What constitutes Software? An Empirical, Descriptive Study of Artifacts

Rolf-Helge Pfeiffer
ropf@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

ABSTRACT

The term *software* is ubiquitous, however, it does not seem as if we as a community have a clear understanding of what software actually is. Imprecise definitions of *software* do not help other professions, in particular those acquiring and sourcing software from third-parties, when deciding what precisely are potential deliverables. In this paper we investigate which artifacts constitute software by analyzing 23 715 repositories from Github, we categorize the found artifacts into high-level categories, such as, *code*, *data*, and *documentation* (and into 19 more concrete categories) and we can confirm the notion of others that software is more than just source code or programs, for which the term is often used synonymously. With this work we provide an empirical study of more than 13 million artifacts, we provide a taxonomy of artifact categories, and we can conclude that software most often consists of variously distributed amounts of *code* in different forms, such as source code, binary code, scripts, etc., *data*, such as configuration files, images, databases, etc., and *documentation*, such as user documentation, licenses, etc.

KEYWORDS

empirical study, software, artifacts

1 INTRODUCTION

Do we really have to write an article that tries to understand what *software* actually is in 2020? Is not the term and concept of *software* well understood? For two reasons, we believe it is not: *a)* There does not seem to be a standard definition that most people refer to, see Sec. 2.4 and *b)* our initial study of the Danish and British standard contracts for software sourcing, see Sec. 2.2, reveals that none of these contracts provides a practical definition of software. Additionally, different versions of the Danish standard contract have varying concepts of what precisely constitutes software, i.e., which artifacts are fundamental to software.

Stakeholders negotiating and signing software sourcing contracts are often generalists, i.e., often they are neither software engineers nor developers by education. Still, the contracts that they sign have to specify what is a deliverable and what is not. This is where the question arises: "What shall be considered software, i.e., what precisely are the artifacts to be delivered?"[1]. Definitions from standard dictionaries, such as the Oxford Dictionary of English which states that *software* are *"the programs and other operating information used by a computer."* [30] are not too helpful as they are too generic and do not provide a list of potential artifacts. On the other hand, the definitions that are provided by the software engineering (SE) community, for example from the *IEEE Software and Systems Engineering Vocabulary (SEVOCAB)* [28] and the *Software Engineering Body of Knowledge (SWEBOK)* [7], share similar shortcomings. They are either too generic, different in scope, contradictory, or they provide only incomplete examples of artifacts that constitute software, see Sec. 2.2. Similarly, definitions meant for educating software engineers provide only incomplete exemplary lists of artifacts. For example, for Sommerville [29] – likely one of the most used textbooks in SE education – software is more than just computer programs, *"but also all associated documentation and configuration data that is required to make these programs operate correctly. [...] It may include system documentation, [...] user documentation, [...] and websites for users to download recent product information."*.

Consequently, we – like others [24] – believe that the term *software* even though widely used is still not well understood in 2020. Furthermore, we agree with Osterweil in that: *"It seems odd [...] that there has been hardly any discussion of [...] the term [...] "software"."* [24] and especially with an updated version of the article [25] that states that the nature of software is likely best understood empirically. Therefore, we investigate empirically in this paper the research question:

RQ: What are the constituents of software and how are they distributed?

For real objects like for example bread, one can easily say that: Bread is the result of baking a variously treated melange of mainly flour, fewer liquids (mostly water), and small amounts of extra ingredients like salt or yeast. We aim to construct a similar understanding of software. In this paper we present –to the best of our knowledge– the first empirical study of artifacts constituting software. Like natural scientists, we set out to study a large amount of software that is stored in repositories on Github, we automatically analyze all the collected artifacts (Sec. 3), we categorize them into a taxonomy (Sec. 3.2), we analyze and discuss the found artifacts and their distributions[2] (Sec. 4), and finally we present a definition of software that is constructed out of the findings of this study (Sec. 5). With the presented taxonomy and definition we hope to help the community and practitioners to better understand what software actually is.

2 MOTIVATION & RELATED WORK

The first two sections of this related work section form the motivation for the study in this paper. In Sec. 2.2 we show that the definitions of the term *software* in industrial standards in our field are vague and

[1] For example, we know of cases where customers got delivered a systems' source code but neither a complete build specification nor the built executable. Since source code alone is unusable, build code as well as executables should be considered constituents of software.

[2] A repository with dataset and results for replication is available under DOI: 10.5281/zenodo.3701443

contradictory when describing which artifacts precisely form software. In Sec. 2.3 we show that standard contracts used in software sourcing are similarly vague and as inhomogeneous as the standards.

2.1 Definition of terms

In this paper we rely on the term *artifact*. Generally, artifacts are considered to be objects that are *"an intentional product of human activity"* [14] or more generally: *"artifacts are objects made intentionally, in order to accomplish some purpose"* [12]. In this paper, we follow the definition of ISO/IEC 19506:2012 [15], which says that an artifact *"is a tangible machine-readable document created during software development. Examples are requirement specification documents, design documents, source code and executables."*. The purpose of these artifacts is to implement, build, execute, understand, maintain, evolve, etc., software systems or to aid with any of these activities. Note, as the title of the MSR conference suggests, we consider all artifacts appearing in software repositories as software artifacts and thereby constituents of software.

2.2 Industrial Standards

To search definitions of the term *software* in standards, we used the SEVOCAB [28] and the SWEBOK [7]. The search results suggest that software is:

D.1 *"computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system [...]"* section 2.1 in [13] *"EXAMPLE: command files, job control language, both executable and non-executable software, such as fonts, graphics, audio and video recordings, templates, dictionaries, and documents"* section 3.3783 in [21]

D.2 *"all or a part of the programs, procedures, rules, and associated documentation of an information processing system. Note [...]: There are multiple definitions of software in use. [...] it is typically important to include both executable and non-executable software, such as fonts, graphics, audio and video recordings, templates, dictionaries, documents and information structures such as database records."* section 3.34 [17]

D.3 *"program or set of programs used to run a computer [...] For the purposes of this International Standard, the term "software" does not include on-screen documentation."* section 4.46 [20]

D.4 *"all or part of the programs which process or support the processing of digital information [...]: For the purposes of this definition, software excludes information per se, such as the content of documents, audio and video recordings, graphics, and databases. [...]: There is both executable and non-executable software. The purpose of non-executable software is to control or support executable software, and includes, for example, configuration information, fonts, and spell-checker dictionaries. Digital information which is managed by executable software (e.g. the content of documents and databases) is not considered software ..., even though program execution may depend on data values."* section 3.49 [16]

D.5 *"part of a product that is the computer program or the set of computer programs Note: Software excludes information per se, such as the content of documents, audio and video recordings, graphics, and databases. Digital information which is managed by executable software (e.g. the content of documents and databases) is not considered software, even though program execution may depend on data values."* section 3.34 [19]

D.6 *"[...] all or part of the programs, procedures, rules, and associated documentation of an information processing system [...]: Software is an intellectual creation that is independent of the medium on which it is recorded."* [18]

All definitions above seem to agree that, *programs* are fundamental to software and that software consists of other *artifacts* too, such as *documentation*, *graphics*, *databases*, etc. The condensed list of all artifacts mentioned in the definitions above is: *programs*, *documentation*, *data*, *documents*, *audio*, *video*, *graphics*, *databases*, *fonts*, *templates*, and *dictionaries*. We call non-executable software "data" and executable software "programs" or just "code".

