493 |

## 6 Patterns of Knowledge

The fundamental motivation for our inquiry was to discover how different knowledge types are distributed through API documentation, a type of observation we call a knowledge pattern (in documentation). We investigated this question by analyzing the distribution of knowledge types from three different perspectives: relative proportions in different subpopulations, co-occurrence of knowledge types in documentation units, and relationship between documentation length and incidence of knowledge types.

### 6.1 Comparisons of Proportions

Figure 3 shows the relative proportion of each knowledge type across all documentation units, weighted by the proportion of the element type (class, method, field) in the total population (see Table 2). This barplot allows us to distinguish between three classes of knowledge types: Pervasive (Functionality), Common (Structure, Non-information), and Rare (all others). We also notice that, although at first glance the distributions of knowledge in the JDK and .NET reference documentation follow the same trend, there are some important differences. In particular, .NET documentation includes a noticeably larger amount of Structure, Patterns, and Examples knowledge, three types generally related to the question of *how* to engineer solutions with an API. Based on this observation, we define the concept of *documentation style* as collection of knowledge patterns applicable to a cohesive subset of documentation units (e.g., all those pertaining to a technology platform, all those pertaining to classes, etc).

From this overview we proceed with a detailed analysis, comparing the proportions of different knowledge types in the different subpopulations. Table 8 reports on this investigation (the top of the table for the JDK, the bottom for .NET). For a given knowledge type, the table compares the proportions of Types vs. Members (left), Classes vs. Interfaces (center), and Methods vs. Fields (right). For a given group, the table reports the relative proportions of units with the knowledge type, the p-value from a $\chi^2$ test of independence, and the $\phi$-coefficient representing the size of the difference (see Section 5.3 for explanations on the $\chi^2$ test and $\phi$-coefficient). For differences with a p-value less than 0.05, we indicate the largest ratio in **bold**.

For example, we observe that in the JDK, the proportion of classes[10] with Purpose is 0.185 and the proportion of interfaces with Purpose is 0.258. The difference is statistically significant (p=0.003) and in our context the magnitude is notable (7.3% absolute difference, $\phi = 0.085$). In other words, JDK interfaces are more likely to contain Purpose knowledge than classes. This makes sense in that interfaces are intended to represent high-level abstractions and are often primary elements in a framework. Surprisingly, this difference is not observed in .NET.

Overall we observe that classes and interfaces contain significantly higher proportions of all knowledge types than members, with the exception that JDK members contain Functionality as often as classes and interfaces. The difference in proportions for Quality is not notable because it is based on a very low number of units.

Comparing between classes and interfaces we notice that, in the JDK, interfaces have significantly more knowledge of type Directives, Purpose, Quality, and Examples. This was expected given that interfaces tend to model higher-level concepts and main points of interaction in a framework. Surprisingly, not only are these differences not significant in .NET, but in fact Directives and Examples are less frequent in .NET interfaces than in .NET classes. Hence, expectations about the location of such knowledge types do not generalize across technologies.

Significant differences between fields and methods relate to other knowledge types. Not surprisingly, Control knowledge is rarely found in fields (JDK and .NET), as is the case for Structure (JDK). However, in .NET, relatively many fields also contain Structure knowledge, probably because of the concept of properties, which offer additional composition features besides basic data storage.

The distribution of Non-information deserves special mention. In both the JDK and .NET, Non-information is overwhelmingly present in members

10. In the rest of the section we use structural terms (e.g., "class") to refer to "the documentation unit associated with" the term.

