Cloning beyond source code : a study of the practices in API documentation and infrastructure as code.

Mohamed Ameziane Oumaziz

ACADÉMIE DE BORDEAUX
UNIVERSITÉ **D**E **B**ORDEAUX
Sciences et **T**echnologies

THÈSE

Présentée au Laboratoire Bordelais de Recherche en Informatique pour obtenir le grade de Docteur de l'Université de Bordeaux

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Mathématiques et Informatique**

Cloning beyond source code: a study of the practices in API documentation and infrastructure as code

par

Mohamed Ameziane OUMAZIZ

Soutenue le 27 Janvier 2020, devant le jury composé de :

Président du jury
Jean-Philippe DOMENGER, Professeur ..... Université de Bordeaux, France

Directeur de thèse
Jean-Rémy FALLERI, Maître de conférences, HDR ..... Bordeaux INP, France

Co-Directeur de thèse
Xavier BLANC, Professeur ..... Université de Bordeaux, France

Rapporteurs
Mireille BLAY-FORNARINO, Professeur ..... Université de Nice, France
Tom MENS, Professeur ..... Université de Mons, Belgique

Examinateurs
Anne ETIEN, Maître de conférences, HDR ..... Université Lille 1, France
Tewfik ZIADI, Maître de conférences, HDR ..... Sorbonne Université, France

Abstract

When developing software, maintenance and evolution represents an important part of the development life-cycle, representing up to 80% of the overall cost and effort. During the maintenance effort, developers sometimes copy and paste source code fragments in order to reuse them. Such practice, seemingly harmless, is more frequent than we expect. Commonly referred to as "clones" in the literature, these source code duplicates are a well-known and well studied topic in software engineering.

In this thesis, we aim at shedding light on copy-paste practices on software artifacts beyond source code. In particular, we chose to focus our contributions on two specific types of software artifacts: API documentation and deployment files (i.e. Dockerfiles) as they respectively: help developers understand how to use and integrate external APIs, and represent the last step between developers and application users. For each software artifact, we follow a common empirical study methodology. As a result, we show that API documentations and software deployment files (i.e. Dockerfiles) contain duplicates and that such duplicates are frequent from 27,69% (API documentation) to nearly 50% (Dockerfiles). We then identify the reasons behind the existence of such duplicates. Also, we perform a survey on experimented developers and find that they're aware of such duplicates, frequently face them. But still have a mixed opinion regarding them. Finally, we show that both types of software artifacts are lacking tools with reuse mechanisms to cope with duplicates, and that some developers even resort to ad-hoc tools to manage them.

Keywords: *Documentation, duplicates, Dockerfile, Docker, reuse*

Résumé

Lors du développement de logiciels, la maintenance et l'évolution constituent une partie importante du cycle de vie du développement représentant 80% du coût et des efforts globaux. Pendant l'effort de maintenance, les développeurs copient et collent parfois des fragments de code source afin de les réutiliser. Une telle pratique, apparemment inoffensive, est plus fréquente qu'on ne le pense. Communément appelés "clones" dans la littérature, ces doublons de code source sont un sujet bien connu et étudié en génie logiciel.

Dans cette thèse, nous visons à mettre en lumière les pratiques du copier-coller sur les artefacts logiciels au-delà du code. En particulier, nous avons choisi de concentrer nos contributions sur deux types d'artefacts logiciels : Documentation d'API et fichiers de déploiement (c.-à-d. Dockerfiles) puisqu'ils respectivement : aident les développeurs à comprendre comment utiliser et intégrer des API externes, et représentent la dernière étape entre les développeurs et les utilisateurs d'une application. Pour chaque artefact logiciel, nous suivons une méthodologie d'étude empirique commune. Comme résultats, nous montrons que les documentations d'API et les fichiers de déploiement de logiciels (c.-à-d. Dockerfiles) sont confrontés aux doublons et que de tels doublons sont fréquents allant de 27,69% (documentation API) à presque 50% (Dockerfiles). Par la suite, nous identifions les raisons derrière l'existence de ces doublons. Aussi, nous effectuons une enquête auprès de développeurs expérimentés et de constatons qu'ils sont conscients de ces doublons, et qu'ils les rencontrent souvent tout en ayant un avis mitigé sur eux. Enfin, nous montrons que les deux artefacts logiciels manquent de mécanismes de réutilisation pour faire face aux doublons, et que certains développeurs ont même recours à des outils ad-hoc pour les gérer.

Mots clés : *Documentation, duplication, Dockerfile, Docker, reutilisation*

LaBRI — 351, Cours de la Libération — 33400 Talence — France

Contents

CHAPTER

1 Introduction

Through this chapter, we introduce the context, motivation and contributions of this thesis. This thesis aims at empirically studying duplicates among two types of software artifacts. We first start by taking a closer look towards duplicates in API documentation for two programming languages: Java and Ruby. We then tackle another duplicates issue plaguing Dockerfiles. in this chapter, we describe these two underlying challenges and our main contributions.

Contents

1.1 Context

When developing a software, maintenance and evolution represents an important part of the development's life-cycle, making up to 80% of the overall cost and effort [Alkhatib, 1992]. During the development or maintenance phase, it happens that developers resort to copying and pasting source code fragments. Commonly referred to as "clones" in the literature, these source code duplicates are a well-known and deeply studied topic in software engineering. Multiple empirical studies [Baker, 1995; Baxter et al., 1998; Rieger et al., 2004; Zibran et al., 2011] have shown that large softwares can have from 5% up to 20% of their code base that is cloned.

During the last two decades, many studies have tried to understand the underlying reasons to resort to code clones [Baxter et al., 1998; Kapser, 2009; Rieger, 2005; Cordy, 2003]. For instance, Cordi [Cordy, 2003] states that in the financial industry, cloning is a common reuse strategy. He explains that since financial products aren't that much different from each others, when a new product has to be developed, developers start by cloning a similar existing project and then adapt it to produce the new product. Developers have to resort to such practices because monetary risks are very high in case of errors. Thus, any financial tool has to be heavily tested (70% of costs in financial softwares are spent on testing [Cordy, 2003]). Therefore, it is cheaper and quicker to simply reuse existing source code by cloning it. Baxter et al. [Baxter et al., 1998] also writes about what is called accidental clones, that are produced out of the awareness of the developer. Such clones may happen during the use of libraries for instance, where developers have to write repetitive boilerplate code such as a sequence of calls to perform a task.

While some studies discuss the possible positive impact of code clones [Kapser and Godfrey, 2006], others state that code cloning isn't inconsequential [Juergens et al., 2009b]. Whether code clones are positive or not, during the maintenance process, if there is a bug in a cloned code fragment, developers have to propagate a bug-fix across all cloned fragments, this process might increase the maintenance costs. Moreover, developers might not all be aware that the bug exists somewhere else in his code base. Therefore, developers are required to know if their code is being cloned, and if so, where the other cloned fragments are located.

During the last decade, multiple code clone detectors have been released. Clone detectors are used to identify similar or identical code snippets in a code base. Existing clone detectors use a plethora of techniques such as AST-based (Abstract Syntax Trees) clone detection [Baxter et al., 1998], token-based code clone detection [Kamiya et al., 2002] or even deep learning techniques [White et al., 2016a].

While the software engineering community was very interested in code clones during the last two decades, there has been little interest in the existence of clones in software artifacts beyond source code. During a software system's development process, several by-products called software artifacts are also produced. These artifacts can be of different types: API documentation, design diagrams, requirement specifications, build configura-

tion files, deployment files, etc. Non-code software artifacts can have a great impact on the final product and thus, play a key role during the development process. Studies have shown that it is also common to have clones in non-code software artifacts. For instance, multiple studies [Liu et al., 2006b; Störrle, 2013] have shown that UML diagrams are also facing duplicates exactly like source code. Juergens et al. [Juergens et al., 2010; Domann et al., 2009] show that requirement specifications can also face such duplicates. McIntosh et al. [McIntosh et al., 2014] show that build configuration files are also facing clone issues.

Among all non-code software artifacts, we chose in this thesis to focus on studying the existence of clones in: API documentation and deployment files (i.e. Dockerfiles). We chose these two specific non-code software artifacts as they're becoming widely available in open-source projects. Further, the availability of these artifacts in open-source projects makes them a perfect fit for us to perform our empirical research studies. Finally, to the best of our knowledge, no research study has studied clones on any of these two types of artifacts in the past.

In particular, we take a closer look at API documentations since it helps developers understand how to use and therefore integrate an external API they don't know into their code. We think that having clones on an API documentation could lead to inconsistencies that can mislead developers and maintainers, making them not correctly understand the behavior of an API, thus extending the cost and effort of software development.

Also, we chose to take a closer look at Dockerfiles which are a proprietary type of deployment files. Dockerfiles are used to package an application with all its dependencies into a single package that can then be easily released. These packages represent the last step between developers and application users. Dockerfiles are written as a simple text file composed of a sequence of instruction written a Domain Specific Language (DSL). We think that as Dockerfiles are similar to source code, if there is a bug in an a cloned instruction, the developer has to propagate a bug-fix across all cloned instructions, making it necessary for developers to be aware of the existence of clones on the first hand.

Therefore, we believe that the community could benefit from these two empirical research studies given the importance of such artifacts.

1.2 Problem statement

Through this thesis we aim at studying the copy-paste maintenance burden in software artifacts beyond code. More precisely, the contributions discussed in this work are towards two specific types of artifacts: API documentation and Dockerfiles (i.e. a type of deployment files). Throughout this section, we chose to present only one extract that we gathered from an API documentation. While we could have indifferently chosen any of the two types of artifacts we're studying, we chose to present on only one of them in order to ease the understanding of our problem statement.

Figure 1.1 showcases a real API documentation extract from the Apache Commons Collection project written in Java. Both methods have an API documentation that is written in a domain-specific language called *JavaDoc*. In this extract, we do notice that a large part of both API documentations is duplicated (highlighted in red).

```
1  /**
2  * @param a the first collection, must not be null
3  * @param b the second collection, must not be null
4  * @return true iff the collections contain the same elements with the
       same cardinalities.
5  */
6  public static boolean isEqualCollection(final Collection a,
       final Collection b) {
7    ...
8    return true;
9  }
```

```
1  /**
2  * @param a the first collection, must not be null
3  * @param b the second collection, must not be null
4  * @param equator the Equator used for testing equality
5  * @return true iff the collections contain the same elements with the
       same cardinalities.
6  */
7  public static  boolean isEqualCollection(final Collection
       a, final Collection b, final Equator equator) {
8    ...
9    return isEqualCollection(collect(a, transformer),
         collect(b, transformer));
10 }
```

Figure 1.1 – Extract of a documentation duplication due to method delegation (in the Apache Commons Collection project). Duplicated documentation is highlighted in red.

We therefore wonder if we've just been lucky encountering it and that such scenarios are rare or if it's actually frequent and that developers often resort to perform copy-pastes. This questioning leads us to the first research question that we investigate in this thesis:

— RQ1: Do developers often resort to copy-pastes?

Then, when looking closer at this extract, we do notice that the bottom method's returned value is computed by calling the upper method (i.e. delegation). When looking even closer, we do notice that both methods have common input parameters and return types.

This is because the bottom is passing its input parameters (a and b) to the upper method and using the returned value from the upper method as its return value. Since both methods are sharing common input parameters (a and b), this causes them to also share their corresponding documentation which is therefore duplicated as we can see in Figure 1.1 (highlighted in red). In this scenario, the API documentation is being duplicated because of a method delegation, we do wonder if there are other underlying reasons that could cause developers to resort to copy-pasting. This leads us to the second research question that we investigate in this thesis:

— RQ2: Why do developers resort to perform copy-pastes?

Finally, the JavaDoc tool provides only a single reuse mechanism called *@InheritDoc*. This reuse mechanism lets developers reuse documentation across methods, however, this mechanism is limited to only overriding methods. Since in Figure 1.1 we only have a delegation scenario, the mechanism isn't applicable. We therefore wonder if there aren't other reuse mechanisms provided by other documentation tools, that can help developers avoid such copy-pastes. This constitutes the reasoning behind the last research question that we investigate in this thesis:

— RQ3: Could copy-pastes be avoided by a proper usage of state-of-the-art approaches?