The definitions seem to disagree on which artifacts precisely constitute software. For example, are *graphics* or *databases* software or not? Definition **D.4** and **D.5** suggest they are not, whereas, e.g., **D.1** suggests they are. Interestingly, none of the definitions (**D.1**–**D.6**) mentions code in various forms such as source code or binary code neither do they describe what for example *databases* are. It is unclear if they are, e.g., CSV files, binary files or the entire DBMS. That is, from reading the various standards it is not completely clear what precisely software is and what artifacts constitute software.

2.3 Standard Contracts

Standard contracts exist is some countries, which public institutions use when sourcing software. In this section we report the results of studying the Danish standard contracts (K01 [2], K02 [3], and K03 [4]) and the British Model Services Contract [5] and associated documents. We study these contracts as we expect to find a crisp definition of software, the product mainly concerned by the contract. We collect all artifacts that are mentioned in the contracts as potential deliverables and categorize them below into three high-level categories *code*, *data*, and *documentation* that are common to all contracts, see Tab. 1. We selected the contracts from Denmark and Great Britain as both countries range under the top five European nations with the highest Digital Economy and Society Index (DESI) [1], an index expressing the degree of digitization of various aspects of societies and we have not found such contracts from the other three top-five nations Finland, Sweden, and the Netherlands.

Currently, there are three Danish standard contracts. One each for short-term, long-term, and agile projects (K01, K02, and K03) respectively. Short-term projects are mostly for sourcing standard off-the-shelf software, long-term and agile projects are for sourcing software, which is either adapted to or exclusively developed for a customer.

In K01 [2] we cannot find an explicit definition of software. However, it mentions the artifacts listed in Tab. 1. Documentation is always mentioned aside of software. It appears that for the authors it is not directly software.

Unlike K01, K02 [3] provides a definition for software. Unfortunately, the definition is recursive and inconclusive. In K02 software is either *standard software*, which is software that is not specified as customer specific, *customer specific software*, which is software that is adapted, or developed by the contractors specially for the customer, and *open source software*, which is provided in form of source code and machine code based on an open-source license. See Tab. 1 for the artifacts listed in K02.

Contract	Category	Artifacts
K01	Code	Code/program, tests
	Data	Data, test data
	Docu.	Requirements specification, on-line documentation
K02	Code	Source code, object code/machine code, scripts, tests
	Data	Data, user interface templates
	Docu.	Licenses, documentation*, requirements specification
K03	Code	Source code, scripts, tests
	Data	Production data and test data, user interface templates
	Docu.	Licenses, documentation*, requirements specification
MSC	Code	Source code, object code, scripts, tests, test scripts, build scripts (linking instructions/compilation instructions), macros
	Data	Databases, configuration details
	Docu.	Licenses, test instructions, documentation*, technical specifications

Table 1: Categorized artifacts mentioned in standard contracts (Docu. abbreviates documentation).

K03 [4] provides a definition for software. The definition is different than the one of K02 but it is still recursive and inconclusive. Translated from Danish it says: *"Software consists of customer-specific software and standard off-the-shelf software, and it includes user interfaces."*. The artifacts which are mentioned in K03 are different than those of K02, see Tab. 1.

The Model Services Contract (MSC) does not seem to define software as such. However, it names more artifacts than the Danish contracts and it associates some of them directly with software: *"Any software (including database software, linking instructions, test scripts, compilation instructions and test instructions)"* [5].

In Tab. 1 we marked the *documentation* entry of K02, K03 and MSC with an asterisk. The reason is, that these standard contracts provide detailed lists of which type of documentation is required. For example, K02 lists more than 25 types of documentation (plus many more sub-types) such as system documentation, architecture description, component diagram, operations manual, etc. K03 lists 20 types of documentation, which are slightly different compared to K02, such as UML, E/R diagrams, maintenance and support documentation, etc. Similarly, MSC lists five types of documentation, such as test documentation, technical architecture documentation, etc. Interestingly, the lists of documentation artifacts types seem to be more precise than other potential artifacts, which are mentioned only briefly. In a future study, see Sec. 6, we plan to provide a more detailed list and categorization of all documentation artifacts named in the contracts.

Consequently, we believe that the definitions of *software* provided in the standard contracts are similarly inconclusive as their counterparts in the standards and that the lists of artifacts appear to be incidental, especially when compared to each other. The standards provide no rationale for the in–/exclusion of a certain artifact type.

2.4 Historical & Software Engineering Work

Shapiro [26] finds and attributes the first occurrence of the term software to Tukey who wrote in 1958 that software *"[comprises] the carefully planned interpretive routines, compilers, and other aspects of automative programming"* [32].

We started searching for definitions of the term *software* in software engineering (SE) literature since we expected software engineers to have a good understanding of what *software* actually is. To our surprise –if defined at all– the notions of software are similar to those in the standards and contracts. For example, the fundamental and often cited report on the 1968 NATO conference on SE discusses widely the challenges of SE without explicitly defining the term *software*. At the time, software was mainly considered to be source code (machine language/assembly, Fortran, Cobol, and Algol), which was used synonymously for programs: *"The present report is concerned with a problem crucial to the use of computers, viz. the so-called software, or programs, developed to control their action."* [23]. However, the lack of a suitable definition of software was already realized back then: *"the central concept in all software is that of a program, and a generally satisfactory definition of program is still needed. The most frequently used definition — that a program is a sequence of instructions — forces one to ignore the role of data in the program."* [23] Implicitly, the report mentions a set of artifacts, such as source code, compiled (binary/executable) code, data, and other artifacts, such as documentation in text and various visual notations.

Current works in SE either still avoid defining the term [9], continue using it synonymously for *program* [8], or provide exemplary lists of artifacts that constitute software: *"software [...] usually consists of a number of separate programs, configuration files [...], system documentation [...], user documentation [...], web sites for users to download recent product information."* [29]

2.5 Metaphysical Work

We find two articles that have the literal title "What is software?"[3], see [24, 31]. Suber [31] defines software metaphysically as *"software is pattern per se, or syntactical form [...] Hardware, in short, is also software, but only because everything is."* [31][4]. Unfortunately, the definition is so generic that it is not useful in practice, which is also criticized by Irmak [14] who seems to use software synonymously for program and who defines software as an abstract artifact similar to music. It is abstract since as long as somebody can remember a certain algorithm and can reimplement it, the corresponding software does not cease to exist. Since metaphysical, Irmak does not/cannot provide a list of constituting artifacts as they would only be representations of software but not the software itself. Similarly, Wang et al. [33] provide an initial ontology of software in which they characterize software via its relation to other concepts, such as, requirements and specifications.

The other article under the title "What is software?" [24] does not study constituting artifacts of software either. Likely that is even

[3]We searched `https://dl.acm.org/`, `https://link.springer.com/`, with the search query "What is software" on the title. Even though the search returned five results on ACM DL and on Springer Link respectively, and ca. 3 730 results on Google Scholar, after manual inspection we find only the mentioned two articles with that title.
[4]Similar to Def. **D.4**, which excludes data from being software, Suber distinguishes between data and software. For example, he states that a word processor text document is not software but data.

impossible as for Osterweil software is intangible and non-physical. Instead, the author describes typical characteristics such as compositional structure, interconnected components, etc.

Even though highly inspiring, the metaphysical work is not directly applicable, when one has to know what artifact to include in a contract as deliverable.