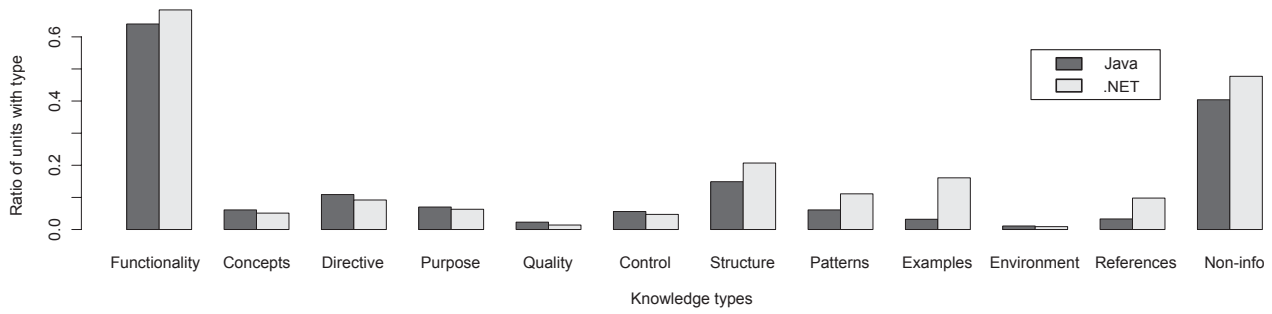


Fig. 3. Proportion of knowledge type by documentation unit

TABLE 8
Proportions of knowledge types in different subpopulations

| **JDK** | **Types vs. members** | | | | **Classes vs. interfaces** | | | | **Methods vs. fields** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Variable | **Type** | **Member** | **p** | $\lvert\phi\rvert$ | **Class** | **Interface** | **p** | $\lvert\phi\rvert$ | **Method** | **Field** | **p** | $\lvert\phi\rvert$ |
| Functionality | 0.638 | 0.640 | 0.925 | | 0.642 | 0.631 | 0.733 | | **0.706** | 0.277 | 0.000 | 0.322 |
| Concepts | **0.167** | 0.049 | 0.000 | 0.194 | 0.165 | 0.172 | 0.811 | | 0.048 | 0.055 | 0.788 | |
| Directives | **0.179** | 0.101 | 0.000 | 0.113 | 0.158 | **0.215** | 0.015 | 0.071 | 0.109 | 0.059 | 0.024 | |
| Purpose | **0.212** | 0.055 | 0.000 | 0.237 | 0.185 | **0.258** | 0.003 | 0.085 | 0.052 | 0.071 | 0.287 | |
| Quality | **0.060** | 0.019 | 0.000 | 0.105 | **0.072** | 0.039 | 0.022 | 0.068 | 0.021 | 0.013 | 0.570 | |
| Control | **0.094** | 0.052 | 0.000 | 0.081 | 0.093 | 0.097 | 0.898 | | **0.059** | 0.013 | 0.004 | 0.076 |
| Structure | **0.345** | 0.128 | 0.000 | 0.259 | 0.345 | 0.345 | 0.967 | | **0.140** | 0.063 | 0.001 | 0.083 |
| Patterns | **0.198** | 0.046 | 0.000 | 0.238 | 0.190 | 0.210 | 0.437 | | 0.050 | 0.021 | 0.068 | |
| Examples | **0.130** | 0.022 | 0.000 | 0.221 | 0.115 | **0.157** | 0.040 | 0.061 | 0.024 | 0.008 | 0.190 | |
| Environment | **0.050** | 0.007 | 0.000 | 0.133 | 0.051 | 0.047 | 0.841 | | 0.005 | 0.021 | 0.018 | |
| References | **0.089** | 0.027 | 0.000 | 0.136 | 0.094 | 0.082 | 0.522 | | 0.026 | 0.034 | 0.649 | |
| Non-info | 0.136 | **0.433** | 0.000 | 0.321 | 0.144 | 0.122 | 0.316 | | **0.455** | 0.311 | 0.000 | 0.039 |
| **.NET** | | | | | | | | | | | | |
| Variable | **Type** | **Member** | **p** | $\lvert\phi\rvert$ | **Class** | **Interface** | **p** | $\lvert\phi\rvert$ | **Method** | **Field** | **p** | $\lvert\phi\rvert$ |
| Functionality | **0.739** | 0.676 | 0.000 | 0.068 | 0.732 | **0.875** | 0.016 | 0.071 | 0.685 | 0.667 | 0.503 | |
| Concepts | **0.100** | 0.045 | 0.000 | 0.108 | 0.100 | 0.094 | 0.960 | | 0.031 | **0.060** | 0.011 | 0.068 |
| Directives | **0.127** | 0.087 | 0.000 | 0.064 | 0.128 | 0.109 | 0.808 | | 0.099 | 0.074 | 0.101 | |
| Purpose | **0.166** | 0.049 | 0.000 | 0.191 | 0.164 | 0.203 | 0.518 | | 0.053 | 0.046 | 0.598 | |
| Quality | 0.016 | 0.014 | 0.724 | | 0.017 | 0.000 | 0.580 | | 0.015 | 0.013 | 0.878 | |
| Control | **0.102** | 0.039 | 0.000 | 0.124 | 0.105 | 0.047 | 0.198 | | **0.060** | 0.016 | 0.000 | 0.115 |
| Structure | **0.493** | 0.168 | 0.000 | 0.348 | 0.498 | 0.406 | 0.192 | | 0.172 | 0.164 | 0.710 | |
| Patterns | **0.241** | 0.093 | 0.000 | 0.200 | 0.244 | 0.188 | 0.382 | | 0.098 | 0.088 | 0.574 | |
| Examples | **0.326** | 0.139 | 0.000 | 0.223 | **0.332** | 0.203 | 0.044 | 0.060 | 0.132 | 0.147 | 0.458 | |
| Environment | **0.019** | 0.007 | 0.007 | 0.054 | 0.019 | 0.031 | 0.814 | | 0.008 | 0.007 | 0.836 | |
| References | **0.178** | 0.087 | 0.000 | 0.135 | 0.180 | 0.125 | 0.335 | | 0.097 | 0.075 | 0.169 | |
| Non-info | 0.235 | **0.510** | 0.000 | 0.282 | 0.239 | 0.156 | 0.171 | | **0.535** | 0.481 | 0.040 | 0.054 |

(as opposed to types), reaching more than 50% of members in the case of .NET. We also observe that Non-information is mostly a problem with methods. For both types and members, the amount of Non-information is significantly larger in .NET (Types $p = 2.7 \times 10^{-10}$, $|\phi| = 0.127$; Members $p = 2.4 \times 10^{-5}$, $|\phi| = 0.077$).

## 6.2 Co-occurrence of Knowledge Types

The analysis described in Section 6.1 tells us about the probability of occurrence of different knowledge types in different subpopulations of documentation units, but does not provide any insight about patterns of co-occurrence of knowledge types. For example, is the Structure knowledge necessarily accompanied by Functionality knowledge? To answer this question, we systematically investigated the statistical relationships between the presence of knowledge types in documentation units using correlation and frequent-itemset mining techniques.

### 6.2.1 Correlation

We analyzed the correlation between knowledge types in our four populations. In other words, we investigated how often certain knowledge types co-occur in documentation units. Because we use binary variables to capture the presence of knowledge types, we use the $\phi$-coefficient to measure the association between knowledge types. In contrast to previous sections, here we use the $\phi$-coefficient directly as a correlation measure, as opposed to using it to represent the magnitude of a difference between proportions. Nevertheless, its interpretation and calculations are

identical (see Section 5.3 for details).

Figure 4 shows the correlation matrix for JDK and .NET types. In each cell, the circle area is proportional to the $\phi$-coefficient between the two knowledge types. Black circles indicate positive correlation, whereas gray circles indicate negative correlation.

Overall, absolute correlations levels vary from negligible to medium [0.0–0.285]. For JDK types the maximum correlation is 0.276 (between Structure and Patterns) and for .NET 0.285 (between Patterns and Examples). The nature of patterns is different for JDK and .NET types. In the JDK we notice a cluster of types that tend to co-occur with one another with some frequency, including Structure, Examples, Control, Patterns, and Directives (each being involved in at least one association with $\phi > 0.2$). In .NET associations are much weaker. Except for the relatively strong association between Patterns and Examples, the only other associations with $\phi > 0.19$ are between References and Examples ($\phi = 0.195$), and between Concepts and Purpose ($\phi = 0.194$). Although the association between Concepts and Purpose seems logical, we do not observe this association in the JDK ($\phi = 0.094$). Finally, for both JDK and .NET the presence of Non-information is negatively correlated with most other knowledge types, in particular, Functionality and Structure. This observation implies that, in an aggregated sense, Non-information is mutually exclusive with the other (effective) knowledge types, which supports the hypothesis that documentation might sometimes be produced as an end in itself, possibly to meet completeness requirements.

The correlations observed in members populations are much weaker, and for this reason we do not include the correlation matrix. For JDK members the most strongly correlated knowledge types are Functionality and Structure ($\phi = 0.186$) and Functionality and Control ($\phi = 0.162$). No other association is noteworthy. For .NET members we again note a correlation between Examples and Patterns ($\phi = 0.216$), Patterns and Structure ($\phi = 0.199$) as well as between Functionality and Structure ($\phi = 0.170$). Interestingly this cluster is a subset of the one observed for JDK types. This indicates that patterns present in JDK types may exist in members in the case of .NET. Finally, for both JDK and .NET members negative correlation between Non-information and all other types is very weak ($|\phi| < 0.09$) except in one case in .NET, where we observe ($\phi = -0.225$) between Functionality and Non-information.