Through our three research questions, we aim at making developers and researchers more aware about the existence of duplicates in their non-code software artifacts. We also aim at making the research community more aware of the reasons that can lead developers to resort to copy-pasting. Finally, we want to identify the different types of reuse mechanisms that could help tool providers better serve their community for avoiding duplicates. Therefore, in this thesis, for both API documentation and Dockerfiles software artifacts, we answer the three following research questions:

— RQ1: Do developers often resort to copy-pastes?

— RQ2: Why do developers resort to perform copy-pastes?

— RQ3: Could copy-pastes be avoided by a proper usage of state-of-the-art approaches?

1.3 Methodology

In this thesis, we perform two empirical research studies. For each study, we rely on two sources of knowledge: (1) a set of Github [1] repositories corresponding to open-source projects that will be analysed in our study, and (2) a set of responses to a survey gathering opinions of experts regarding duplicates in the studied artifact. We manually build the survey and send it to developer experts that we contact online through e-mails or social

1. https://github.com

medias (e.g. LinkedIn, Reddit, Github). Relying on these two sources of knowledge, we follow a triangulation-based approach to cross-validate our results [Seaman, 1999; Wood et al., 1999; Miller, 2008; Bratthall and Jørgensen, 2002]. Through this approach, we aim at increasing our confidence in the gathered results as they're validated by two different knowledge sources. This approach also reduces our overall bias as our sources of knowledge have each separate sets of biases which aren't overlapping.

In order to answer our first research question: *RQ1: Do developers often resort to copy-pastes?*, we start by building a tool that automatically parses all github repositories in our data collection (relying on the Diggit tool [2]) and identifies all copy-pastes in the repository's artifacts. We then apply our tool and identify all duplicates which are showcased in a web application to ease their manual analysis. We then cross-validate these results with the responses we gathered from our survey.

Then, to answer our second research question: *RQ2: Why do developers resort to perform copy-pastes?*, we take a random sample set from all identified duplicates and ask three experts to manually analyse it using a web application we've developed for duplicates analysis. To perform this analysis we ask our experts to look for the underlying reason explaining why the duplicate actually exists and tag each duplicate accordingly. We do not allow the experts to discuss between them during the tagging process. Also, if two experts do not agree on the underlying reason behind a duplicate, the third expert has to decide which one is the most plausible.

Finally, to answer our final research question: *RQ3: Could copy-pastes be avoided by a proper usage of state-of-the-art approaches?*, we start by constructing a list of all available tools for the type of artifact we're studying. Then, we manually look at every tool that is in our list and read all user-guides looking for all reuse mechanisms they provide. Then, for every underlying reason we've identified in RQ2, we look if there is a reuse mechanism that could be used to avoid it. We then cross-validate these results with the responses we gathered from our survey.

1.4 Contributions

Through our work, we were aiming on duplicates in artifacts other than the source code. We chose to focus on two main artifacts: API documentation and Dockerfiles. For both artifacts, the main contributions can be summed-up as follows:

— We show that API documentations and software build files actually face duplicates issues and that such duplicates are frequent.

— We identify the reasons behind the existence of such duplicates.

— We find that they're aware of such duplicates and frequently face them.

2. https://github.com/jrfaller/diggit

— We show that both software artifacts lack reuse mechanisms to cope with duplicates, and that some developers even resort to ad-hoc tools to manage them.

These contributions led us to the publication of two research papers:

— Documentation Reuse: Hot or Not? An Empirical Study, at the 16th International Conference on Software Reuse (ICSR 2017) [Oumaziz et al., 2017].

— Handling duplicates in Dockerfiles families: Learning from experts, at the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019) [Oumaziz et al., 2019].

1.5 Thesis outline

The remainder of this thesis is organized as follows. We first present in Chapter 2 an overview of the state of the art in the field of duplicates in software engineering. Then, in Chapter 3, we present an empirical study on duplicates in API documentation for two programming languages: Java and Ruby. In Chapter 4, we present an empirical study on duplicates in Dockerfiles. Finally, in Chapter 5, we conclude this document by summarizing the contributions and the main perspectives.

CHAPTER

2 Background

In this chapter, we start by setting some context surrounding this thesis. We first present the concept of clones and its various definitions in the literature. We then present a taxonomy of existing clone detection approaches with well-known tools implementing them. Then, we present the different empirical studies on clones in all types of artifacts. Finally, we showcase the different techniques that are used to manage clones in software engineering projects.

Contents

2.1 Context

In this section, we define the context surrounding this thesis. We present the concept of clones and their multiple definitions. We then present the different existing approaches used to detect clones. All definitions and detection approaches discussed in this section are gathered and summarised from a research work performed by Roy et al. [Roy et al., 2009].

2.1.1 Clone definitions

When looking at the literature, a large set of studies looked at clones in source code, however, there's actually no single definition for code clones. Every new code clone detection tool article defines what they've detected as code clones with their own definitions. In 1998, Baxter et al. [Baxter et al., 1998] consider clones as a group of code fragments that are almost identical based on a similarity threshold computed upon a group of parameters. In 2002, Kamiya et al. [Kamiya et al., 2002] define code clones as fragments of code that can be either perfect duplicates (i.e. copy-pastes) or as similar fragments of code. However, they don't formally define what they mean by similar clones as done previously by Baxter et al. [Baxter et al., 1998]. Another vague definition is proposed by Burd et al. [Burd and Bailey, 2002] who consider a code fragment as being cloned (i.e. a code clone) if there's at least another occurrence in the source of the same code fragment even if it has some minor modifications.

In order to avoid all this vagueness in code clone definitions, some researchers tried to classify all these code clone definitions in form of taxonomies. Mayrand et al. [Mayrand et al., 1996] suggested a scale of eight different categories of clones. For instance, a category called *DistinctName* was composed of cloned code fragments that had only their identifier names that were different. However, all the categories that were proposed aren't sill precise in their definitions. For instance, the category *SimilarExpression* was composed of cloned code fragments that had expressions that were different but still "similar" enough to be considered as clones. Balazinska et al. [Balazinska et al., 1999], proposed another taxonomy composed of 18 different categories. However, it still lacked precision as it has categories "One long difference", "Two long differences" and "Several long differences" which introduces some vagueness in their classification.

All of this showcases how hard it is to formally define what is a code clone. Nowadays, researchers finally agree upon four types of code clones [Bellon et al., 2007a]. They chose to define code clones based on four clone categories called types. The first three types focuses on clones sharing a similarity in their source code. The fourth clone type focuses on clones sharing a similarity in their functionalities without having a similarity in their source code. All four clone types are defined as follows:

```
1  public Player winner(Player player1, Player player2) {
2    if(player1.score() > player2.score()) {
3      return player1; // Comment1
4    } else {
5      return player2; // Comment2
6    }
7  }
```

```
1  public Player winner(Player player1, Player player2) {
2    if( player1.score() > player2.score() )
3    {
4      return player1; // Comment1'
5    }
6    else
7    {
8      return player2; // Comment2'
9    }
10 }
```

Figure 2.1 – Example of a Type-I clone.

Type-I clones

Definition 2.1: Type-I clones
Identical code fragments except for variations in whitespace, layout and comments.

Figure 2.1 showcases un example of a Type-I clone. In this example, the *winner* method receives two input parameters which are *player1* and *player2* and returns as output the player having the highest score. In case of a draw, *player2* is considered as a winner. In this example, we can see that both fragments have the exact same code, except with regard to formatting. While the bottom code fragment uses more lines of code, if we remove all the white-spaces, comments, tabulations and new lines, we can see that both fragments are identical.

Type-II clones

Definition 2.2: Type-II clones
Syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments.

```
1  public Player winner(Player player1, Player player2) {
2    if(player1.score() > player2.score()) {
3      return player1; // Comment1
4    } else {
5      return player2; // Comment2
6    }
7  }
```

```
1  public Player winner(Player computer, Player host) {
2    if(computer.score() > host.score())
3    {
4      return computer; // Comment1'
5    }
6    else
7    {
8      return host; // Comment2'
9    }
10 }
```

Figure 2.2 – Example of a Type-II clone.

Figure 2.2 presents an example of a Type-II clone. Apart from the small differences in their formatting as we've previously seen in Figure 2.1 which is a characteristic of Type-I clones, Type-II clones also have small differences in variable identifiers, types or literals. These differences can be seen in our example where in the upper fragment, the *winner* method has two parameters which are *player1* and *player2*, while the *winner* method in the bottom fragment has *computer* and *host* as parameters. Still, these two fragments are considered as a clone as they are syntactically similar.

Type-III clones

Definition 2.3: Type-III clones
Copied fragments with further modifications such as changed, added or removed statements, in addition to variations in identifiers, literals, types, whitespace, layout and comments.

Figure 2.3 presents an example of a Type-III clone. In this type of clones, in addition to the characteristics of type-II clones, new code statements are added, modified or removed. For instance, in our example, if we look at the bottom fragment, we can see that in addition to the changes we've previously seen, there's a new code statement. The new code state-

```
1  public Player winner(Player player1, Player player2) {
2    if(player1.score() > player2.score()) {
3      return player1; // Comment1
4    } else {
5      return player2; // Comment2
6    }
7  }
```

```
1  public Player winner(Player computer, Player host) {
2    host.setScore(host.score() + DIFFICULTY_POINTS); // Bonus
       points
3
4    if(computer.score() > host.score())
5    {
6      return computer; // Comment1'
7    }
8    else
9    {
10     return host; // Comment2'
11   }
12 }
```

Figure 2.3 – Example of a Type-III clone.

ment lets the host have more points with bonus points that are added based on the game's difficulty. Both fragments in Figure 2.3 are considered as clones as they're almost similar.

Type-IV clones

Definition 2.4: Type-IV clones
Two or more code fragments that perform the same computation but are implemented by different syntactic variants.

Figure 2.4 depicts an example of a Type-IV clone. In this example, we can clearly see that the two code fragments aren't syntactically similar with the upper fragment using a classical if-else block statement for its condition while the bottom fragment uses a ternary operator (?:) as means to express the same condition as in the upper fragment. However, while they are written differently, both fragments are performing the same computation, and thus are considered as a clone.

```
1 public Player winner(Player player1, Player player2) {
2   if(player1.score() > player2.score()) {
3     return player1; // Comment1
4   } else {
5     return player2; // Comment2
6   }
7 }
```

```
1 public Player winner(Player computer, Player host) {
2   return (computer.score() > host.score())?computer:host;
3 }
```

Figure 2.4 – Example of a Type-IV clone.

Summary

Throughout this section, we've seen that there isn't a clear definition of code clones. However, a consensus has emerged categorizing clones into four main types: Type-I, Type-II, Type-III and Type-IV. In this thesis, we chose as an initial step to focus our research on Type-I duplicates which are Identical code fragments (e.g. copy-pastes) except for variations in whitespace, layout and comments. We chose this type as a mean to have a higher trust level regarding our results as shown by Charpentier et al. [Charpentier et al., 2015].

2.1.2 Clone detection approaches

There are multiple clone detection approaches available in the literature. Roy and Cordy [Roy and Cordy, 2007] tried to classify these approaches into four main categories: text-based approaches, lexical-based approaches, syntactic-based approaches and semantic-based approaches. In the following sub-sections, we present each approach.

Text-based approaches

In this approach, the source code is considered as a sequence of text lines. In order to compare two code fragments, we simply compare them line by line and identify common sequences. If a common sequence is identified, the tool implementing this approach will return it as a clone pair. Figure 2.5 showcases an example of a code clone identified through the use of a text-based detection technique.

In order to apply a text-based detection technique on source code, one has to first apply some transformations and normalisations to the source code in order to have better

```
1  public Player winner(Player player1, Player player2) {
2   if(player1.score() > player2.score()) {
3    return player1;  // Comment1
4   } else {
5    return player2;  // Comment2
6   }
7  }
```

```
1  public Player winner(Player player1, Player player2) {
2   if( player1.score() > player2.score() )
3   {
4    return player1;  // Comment1'
5   }
6   else
7   {
8    return player2;  // Comment2'
9   }
10 }
```

Figure 2.5 – Example of a code clone identified with a Text-based approach.

results. These transformations are necessary as that when performing a line by line comparison even a simple double whitespace rather than a single whitespace could mislead the detection process. Therefore, applied transformations are usually: code comment removal, whitespace normalisation (removing multiple whitespaces, tabs and new lines) and some normalisation. Figure 2.6 showcases an example of code transformation.