2.6 Empirical Work

Bigliardi et al. [6] study 35 open-source projects and they too wonder what – other than source code – constitutes software. They call *"[. . .] artifacts other than code, such as documentation and examples, build system and configurations, or graphics and translations."* non-code artifacts. They find that on average almost 50% of the artifacts in each project are non-code artifacts. Our results, see Sec. 4 do not confirm this high amount. Bigliardi et al. [6] provide no explicit taxonomy but an exemplary list of artifacts, such as, documentation, images, multimedia, etc. which makes it hard to understand what precisely non-code artifacts are. We collect all mentioned artifacts in Tab. 2 in the same way as we did for the contracts. Furthermore, it seems as if they consider source code to be code only. A possible classification of executables (binary code) as non-code artifacts appears possible but counter-intuitive. Lastly, they consider build files, such as Makefiles non-code artifacts. We do not agree on that, as they are code files that are interpreted to perform a certain task.

Robles et al. [27] study non-code artifacts in KDE, which at the time was stored in a mono-repository. That is, one repository containing a wide range of software systems, such as the actual KDE GUI, an office suite, a file explorer, and many more. Robles et al. study the artifacts of all the sub-software uniformly and mention the artifacts that are listed in Tab. 2. They find that source code accounts for only 24.4% of the total amount of artifacts, even fewer than in Bigliardi et al. [6] and fewer than in our results Fig. 3.

Ma et al. [22] study which types of artifacts appear in open-source software repositories. They suggest a technique based on Machine Learning to identify various artifact types and present a corresponding taxonomy. They identify the artifacts given in Tab. 2. The main focus of their study is to investigate various techniques to classify artifacts, which is not a simple task. For example, Ma et al. categorize manifest files, Makefiles, configuration files, and version requirement files as *setup files*, where we would consider them either data, such as configuration files, or code since for example Makefiles are meant to be executed with a corresponding interpreter (make).

To allow for automatic requirements tracing, Dekhtyar et al. [10] suggest that *"Text is Software Too"* [10] and that text documents should be part of software repositories. They name requirements specifications, design specifications, user manuals and comments in the code; as examples for natural language documents of interest, see Tab. 2. Additionally, they mention source code and code in general but it remains unclear what precisely they mean there. They do not investigate other kinds of data. Our results suggest, that textual documentation is actually stored in repositories.

Previous studies [6, 27] usually identify different artifact types via regular expressions targeting mainly file extensions. We find that ca. 5% (707 012) of the artifacts in our dataset do not contain file extensions (often the interesting ones, such as Makefiles, configuration,

Paper	Category	Artifacts
[6]	Code	Source code, binaries
	Docu.	documentation and examples
	Data	Configurations, graphics, translations, build files
[22]	Code	Source code, testing code, application (a mix of script and binary code)
	Data	Font, image, archive, audio, disk image, project
	Docu.	contributors' guide, design documents, license, list of contributors, release notes, requirement documents, setup files
	Other	Non-readable, non-english, empty or too small files.
[10]	Code	Source code, code
	Docu.	requirements specifications, design specifications, user manuals
[27]	Code	Source code, build code,
	Docu.	Documentation, devel-doc
	Data	Images, translation, user interface, multimedia
	Other	Unknown

Table 2: Categorized artifacts mentioned in the empirical studies (Docu. abbreviates documentation).

licenses, etc.). Therefore, we suggest an artifact categorization technique similar to [22] but more lightweight (no machine learning), which relies in essence on a set of heuristics on file contents and file naming conventions in combination, which is explained in the following section.

3 THE EXPERIMENT

3.1 Research Questions

To investigate our research question (What are the constituents of software and how are they distributed?) we formulate the following more concrete research questions, which are studied in Sec. 4:

RQ1: To which degree is software more than code?
RQ2: Is documentation an integral constituent of software?
RQ3: Does software without data artifacts exist?
RQ4: Does software without code exist?
RQ5: Does a characteristic distribution of frequencies of artifact categories exist?
RQ6: Does the ratio between frequencies of artifact categories depend on the size of software?

To study these research questions, we have to collect and prepare data. The following section describes the experimental setup.

3.2 Experimental Setup

To study software from different domains and application areas, we query the Github search API[5] for popular[6] repositories of the following 25 languages respectively: ABAP, Ada, Assembler, C, COBOL,

[5]`https://developer.github.com/v3/search/`
[6]By popularity we mean the *starred* criteria with which Github users express liking similar to likes in social networks.

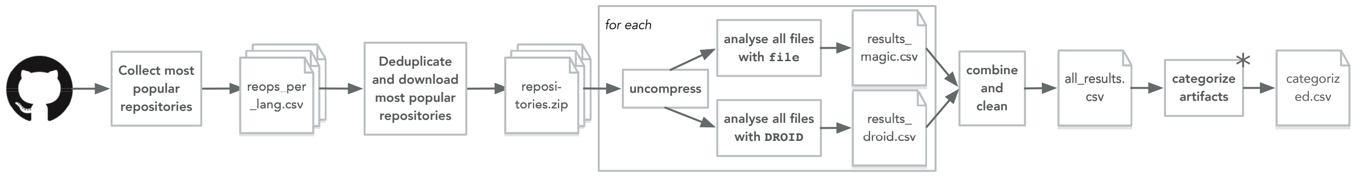

Figure 1: The analysis engine pipeline. Development of artifact categorization (*) is explained under Method 3.3.

C++, C#, D, Erlang, Fortran, F#, Go, Groovy, Java, JavaScript, Kotlin, Lua, Objective-C, OCaml, Perl, PHP, Python, Ruby, Scala, and Swift.

A priori, we chose 25 separate languages as we know that the API returns 1 020 repositories per language qualifier, resulting in more than 25 thousand repositories for analysis. Without language qualifier, the API returns only 1 020 repositories in total, which we decided is not enough for our study. The 25 languages were selected – out of the more than 370 programming languages that Github recognizes[7] – to be exemplar for different software domains from system-level (in C, D, etc.) to high-level (e.g., Java, C#, etc.) software and ranging from scientific software over mobile-, web-applications and operating systems to games.

Thus, querying the search API results in 25 500 repositories. We collect all of them without applying a threshold. After deduplication[8] we download 23 718 unique repositories. For each of the repositories, we download a recent snapshot as ZIP file. To save space on the analysis machine[9] we do not download the entire history of the Git repositories.

Fig. 1 illustrates the dataflow in our analysis engine (a set of Python and Bash scripts), which follows a pipeline architecture[10]. Our analysis engine relies on two readily available tools for classification of files. These are the *Fine Free File Command* (`file`)[11] and the Digital Record and Object Identification (`DROID`)[12] from the UK National Archive. Both tools automatically detect the kind of a given file, its media type (mime type), encoding, etc., by searching heuristically for *magic numbers*[13] or similar patterns. A magic number is a combination of bytes, i.e., patterns of content in files, which are unique to a certain file type.

After download, the analysis engine uncompresses each single repository. Each file within each repository is analyzed separately by the `file` tool and by the `DROID` tool. Per repository, the engines store a CSV file with the identification results. Lst. 1 illustrates abbreviated output of this analysis step.

After a complete analysis (which takes ca. five days on a machine with eight processor cores, on which the analysis is executed in parallel on all cores), results per repository are combined into a single CSV file containing more than 13M rows, one row per artifact.

The last step that the analysis engine executes is categorization of each artifact based on the information generated by the two tools.