### 6.2.2 Frequent Itemset Mining

Correlation provides an aggregated perspective on the relationships between knowledge types but it is limited to relations between pairs of variables. To gain a different view of co-occurrences between knowledge types, we applied *Frequent Itemset Mining* to our data set [11].

Frequent Itemset Mining is a data mining technique that computes frequent subsets in a data set. A Frequent Itemset is defined as a recurring subset of the set of all possible elements that is also a subset of a high number of instances in a data set. In our case, elements are the 12 knowledge types and dataset instances are documentation units. There are two important parameters for computing frequent itemsets: *minimum support* (minsup) and *minimum length* (minlen). The minimum support is the minimum number of instances that must contain an Itemset for it to be considered frequent. This parameter is often expressed as a ratio of the total number of items in a dataset. Minimum length is simply the minimum cardinality of an itemset. For example, if items "bread" and "milk" are found in at least 10 of the instances in a data set consisting of 100 purchase transactions and Frequent Itemset Mining is applied with minsup=0.1 and minlen=2, the set {"bread", "milk"} would be reported as a frequent itemset.

There are two major differences between frequent itemsets and correlation. First, frequent itemsets describe a subset of the data (represented by the support value), whereas correlation summarizes the entire data. Second, frequent itemsets describe relations between potentially more than two knowledge types, as opposed to pairs only.

Applying Frequent Itemset Mining to our data sets is straightforward because we can trivially convert our database of documentation units $\times$ ratings into a binary incidence matrix, where rows are documentation units in one of our four populations, and columns (12 in total) contain a value of 1 if the knowledge type corresponding to the column was found in the documentation unit. We applied Frequent Itemset Mining by using the Eclat algorithm as implemented by the `arules` package of the R statistics program [11].

Table 9 reports the frequent itemsets in the populations of JDK and .NET types, and Table 10 provides the same information for the member subpopulations. The tables lists *all* the itemsets with support ratio above 0.1. To obtain the exact number of documentation units with the corresponding knowledge types requires simply to multiply the reported support ratio with the number of instances in the data set (listed in the first row). For example, $0.265 \times 1243 = 330$ JDK type documentation units contained both Functionality and Structure. In the tables, the frequent itemsets in **bold** are the ones found in both JDK and .NET.

Because most frequent itemsets are pairs, we also obtained itemsets with minsup=0.08 and minlen=3. This configuration allowed us to discover larger itemsets with a comparable level of support. Larger itemsets provide a better insight on how knowledge types tend to cluster.

Not surprisingly, the relation between Functionality and Structure discovered through the correlation anal-

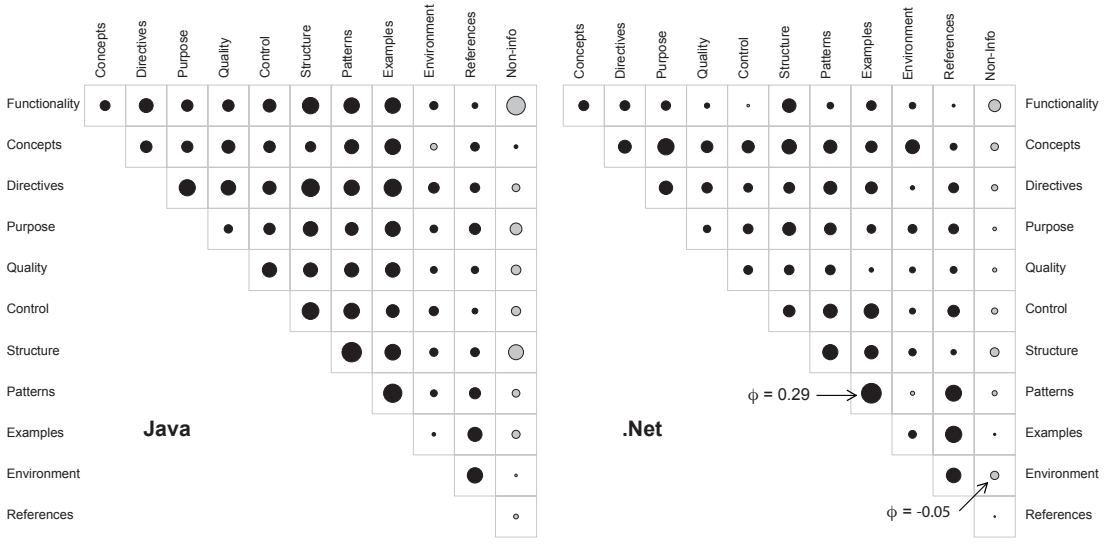


Fig. 4. Correlation analysis of JDK and .NET types

TABLE 9
Frequent Itemsets in JDK and .NET types (minsup = 0.1, minlen=2) *(minsup=0.08, minlen=3)

| JDK Types (1243 Items) | | .NET Types (1283 Items) | |
|---|---|---|---|
| **Itemset** | **Support** | **Itemset** | **Support** |
| {**Functionality, Structure**} | 0.265 | {**Functionality, Structure**} | 0.394 |
| {**Functionality, Patterns**} | 0.160 | {**Functionality, Examples**} | 0.255 |
| {**Functionality, Purpose**} | 0.154 | {Structure, Examples} | 0.192 |
| {**Functionality, Directives**} | 0.141 | {**Functionality, Patterns**} | 0.184 |
| {**Structure, Patterns**} | 0.121 | {Functionality, Structure, Examples} | 0.163 |
| {Functionality, Concepts} | 0.120 | {**Structure, Patterns**} | 0.155 |
| {**Functionality, Examples**} | 0.112 | {Functionality, Non-information} | 0.154 |
| {**Structure, Purpose**} | 0.104 | {Patterns, Examples} | 0.136 |
| {Structure, Directives} | 0.103 | {**Functionality, Purpose**} | 0.133 |
| {**Functionality, Structure, Patterns**}* | 0.099 | {Functionality, References} | 0.133 |
| {Functionality, Structure, Directives}* | 0.090 | {**Functionality, Structure, Patterns**} | 0.125 |
| {**Functionality, Structure, Purpose**}* | 0.083 | {**Structure, Purpose**} | 0.105 |
| | | {**Functionality, Directives**} | 0.104 |
| | | {Functionality, Patterns, Examples} | 0.104 |
| | | {Structure, Non-information} | 0.104 |
| | | {**Functionality, Structure, Purpose**}* | 0.090 |
| | | {Structure, Patterns, Examples}* | 0.088 |
| | | {Functionality, Structure, Non-information}* | 0.081 |