In 1993, Johnson [Johnson, 1993] proposed a redundancy detection mechanism based on fingerprints corresponding to source code substrings. In this technique, the source code is split into substrings where each substring has a corresponding signature. These signatures are computed directly from the substring using Karp-Rabin fingerprinting algorithm [Karp and Rabin, 1987]. The substrings are extracted from the source code, this extraction starts with substrings from the largest length to the smallest length (i.e. one character). They then apply this text-based detection tool to the GNU C Compiler project and identify several large matches (more than 50 lines of code) which are considered as being the result of copy-paste operations (Type-I clones).

Later, in 1999, Ducasse et al. [Ducasse et al., 1999] developed a language independent technique for detecting duplicated code without using any parser through the use of line-based string matching. They considered the source code a sequence of lines, they transformed each line by removing whitespaces and comment lines. Based on these transformed lines, they then hashed each line using a hash function in order to speed up the

```
1  /**
2  * Some code comments
3  */
4
5  public Player winner(Player player1, Player player2) {
6   if( player1.score() > player2.score() )
7   {
8    return player1; // Comment1'
9   }
10  else
11  {
12   return player2; // Comment2'
13  }
14 }
```

```
1  public Player winner(Player player1, Player player2){
2   if(player1.score() > player2.score()){return
       player1;}else{return player2;}
3  }
```

Figure 2.6 – Example of code transformation.

line by line comparison process. They then used a string-based pattern matching algorithm which outputs the line numbers of clone pairs.

Token-based approaches

Token-based approaches are similar to text-based approaches. While for text-based approaches we consider the source code as a sequence of text lines, in the Token-based approaches we consider the source code as a sequence of tokens, where tokens are the result of the tokenization step at the beginning of a parsing process. Therefore, when applying a token-based duplicate detection technique, we compare sequences of tokens and not sentences. However, when a duplicate is identified based on the sequence of tokens, the returned result is actually the initial sentence. Similarly to text-based approaches, token-based approaches identifies the same types of clones. However, token-based approaches are more robust to code changes such as formatting and spacing.

One of the first Token-based detection tools was proposed in 1993 by Baker [Baker, 1993]. His tool called *dup* represents the code as a sequence of lines in order to detect clones. Therefore, it uses a lexer and a line-based string matching algorithm on the sequence of tokens extracted from each line. *Dup* also applies normalisations of each token

(replacing identifies of functions, variables and types with a special parameter) to help identify Type-II clones and hashes all token sequences to speed up the comparison process. The tool then extracts a set of pairs of longest duplicate matches using a suffix tree algorithm. *Dup* manages to identify Type-I and Type-2 clones.

CCFinder [Kamiya et al., 2002] is another token-based clone detection tool. Similarly to the *Dup* tool, *CCFinder* starts by splitting each source file into a single large sequence of tokens through the use of a lexer. In addition to a set of transformations based on specific rules, the tool replaces identifiers that are related to constants, types and variables with a special token. Through this process the tool aims at easing the duplicate identification as after this process fragments having identical code but different variable names can now be identified. Then, a suffix-tree based sub-string matching algorithm is used to identify all similar sub-sequences from our transformed large token sequence. As a result, sub-sequence pairs are returned as code clone pairs. Afterwards, a mapping is applied based on the cloned sub-sequence in order to find the corresponding original source code.

Another token-based clone detection tool is *CP-Miner* by Li et al. [Li et al., 2006]. Their tool uses a frequent sub-sequence mining technique to identify similar sequences of token statements. *CCFinder* and *dup* tools were actually vulnerable to code insertions and reordering because of their analysis which is sequential. Therefore, token sequences are broken if a new statement or a reordered statement is added while there could be a potential duplicate in them. *CP-Miner* overcomes all these limitations through the use of the frequent subsequence mining technique. The use of an extended version of CloSpan [Yan et al., 2003] lets *CP-Miner* tolerate one to two statement insertions, modifications or deletions in duplicate code.

Another well-known token-based clone detection tool is *SourcererCC* by Sajnani et al. [Sajnani et al., 2016]. *SourcererCC* uses an optimized index to scale to hundred of millions of lines of code. Their tool compares code fragments through the use of a bag-of-tokens strategy which can be seen as multiple sets of tokens that are extracted from each compared code fragment. Relying on such strategy makes their tool resilient to Type-III changes. It also uses some heuristics to speed up the comparison process such as a heuristic that is used to build an inverted index which maps tokens to the blocks of code containing them. They also show that their tool can scale to up to 250MLOC on a standard workstation.

Hummel et al. [Hummel et al., 2010] proposed a new token-based clone detection technique that they called index-based code clone detection as their technique also relies on indexes. Their technique is composed of three main steps. The first step consists of reading the source code and splitting into a set of tokens through the use of a lexer. In this step, a normalization is also performed on each token in order to remove comment or variable names. All tokens are then regrouped into statements which are the output of this first step. Secondly, a clone index is built, which contains information for each sequence such as: filename, a hash of the sequence and the position of the original code for the statement. Then, each statement is split into chunks of all possible sizes and added to the clone

index. All chunks' hashes are then compared to each other starting with the larger chunks to smaller ones if the largers didn't match. As a last step, all identified duplicates are then mapped to their original source code through the use of the clone index and returned as a result.

Tree-based approaches

In this approach, the source code is converted into a parse tree or an abstract syntax tree (AST) through the use of a parser that has to be specific to the source code's language. Such parse tree or AST contains all required information from the source code. However, functions, variable names and types extracted from the source code can be discarded in the tree representation. Such representation also helps to apply more advanced normalisation techniques that can be specific to either functions, variables or types. After the parse tree or AST is built and normalised, similar subtrees are then searched in the tree through the use of tree matching techniques. The initial source code corresponding to the matching subtrees is then returned as a clone pair.

Baxter et al.'s *CloneDR* [Baxter et al., 1998] AST-based clone detection tool is one of the pioneers in the AST-based clone detection approaches. The tool starts by generating an abstract syntax tree and compares all its sub-trees through a tree matching technique that uses metrics that are based on a hash function. The hash function helps the tool to detect clones from code fragments which had some of their statements that were reordered. Then, the source code corresponding to similar sub-trees is returned as clones.

A variant of *CloneDR* is *ccdiml* proposed by Bauhaus research project [Raza et al., 2006]. *Ccdiml* is a bit different as it avoids the use of the similarity metric, sequences handling and hashing. Also, contrary to *CloneDR*, *ccdiml* isn't able to work concurrently and check for consistent renaming. *Ccdiml* also represents the source code as an intermediate language rather than an AST during the comparison phase as *CloneDR* does.

Yang [Yang, 1991] proposes an approach to find syntactic differences between two versions of a same program through the generation of a variant parse tree for both program versions and then applies dynamic programming to identify similar sub-trees. Wahler et al. [Wahler et al., 2004] proposes a technique for finding exact and parameterized clones. To do so, they choose to further abstract the AST by converting it to an XML representation. They then apply the frequent item-set technique [Han et al., 2011] on the XML representation to identify clones.

Gitchell and Tran [Gitchell and Tran, 1999] propose a tree-based detection tool using parse trees. Their tool focuses on clone detection in C programs. It uses a lexical analyser to convert C source code to parse trees. The tool then views the two parse trees as simple strings and looks for the maximal common sub-sequence of tokens. The corresponding source code to the duplicate sub-sequence is then returned as a result.

Metrics-based approaches

Another approach to detect duplicates in source code is the use of a metric-based approach. In this approach, detection tools rely on metrics gathered from code fragments and compare vectors of such metrics rather than comparing the code directly. Several clone detection tools use this approach. They all start by computing a set of metrics through the use of fingerprinting functions. These fingerprinting functions are applied on classes, functions, methods or statements and the resulting vectors of metrics are then compared to identify clones. Generally, in order to compute the metrics, tools start by first converting the source code to an abstract syntax tree, and then apply fingerprinting functions on it.

[Mayrand et al., 1996] propose a tool that computes several metrics such as: number of lines of code, number of function calls, etc. for each function in the analysed program. The tool used an intermediate representation language to represent the source code. The tool defines clones as pairs of functions who have their whole body that has similar metrics.

[Kontogiannis et al., 1996] propose two ways to identify code clones. Both methods rely on an abstract syntax tree constructed from the source code. Once the tree is built, the tools use it to compute metrics that are later annotated in the AST's nodes. Then, the metrics are extracted from the AST to construct a reference table composed of source code entities sorted by their metrics. This reference is then used to identify clones. In the first method, metrics generated from blocks of instructions are directly compared to each other. They use the assumption that two code fragments are similar if their corresponding metrics are proximate. In the second method, they use a dynamic programming technique to compare blocks of instructions in a statement-by-statement basis. They compute the distance between blocks of instructions as being the smallest sequence of insert, edit and delete steps required to transform one block to the other one.

[Di Lucca et al., 2001] apply a metrics-based approach to identify similar HTML pages through the use of a distance function to compare items in web pages. Their tool first starts by computing a string representation for each HTML web page where each HTML element was replaced with a symbol taken from a set of alphabets. They then apply the Levenshtein distance on pairs of strings to identify similar web pages. [Lanubile and Mallardo, 2003] propose a semi-automated technique to detect function clones in source code. Their technique was a two-step process where first an automated process is applied to detect potential function clones. Then a manual visual inspection is applied to classify suspected clones.

PDG-based approaches

In a Program Dependency Graph based approach, the source code is transformed into a higher abstraction compared to the previous approaches since this approach takes into account the semantic information that is available in the source code, since the Program

Dependency Graph (PDG) contains control flow and data flow information from the source code. In such a graph, the nodes represent statements while vertices represent the dependencies between the data and control structures. Therefore, such a graph also carries semantic information about the analysed program that can be used to better identify clones. In this approach, an isomorphic sub-graph matching algorithm is applied to identify similar sub-graphs which are return as clones. While PDG-based techniques are robust to reordered statements, insertions and deletions of code they're however not scalable to large size programs.

A well known PDG-based clone detection tool is PDG-DUP by Komondoor et al. [Komondoor and Horwitz, 2001]. Their tool uses a program slicing technique to identify isomorphic PDG sub-graphs. Krinke [Krinke, 2001] proposes an iterative approach to identify maximal similar PDG sub-graphs. Another PDG-based tool called GPLAG [Liu et al., 2006a] was proposed for plagiarism detection. Gabel et al. [Gabel et al., 2008] proposed a PDG-based clone detection that is scalable. To do so, they propose a technique where they reduce the difficulty of graph similarity to a tree similarity comparison by mapping selected PDG sub-graphs to their related structured syntax.

Hybrid-based approaches

In this section, we discuss clone detection approaches that use hybrid approaches which could be classified in the categories we've seen earlier but are different enough to have their own category.

[Koschke et al., 2006] proposed a technique where the AST nodes are serialized in preorder traversal. Then, a suffix tree is created for these serialized AST nodes and their approach then compares the tokens of the AST nodes using a suffix tree-based algorithm to identify clones making their approach linear in time and space. Another clone detection technique was used for Microsoft's Phoenix Framework using abstract syntax trees and suffix trees that can find exact and parameterized clones [Tairas and Gray, 2006]. AST nodes were used to generate a suffix tree which can be performed in linear time and space as we've just seen [Koschke et al., 2006].

[Jiang et al., 2007] proposed a clone detection tool called *DECKARD* that is used to identify similar trees. They compute characteristic vectors to approximate the structural information within ASTs in the Euclidean space. They then apply a locality sensitive hashing to cluster similar vectors based on their Euclidean distance and therefore identify code clones.

[Cordy and Roy, 2011] proposed a hybrid scalable clone detection tool called *NiCad*. Their tool uses a mix between a string-based and metrics-based approach where their tool first starts by parsing the source code into a textual form which is then normalized. Later, they apply a longest common sub-sequence algorithm to compare the source code in a line-by-line basis and identify code clones.

Learning-based approaches

In this section, we discuss clone detection approaches that rely on neural networks and deep learning techniques and cannot be considered as hybrid approaches.

[Davey et al., 1995] proposed a neural network-based clone detection tool. Their tool started by converting the source code to vectors where two vectors corresponding to very similar fragments have to have heir distance in the Euclidean space as small as possible. Then, they provide their input vectors to an unsupervised neural network that is able to cluster the vectors and therefore determine to which class the pair of vectors belongs to (i.e. Type-1, Type-II or Type-III).

[White et al., 2016b] used a learning-based detection technique which relies on deep learning to detect code duplicates. They found that their approach detected duplicates that where sub-optimally detected by traditional techniques.