[7]`https://github.com/github/linguist/blob/master/lib/linguist/languages.yml` many of the languages are esoteric or niche languages, such as NetLogo, Moocode, Wollok, Redcode, etc., which we decided not to include in this study.
[8]The Github API can return repositories containing a mix of languages as most starred for more than one language.
[9]The entire dataset in compressed format (ZIP) is bigger than 208GB, which corresponds to more than 648GB uncompressed data.
[10]Note, the entire experimental setup with all scripts and necessary data can be accessed and downloaded for replication DOI: 10.5281/zenodo.3701443
[11]`http://www.darwinsys.com/file/`
[12]`https://digital-preservation.github.io/droid/`
[13]`https://en.wikipedia.org/wiki/File_format#Magic_number`

```
path,kind,mime_type,encoding,ext
llvm/.arcconfig,ASCII text,text/plain,charset=us-ascii,
llvm/polly/CREDITS.txt,ASCII text,text/plain,charset=us-ascii,.txt
llvm/README.md,ASCII text,text/plain,charset=us-ascii,.md
llvm/polly/test/update_check.py,"a /usr/bin/env
    ↪ python3 script, ASCII text executable",text/plain,charset=us-ascii,.py
llvm/polly/test/create_ll.sh,"POSIX shell
    ↪ script, ASCII text executable",text/x-shellscript,charset=us-ascii,.sh
```

Listing 1: Abbreviated output of the `file` tool's analysis

The following section (Sec. 3.3) explains in detail how we developed the categorization logic, which is marked by an asterisk in Fig. 1. After categorization, each artifact is associated with one of the four high-level categories *programming*, *data*, *documentation*, or *other* and one of the 19 concrete categories, from our taxonomy in List 1.

Code: *source code*, *specific source code*, *script*, *binary code*, *build code*, *infrastructure code*
Data: *image*, *video*, *music*, *configuration*, *database*, *font*, *archive*, *markup*, *document*, *app data*
Documentation: *prose*, *legalese*
Other: *other*

List 1: Taxonomy of high-level categories (in bold) and concrete sub-categories for artifacts.

In the following we explain the concrete categories: *Specific source code* is source code for which the tools `file` and `DROID` provide more detail about its kind, such as Maple, Matlab, Qt C code, etc. *Script code* is source code in languages, such as, Ruby, Bash, etc., which are executed by an interpreter and often contain a corresponding header line specifying the interpreter. By *binary code* we mean everything that is compiled machine code no matter if for a real machine architecture or for virtual machines. *Build code* is code meant to build source code or to bundle packages, such as Makefiles, Rakefiles, Gradle scripts, etc. *Infrastructure code* is meta-code to create certain environments, such as Dockerfiles, Vagrantfiles, etc. As *music*, we classify any audio artifact also just sound effects or other recordings. *Configuration* are plain text files with file endings, such as, `ini`, `cfg`, `config`, etc. As *database* we categorize binary files storing data from DBMS like SQLite, MySQL, Redis, etc. *App data* are files that store data for specific applications, such as diff output, Ctags tag files, Git indexes, etc. As *archives* we categorize all archives no matter if compressed or not, such as TAR, ZIP, or RAR archives. We categorize *markup* as data since these artifacts are most often XML or HTML files storing data for certain applications or they serve as templates to be filled with more data at runtime. *Documents* are all artifacts, which are meant to be used via/generated by a particular application, such as Excel spreadsheets, FrameMaker documents, CAD files, etc. Documents are no documentation since, e.g., CAD files and Excel

```
mask = (((DF.kind.str.startswith('PDF_document')) |
    (DF.kind.str.contains('Postscript')) |
    (DF.kind.str.contains('WordPerfect')) |
    (DF.kind.str.contains('Microsoft_Word_2007+')) |
    (DF.kind == 'OpenDocument_Text') |
    (DF.name == 'README') | (DF.name == 'NOTICE') |
    (DF.name == 'TODO') |
    (DF.ext == '.rst') | (DF.ext == '.txt') |
    (DF.ext == '.md')))
DF['concrete'][mask] = 'prose'
DF['high_lvl'][mask] = 'documentation'
```

Listing 2: Illustration of categorization constraints.

spreadsheets usually are artifacts not documenting software but that are more meant to store certain information, i.e., data.

Legalese are all text artifacts that are concerned with licenses, copyright notes, or patents, which are identified by respective file names, such as `LICENSE`, `LICENCE COPYRIGHT`, or `PATENTS`. *Prose* are all artifacts, which are either plain text files with a file extension, such as `txt`, `md`, `adoc`, etc. or, a representative name such as `README`, `NOTICE`, `CHANGELOG`, etc. Additionally, we categorize all word processor files and Postscript and PDF files as prose. We think they are most often used to store natural language documents. *Other* is the category for all artifacts that appear so rarely in the dataset that we did not develop a categorization rule, such as GKS metafiles, Java JCE KeyStore, FoxPro FPT files, etc., see Sec. 3.3.

Even though all the contracts discussed in Sec. 2.3 mention *tests* under the category code and *test data* under the category data, see Tab. 1, our taxonomy in List 1 does not contain a category for tests or test data. The reason is that we consider tests as source code and test data just as data. We do not consider tests to be a proper category for an artifact. Instead, it is a role that an artifact plays. Roles form another dimension. In this paper we do not address this issue but will consider it in future work, see Sec. 6. Similarly, we do not search for certain kinds of documentation as that would require an even deeper inspection of the corresponding artifacts and association of heuristics that assign potential roles to artifacts. In this paper we are studying artifacts by their syntactic appearance but not by semantics, which could be inferred out of them.

3.3 Method

This section describes how we developed the artifact categorization, i.e., the taxonomy in List 1 and the program marked with an asterisk in Fig. 1.

The four high-level categories *code*, *data*, *documentation*, *other* are inspired by *a)* our work on the contracts for which we tried to find a suitable high-level categorization and *b)* Linguist[14] an open-source programming language identification tool. The only difference to Linguist's high-level categories[15] is that we decided to categorize mark-up artifacts as a subcategory of data and not as a separate high-level category.

From the related work, we had an initial set of concrete categories, such as, font, image, configuration, etc. Via an iterative and greedy process over artifacts of ca. 2 400 repositories (ca. 10%) we develop a set of boolean constraints over kind, mime_type, file name and extension fields of the data generated by `file` and `DROID`. We sort

```
mask = ((DF.kind.str.contains('.*_script,', regex=True)) |
        (DF.kind.str.contains('batch')))
DF['concrete'][mask] = 'script'
DF['high_lvl'][mask] = 'code'
```

Listing 3: Illustration of categorization constraints.

all uncategorized artifacts by the most frequent occurrence of data that the tools report and create a suitable boolean constraint as illustrated in Lst. 3. We manually verify that the vast majority of artifacts that are now matched by this constraint actually are of the assigned artifact category. We repeat this process until all but 10 414 (0.07%) of the more than 13M artifacts are categorized into the three high-level categories *code*, *data*, *documentation*, and 18 concrete categories (except *other*). The remaining 0.07% artifacts are categorized as *other*, i.e., as artifacts which either appear so rarely in the dataset that we did not develop a categorization rule or for which we do not know how to categorize them, e.g., lif files, Micro Focus Index files, etc.

Due to space limitation, the complete list of categorization rules is only available online as part of the analysis engine[16]. For the 19 concrete categories we have in total 340 atomic patterns, 134 for code, 117 for data, 88 for documentation and 1 for other. For example, 84 atomic patterns form by disjunction one large rule to categorize prose, see an excerpt in Lst. 2 listing 11 of the 84 atomic patterns. Lst. 3 illustrates the smallest categorization rule that identifies script code. It can be read as follows: all artifacts for which the `file` tool reports some kind of script or some kind of batch file (compare with, e.g., Lst. 1) are categorized as *scripts* which are a sub-category of the high-level category *code*.