TABLE 10
Frequent Itemsets in member subpopulations (minsup = 0.1, minlen=2)

| JDK Members (1549 Items) | | .NET Members (1499 Items) | |
|---|---|---|---|
| **Itemset** | **Support** | **Itemset** | **Support** |
| {**Functionality, Non-information**} | 0.267 | {**Functionality, Non-information**} | 0.292 |
| {**Functionality, Structure**} | 0.112 | {**Functionality, Structure**} | 0.142 |
| | | {Functionality, Examples} | 0.113 |

ysis is also represented here. However, the other frequent itemsets illustrate the difference in style between the JDK and .NET documentation. .NET is richer in examples. Its documentation units that include Functionality and Structure also include Examples much more often. For instance, the itemset {Functionality,Examples} is more than twice as common in .NET documentation than in the JDK. In contrast, the JDK conveys more of this knowledge through Directives. For example, in the JDK we find the {Functionality, Directives} itemset to be more frequent than in .NET, and that Directives is also part of the larger itemset {Functionality, Structure, Directives}.

For itemsets that exclude Functionality, we notice that {Structure, Patterns} and {Structure, Purpose}

TABLE 11
Distribution of element lengths (word count)

| | 1st Q. | Med. | Mean | 3rd Q. | 0 | Out. |
|---|---|---|---|---|---|---|
| **JDK types** | 20 | 51 | 107.2 | 120 | 37 | 7 |
| **.NET types** | 23 | 55 | 95.0 | 121.5 | 16 | 3 |
| **JDK mem.** | 14 | 32 | 48.7 | 65 | 140 | 2 |
| **.NET mem.** | 21 | 34 | 54.9 | 59 | 23 | 3 |

occur in slightly more than 10% of the type documentation units for both JDK and .NET.

Finally, in the case of members, the results are very similar for both the JDK and .NET. Over 25% of the member documentation units show a combination of Functionality and Non-information.

### 6.3 Length Analysis

We also studied the relation between the length (in number of words) of each documentation unit in our sample and its relation to the number of knowledge types it contains. This analysis contributes additional evidence for the distinctiveness of knowledge types (research question 1).

The text of a documentation unit consists of the entire content of the document except automatically-generated content (e.g., syntax summaries), source code, and sections irrelevant to this study (e.g., version information in .NET). Overall, our study involved the close reading of 431 136 words, or roughly the size of a very long novel.

Table 11 summarizes the distribution of element lengths in our sample. For each subpopulation, the table shows the values of the mean and the different quartiles, along with the number of 0-length elements ("0"), and the number of outliers (types longer than 1000 words and members longer than 500 words). Overall, the JDK and .NET distributions are very similar, with only a larger number of extremely long elements in the JDK. The distributions of member lengths have similar exponential shapes, but with smaller values.

Figure 5 explores the relation between the number of distinct knowledge types found in a documentation unit and the length of the unit. It shows a boxplot of the distribution of lengths for each subpopulation containing a certain number of knowledge types. Note the change in scale between the top two (types) and bottom two rows (members). For example, the top row shows the distribution of lengths of JDK types containing, respectively, zero knowledge types, one knowledge type, two knowledge types, etc. We exclude data for types with more than seven knowledge types and members with more than five knowledge types as these subgroups are very small, all containing less than 13 elements.

As the figure shows, the documentation units containing more knowledge types tend to be longer. We statistically tested this hypothesis through sets of one-tailed Wilcoxon rank sum tests with continuity correction. We chose a non-parametric test given the obvious non-normality of the length distributions. For each of the four populations, we performed a set of tests to verify that in terms of overall population the length of an element containing $n$ distinct knowledge types was significantly less than that of an element containing $n+1$ distinct knowledge types.[11] This analysis revealed that the comparative increase in length for types (both JDK and .NET) is statistically significant for all pairs in the sequence up to and including $4 \rightarrow 5$ (which has the highest p-value of $0.00020$ for JDK and $6.30 \times 10^{-5}$ for .NET). In the case of the population for members documentation, the relations are significant up to and including $3 \rightarrow 4$ (p = 0.002 for both JDK and .NET). Interestingly, if we omit contributions to the total number of knowledge types from References and Non-information knowledge types, the statistical differences are much more significant, and in the case of types the sequence $5 \rightarrow 6$ also shows a significant difference. This observation makes sense from a theoretical perspective, because describing cross-references and non-information typically involves a few words.

That very small documentation units would contain few knowledge types (and vice-versa) is not surprising. However, step-wise increase in length related to the number of knowledge types confirms the intuition that longer documentation units are longer because they discuss many different aspects of the API (as opposed to long developments in one type of knowledge).

## 7 Discussion

The content analysis research method is fundamentally quantitative, and the insights it provides answer general questions of distributions across populations. Through multiple dimensions of analysis complemented by focused qualitative investigations, we derived from our study a number of primary outcomes, implications for API documentation practices, and future research research directions.

### 7.1 Outcomes

A primary outcome of this work is the validation of our knowledge type taxonomy. Although it is trivial to come up with arbitrary labels for what one might find in a documentation page, our coding guide has by now undergone empirical validation. The agreement analysis (Section 5.2) showed that different coders can independently agree on classifications based on it. The correlation analysis (Section 6.2) showed no systematic co-occurrence. Finally,