[Li et al., 2017a] proposed a token-based detection tool called *CCleaner* that leverages deep learning techniques to identify code clones. Their tool receives as input a pair of code fragments that are then split into token using the ANTLR lexer. Then, the tool computes a list of token frequency for each token. This frequency list and list of tokens will then be fed to an already trained deep-learning model to determine whether the two initial fragments are code clones or not.

Summary

As we've seen in this section, there isn't a single technique to identify code clones. Each technique has its advantages and drawbacks. Some tools try to mix techniques to have advantages from both worlds. In this thesis, we use a token-based detection approach on all discussed studies. This technique fits well with our research as both Javadoc/-Yard documentation and Dockerfiles are structured in a way that can easily be split into tokens. In particular, since we only focus on Type-I duplicates (i.e. perfect duplicates), we chose to use the index-based clone detection technique [Hummel et al., 2010] which is a token-based clone detection technique as we've described earlier.

2.2 Empirical research on clones in software artifacts beyond code

In this section, we present different empirical studies and tools regarding the existence of clones outside the source code, more particularly, clones in non-code software artifacts. We chose to split this section into four main subsections: requirement specifications, modeling languages, build files and other artifacts.

Requirement specifications

[Domann et al., 2009] perform an empirical study on requirement specification clones. In their study, they analysed 11 real-world requirement specifications with more than 2,500 pages. To do so, they chose to rely on an already existing text-based clone detection tool that they adapted for their needs. Through their study, they aim at answering three main research questions. First, they wanted to know if real-world requirement specifications are actually facing clones. After applying their tool they show that indeed requirement specifications are facing clones, however, projects face it at different degrees. While two projects didn't face any clones, other faced up to two thirds of duplicate content. However, they didn't find any correlation between the size of the requirement specification file and the number of clones. Secondly, they wanted to see if the identified clones weren't false positives. They therefore manually analyse identified clones and find an average precision of 87%. Finally, they apply a qualitative analysis to understand what kind of information are cloned in requirement specifications. They manually analyse a sample of detected specification clones and classify them. They identify three categories: long clones, clone groups of high cardinality (high occurrence), and other patterns.

[Juergens et al., 2010] also perform an empirical study on requirement specifications clones from a quality assurance standpoint. In their study, they analyse 28 requirement specifications with more than 8,667 pages. They choose to rely on an existing token-based clone detection tool called *CloneDetective* based on *ConQAT* tool [Juergens et al., 2009a] contrary to [Domann et al., 2009] with their text-based tool. In this study, they aim at answering at four main research questions. First, just as [Domann et al., 2009], they try to determine how much clones requirement specifications face in real-life. They find that clones are frequent in their corpus with clone pairs being more frequent than clone groups (a group of three identical duplicates or more). Secondly, they investigate which kind of information is cloned in requirement specifications. After a manual analysis, they identify 12 different categories, the most frequent ones being: Detailed steps, Reference, UI and Domain knowledge. Thirdly, they investigate the consequences for requirement specification cloning through three perspectives: (1) specification reading, (2) specification modification and (3) specification implementation. For instance, in (1) they find that cloning makes reading harder as the size of the specification file increases. Finally, they investigate whether clones can be correctly identified with existing clone detection tools. They identify cases where duplicates are falsely considered as clones and categorize them: Document meta-data, indexes, page decoration, open issues and specification template information.

Modeling Languages

[Liu et al., 2006b] propose a suffix-tree based approach to detect duplicates in Unified Modeling Language (UML) sequence diagram. Their tool automatically arranges all elements of the sequence diagram into an array. They then concatenate all generated

arrays for every diagram that is analysed into one large array that is turned into a suffix tree. They then apply an algorithm that detects common prefixes of suffixes to identify diagram clones. They later apply their approach on two industrial projects and find that both projects face diagram duplicates. They then provide the identified duplicates to the maintainers and find out that maintainers also consider them as duplicates and applied refactoring techniques to remove them.

[Antony et al., 2013; Alalfi et al., 2018] propose another automated approach to identify duplicates in UML sequence diagrams. To identify duplicates, they chose to use an already existing tool that we've discussed earlier called *NiCad* developed by [Cordy and Roy, 2011]. As *NiCad* better identifies duplicates when analysing languages with a nesting structure, in order to detect duplicates, they had to transform the original representation of sequence diagrams which is in XMI and didn't have a nesting structure. Then, as sequence diagrams can be large, in order to detect smaller clones, they chose to break sequences into smaller windows to identify smaller clones. They then apply their approach on the problem of identifying access-control security vulnerability in models of interaction with web applications. They then compare their tool with state-of-the-art model-checking tools and find an increased precision and recall.

[Störrle, 2013] also explores duplicates in UML models and implement algorithms and heuristics for detecting duplicates with the *MQlone* tool. They first start by proposing a definition for model clones. They chose a similar categorization to the generally accepted definition for code clones [Bellon et al., 2007a]. They define four types: Exact model clones (Type-A), Modifier model clones (Type-B), Renamed model clones (Type-C) and Semantic model clones (Type-D). Their tool *MQlone*, relies on a set of heuristics to identify similar diagrams. They chose to focus on UML nodes as they noticed that UML diagrams are loosely connected graphs of heavy nodes. They then applied their tool on 7,181 model elements gathered from a sample project. They used different heuristics configurations and identified the optimized configuration for their tool. They also find that their heuristics are enough fast to be practical on medium sized models. They later expect to perform more experiment on larger models to derive reliable conclusions.

[Nejati et al., 2007] proposed an approach for matching and merging UML statecharts models. Statecharts are a design and implementation language for specifying dynamic behavior of software systems. In order to match similar Statecharts, their tool uses a hybrid approach combining static matching and behavioral matching. For static matching, they use typographic and linguistic similarities between state names, and similarities between state depths in the model's hierarchy tree. For behavioral matching, they compute similarity between states using their behavioral semantics. They then evaluate their tool with statechart models describing different telecom features at AT&T, and find that their hybrid approach produces higher precision than relying only on static or behavioral matching. They however expect to perform more evaluations in the future to validate their tool's practicality.

[Deissenboeck et al., 2008a] propose and automated approach for the detection of clones in large control system models. Their approach relies on graph theory which makes it easily applicable to any graph-based models such as graphical data-flow languages such as in Figure 2.7 presenting an example of a PI-controller model. As the problem of finding the largest clone pair in a graph is NP-complete, their approach relies on a heuristic to find clones. They then perform a case study on their approach by applying it to models provided by MAN Nutzfahrzeuge Group. The provided model was composed of more than 20,000 blocks distributed over 71 files. They then identify 166 clones with a clone going up to having a 101 size (number of nodes).

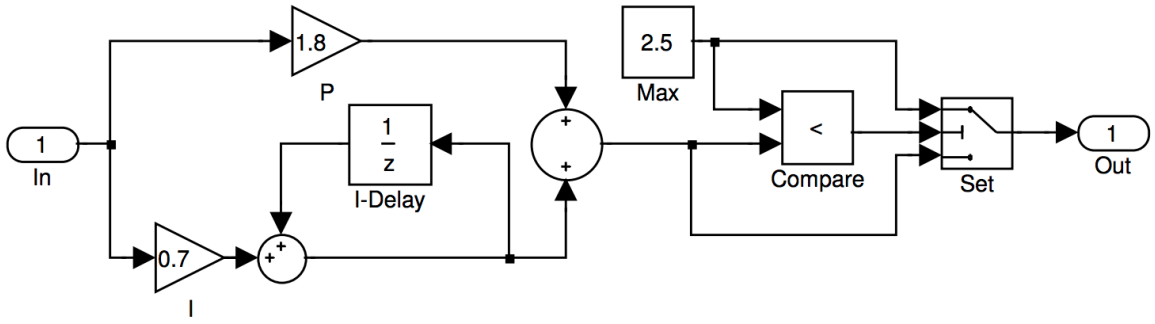

Figure 2.7 – Example of a PI-controller model gathered from [Deissenboeck et al., 2008a].

Build files

[McIntosh et al., 2014] performed an empirical analysis on 3,872 build systems looking for clones. They chose to rely on an existing line-based clone detection tool called *Conqat* [Deissenboeck et al., 2008b]. In this study, they perform a quantitative and qualitative empirical research. They first quantitatively answer three research questions, and then qualitatively answer two research questions. First, they investigate if clones in build systems are common. They find that while identified clones tend to be small, they're more frequent in Java build files with nearly 50% being cloned compared to other software artifacts. Secondly, they investigate if the technology that is used influences clones in build systems. They find that for Java projects, clones are more frequent if the project uses Maven rather than Ant. For C/C++ projects, clones are more frequent in build files when using CMake rather then autotools. Thirdly, they investigate if benchmark-derived thresholds vary depending on the build technology. They find that thresholds vary more for Java build systems with low amounts of clones, and Cmake/autotools and Maven/Ant with high amounts of clones. They then investigate their two qualitative research questions.

They first start with an investigation regarding the type of information that is cloned in build systems. They find that configuration details are usually more cloned in Maven and CMake build technologies. Finally, they investigate how build systems with a low number of clones manage to do it. They find that Java build clones could by avoided if the underlying XML representation would be exploited. For C/C++ projects, clones could be avoided by automatically duplicating templates during build.

[Sharma et al., 2016] perform an empirical study on Infrastructure as code (IaC) configuration files and propose 11 implementation and 13 design code smells. In this study, they analyse 4,621 Puppet projects with more than 142,662 Puppet files and investigate four research questions. One of the smells they investigate is called *Duplicate Block* which corresponds to modules having more than 150 tokens that are duplicated. In order to identify duplicates in their corpus, they relied on the PMD-CPD [1] token-based clone detection tool. They find out that up to 25% of their analysed projects have code that is duplicated and that developers of Puppet projects either don't duplicate code at all or duplicate it in a massive scale.

Web services

[Martin and Cordy, 2011] proposed a new method for detecting clones in Web Service Description Language (WSDL) leveraging existing clone detection tools. In their approach, they first start by introducing a new idea of *contextual clones* which are clones which have their original fragment that's being analysed augmented with related information to add context to it. To perform their analysis, they leveraged the hybrid-based clone detection tool called *NiCad*. They then applied their tool on more than 700 WSDL descriptions and found that using contextualized fragments to identify clones reduces the number of detected clones, but identified clones are less likely to be false-positive. They even noticed that with contextualized fragments, their tool managed to identify clones that weren't flagged as clones when not using context information.

[Rajapakse and Jarzabek, 2007] performed an empirical study on clones in web applications. They chose for their study to focus on a single web application that they built themselves based on requirements from one of their industry partners. They then perform three implementation iterations where each time they try to further reduce the number of clones. They did this in order to quantitatively and qualitatively analyse the trade-offs that web developers would have to take to reduce the number of clones in their web applications. Through this process, they managed to reduce the code size of their web application by 78%. To do so, they used the well-known *CCFinder* token-based clone detection tool to identify duplicates in PHP templating scripts. Once clones were identified and selected for editing, they then applied a combination of strategies to reduce them. They used three strategies as follows: applying design patterns, applying known refactoring techniques and

1. https://pmd.github.io/latest/pmd_userdocs_cpd.html

context-specific restructurings. Through their study, they showed that the trade-offs developers have to face in order to reduce the number of duplicates actually reduces their ability to remove duplicates and thus shed some light on why clones persist in software.

[Di Lucca et al., 2001] propose an approach for detecting clones in HTML pages which relies on existing clone detection approaches. Their tool identifies duplicate HTML fragments by computing a Levenstein distance similarity metric between fragments. They then perform a case study of their tool on an industry Web application composed of 201 HTML files and identify 46 couples of perfect clones.

[Charpentier et al., 2016] propose an approach for detecting clones in Cascading Style Sheets (CSS) files and automatically remove them. Their tool relies on Formal Concept Analysis techniques to identify duplicate CSS code and extract mixins. They then perform an experiment on 108 CSS files extracted from real projects and find that their tool has good performance results. They later perform conduct several case studies and demonstrate that their tool helped developers identify and avoid duplicates through the automatic use of mixins.

Other artifacts

In this subsection, we discuss literature research on clones on a larger set of artifacts related to software such as: cloned bug reports on tracking systems, cloned questions on developer forums, clones on binary executables and spreadsheets.

[Sun et al., 2011] propose a retrieval function to measure the similarity between bug reports using all available information in the bug tracking system such as: the summary, description, product, component, version, etc. To do so, they extend the BM25F similarity model [Pérez-Agüera et al., 2010] which is known in the information retrieval community in order to better fit the duplicate bug report detection. They then apply their tool on four different bug tracking systems and find that their tool is able to better identify duplicate bug reports than existing tools (i.e. [Sureka and Jalote, 2010; Sun et al., 2010]).