Essentially, our categorization rules aggregate the classifications from the `file` and `DROID` tools into our taxonomy. In case the tools classify an artifact wrongly this wrong classification may propagate into our categorization. For example, the tools may classify a Ruby script generating HTML code wrongly as an HTML markup file in case they find a corresponding pattern before finding a suitable Ruby pattern. In this case the misclassification is carried over into our categorization and the Ruby code gets wrongly categorized as HTML, i.e., markup and thereby data. Sec. 5.1 briefly discusses the quality of classification and possible effects on our categorization showing that misclassifications affecting our categorization are quite rare.

3.4 Characteristics of the Dataset

The dataset (`categorized.csv` in Fig. 1) on which all analysis in Sec. 4 is based, consists of 23 715 repositories containing 13 840 142 artifacts. After deduplication of the originally 25 500 repositories, 23 718 unique repositories remained from which three were excluded[17] as the received ZIP files were not valid archives. Another single repository in the dataset[18] (a security oriented repository) was only analyzed with `file` and not with `DROID` as it contains Zip Bombs [11] and `DROID` by default uncompresses archived artifacts for analysis, which would cause a crash of the entire analysis system. In total, we analyzed 648GB of uncompressed artifacts.

The repositories are quite diverse in that they range from big projects from companies such as Google, Microsoft, IBM, Facebook

[14]`https://github.com/github/linguist`
[15]field type in `https://github.com/github/linguist/blob/master/lib/linguist/languages.yml`

[16]`https://github.com/HelgeCPH/msr2020_what_constitutes_software/blob/ffffc004547ecadc1524f95c726accadae03b706/src/interpretation.py#L655`
[17]illera88/Ponce, Entomy/libAnne, michael-valdron/course-transaction-system
[18]danielmiessler/SecLists

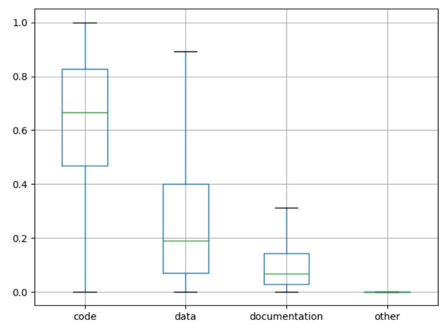

Figure 2: Distribution of relative frequencies per high-level artifact category.

over big organizations, such as the Apache, Eclipse, or Python Software Foundation over projects of lesser known organizations or groups to small repositories of single users. They cover various domains from scientific computing (many of the Fortran repositories) over mobile and web-applications (many of the Swift, Kotlin, and JavaScript repositories) and operating systems (Linux kernel, FreeBSD, OpenBSD, Minix, etc.) to games and retro computing (many of the assembler repositories). The over 13M artifacts are not distributed equally over the the analyzed repositories. There is at minimum one artifact and at maximum 90 976 artifacts in each repository. On average there are ca. 583, median 73 (q_1: 25, q_3: 254, std. ca. 2 803) artifacts per repository.

4 RESULTS

In this section we use the following symbols for more terse presentation median: q_2, mean: μ, standard deviation: σ, minimum: $\wedge$, maximum: $\vee$. The numbers given in the text are rounded when nothing else is indicated. Remember when reading this section, that Fig. 2, Fig. 5, and Tab. 3 are about high-level categories and Fig. 3, Fig. 4, and Tab. 4 are about concrete categories. Also all figures, tables and numbers in text can be found online for traceability and reproducibility[19].

4.1 RQ1: To which degree is software more than code?

Fig. 2 illustrates the relative amount of artifacts of the corresponding high-level categories over all repositories of the dataset. The upper and lower box boundaries represent q_1 and q_3, the 25% and 75% quartile respectively and the green-line represents the median (q_2, 50% quartile). It shows that some repositories contain solely code as the upper whisker of the *code* category is placed directly on one, i.e., 100% code, and that usually $\frac{2}{3}$ (q_2: 67%, μ: 63%, σ: 0.24) of a repository is code. Tab. 3 shows, that only 234 (ca. 1%) repositories contain solely code artifacts in their various forms. These 234 code-only repositories, contain most often (71%) only source code, 13% of them contain also *binary code*, and rarely (3%) *build code*. Software consisting solely of scripts is virtually non-existent. Only two repositories are script-only (`iZsh/SSLSniffer` a DTrace script for network communication analysis and `lpw25/girards-paradox` an OCaml script implementing a type-theoretical paradox, both with

Category combination	Occurrence Frequency	Occurrence in %
code, data, documentation	20 650	87.076%
code, data, documentation, other	1 068	4.503%
code, documentation	1 406	5.929%
code, data	286	1.206%
code	234	0.987%
documentation	23	0.097%
data	3	0.013%
data, documentation	35	0.148%
code, documentation, other	5	0.021%
code, data, other	3	0.013%
code, other	2	0.008%

Table 3: Overview of amount of occurrences of the high-level category combinations in the dataset.

a single script each). The 234 code-only repositories are all small ranging from one to 152 artifacts (q_2: 2, μ: 6.65, σ: 15.25).

The data shows that usually (99%) software consists of heterogeneous artifacts of more than a single artifact type. When considering the 19 concrete artifact categories, there are only 212 repositories (less than 1%) containing artifacts of a single concrete category only: *source code* 165, *prose* 21, *spec. source code* 16, *binary code* 5, *app data* 2, *script* 2, *configuration* 1. Most commonly (more than 96%) of the repositories consist of artifacts of at least three different concrete categories. This diversity is illustrated in Fig. 4, which shows the cumulative relative frequencies of concrete category combinations, i.e., amount of repositories with a single artifact category, amount of repositories with any combination of two categories, etc. Cumulative means that, e.g., 405 different combinations with nine concrete categories exist in the dataset and none of these appears with more than 1.5% but together they sport more than 12%. Additionally, Tab. 4 lists the five most common concrete category combinations and their relative frequency of occurrence in the dataset, which are all below 4% for each single combination. Note, software with artifacts of all possible concrete categories is unlikely to be found. The dataset, contains only a single instance, the rapid7/metasploit-framework tool, a penetration testing framework.

Since such a high amount (more than 96%) of repositories contain artifacts from at least three concrete categories and since frequencies of each single combination are quite low, we do not only conclude that software is usually more than code but also that it is usually quite diverse in terms of kinds of artifacts.

4.2 RQ2: Is documentation an integral constituent of software?

Out of the 23 715 repositories only 639 (roughly 3%) do not contain any *prose* and 7 383 (31%) do not contain any *legalese*. Only 528 (2%) of all repositories contain neither prose nor legalese – the two concrete categories which form together the high-level category *documentation* –, see Tab. 3. Similar to above, repositories without documentation are consistently smaller than those with documentation (q_2: 7, $\wedge$: 1, $\vee$: 1 733, μ: 45, σ: 165.7 compared to q_2: 76, $\wedge$: 1, $\vee$: 90 976, μ: 596, σ: 2 833.6 in the repositories with documentation). In the 23 187 (ca. 98%) repositories that contain documentation, usually there are ca. 5% (q_2: 5%, μ: 10%, σ: 0.12) prose artifacts and 0.5% (q_2:

[19]Repository: `https://github.com/HelgeCPH/msr2020_what_constitutes_software`, DOI: 10.5281/zenodo.3701443

Category combination			# Concr. Categ.	Abs. freq.	Rel. freq.
Code	Data	Docu.			
source code, spec. source code, script, build	app data, archive, config., image, markup	prose, legalese	11	930	3.92%
source code, spec. source code, script, build	app data, config., image, markup	prose, legalese	10	546	2.3%
source code		prose	2	445	1.88%
source code	markup	prose	3	351	1.48%
source code, spec. source code	app data, config., image, markup	prose, legalese	8	342	1.44%

Table 4: Occurrence frequencies of the five most common concrete category combinations grouped by high-level category.