11. Using Bonferroni corrections for multiple comparisons, with $n = 7$ and $n = 5$ for types and members, respectively.

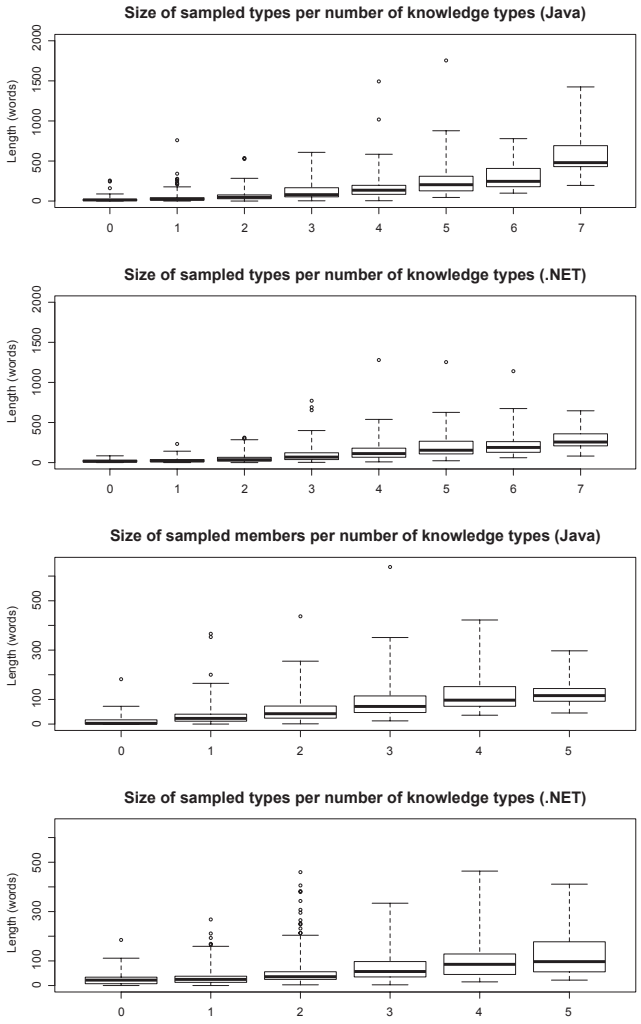


Fig. 5. Length (word count) of documentation units by contained number of knowledge types

the length analysis (Section 6.3) showed how knowledge types are generally additive. These are important properties for a taxonomy of knowledge types.

In Section 6.1 we reported on the relative proportions and frequency of co-occurrence of different knowledge types in the API reference documentation of two major technology platforms. We refer to such observations as patterns of knowledge in API documentation. Given our use of random sampling, these patterns of knowledge are expected to be representative of the documentation for the two entire platforms.

Some findings were expected but others, much less so. In particular, we were surprised to observe such low frequencies of Concepts, especially in .NET (only present in 10% of types). Given that an underlying principle of object-oriented programming is to have objects roughly map to domain concepts, type-level API reference documentation could be a reasonable location for this knowledge. At the limit, we could expect that most type-level documentation units should contain some explanation of the concept it represents. Apparently, it is not the case. A potential explanation is that this knowledge is instead found in conceptual overviews, such as the Microsoft Developer Network Library and the Java Tutorial.

Both the JDK and .NET are large, general-purpose object-oriented frameworks and our stratified random sampling scheme ensured that all major functional units from both frameworks were represented. We can thus claim that to a large extent the target of the documentation is reasonably equivalent for both JDK and .NET. In addition, the distribution of the length of documentation units is remarkably similar between JDK and .NET. For this reason, major differences between platforms in how knowledge types appear in documentation units are indicative of differences in *documentation styles*. A few trends are evident:

- The JDK contains more "conceptual" types of knowledge, namely Concepts and Purpose, whereas .NET contains more "structural" types of knowledge, namely Structure and Patterns.
- .NET contains more Examples whereas the JDK contains more Directives.
- The JDK tends to provide more Concepts, Purpose, and Quality in interfaces (vs. classes), whereas in .NET this is not true.
- The JDK and .NET document Directives differently. In the JDK types are correlated with Structure ($\phi = 0.225$) and Examples ($\phi = 0.218$) whereas in .NET not really ($\phi < 0.105$).

### 7.2 Implications

The outcomes of this study have a number of practical implications on the development and use of API documentation.

First, practitioners can **evaluate the content** of their API documentation in relation to the knowledge types described in our taxonomy. The patterns of knowledge in the JDK and .NET API documentation offer a point of reference for the amount of content of each knowledge type that would be expected by users of the respective technologies. These ratios are by no means prescriptive, but they can help API documentation teams think about whether their documentation covers all necessary aspects of the API.

Second, practitioners can use our definitions of knowledge types to better **organize their documentation**, for example by creating templates that explicitly highlight specific knowledge types. The patterns of knowledge we document can also help reason about what knowledge types are most appropriate for different types of API elements.

Third, our taxonomy provides a **vocabulary** that can help structure and facilitate discussions about the content of API documentation, for example between development and documentation teams. The patterns of knowledge further complement this vocabulary by describing typical combinations of knowledge types

that may be adopted by documentation teams (e.g., the "Structure and Examples" pattern).

Finally, **documentation quality can potentially be improved** by the detection and removal of non-information. In previous work, we observed that many professional developers were hindered by "boilerplate documentation that merely rehashes the name of an API method, bloats the presentation with derived information such as inherited members, or provides overly trivial examples..." [27, p. 722]. The proportions of Non-information we report motivate the need to address this concern, and our taxonomy provides a detailed description of this problematic knowledge type that can be used as a starting point for avoiding or removing it from documentation.

### 7.3 Future Work

For researchers, our taxonomy and the patterns of knowledge we report have implications that motivate further research in three main directions.

First, the study represents the first step toward a systematic identification of knowledge types in documentation. Each of the knowledge types implies an individual research question: How can we automatically detect the presence of that knowledge type in a documentation unit? Monperrus et al. recently studied the detection of Directives in framework documentation [21]. However, to the best of our knowledge, this question is unanswered for the eleven remaining knowledge types.

Second, our taxonomy helps to study the gap between information seekers and information providers. Several studies summarized in Section 8.1 focused on types of knowledge sought by developers during their work. Our taxonomy provides a dimension of analysis that can be used to assess the balance between information sought by users and provided by documentation writers.

Finally, we plan to continue this work by qualitatively studying documentation units that present unusual combinations of features. For example, methods with Concepts and without Functionality, elements with short documentation and many knowledge types, or singleton knowledge types with no Functionality. In addition, our analysis of disagreements between coders provided a way to detect confusing documentation snippets (see Section 5.3). From these observations we expect to identify a number of deviant documentation patterns that will help us automate the detection of low-quality documentation and eventually develop quality instruments for API documentation.

## 8 Related Work

As part of the contributions of this paper, we provide a detailed comparative analysis of ten different categorizations of software engineering knowledge (Section 8.1). We also discuss related studies of API documentation (Section 8.2).

### 8.1 Knowledge Categorization

Other researchers have attempted to categorize knowledge in software engineering. This section explains how previous work relates to and informs our taxonomy of knowledge types (Section 3). Table 12 provides an overview of the comparison.

**Mylopoulos et al.** discussed how knowledge representation techniques from the field of Artificial Intelligence can be applied to software engineering. The authors presented a categorization of different knowledge types [22], presumably derived from their experience. The categorization includes the following types: Domain knowledge, requirements knowledge, design knowledge, implementation knowledge, programming knowledge, quality factors, design rationale, and historical knowledge. This categorization has helped orient our taxonomy. Our notion of Concepts relates to their notion of *Domain knowledge*. Our notion of Structure and Control are both specialization of their category *Design*. Finally, we both share a Purpose (Rationale) category. Our adaptations have mostly to do with the differences between API documentation (for reusable components) and general software documentation (including the internal implementation aspects).