Similar to [Sun et al., 2011] research work, another duplicate bug report approach was proposed by [Zhou and Zhang, 2012]. In their research, they propose an tool called *BugSim* based on learning to rank techniques. They consider existing bug reports as documents to be searched and new bug reports as a query to search an existing document. They then rank the query results to identify if the new bug report isn't actually a duplicate. To do so, they identify 9 features based on bug summary and bug description just as what we've previously seen. Finally, they evaluate their model on more than 45,000 real bug reports and show that their tool outperforms previous state-of-the-art approaches including [Sun et al., 2011] approach.

[Li et al., 2017b] propose a tool to automatically identify similar pull-requests in Github [2]. In their work, they rely on Natural Language Processing (NLP) techniques (i.e.

2. https://github.com/

Cosine similarity) to identify textual similarity between two pull-requests. To do so, they mainly use the information extracted from the pull-request title and description, they therefore don't look at the source code attached to the pull-request. They then perform a case study on three well-known Github projects and find that their technique can find about 55.3% to 71.0% of the duplicates.

[Ahasanuzzaman et al., 2016] perform an empirical study on duplicate questions in the popular question and answer site StackOverflow[3]. In their study, they aim at applying machine learning techniques to automate the process of duplicate questions identification. They investigate three main research questions. First, they investigate why developers resort to ask duplicate questions. To do so, they took a sample of 600 duplicate questions and manually analysed them. As a result, they found multiple potential reasons being: not searching on Stackoverflow first, titles do not match, domain differences, difficult to comprehend, too concise to understand, lack of knowledge, lack of terminology. Secondly, they investigate the possibility of building a tool that could automate the detection of duplicate questions. To do so, they chose to train a logistic regression model on a corpus composed of 130,000 duplicate questions provided by StackOverflow. For the training, they tried different sets of features and selected the best ones. Finally, they evaluated their tool and found that their tool outperforms all existing tools.

[Sæbjørnsen et al., 2009] propose an algorithm for binary executables clone detection. Their tool follows a tree similarity framework, it models the structural information of instruction sequences into numerical vectors and groups similar vectors in order to identify clones. They also perform a large scale study on their tool using the disassembled Windows XP system executables and libraries and find large numbers of clones.

[Hermans et al., 2013] perform an empirical study on clones in spreadsheets. For their study they perform a quantitative evaluation in which they analyze the EUSES corpus and a qualitative evaluation composed of two case studies. For their study, they define clones as being *data clones* which are formula results that are copied as data in other parts of a spreadsheet. They then develop a data clone detection algorithm based on existing text-based clone detection techniques. They then investigate three research questions. First, they investigate if data clones are frequent. To do so, they apply their tool on their corpus and find that data clones are actually common in spreadsheets. Secondly, they investigate the impact of data clones and learn that clones have impact on the awareness of users as they don't necessarily know that that data was actually copied from another place. Finally, they investigate if their approach helps users and find that their technique had helped users identify data that was duplicated and they weren't aware of.

Summary

3. https://stackoverflow.com/

In this section, we have seen that clones can actually exist outside source code. In particular, we've seen that they are quite frequent in software artifacts such as UML models, build files and even spreadsheets. In this thesis, we investigate the existence of duplicates in two types of software artifacts: API documentation and Dockerfiles. To the best of our knowledge, both of these artifacts weren't previously studied in the literature.

2.3 Clone management

In contrast with code clone detection, the field of clone management activities (i.e. techniques and tools for managing code clones) hasn't gained as much attention [Roy et al., 2014]. In this section, we present different approaches that have been undertaken in the literature to manage clones during their lifetime.

In 1996, Mayrand et al. [Mayrand, 1996; Lague et al., 1997] proposed to split clone management activities into two main categories: problem mining and preventive control. They define problem mining as the activity that aims at coping with the existing base of clones in a software that's already in service. Through this they aim at always keeping clones with a common context up-to-date across changes. Also, they define preventive control as the activity of controlling the introduction of new clones meaning that a new clone would be introduced in a software system only if good reasons indicate that it's necessary.

[Giesecke, 2007] suggest a new possibility to integrate clone management into the software development process. Based on Mayrand et al.'s work [Mayrand, 1996; Lague et al., 1997], they propose three new categories similar to Swanson's [Swanson, 1976] maintenance categories for clone management activities:

— Corrective clone management: activity consisting of removing existing clones from the source code.

— Preventive clone management: activity consisting of avoiding the introduction of clones into the source code.

— Compensatory clone management: activity consisting of avoiding negative consequences of existing clones in the source code.

[Zhang et al., 2013] proposed a preventive clone management approach that is contextual and on-demand called *CCEvents* (Code Cloning Events). In their approach, they propose timely notifications regarding specific code cloning events based on continuous monitoring of code repositories. In order to provide such notifications, they built a code cloning domain model capturing the introduction and evolution of code clones. Their domain model was also capturing various clone context information such as: the time when a code clone is introduced or modified, entities in which a code clone resides and the developer who introduced or modified the code clone. They then defined a domain-specific language called *CCEML* allowing users to define their own on-demand clone monitoring

strategies. They then evaluate their tool and show that such approach is actually effective for helping ease code clones management.

[Duala-Ekoko and Robillard, 2007, 2008] propose a compensatory clone management approach through an Eclipse plug-in called *CloneTracker* that provides developer support for tracking code clones. Their tool relies on *clone region descriptors* (CRD) describing a clone region based on a combination of syntactic, structural and lexical information. *CloneTracker* takes as input the output of a traditional code clone detection tool. Based on that input, the tool automatically generate CRDs for each clone region representing a clone group. Clones are then automatically tracked, and developers are automatically notified whenever a clone is facing a code change. The tool also provides mechanisms to apply simultaneous code changes in cloned regions. *CloneTracker* even provides developers the possibility to share their project's clone model with their collaborators.

[Kawaguchi et al., 2009] proposed a preventive clone management tool for automatic clone detection and management in the IDE called *SHINOBI*. They implemented their tool as an add-on to Microsoft Visual Studio IDE. Contrary to *CloneTracker*, their tool doesn't need to re-run code clone detection to detect newly created code clones. *SHINOBI* automatically and quickly detect code clones when the source code is being edited. Whenever a developer makes a code fragment copy, the tool automatically detects the event and notifies the developers of all duplicate code fragments in the source code, making it possible for developers to handle the clone issues early on during the development process. Also, whenever the developer applies modifications to a code fragment, the tool automatically notifies the developers of similar code fragments that should also have the changes applied on.

[Synytskyy et al., 2003] proposed a corrective clone management technique to automatically identify and reduce the number of clones in HTML pages. Through their approach, they aim at automatically using existing reuse mechanisms to reduce the number of existing clones. To do so, after the tool identifies HTML clones, it will automatically choose a reuse technique from four main techniques. The first is *server-side includes* which is a functionality provided by the server rather than the HTML language. Such technique can be used to reduce any detect clone. Another technique is *executable sub-routines* which rely on reuse mechanisms provided in the server-side language such as Java. This technique is similar to the previous one, however, in the first one, clones had to be in a separate file to be included while with this method they can be in a single file. Another technique is *external script files* which relies on the include mechanism existing in the Javascript language to reuse clones written in Javascript. The last technique they propose is the use of resolution methods provided by the server-side includes to reduce all clones.

[Jablonski and Hou, 2007] proposed a compensatory clone management tool that tracks copy-paste clones and automatically infers rules capturing the programmer's intentions. They proposed their tool called *CReN* as an Eclipse plug-in working with Java projects. Using the abstract syntax tree (AST) API provided by Eclipse, the tool automatically starts observing project's activity. The tool relies on the AST because it allows it to

automatically establish relationships of the copy-pasted code and infer knowledge for consistent renaming. As *CReN* automatically tracks new clones when copy-paste operations are happening the tool doesn't require to perform any code clone detection. The tool keeps track of all created clones and whenever a new update on a code clone is performed, the update can be automatically propagated to all instances of the clone. *CReN* also uses highlights on the user interface to let developers visualize when a code fragment is duplicated and even provide an interface to manage clones on the IDE.

[Nguyen et al., 2011] proposed a compensatory clone management tool called *JSync*. Their tool aims at helping manage code clones for Java projects. It is implemented as an Eclipse plug-in and it uses an SVN version control repository in order to store its data on clones and access source code files with their changes. The *JSync* tool has three purposes: it helps developers be more aware about the existence of cloned code in their projects, it prevents defects due to inconsistent changes, and it helps developers propagate changes across code clones. The tool is divided into two main modules: Clone Relation Management (CRM) which is responsible for managing code clones and groups of code clones, and Clone Consistency Management (CCM) which is responsible for the management of code clones such as maintaining consistency across groups of clones. *JSync* just as *CReN*, relies on AST to represent the project's source code. The tool represents code changes as a sequence of tree edit operations (addition, modification, deletion of a tree node). They also evaluate their tool on Bellon's clone benchmark [Bellon et al., 2007b] and several open-source project and find that their tool is scalable and efficient.

Summary

In this section, we have seen that the clone management research field lacks researcher's attention compared to the clone detection field. While a number of studies tried to provide solutions for managing clones in source code, the field lacks research on clone management for other software artifacts. In this thesis, we chose to investigate the different clone management approaches that exist for two types of software artifacts: API documentation and Dockerfiles. To the best of our knowledge, both of these artifacts' clone management practices weren't previously studied in the literature.

2.4 Summary

In this background chapter, we've seen that there's a large set of research studies on clones in software engineering ranging from clone definitions to clone management. As a summary, we list the three main lessons we've learned from this chapter in order to answer our research questions:

— In our first research question, we aim at determining if clones exist in two types of software artifacts: API documentations and Dockerfiles. We've seen in this chapter that there exists a large number of clone detection approaches. Some approaches can be tied to the source code language such as tree-based approaches, while others such as text-based approaches are totally independent from the type of artifact that is analysed. In this thesis, we chose to rely on a token-based approach to answer our first research question, as both API documentation and Dockerfiles can be seen as a set of tokens and token-based detection techniques can easily handle different types of languages..

— In our second research question, we aim at identifying the different causes for clones in the software artifacts we're analysing. We've seen in this chapter that there exist a number of empirical studies on clones on software artifacts such as UML models and web services. However, to the best of our knowledge no empirical study has tried to investigate the underlying reasons for clones in API documentation and Dockerfiles.

— In our third research question, we aim at identifying the different approaches that are used by developers to avoid clones in their software artifacts. We've seen in this chapter that clone management activities can be split into three categories: corrective, preventive and compensatory. While some research work has been done on preventive clone management approaches, to the best of our knowledge, no research study has been done on preventive clone management approaches for API documentation and Dockerfiles.

CHAPTER

3 Duplicates in API documentation

Code documentation is a crucial part of software development. However, studies have shown that such documentation is rarely up-to-date with the code and is perceived as very expensive to maintain. In this chapter we investigate the issue of duplicates in API documentation and show that it is actually frequent. We perform a survey of 39 developers and show that they're aware of these issues and are lacking solutions to solve them. Finally, we show that out of 19 documentation tools, only one tool could help at easily solve most duplicate issues that we've found in our study.

Contents

3.1 Introduction

As previously stated in Chapter 1, API documentation plays a crucial role in software development as it helps developers understand someone else's code without reading it [Lakhotia, 1993; Vanter, 2002]. It is even more critical in the context of APIs, where the code is developed with the main intent to be used by other developers (the users of the API), who don't want to read the code [Parnas, 1972; Kramer, 1999; Monperrus et al., 2012]. In this context, having a high quality reference documentation is critical [Dagenais and Robillard, 2010].

Further, it has been shown that the documentation has to be close to the corresponding code that it describes [Forward and Lethbridge, 2002; Lethbridge et al., 2003; Fluri et al., 2007], and that developers prefer to write the documentation directly in comments within the code files rather than in external artifacts [de Souza et al., 2005]. Popular documentation tools, such as JavaDoc, Yard or Doxygen, all share the same principle which is to parse source code files to extract tags from documentation comments and to generate readable web pages [Pollack, 2000; Van Heesch, 2004].