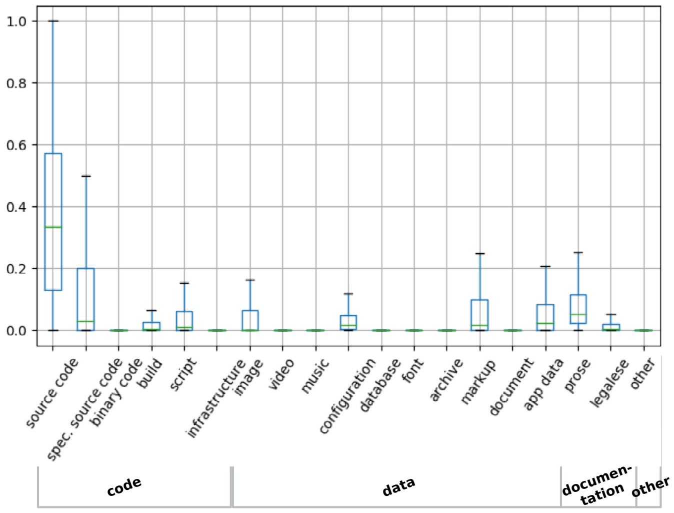

Figure 3: Distribution of relative frequencies per concrete artifact category.

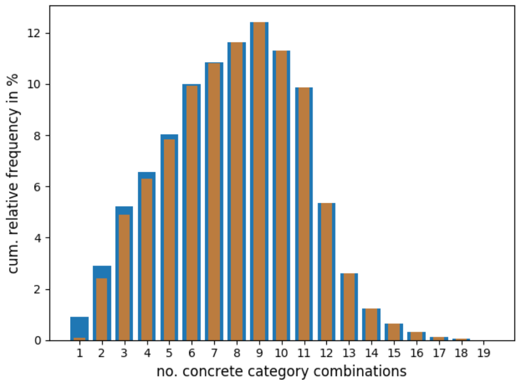

Figure 4: Cumulative relative frequencies of repositories with artifacts of x different concrete categories (blue). In orange the amount of those containing documentation.

0.5%, μ: 2%, σ: 0.04) legalese artifacts. The high amount of documentation in all kinds of repositories is also illustrated by the orange bars in Fig. 4, which show the cumulative frequencies of concrete category combinations of the repositories containing documentation.

Since ca. 98% of the repositories contain documentation, of which the majority is usually prose, and since on average about a tenth of the artifacts in these repositories are prose, we consider documentation to be an integral constituent of software, i.e., in the “wild” it is quite unlikely to find code or data artifacts that are not associated with documentation artifacts.

4.3 RQ3: Does software without data artifacts exist?

Surprisingly, we find 1 670 repositories (ca. 7%) that do not contain any data, such as, images, videos, configuration, etc. On deeper investigation we find that repositories without data are most often either libraries, i.e., software to be reused by other software, system-near software, such as, low-level code for embedded/mobile devices, scientific code that collects data at runtime, or they are just very small projects. Again repositories without data are smaller compared to the size of those containing data (without data: q_2: 6, $\wedge$: 1, $\vee$: 1 824, μ: 21, σ: 70.9 versus q_2: 84, $\wedge$: 1, $\vee$: 90 976, μ: 626, σ: 2 902.9 with data). On the other extreme, repositories with high amounts of data ($\geq$ 95%) are often either repositories containing fonts, icons, themes, or games with many images or they are just repositories that document other software, such as, API documentation.

We realize that software without data exists as ca. 7% of all repositories do not contain data in separate artifacts. For small software ($\leq$ 10 artifacts) that share increases to above 13%. Additionally, it seems as if software without data mainly appears in certain domains.

4.4 RQ4: Does software without code exist?

Only 61 (0.26%) repositories do not contain any kind of code, i.e., neither source code, scripts, nor binary code, etc. Further inspection reveals, that most of these no-code repositories are quite small in terms of number of artifacts (q_2: 2, $\wedge$: 1, $\vee$: 342, μ: 22, σ: 58.4 compared to: q_2: 73, $\wedge$: 1, $\vee$: 90 976, μ: 585, σ: 2 806.6 in the entire dataset). The no-code repositories contain either mostly prose artifacts, which is especially true for the smallest ones or they are resource storages for images, configuration and app data. It seems as if these repositories are informational artifact libraries containing artifacts meant for reuse either in other software (e.g., images or icons), they are training manuals/websites, or collections of links. Note, we find all of the artifacts of no-code repositories (app data, configuration, image, legalese, markup, prose, etc.) in other repositories, which contain code too, i.e., they are not special themselves.

Osterweil [24] presents the idea of software that is not computer software, i.e., software that is executed for example by humans. He refers to law texts or recipes as examples for such non-computer software. Here, we can confirm that such software really exists for example as manuals or tutorials that explain humans how to perform a certain task.

Even though quite rare, we find that software without code exists. Remember, that this only means that there are no separate code artifacts. Multiple of the no-code repositories contain code embedded in *prose* artifacts but not as separate *code* artifacts.

4.5 RQ5: Does a characteristic distribution of frequencies of artifact categories exist?

When starting this study we had the following hypothesis concerning the distribution of frequencies of high-level artifact categories: *Software consists mainly of code (in various forms such as source code, scripts, binary code, etc.) to a medium amount of data such as configuration files, images, databases, etc. and to minor amount of documentation, licenses, etc.* That is, we hypothesize that software is usually more code than data and more data than documentation.

To test our hypothesis, we select all repositories, that contain artifacts of the three high-level categories *code*, *data*, and *documentation* and that contain at least six artifacts, as this is the lowest amount of artifacts that could potentially produce a major amount of code, medium amount of data, and minor amount of documentation distribution (Three code, two data, and one documentation artifact respectively. In that case, there would be 1.5 times more code than data and two times more data than documentation).

There are 21 309 repositories (ca. 90% of the entire dataset) that contain at least six artifacts of the thee high-level categories *code*, *data*, and *documentation*. For each of these repositories, we compute the code-to-data-ratio (r_{c_dat}) in between relative frequencies of *code* and *data* artifacts, and the data-to-documentation-ratio r_{dat_doc} in between relative frequencies of *data* and *documentation* artifacts. For example, with a code-to-data-ratio $r_{c\_dat} > 1$, a repository contains more code than data and with data-to-documentation-ratio $r_{dat\_doc} \leq 1$ a repository contains more documentation than data. There are four different distribution types as illustrated in Fig. 5: *a)* more code than data and more data than documentation ($r_{c\_dat} > 1$ and $r_{dat\_doc} > 1$), *b)* more data than code and more data than documentation ($r_{c\_dat} \leq 1$ and $r_{dat\_doc} > 1$), *c)* more code than data and more documentation than data ($r_{c\_dat} > 1$ and $r_{dat\_doc} \leq 1$), and *d)* more documentation than data and more data than code ($r_{c\_dat} \leq 1$ and $r_{dat\_doc} \leq 1$). In Fig. 5 we plot the median of the relative occurrences of code, data, and documentation respectively for the repositories categorized by these four different distribution types together with their occurrence likelihood compared to the entire dataset.