**Herbsleb and Kuwana** classified questions asked during design meetings to study the kinds of knowledge that may benefit from explicit capture at the requirements and design stages [12]. The six example questions provided in the paper point to our Structure, Control, Functionality, and Patterns knowledge types.

Based on their general experience, **Erdös and Sneed** proposed seven questions developers ask when performing software maintenance [7]. All the questions have to do with understanding the behavior of the program, and thus relate to our Functionality type.

**Johnson and Erdem** classified 249 questions asked on the comp.lang.tcl newsgroup in 1995. The authors analyzed various characteristics of the person asking, of the related task, and of the type of questions [15]. They categorize questions as *Goal-oriented*, *Problem-oriented*, or *System-oriented*. Goal-oriented questions map to both our Patterns type ("How can I read a file into an array?") and our Functionality type ("Is it possible to display a picture on a button widget?"). The notion of problem-oriented question (e.g., "Tcl installation fails [...] What am I doing wrong?") is interesting because, although it represents seeking knowledge about a software component, the knowledge is context-specific, not general. Since troubleshooting questions will generally be answered by variants of the "How do I do..." question, we associate them to our Patterns type. Finally, their *System-*

TABLE 12
Knowledge categorization studies in software engineering

| Reference | Focus | Structure | Origin |
|---|---|---|---|
| Mylopoulos et al. 1997 [22] | Software engineering | 8 Question categories | General experience |
| Herbsleb & Kuwana 1993 [12] | Software design | Question and attributes | Design meetings |
| Erdös & Sneed 1998 [7] | Software maintenance | 7 Questions | General experience |
| Johnson & Erdem 1995 [15] | Tcl/Tk programming | 3 Question categories | 1250 newsgroups |
| Erdem et al. 1998 [6] | Software engineering | Question model | Literature, newsgroups, theoretical |
| Butler at al. 2000 [2] | Framework usage | 6 Documentation primitives | General experience |
| Hou et al. 2005 [13] | 2 Swing components | Questions mapped to features | Swing forum |
| Ko et al. 2007 [17] | Software engineering | 21 General information needs | Observation of 17 developers |
| Kirk et al. 2007 [16] | JHotDraw | 4 Categories of reuse problems | Various data from user studies |
| Sillito et al. 2008 [30] | Software maintenance | 44 Questions in 4 categories | Think-aloud utterances |
| Maalej & Happel [20] | Work description | 9 categories and 10 granularities | Computer-based analysis of informal artifacts |
| Maalej & Robillard (this paper) | API documentation | 12 knowledge types | Content analysis of reference documentation |

*oriented* category is very broad, encompassing questions related to design rationale, conceptual questions mapping system with domain objects, and questions on the internal behavior of the system. Noticing that their questions do not encompass the entire range of potential questions that can be asked in software engineering, **Erdem et al.** refined their catalog into a more extensive framework [6] that can be used to systematically construct questions by combining predefined elements, such as a question type (e.g., What vs. How), with a question topic (e.g., output vs. structure). This results in generalized questions such as "How is it structured?" Erdem et al. were inspired by a number of older question models and followed a theoretical approach, exploring the set of possible questions. Our work is complementary and has an empirical nature, reflecting the reality of programming with modern APIs and tools.

Based on their general experience and research on frameworks, **Butler et al.** proposed a "framework for framework documentation" [1], [2], which includes six "framework documentation primitives". These correspond to basic information one may need to document a framework. The six proposed primitives are exclusively system-focused. They include the name of the framework participants, their static and dynamic specifications, and the dependencies between primitives. We observed in reference documentation additional knowledge types such as Purpose or Concepts.

**Hou et al.** studied 300 questions related to two specific Swing widgets (JButton and JTree) posted on the Swing forum [13]. They then mapped the questions to the different design features of the widgets. The study provides insights about the types of features that are not well understood in a framework. However, the classification focuses more on the target of the question (e.g., inherited vs. widget-specific) and less on discovering the different types of knowledge provided to and sought by API users.

More recently, **Ko et al.** observed 17 developers at Microsoft for a 90 minutes session each, studying their information needs as they perform their software engineering tasks [17]. From the observation data the authors collected 334 specific information needs, which they abstracted into 21 general (or abstract) information needs. These were then partitioned into seven categories based on the type of work they pertain to. The information needs collected by Ko et al. cover a broad range of tasks, from writing code to submitting a change or reproducing a failure. Interestingly, the three questions they collected under the "writing code" category all have a strong "how to" flavor, in some case very closely matching our Patterns-related questions (e.g., "how do I use this data structure or function?"). In their category "Reasoning about Design", the question ("Why was this code implemented this way?" could also be related to API usage. In brief, while the Ko et al. catalog captures a much broader range of information needs, it is sparse in terms of information needs associated with API usage. Our catalog reflects what is being documented in API reference documentation as opposed to what is being sought be developers when programming with API.

**Kirk et al.** investigated the knowledge problems faced by them and their students when trying to develop applications by extending the JHotDraw framework [16]. After collecting 209 instances of "reuse problems" from notebooks, newsgroup postings, and student assignments, the authors identified four main categories of framework reuse problems: *Mapping* (converting the abstract solution to a concrete implementation), *Interactions* (understanding how the framework works), *Functionality* (determining the features offered by different parts of the framework), and *Architecture* (understanding the general design of the framework and its rationale). The authors hypothesized that specific types of documentation formats are better suited to address different types of problems. They conducted a second empirical study to investigate how these types of documentation formats help mitigate the problems discovered. The categorization of Kirk et al. provides insights into the type of knowledge that can be useful to capture in API documentation and that may be expected

therein. We notice very close relationships between the categories of Kirk et al. and ours, which may be explained by the facts that both categorizations were obtained from grounded methods focusing on API (or frameworks). The *Mapping* category corresponds to our Patterns and Concepts types. According to Kirk et al. the mapping category is represented by questions such as "How do I achieve...?", which is our landmark question for the Patterns type. Their *Interaction* category closely matches our Functionality type, and likewise their *Architecture* our Structure and Control types. They also have a *Functionality* category very similar to ours. In summary, given their different origin (user- vs. documentation-centered) the close similarity between our categorization and that of Kirk et al. provides confirmatory evidence that this clustering represents a meaningful distinction between knowledge types describing APIs. Our study also presents the distribution of these knowledge types across a documentation system.