Writing documentation and code are highly coupled tasks. Ideally, developers should write and update the documentation together with the code. However, it has been shown that the documentation is rarely up-to-date with the code [Forward and Lethbridge, 2002; Lethbridge et al., 2003; Fluri et al., 2007] and is perceived as very expensive to maintain [Dagenais and Robillard, 2010; Correia et al., 2009].

We think that one possible reason for this maintenance burden is that documentation tools lack reuse mechanisms whereas there are plenty of such mechanisms in programming languages. Developers that write documentation therefore copy- paste many documentation fragments, which is suspected to increase the maintenance effort [Juergens et al., 2009c].

In this chapter, we investigate our hypothesis by answering the following research questions:

— RQ1: Do developers often resort to copy-paste documentation tags?

— RQ2: What are the causes of documentation tags copy-paste?

— RQ3: Could duplicate documentation be avoided by a proper usage of documentation tools

This chapter is organized as follows: Section 3.2 presents a background regarding API documentation. Section 3.3 presents the 100 repositories (50 for the Java language, 50 for the Ruby language) composing our corpus, our survey methodology and duplicates detection technique. Section 3.4 presents the methodology and results for our three research questions. Section 3.5 presents the threats to validity regarding our experiments. Finally, Section 3.6 concludes this chapter.

3.2 Background

Regardless of the programming language, when writing documentation in the source code, the documentation is always written as a set of in-line comments (single line) or blocks (spanning across multiple lines) of comments. Such comments are usually placed just before the code element they refer to. Listing 3.1 depicts an extract of such block of comments from the RbNaCL open-source project (A ruby cryptography library). In this extract, the developer describes the main purpose, inputs and outputs of the method.

LISTING 3.1 – Example of a block of comments describing a method from the Apache Commons IO project.

```
1  # Use a secret key to create a SimpleBox
2  #
3  # This is a convenience method.  It takes a secret key and
       instantiates a
4  # SecretBox under the hood, then returns the new SimpleBox.
5  #
6  # @param secret_key [String] The secret key, 32 bytes long.
7  #
8  # @return [SimpleBox] Ready for use
9   def self.from_secret_key(secret_key)
10   new(SecretBox.new(secret_key))
11  end
```

In order to help developers maintain such documentation, a plethora of documentation tools such as Javadoc (Java), Yard (Ruby) and Doxygen (C, C++, Python, PHP, Java) have been built. Such documentation tools are used to automatically parse the source code looking for documentation comments in order to construct a readable documentation that will exported as an artefact, such as a PDF or a web-site [Pollack, 2000; Van Heesch, 2004]. For instance, the generated documentation for Listing 3.1 using the Yard documentation tool is shown in Figure 3.1.

In order to generate a well-formatted API documentation, documentation tools provide additional constructs that are used in code comments. For instance, they divide the block of comments into two sections: a main description at the beginning, followed by what is called a *tags* section. The main description is used to describe the element that we want to document (e.g. a method, a class, an attribute). The tags section on the other hand, is composed of zero to as many tags as we want. Tags are used to enrich the documentation with specific information, such as a description of each input parameter of a method for instance. Such description is directly placed after the tag itself. Tags are easily recognizable as they often start with a @ sign (e.g. @author, @param, @return...). There exist around 20

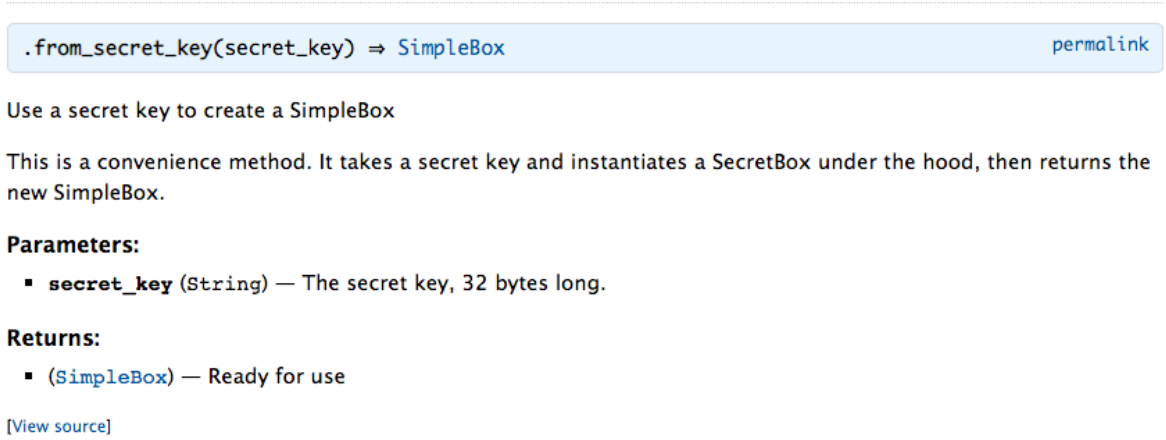

Figure 3.1 – The generated documentation by Yard for the *from_secret_key* method from the RbNaCL project.

documentation tags for the Javadoc tool[1] for instance. The *@param* tag for example, lets the documentation tool know that the following line describes a method's parameter.

Just as for comments where we can have in-line and block comments, there are two main types of tags: in-line and block tags. Block tags (such as @param, @return) must always appear at the beginning of a line. They are always followed by a description sentence that is used in the documentation. An in-line tag is always enclosed by curly braces (i.e. *{@tag}*), contrary to block tags, such tag can appear anywhere in a sentence whether in the main description or in the description of a block tag (see Listing 3.3).

LISTING 3.2 – Example of a block of comments describing a method from the Apache Commons IO project.

```
1  /**
2  * Invokes the delegate's <code>skip(long)</code> method.
3  *
4  * @param ln the number of bytes to skip
5  * @return the number of bytes to skipped or EOF if the end
       of stream
6  * @throws IOException if an I/O error occurs
7  */
8  public long skip(final long ln) throws IOException {
9    try {
10    return in.skip(ln);
11   } catch (final IOException e) {
12    handleIOException(e);
13    return 0;
```

1. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

```
14     }
15 }
```

Listing 3.2 is an extract from the Apache commons IO project using the Javadoc documentation tool. We can see that the method is preceded with a block of comments, which is divided in two sections: a single sentence as a main description, followed by three tags in the tags section. The first tag is *@param* that defines the input parameters of a method. We can see that it presents two information: the name of the input (i.e. In) and its description. These two information are separated by a single space, where the first word of the describing sentence will always be the name of the attribute. The second parameter is *@return*, it describes the returned value. As there is only one possible return value in Java, we don't need to set a name to the return value as for the *@param* tag. Finally the *@throws* (or *@exception*) presents two types of information: the exception type (here IOException) and its description.

LISTING 3.3 – Example of a block of comments using an in-line tag from the AssertJ project.

```
1 /**
2 * Check that the {@link ZonedDateTime} to compare actual
     {@link ZonedDateTime} to is not null, otherwise throws a
     {@link IllegalArgumentException} with an explicit message
3 *
4 * @param dateTime the {@link ZonedDateTime} to check
5 * @throws IllegalArgumentException with an explicit message
     if the given {@link ZonedDateTime} is null
6 */
7 private static void
     assertDateTimeParameterIsNotNull(ZonedDateTime dateTime)
     {
8  checkArgument(dateTime != null,
      NULL_DATE_TIME_PARAMETER_MESSAGE);
9 }
```

Listing 3.3 is an extract from the AssertJ project. We can see that the in-line tag *@link* has been used. Such tag is used to automatically insert a link pointing to the right documentation. For instance, in our extract, the *{@link ZonedDateTime}* will automatically be replaced by a link to the class' documentation when the documentation is generated.

3.3 Data collection

In this section, we present our three main data sources that are used in the remainder of this study: a list of 100 GitHub repositories, each one corresponding to an open-source library (50 repositories using the Javadoc tool and 50 using the Yard tool), an on-line survey

performed on 39 developers who have experience writing code documentation and more than 41,000 duplicate tags across our repositories.

3.3.1 Repositories

For our study we arbitrarily chose to focus on the API documentation of libraries written in two specific programming languages: *Java* and *Ruby*. Developers using these two programming languages usually write their documentation directly in the source code. They then use documentation tools to generate the corresponding API documentation: JavaDoc for generating Java documentation and Yard for generating Ruby documentation.

After analysis, we chose to focus only on methods' documentation, as this is where there is most of the documentation [Oumaziz et al., 2017]. In the remainder of this study, we therefore only discuss about the documentation of Java and Ruby methods written in JavaDoc and Yard. We also just consider the source code used to generate the documentation displayed on their websites.

For this study, we chose to construct a corpus composed of 100 libraries (50 per programming language). We therefore explain the process we followed to select these libraries. For each language, we go to Github and search for the keyword "library" and specify the language it is written in as respectively Java and Ruby. We then rank these libraries for each programming language based on the number of stars they have. Finally, we select the resulting first 50 libraries as our corpus for each language.

Figure 3.2 depicts the number of classes per project for each programming language in our corpus. We can see that Java projects are consequently larger compared to ruby projects with respectively a median at 206 and 29.

Figure 3.3 presents the of the left the number of methods that each project in our corpus has. It shows how Java projects are larger than Ruby projects. However, while Java projects happen to be larger, the right figure indicates that they also lack documentation compared to Ruby projects as more than 75% of Ruby methods have at least 30% of their methods documented compared to less than 20% of Java methods (we consider a method as being documented if it has a main description or any documentation tag: @param, @return, @throws, @raise). We also do note that half of Ruby projects have half of their methods documented.

Through our analysis, we do notice that out of all existing tags used by documentation tools, only 4 tags are most frequently used: **@description** (main drescription of a method), **@param**, **@return** and **@throws/@raise** for Java and Ruby. Finally, both JavaDoc and Yard propose a mechanism to reuse a documentation tags across method's documentations. For JavaDoc, the in-line tag *InheritDoc* is used to reuse tags when there is inheritance in Java. It was used 450 times across 15 projects. On the other hand, Yard proposes a reuse mechanism "*(See #method)*" that lets developers reuse any tag from any method's documentation existing in the project. This mechanism was used 95 times across 4 projects.

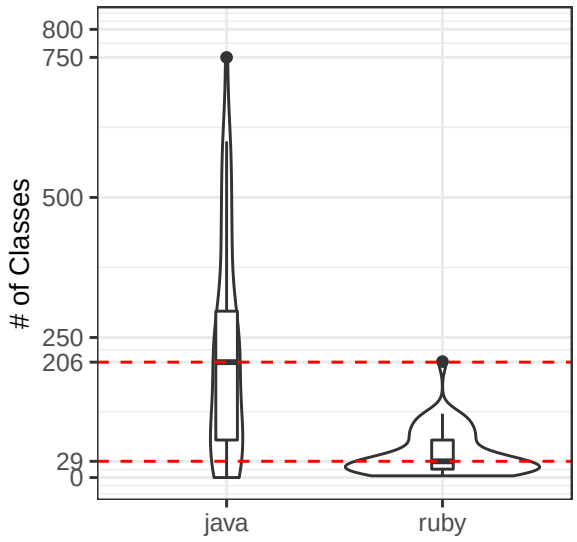

Figure 3.2 – Violin plot for the number of classes of each project in our for corpus (for both Java and Ruby).

3.3.2 Survey

Our study involves a survey of developers maintaining code documentation to get their opinions regarding duplicates in code documentation and understand their practices.[2] Therefore, we create a survey composed of two main sections asking questions about 1) duplicates and 2) management tools to handle duplicates. We then share this survey across multiple reddit's sub-reddit such as: r/ruby, r/learnjava, r/compsci sub-reddits. We also share this survey across different social medias such as Twitter an LinkedIn. We then manage to gather 48 responses for our survey. Out of these 48 responses, 25% (12 out of 48) said that they always write documentation, 54,2% (26 out of 48) stated that they sometimes write documentation, while 20,8% (10 out of 48) stated that they never write documentation. One developer who stated that he never writes documentation, still wanted to continue the survey, which therefore leaves us with 39 responses for our study. Finally, 56,4% (22 out of 39) of respondents stated that they use Javadoc as their documentation tool, the other used documentation tools being: Doxygen, JSDoc and Yard.