Interestingly, only 45% of all repositories contain more code than data and more data than documentation. Ca. 19% of the repositories contain more data than code ($r_{c\_dat} \leq 1$) and more data than documentation $r_{dat\_doc} > 1$. Around a quarter of all repositories contain more code than data ($r_{c\_dat} > 1$) but less data than documentation $r_{dat\_doc} \leq 1$. An absolute minority (ca. 1%) of repositories consists of a major amount documentation, a medium amount data, and a small amount code ($r_{dat\_doc} \leq 1$ and $r_{c\_dat} \leq 1$). The remaining repositories are those that do not contain artifacts of each of the three high-level categories *code*, *data*, and *documentation*.

Consequently, we reject our initial hypothesis as less than half of the repositories contain more code than data and more data than documentation. It seems as if there does not exist one clear distribution pattern of high-level artifact categories for software. However, we can say that it is quite unlikely (1%) to find software with more documentation than data and more data than code.

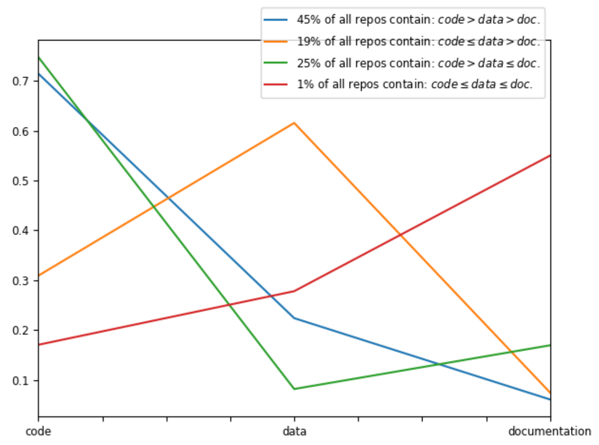

Figure 5: Median (y-axis) of relative frequencies of code, data, and documentation respectively in repositories with at least six artifacts and per category distribution as given in legend.

4.6 RQ6: Does the ratio between frequencies of artifact categories depend on the size of software?

The question means: Is it that the bigger the software the more it contains code than data and the more it contains data than documentation? When starting this study we had the following hypothesis concerning the size of software and frequencies of artifact categories: *The ratios of code to data and data to documentation are independent of the size of the software.*

To test our hypotheses, we use the same 21 309 repositories that contain at least six artifacts of all three high-level categories code, data, and documentation as in Sec. 4.5. Neither size (number of artifacts), the code-to-data-ratio r_{c_dat}, nor the data-to-documentation-ratio r_{dat_doc} are normally distributed[20]. Therefore, we compute Spearman's ρ to test for a correlation between *a)* the size and r_{c_dat}, and between *b)* the size r_{dat_doc} respectively. For both cases, we use the null hypotheses $H_{0a)}$ size and the r_{c_dat} are uncorrelated and $H_{0b)}$ size and the r_{dat_doc} are uncorrelated.

The results of the test are: $\rho_{size\_code\_data} \approx 0.1207$, $n = 21309$, $p < 0.05$, i.e., a very weak positive monotonic correlation, and $\rho_{size\_data\_doc} \approx 0.2672$, $n = 21309$, $p < 0.05$, i.e., a weak positive monotonic correlation. Since the p-values are in both cases very close to zero(< 0.05) we reject the null hypotheses. Thereby, we also reject our hypothesis.

Since size and the code-to-data-ratio are very weakly correlated and since size and the data-to-documentation-ratio are weakly correlated one may formulate that the bigger the software, the more

[20]Anderson-Darling test for normality with null hypothesis H_0: the size, r_{c_dat}, r_{dat_doc} respectively are normally distributed returns very high values (5 655.11, 7 760.92, 6 479.58), all way above the critical values for any p-values. That is, we reject the null hypotheses, i.e., none of the data is normally distributed.

it contains code than data and the more it contains data than documentation. But since the correlations are so weak, we are prudent to conclude that in general.

5 DISCUSSION

With this study we confirm the common belief that software is usually more than just code as 99% of the studied repositories are not code-only. Even though code in its various forms plays a major role in software – commonly $\frac{2}{3}$ of software is code –, our results suggest that using *software* as synonym for *code* or *program* may be inaccurate. Software is more diverse, which is demonstrated by the fact that the most common distribution of 11 artifact categories accounts for only 3.92% of all studied software and the second most common accounts for only 2.3%. Documentation is omni-present in software. 98% of the studied repositories contain it and on average 10% of a repository are documentation artifacts.

So far, we cannot find a certain distribution of artifact categories that is characteristic for software in general, see Sec. 4.5 and 4.6 but it seems as if certain distributions are characteristic for certain kinds of software, such as no data for libraries, high amount of images for games, broad coverage of categories in large software, see Sec. 4.1. We find weak correlations between size of software and code-to-data ratio and data-to-documentation ratio, which suggests the larger the software the more it contains more code than data and more data than documentation. The lack of clear distribution patterns and the diversity in artifact combinations suggest that software is any possible combination of artifacts though seldomly more documentation than data and more data than code.

The concrete categories of our taxonomy (List 1) may be divided further into even more concrete categories. For example, archives can be compressed or uncompressed, binary code can be categorized further regarding a target architecture, etc. Since we find for all concrete categories artifacts, we do not expect that the 19 concrete categories will change drastically in future except for further sub-division.

5.1 Threats to Validity

The scope of analysis in our experiment is a Github repository. However, there is not necessarily a one-to-one relationship between a *repository* and *software*. Some repositories, such as the Minix OS or many of the games, contain self-contained software in the sense of software products whereas others contain just libraries, i.e., software to be reused by other software. That is, software may be an aggregate of the contents of multiple repositories or of artifacts from entirely different sources. Additionally, artifact distributions may be different at development, delivery, or production time for the same software. A closer study of artifacts and their distributions across inter-dependent repositories, which together form a software product remains future work.

Github does not list many highly starred Cobol or ABAP repositories compared to Python repositories (Cobol stars: q_2: 0, $\wedge$: 0, $\vee$: 542, μ: 2.2, σ: 26.1 vs. Python stars: q_2: 3 774, $\wedge$: 1, $\vee$: 72 944, μ: 5 869, σ: 6 905). This skew distribution of stars may affect our results as low starred repositories might be less mature and less representative. Another 2 000 highly starred projects might have changed the distribution of artifact categories especially in Sec. 4.5. Though, we think that the big amount of analyzed repositories minimizes this risk.

Besides that repositories on Github may not be representative, projects may have differing politics on distributing executables and documentation. Binary code and documentation may not be provided via the analyzed repositories. Due to the experimental setup we do not analyze artifacts from other channels. We do not consider this a major risk. For documentation we categorize its sources correctly and for binaries we categorize build code, which can serve as a proxy for potential binary code.

Since based on heuristics, the tools `file` and `DROID` might classify some files wrongly. For example, a Ruby script generating an HTML file might get classified as markup when the tools find an HTML header encoded in a string. Similarly, binary data that accidentally contains a magic number can be classified wrongly. Our categorization does not fix misclassifications of the underlying tools. We randomly sampled 20 artifacts from each concrete category (320 files) and compared the tool's verdict against a human classification. We find that 12 files (3%) are classified wrongly. The majority of misclassifications remain within the same high-level category on which much of our analysis is based. Four code artifacts (1%) are wrongly categorized as data and three data artifacts (0.8%) are erroneously categorized as code.