Similarly to Ko et al.'s study, **Sillito et al.** produced a catalog of 44 types of questions developers ask during software evolution tasks [30]. This catalog was elaborated through grounded theory techniques based on the think-aloud utterances of participants involved in two user studies. The questions in the catalog do not focus exclusively on API usage, but rather relate to software evolution and maintenance tasks. Sillito et al. cluster their questions into four categories based on "the amount and type of information required to answer a question" [30, p.438]. Their categories are: *Finding focus points*, *Expanding focus points*, *Understanding a subgraph*, and *Questions over groups*. In the category *Finding a focus point* the question "Which type represents this domain concept or this UI element or action?" is related to either our Concepts or Patterns types, depending on how the answer is expressed. The category *Expanding Focus Points* overlaps with our notion of Structure. For example, their questions in this category include "What are the parts of this type?", or "Who implements this interface or these abstract methods?". Their *Understanding a Subgraph* category is most closely matched to our notion of Patterns knowledge, with questions such as "How are instances of these types created and assembled?", or "What is the 'correct' way to use or access this data structure?". Other questions in this category map instead to our notion of Functionality (e.g., "How does this data structure look at runtime?"). Finally, their last category, which clusters questions having to do with groups of elements, presents questions mostly unrelated to APIs (e.g., impact analysis questions). The questions of Sillito et al. shows no obvious absence in our catalog. However, it is interesting to note that many questions asked by developers about the structure or design of a class hierarchy will find their answer implicitly in the structural organization of reference documents for OO frameworks, as opposed to explicit mention in the documentation. For example "What are the parts of this type?" would be obvious from the list of all public members for a type.

Finally, **Maalej and Happel** analyzed the content of informal artifacts such as commit messages and work logs, and identified nine information entities and ten granularity levels. They also quantified the frequencies of these categories in the informal documents. Despite the different goals, we note interesting overlaps and differences between informal documents and formal API reference documentation. On the one hand, their *Rationale* and *Reference* entities corresponds to our knowledge types Purpose and References. Their *Requirement* granularity level corresponds to our type Functionality. On the other hand, while the authors identified knowledge describing the programming tasks, our focus is on ancillary knowledge for component interfaces.

To summarize, although many taxonomies of knowledge types in software engineering have been proposed, researchers have mainly focused on the information needs of developers and their encountered questions when programming with API—targeting the perspective of the knowledge seekers. Our study investigates the nature of the information included in the API documentation, its distribution in the documentation system and its quality—-targeting the perspective of the knowledge providers.

### 8.2 Studies of API Documentation

There exists numerous studies on APIs, their design [31], their learnability [32], and their usability [5]. In this section we discuss studies of *API documentation*, which are rarer.

**Nykasa et al.** [24] performed a study to assess the documentation needs for a domain-specific API, using surveys and interviews of developers. This study identified, among other requirements, the importance of an overview section in API documentation.

More recently, **Jeong et al.** conducted a lab study with eight participants to assess the documentation of a specific service-oriented architecture [14]. This study identified 18 guidelines they believe would lead to increased documentation quality for the system under study, including "explaining starting points" for using the API. Our quantitative results on the frequency and distribution of Non-information also reveal similar insights about the quality of status-quo reference documentation and how it can be improved.

Through a set of surveys and interviews with Microsoft developers, **Robillard and DeLine** investigated the obstacles faced by developers when trying to learn new APIs [26], [27]. The study revealed that many obstacles were related to aspects of the documentation, but did not include the systematic analysis of API documentation content. **Dagenais and Robillard** [4] also used surveys and interviews to

understand how open-source contributors make decisions about the development and maintenance of API documentation.

Similarly, **Shi et al.** reported on a study of API documentation evolution [29]. The authors apply data mining techniques over the source repository of five open-source APIs. Their study provides various quantitative measures of which parts of the API documentation are most frequently revised, and how often API documentation is changed consistently with the corresponding elements. Although these studies also elicited insights related to the documentation content, they focused on the *evolution* of the content, not its nature.

In a case study of the JQuery API, **Parnin and Treude** investigated the types of documentation available on-line for each of the JQuery functions [25]. They conducted their study by executing a Google search query for each function, one-by-one, documenting in each case the nature of the top ten results. Among their observations, 99.4% of the searches, the official reference API documentation "was part of the first 10 search results and usually appeared on top " [25, p. 3]. One potential interpretations of this observation is that for JQuery developers the API reference documentation is highly visible and available even through general web searches, and thus is likely to be a heavily used source of information. This finding provides additional motivation for research aimed at understanding and improving the content and organization of reference documentation.

Finally, **Monperrus et al.**'s recent study of API directives is the closest to this work, in that it also studies the content of reference documentation [21]. In their work, the authors elicited a number of linguistic patterns indicative of the presence of different types of "programming directives". They then automatically applied heuristic searches to detect such patterns in three Java APIs (a subset of the JRE, Eclipse JFace, and the Commons Collection). The findings can be used to automatically detect directives in existing API documentation. We also share the ultimate goal of automatically detecting knowledge types in documentation by first establishing empirical evidence on its content. The qualitative results also confirms that Directives are an important and meaningful type of API documentation, which is included in our taxonomy. However, our quantitative results show that Directives are not the dominant form of documentation found in reference pages. Finally, while the authors focus on an in-depth analysis of one knowledge type in API reference documentation, our study provides rather a comprehensive analysis of the general content of reference documentation and its relation with the API structure.

# 9 Conclusion

Developers read API reference documentation to learn how to use the API [27] and answer specific questions they have during development tasks [30]. To advance the field toward evidence-based improvement in API documentation, we report on the first empirical study to systematically investigate the knowledge contained in API reference documentation. By using a combination of qualitative grounded methods and quantitative content analysis techniques, we elaborated a taxonomy of 12 distinct knowledge types found in API documentation. We then analyzed the distribution of these knowledge types in different subset of documents. We call individual observations about such distributions *patterns of knowledge* in API documentation.

We found that Functionality knowledge is pervasive and Structure is common, while other types, such as Concepts and Purpose, and much rarer. We also found that Non-information, a deviant type of knowledge representing low-value content, is prevalent in the documentation of methods and fields of the JDK and .NET APIs. Comparisons of patterns of knowledge types in different populations revealed many significant differences on, e.g. how classes are documented vs. methods, how knowledge types tend to co-occur, and how these patterns take different forms in different technology platforms. Collections of knowledge patterns applicable to a cohesive subset of API documentation unit can be seen as a form of *documentation style*.