2. https://forms.gle/75Z5hpEMb57mNRT59

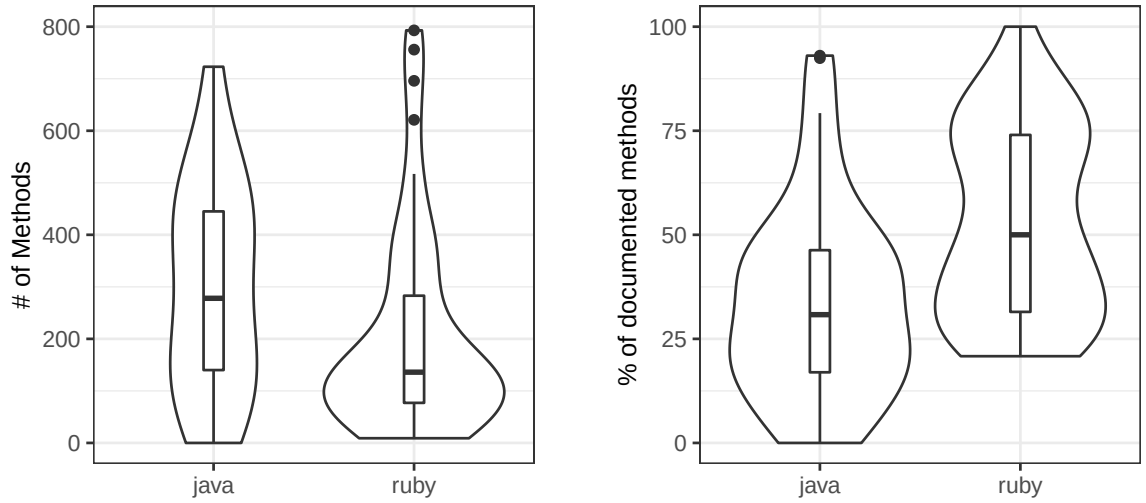

Figure 3.3 – Left: Violin plot for the number of methods in every project in our corpus. Right: Violin plot for the percentage of documented methods in every project in our corpus.

3.3.3 Duplicates detection tool

A duplicate documentation tag, is a tag that is duplicated across at least two method's documentation as it is the case in Figure 3.4. In this figure, we have an extract from the Apache Commons IO project, where both methods are sharing a duplicate tag (in red).

We then implement a duplicate tags detector that inputs a set of Java or Ruby repositories and outputs the so-called duplicate documentation tags. The detector first starts by parsing all the source files in a repository and identifies all the documentation tags they contain. If the repository has Java source files, it uses the GumTree tool [Falleri et al., 2014] to extract all existing documentation tags through the use of the AST (Abstract syntax tree) generated by gumtree (which let's us link each tag to its corresponding method). If the repository has Ruby source files, it uses the Yard library [3] to extract all documentation tags with their corresponding method. To detect only meaningful duplicates, it only extracts the most important tags in JavaDoc and Yard (as we've seen in Section 3.3.1): @param, @return, @throws (or @raise for Yard). It also extracts the main description of each method as if it has a corresponding tag (with an imaginary @description tag). Finally, to avoid missing duplicates because of meaningless differences in the white-space layout, it cleans the text contained in the documentation tags by normalizing the white-spaces (replacing tabs

3. https://github.com/lsegal/yard

```java
/**
* Writes a String to the {@link StringBuilder}.
*
* @param value The value to write
*/
@Override
public void write(final String value) {
 if (value != null) {
  builder.append(value);
 }
}
```

```java
/**
* Writes a portion of a character array to the {@link
    StringBuilder}.
*
* @param value The value to write
* @param offset The index of the first character
* @param length The number of characters to write
*/
@Override
public void write(final char[] value, final int offset,
    final int length) {
 if (value != null) {
  builder.append(value, offset, length);
 }
}
```

Figure 3.4 – Extract of a documentation duplication from the Apache Commons IO project. The duplicated tag is highlighted in red.

by spaces, removing carriage returns and keeping only one space between two words). For the same reasons, it also transforms all text contained in documentation tags to lower-case

As a final step, the detector makes comparisons between the hashes of each identified tags, and checks if they are shared across different methods and returns a list associating each duplicate tag with the corresponding list of methods owning it. In the remainder of this study, we call *owners* the list of methods having an identical tag in common.

3.4 Contributions

In this chapter, we then present the three main contributions of this study:

3.4.1 RQ1: Do developers often resort to copy-paste documentation tags?

In this section, we answer our first research question. To investigate if documentation duplicates are frequent, we simply apply our documentation duplicate detector to our corpus and report statistics about the extracted duplicates.

Methodology

Using the tool we've described in Section 3.3.3, we go and look for all duplicated documentation tags in our repositories (50 per programming language). Using these results, we then compute several statistics about the characteristics of these identified duplicates and compare them with the results we gathered from our survey (Section 3.3.2).

Duplicate tags

After applying our detection tool, we've found that out of 8,170 tags identified in Ruby projects, 2,263 of them were duplicates (27,69%). We also found that out of 71,514 tags identified in Java projects, 33,184 of them were duplicates (46,40%). Figure 3.5 depicts the percentage of duplicates across each project in our repositories. We notice that more than 75% of Java projects have at least 25% of their documentation that is duplicated, while 75% of Ruby projects have at least 20% of their documentation that is duplicated. This clearly indicates that independently from the programming language that is used, projects often face duplicates in their documentation. When asking developers if they have faced duplicates in the past: 64% (25 out of 39) said that they did, while 23% (9 out of 39) said they've never faced duplicates and 12% (9 out of 39) didn't know. Also, out of the 25 developers who have faced duplicates in the past, 16 (64%) stated that they had updated several identical documentation tags simultaneously to make them co-evolve. They qualify such co-evolution practices as being mainly annoying and error-prone.

We then take a closer look at the number of owners for a duplicate documentation tag (i.e. the number of methods sharing a common tag) as depicted in Figure 3.6. We do notice that both Java and Ruby share a common pattern, for both languages, a large majority of duplicates has 2 to 4 owners. While duplicates having more than 4 a less frequent, but still important with 25% of duplicates. This indicates that while tags might be often duplicated, their spread can be somehow limited. When asking developers: *How many times a tag should be duplicated in order to be detected by a tool?*, 84,6% (33 out of 39) said that it should be the minimum possible number which is 2, 12,8% (5 out of 39) said that it should be 3 and 2,6% (1 out of 39) said that it should be at least 5 times. These responses match with our results, with detected duplicates usually spanning across 2 to 4 methods.

We now take a step further and look at how often each kind of tags (*@description*, *@param*, *@return*, *@raise*/*@throws*) is duplicated in every project in our repositories. We

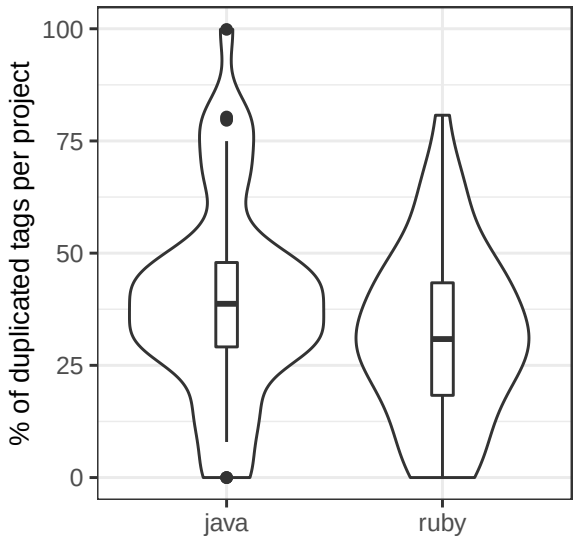

Figure 3.5 – Violin plot for the percetange of duplicated tags per project (for both Java and Ruby).

then obtain the results depicted in Figure 3.7. We first notice in the upper-left Violin plot, representing the percentage of duplicate *@description* tags, that they're actually less frequently duplicated compared to the other tags. We also note that the percentage of duplicate *@description* is quite similar between both Java and Ruby languages, with a mean at around 6%. Then, in the upper-right figure, we notice that *@param* are largely duplicated in Java (mean at nearly 50%) while not so much in Ruby (mean at 0%). The same goes for the bottom-right figure showing the percentage of duplicate *@raise*/*@throws* tags, Java faces way more duplicates for such tags with a mean at 60%. Next, in the bottom-left figure, we can see that both Java and Ruby are facing nearly the same duplicates issues with *@return* tags, with nearly 50% having more than 40% of their *@return* tags duplicated. We also notice that for Java projects, more than 60% of their *@throws* tags are being duplicated.

Finally, when we ask developers: *Would you like your documentation tool to help you avoid duplicates?*, 71,1% (27 out of 38) stated yes, while the 11 developers who responded no, said so because: they think that such tool isn't useful, they don't want another tool, they don't use documentation tools.

Summary
Duplicates in documentation are frequent. They usually span across 2 to 4 methods, but they can also be very large. Developers are frequently facing duplicates in their documentation. They often have to make them co-evolve, and find such practices as

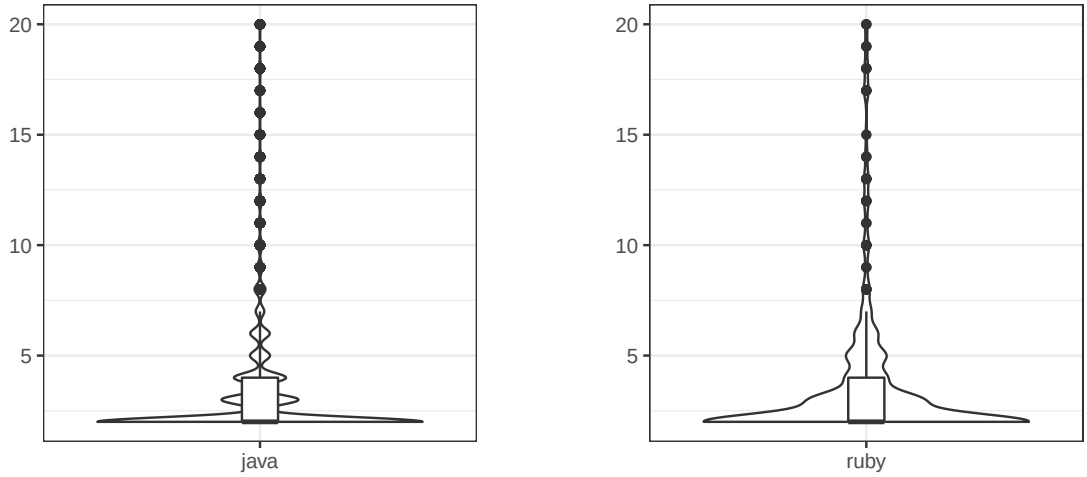

Figure 3.6 – Left: Violin plot for the number of methods sharing a common tag in Java. Right: Violin plot for the number of methods sharing a common tag in Ruby.

being annoying and error-prone. Finally, they would love that their documentation tools can help them avoid such duplicates.

3.4.2 RQ2: What are the causes of documentation tags copy-paste?

In this section, we answer our second research question. We draw at random a subset of the extracted duplicates and ask two authors to manually determine the underlying reason for their existence.

Methodology

In order to perform this study, we asked the two authors of this study to manually take a look at a random sample of 200 duplicates (100 for each programming language). We chose to limit ourselves to look at only 200 because of time constraints reasons, since for each duplicate, the authors have to manually inspect the code, identify the purpose of the method and then determine if the duplicate wasn't a mistake and if so, what did cause its existence. As this process is quite long, 200 duplicates seemed to be a reasonable amount. Also, if the two authors' duplicate existence reason isn't the same, a third author has to decide which one is the most plausible.