6 CONCLUSIONS & FUTURE WORK

Instead of understanding *software* metaphysically or by example, see Sec. 2, we conduct an empirical study of 23 715 repositories containing 13 840 142 artifacts. We find that artifacts of our 19 concrete categories *source code*, *specific source code*, *script code*, *binary code*, *build code*, *infrastructure code*, *image*, *video*, *music*, *configuration*, *database*, *font*, *archive*, *markup*, *document*, *app data*, *prose*, *legalese*, and *other* are constituents of software. Unlike in earlier definitions, these are not examples but empirically confirmed.

Thus, our current working definition of *software* is: Software is the collection of all of the above artifacts that humans collect and store and these artifacts are more than just code, documentation is an integral part of them, depending on the domain data is not necessary, and – even though rare – code is not strictly necessary.

We plan to complement this study by *a)* extending our initial study of standard software sourcing contracts (Sec. 2.3), *b)* a literature study of what software quality models consider to be software, and *c)* a survey amongst experts about their conceptions of software.

We aim for two major results *a)* a comprehensive taxonomy of artifacts (a more detailed version of the one presented in List 1) that can be used by practitioners as a kind of checklist of constituents of software and *b)* a practical definition of software that can assist in deciding which artifacts should be considered software.

ACKNOWLEDGMENTS

We thank Sami Brandt and Paolo Burelli for helping us understand and interpret our data and Paolo Tell, Mircea Lungu and the anonymous reviewers for their valuable feedback on earlier versions of this paper.

REFERENCES

[1] [n.d.]. Digital Economy and Society Index Report 2019. https://ec.europa.eu/newsroom/dae/document.cfm?doc_id=59975. Accessed: 2020-01-08.
[2] [n.d.]. K01 Standard Contract for Short Term IT Projects. https://digst.dk/styring/standardkontrakter/k01-standardkontrakt-

for-kortvarige-it-projekter/. Accessed: 2019-10-11.

[3] [n.d.]. K02 Standard Contract for Long Term IT Projects. https://digst.dk/styring/standardkontrakter/k02-standardkontrakt-for-laengerevarende-it-projekter/. Accessed: 2019-10-11.

[4] [n.d.]. K03 Standard Contract for Agile IT Projects. https://digst.dk/styring/standardkontrakter/k03-standardkontrakt-for-agile-it-projekter/. Accessed: 2019-10-11.

[5] [n.d.]. Model services contract. https://www.gov.uk/government/publications/model-services-contract. Accessed: 2019-10-11.

[6] Luca Bigliardi, Michele Lanza, Alberto Bacchelli, Marco DAmbros, and Andrea Mocci. 2014. Quantitatively exploring non-code software artifacts. In *2014 14th International Conference on Quality Software*. IEEE, 286–295.

[7] Pierre Bourque and Richard E. Fairley (Eds.). 2014. *SWEBOK: Guide to the Software Engineering Body of Knowledge* (version 3.0 ed.). IEEE Computer Society, Los Alamitos, CA. http://www.swebok.org/

[8] JN Buxton. 1990. Software engineering—20 years on and 20 years back. *Journal of Systems and Software* 13, 3 (1990), 153–155.

[9] Bill Curran. 2001. What is Software Engineering? *Ubiquity* 2001, October, Article 5 (Oct. 2001). https://doi.org/10.1145/501305.763745

[10] Alexander Dekhtyar, Jane Huffman Hayes, and Tim Menzies. 2004. Text is software too. In *MSR 2004: International Workshop on Mining Software Repositories at ICSE'04: Edinburgh, Scotland*. 22.

[11] David Fifield. 2019. A better zip bomb. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/fifield

[12] Risto Hilpinen. 1992. On artifacts and works of art 1. *Theoria* 58, 1 (1992), 58–82.

[13] IEEE 828-2012 2012. *IEEE Standard for Configuration Management in Systems and Software Engineering*. Standard. IEEE Standards Association. https://ieeexplore.ieee.org/document/6170935

[14] Nurbay Irmak. 2012. Software is an Abstract Artifact. *Grazer Philosophische Studien* 86, 1 (2012), 55–72. https://doi.org/10.1163/9789401209182_005

[15] ISO/IEC 19506:2012 2012. *Information technology – Object Management Group Architecture-Driven Modernization (ADM) – Knowledge Discovery Meta-Model (KDM)*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:19506:ed-1:v1:en

[16] ISO/IEC 19770-1:2017 2015. *Information technology – IT asset management – Part 1: IT asset management systems–Requirements*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:19770:-1:ed-3:v1:en

[17] ISO/IEC 19770-5:2015 2015. *Information technology–IT asset management–Overview and vocabulary*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:19770:-5:ed-2:v1:en

[18] ISO/IEC 2382:2015 2015. *Information technology – Vocabulary*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en

[19] ISO/IEC 26513:2017 2017. *Systems and software engineering–Requirements for testers and reviewers of information for users*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:26513:ed-1:v1:en

[20] ISO/IEC 26514:2008 2008. *Systems and software engineering–requirements for designers and developers of user documentation*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:iso-iec:26514:ed-1:v1:en

[21] ISO/IEC/IEEE 24765:2017 2017. *Systems and software engineering – Vocabulary*. Standard. International Organization for Standardization, Geneva, CH. https://www.iso.org/obp/ui/#iso:std:71952:en

[22] Yuzhan Ma, Sarah Fakhoury, Michael Christensen, Venera Arnaoudova, Waleed Zogaan, and Mehdi Mirakhorli. 2018. Automatic classification of software artifacts in open-source applications. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 414–425.

[23] Peter Naur. 1968. Software Engineering-Report on a Conference Sponsored by the NATO Science Committee Garimisch, Germany. *http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF* (1968). https://ci.nii.ac.jp/naid/10021847939/en/

[24] Leon J Osterweil. 2008. What is software? *Automated software engineering* 15, 3-4 (2008), 261–273.

[25] Leon J Osterweil. 2013. What Is Software? The Role of Empirical Methods in Answering the Question. In *Perspectives on the Future of Software Engineering*. Springer, 237–254.

[26] F R. Shapiro. 2000. Origin of the Term Software: Evidence from the JSTOR electronic journal archive. *IEEE Annals of The History of Computing - ANNALS* (01 2000).

[27] Gregorio Robles, Jesus M Gonzalez-Barahona, and Juan Julian Merelo. 2006. Beyond source code: the importance of other artifacts in software development (a case study). *Journal of Systems and Software* 79, 9 (2006), 1233–1248.

[28] SEVOCAB. 2019. Software and Systems Engineering Vocabulary. (2019). https://pascal.computer.org Accessed: 2019-09-15, with search term *software*.

[29] Ian Sommerville. 2010. *Software Engineering* (9th ed.). Addison-Wesley Publishing Company, USA.

[30] Angus Stevenson. 2010. *Oxford dictionary of English*. Oxford University Press, USA.

[31] Peter Suber. 1988. What is software? *The Journal of Speculative Philosophy* (1988), 89–119. https://dash.harvard.edu/handle/1/3715472

[32] John W Tukey. 1958. The teaching of concrete mathematics. *The American Mathematical Monthly* 65, 1 (1958), 1–9.

[33] Xiaowei Wang, Nicola Guarino, Giancarlo Guizzardi, and John Mylopoulos. 2014. Towards an Ontology of Software: a Requirements Engineering Perspective.. In *FOIS*. 317–329.