Our findings can inform software development practice in four different ways. First, they allow practitioners to evaluate the content of their API documentation in relation to well-defined knowledge types. Second, they can guide the development of documentation templates that are adapted to the knowledge commonly associated with different types of API elements. Third, our taxonomy provides a vocabulary that can facilitate discussions about the content of API documentation. Finally, they document the extent of low-value content in documentation units which we hope will serve as a motivation for curtailing this practice.

Our study also motivates additional research in at least three areas. First, our taxonomy provides a foundation for the automated classification of knowledge types in API documentation. Second, it provides a framework for studying the gap between the knowledge provided by different types of documents and the information needs of software developers. Finally, classifying documentation according to knowledge types can support quantitative analyses linking patterns of knowledge with more subjective quality features. In general, we hope that this study will serve as a stepping stone toward a more systematic study of API documentation.

## ACKNOWLEDGMENTS

This work has been made possible by the generous support of the Alexander von Humboldt Foundation. We thank the coders, who participated in this study: Yam Chhetri, Vincenz Doelle, Aparna Halder, Zardosht Hodaie, Taha Koltukluoglu, Amel Mahmuzic, Afaq Mustafa, Hoda Naguib, Nitesh Narayan, Helmut Naughton, Dennis Pagano, Enrique Perez, Tobias Roehm, Blagina Simeonova, and Alexander Waldman. We also thank Yam Chhetri, Barthélémy Dagenais, Helmut Naughton, Dennis Pagano, Peter Rigby, Christoph Treude, Gias Uddin, and the anonymous reviewers for comments and suggestions.

## REFERENCES

[1] G. Butler, P. Grogono, and F. Khendek, "A reuse case perspective on documenting frameworks," in *Proceedings of the 5th Asia Pacific Software Engineering Conference*, 1998, pp. 94–101.

[2] G. Butler, R. K. Keller, and H. Mili, "A framework for framework documentation," *ACM Computing Surveys*, vol. 32, March 2000.

[3] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, 3rd ed. Sage Publications, 2007.

[4] B. Dagenais and M. P. Robillard, "Creating and evolving developer documentation: Understanding the decisions of open source contributors," in *Proceedings of the 18th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, November 2010, pp. 127–136.

[5] B. Ellis, J. Stylos, and B. Myers, "The Factory pattern in API design: A usability evaluation," in *Proceedings of the 29th ACM/IEEE International Conference on Software Engineering*, May 2007, pp. 302–312.

[6] A. Erdem, W. Johnson, and S. Marsella, "Task oriented software understanding," in *Proceedings of the 13th IEEE International Conference on Automated Software Engineering*, 1998, pp. 230–239.

[7] K. Erdös and H. Sneed, "Partial comprehension of complex programs (enough to perform maintenance)," in *Proceedings of the 6th International Workshop on Program Comprehension*, 1998, pp. 98–105.

[8] T. Fritz and G. C. Murphy, "Using information fragments to answer the questions developers ask," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, 2010, pp. 175–184.

[9] F. J. Gravetter and L. B. Wallnau, *Essentials of Statistics for the Behavioral Sciences*, 5th ed. Thomson Wadsworth, 2005.

[10] M. Gunderloy, *Understanding and Using Assemblies and Namespaces in .NET*, Microsoft Corporation, 2002, http://msdn.microsoft.com/en-us/library/ms973231.aspx.

[11] M. Hahsler, B. Gruen, and K. Hornik, "arules – A computational environment for mining association rules and frequent item sets," *Journal of Statistical Software*, vol. 14, no. 15, pp. 1–25, October 2005.

[12] J. D. Herbsleb and E. Kuwana, "Preserving knowledge in design projects: what designers need to know," in *Proceedings of the Joint INTERACT '93 and CHI '93 Conferences on Human Factors in Computing Systems*, 1993, pp. 7–14.

[13] D. Hou, K. Wong, and J. H. Hoover, "What can programmer questions tell us about frameworks?" in *Proceedings of the 13th International Workshop on Program Comprehension*, 2005, pp. 87–96.

[14] S. Y. Jeong, Y. Xie, J. Beaton, B. A. Myers, J. Stylos, R. Ehret, J. Karstens, A. Efeoglu, and D. K. Busse, "Improving documentation for eSOA APIs through user studies," in *Proc. 2nd Int'l Symp. on End-User Development*, ser. LNCS, vol. 5435. Springer, 2009, pp. 86–105.

[15] W. Johnson and A. Erdem, "Interactive explanation of software systems," in *Proceedings of the 10th Conference on Knowledge-Based Software Engineering*, 1995, pp. 155–164.

[16] D. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Empirical Software Engineering*, vol. 12, pp. 243–274, June 2007.

[17] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 344–353.

[18] D. Kramer, "API documentation from source code comments: A case study of Javadoc," in *Proceedings of the Conference of the ACM Special Interest Group for Design of Communication*, 1999, pp. 147–153.

[19] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, March 1977.

[20] W. Maalej and H.-J. Happel, "Can development work describe itself?" in *Proceedings of the 7th IEEE International Working Conference on Mining Software Repositories*, 2010, pp. 191–200.

[21] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? An empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[22] J. Mylopoulos, A. Borgida, and E. Yu, "Representing software engineering knowledge," *Automated Software Engineering*, vol. 4, no. 3, pp. 291–317, 1997.

[23] K. A. Neuendorf, *The Content Analysis Guidebook*. Sage Publications, 2002.

[24] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon, "What programmers really want: Results of a needs assessment for SDK documentation," in *Proceedings of the 20th Annual ACM SIGDOC International Conference on Computer Documentation*, 2002, pp. 133–141.

[25] C. Parnin and C. Treude, "Measuring API documentation on the Web," in *Proceedings of the 2nd International Workshop on Web 2.0 for Software Engineering*, 2011.

[26] M. P. Robillard, "What makes APIs hard to learn? Answers from developers," *IEEE Software*, vol. 26, no. 6, pp. 26–34, 2009.

[27] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[28] R. L. Rosnow and R. Rosenthal, *Beginning Behavioral Research: A Conceptual Primer*, 6th ed. Pearson, April 2007.

[29] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of API documentation," in *Proceedings of the Conference on Fundamental Approaches to Software Engineering*, 2011, pp. 416–431.

[30] J. Sillito, G. C. Murphy, and K. D. Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, July-August 2008.

[31] J. Stylos, B. Graf, D. K. Busse, C. Ziegler, and R. E. J. Karstens, "A case study of API redesign for improved usability," in *Proc. Symp. on Visual Languages and Human-Centric Computing*, 2008, pp. 189–192.

[32] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proceedings of the 16th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2008, pp. 105–112.