Results

Through our manual analysis, we've noticed that both programming languages are actually facing the same source causes for duplicates. During the analysis, the two authors

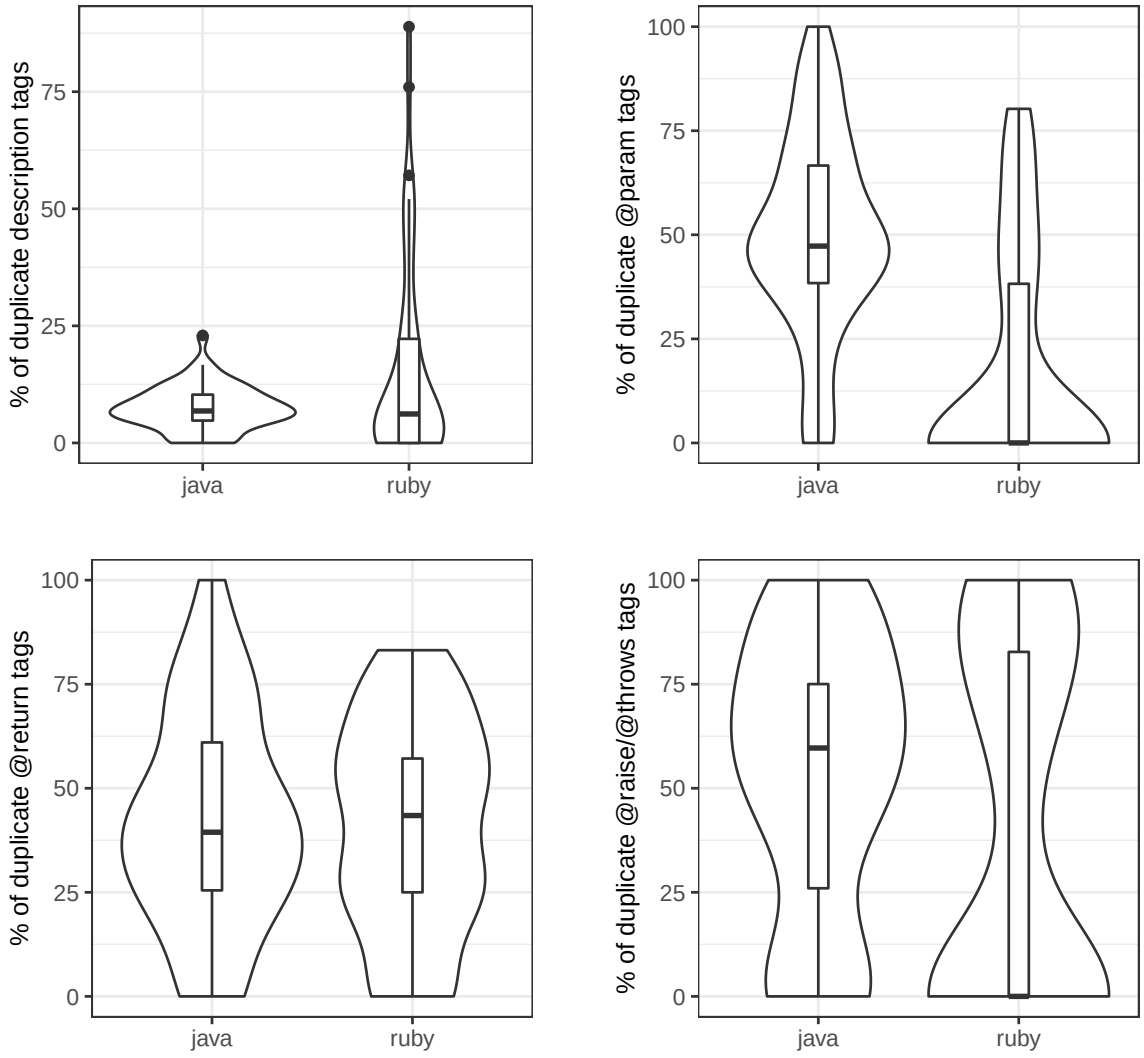

Figure 3.7 – Upper-left: Violin plot for the number of duplicate *@description* tags per project. Upper-right: Violin plot for the number of duplicate *@params* tags per project. Lower-left: Violin plot for the number of duplicate *@return* tags per project. Lower-right: Violin plot for the number of duplicate *@throws* (*@raise* for ruby) tags per project.

agreed on the underlying reason for all duplicates and identified four main reasons for duplicates in API documentation: delegation, sub-typing, code clone, similar intent, similar use.

A *delegation*, as shown in Figure 3.8, appears when a method calls another one, and thus has a part of its documentation coming from the called one. In Figure 3.8 the upper

```
1 #  Shows objects
2 #  @param [String|NilClass] objectish the target object reference (nil
      == HEAD)
3 #  @param [String|NilClass] path the path of the file to be shown
4 #  @return [String] the object information
5 def show(objectish=nil, path=nil)
6   self.lib.show(objectish, path)
7 end
```

```
1 #  Shows objects
2 #  @param [String|NilClass] objectish the target object reference (nil
      == HEAD)
3 #  @param [String|NilClass] path the path of the file to be shown
4 #  @return [String] the object information
5 def show(objectish=nil, path=nil)
6   arr_opts = []
7   arr_opts << (path ? "#{objectish}:#{path}" : objectish)
8   command('show', arr_opts.compact)
9 end
```

Figure 3.8 – Extract of duplicate due to a delegation between two methods in the Ruby/Git library project. Duplicated tags are displayed in red.

show is calling the lower method *show* and therefore shares the same documentation since it does only a single task which is calling the lower *show* method with the right parameters. We encountered this reason for nearly 46% of Java projects and nearly 47% of Ruby projects.

A *similar intent* appears when a method performs a computation that is similar to another method, which is why they share some documentation tags. Figure 3.9 shows such an example. Here the two methods only differ because of the return type (float or int). It is not a clone because there is no common line between them. Further, a funny thing is that the developer made a mistake as he clearly copied the documentation of the long method but didn't change the documentation of the int and float ones. One could therefore think about extending existing documentation tools to automatically detect such basic inconsistencies during documentation generation and notify the developer or even automatically suggest corrections. Most of similar intent cases we observed are due to developers implementing several times a same feature for each primitive type. We encountered this reason for 24% of Java projects and nearly 10% of Ruby projects.

A *sub-typing* appears when a method overrides another one that is defined in a same hierarchy. In this case, it is common that the overriding method's documentation comes from the one of the overridden method. Figure 3.10 shows an example of sub-typing

```
1 /**
2 * Delegates to {@link
     EndianUtils#readSwappedInteger(InputStream)}
3 * @return the read long
4 * @throws IOException if an I/O error occurs
5 * @throws EOFException if an end of file is reached unexpectedly
6 */
7 public int readInt() throws IOException, EOFException {
8   return EndianUtils.readSwappedInteger(in);
9 }
```

```
1     /**
2 * Delegates to {@link
     EndianUtils#readSwappedFloat(InputStream)}
3 * @return the read long
4 * @throws IOException if an I/O error occurs
5 * @throws EOFException if an end of file is reached unexpectedly
6 */
7 public float readFloat() throws IOException, EOFException {
8   return EndianUtils.readSwappedFloat(in);
9 }
```

Figure 3.9 – Extract of duplicate due to two methods with a similar intent in the Guava project. Duplicated tags are displayed in red.

where the whole documentation has been copy-pasted. In this example extracted from the Apache Commons Collections Java project, we have an interface declaring an abstract method with its documentation (upper example). The lower example corresponds to the implementation of the method declared in the interface. We do notice that the documentation has actually been duplicated in both methods (in red color). A solution to avoid such situation is to use reuse mechanisms that are available in the Javadoc tool for instance, we discuss in more detail in Section 3.4.3. We encountered this reason for nearly 13% of Java projects and nearly 17% of Ruby projects.

A *code clone* appears when a method shares similar lines of code with another one, hence duplicating a part of its body. Figure 3.11 shows an example of code clone from the Apache Commons IO Java project, where two methods share a nearly identical line of code. Due to this line of code that requires a value as input for a method call, both *write* methods are therefore aslo requiring a value attribute as input which causes a duplicate line in their documentation. We encountered this reason for nearly 6% of Java projects and nearly 16% of Ruby projects.

```
/**
* Gets the value from the pair.
* @return the value
*/
V getValue();
```

```
/**
* Gets the value from the pair.
* @return the value
*/
public V getValue() {
  return value;
}
```

Figure 3.10 – Example of duplicate due to sub-typing in the Apache Commons Collections project. Duplicated tags are displayed in red.

```
/**
* Writes a String to the {@link StringBuilder}.
*
* @param value The value to write
*/
@Override
public void write(final String value) {
  if (value != null) builder.append(value);
}
```

```
/**
* Writes a portion of a character array to the {@link
    StringBuilder}.
*
* @param value The value to write
* @param offset The index of the first character
* @param length The number of characters to write
*/
@Override
public void write(final char[] value, final int offset,
    final int length) {
  if (value != null) builder.append(value, offset, length);
}
```

Figure 3.11 – Example of duplicate due to code clone in the Apache Commons IO project. Duplicated tags are displayed in red.

```ruby
# Create partitions for a topic.
# @param name [String] the name of the topic.
# ...
def create_partitions_for(name, num_partitions: 1, timeout:
    30)
  @cluster.create_partitions_for(name, num_partitions:
     num_partitions, timeout: timeout)
end
```

```ruby
# Alter the configuration of a topic.
# ...
#@param name [String] the name of the topic.
# ...
def alter_topic(name, configs = {})
  @cluster.alter_topic(name, configs)
end
```

Figure 3.12 – Extract of duplicate due to a similar use between two methods in the Ruby/Git library project. Duplicated tags are displayed in red.

Finally, a *similar use* happens when two methods or more share a common input parameter, output or exception but the task that each method has to do is totally different. Figure 3.12 is an extract from the Kafka library in Ruby. We can see that methods aren't performing a similar task, however, they both have to interact with what is called a topic. In order to interact with such topic, they need to have its name which is duplicated across the documentation of both methods (highlighted in red), and in such situation, whenever there's for example an hypothetical new constraint on the topic parameter, such constraint has to be propagated to the documentation of any method using that name as an input parameter. We encountered this reason for nearly 11% of Java projects and nearly 10% of Ruby projects.

Summary
Both Java and Ruby programming languages are facing the same underlying reasons for duplicates in their documentation. The two authors identified five main reasons being: delegation, sub-typing, code clone, similar intent and similar use.

3.4.3 RQ3: Could duplicate documentation be avoided by a proper usage of documentation tools?

In this section, we answer our third and final research question. We perform a manual analysis of more than 20 documentation tools seeking for all reuse mechanisms they provide. We then describe each mechanism with their advantages and limitations.

Methodology

In order to answer our last research question, we first manually look at the different documentation tools to determine the mechanisms they provide for reusing documentation. As there are too many documentation tools, and due to time constraints, we choose to focus on a list of 22 different documentation tools [4]. Out of these 22 tools, 3 were paid tools, we therefore remove them from our list as we didn't manage to easily have access to their documentation. We then manually analyse in detail the whole user-guide of all 19 tools in our list to find out all types of reuse mechanisms that they provide, whether they are compatible with Java and Ruby or not, in order to be sure that there is no mechanism available for other languages that could avoid duplicates and therefore should be implemented for Java and Ruby.

Results

The results for our manual analysis on the 19 documentation tools are shown in Table 3.1.

Through our analysis, we find that there are three main types of reuse mechanisms all related to the reused documentation's location and their granularity:

— **Override** is a reuse mechanism that allows developers to reuse a documentation only for methods inheriting from the method that has its documentation reused. Some tools like *Javadoc* allow to reuse the whole block of documentation (depicted as *Whole* in Table 3.1), or even sub-parts of the documentation (depicted as *Choice* in Table 3.1), for example, reuse only a specific *@param* description. To do so, they simply provide an in-line documentation tag (as described in Section 3.2) called *@InheritDoc* that is used to specify which sub-parts of the documentation they want to reuse are implicitly used when a method without any documentation is inheriting from a documented method.

— **Anywhere** is a reuse mechanism that allows developers to reuse the documentation of any method from anywhere in the source code. Some tools like *Yard* allow developers just as with the *Override* mechanism, to reuse the whole block of documentation or sub-parts of the origin method's documentation. They do so by providing a

4. https://en.wikipedia.org/wiki/Comparison_of_documentation_generators

Table 3.1 – The list of 19 documentation tools analysed in the study.

		Override		Anywhere		
Tool	**Language**	**Whole**	**Choice**	**Whole**	**Choice**	**Include**
Sandcastle	.NET	✓	✓			✓
JavaDoc	Java	✓	✓			
YARD	Ruby			✓	✓	
JSDoc	JavaScript	✓		✓		
Doxygen	C, C++, Java, C#, VBScript, IDL, Fortran, PHP, TCL			✓		
Doc++	C, C++, IDL, Java					✓
Haddock	Haskell					✓
NDoc	.NET					✓
Sphinx	Python					✓
RDoc	Ruby					✓
VSdocman	C#, .NET					✓
PhpDoc	PHP	✓				
PerlPod	Perl					
PyDoc	Python					
DDoc	D					
Epydoc	Python					
ROBODoc	C, C++, Fortran, Perl, Shell					
HeaderDoc	Objective-C, C, C++, Java, JS, Perl, PHP, Python, Ruby					
Natural Docs	20 languages					

tag (i.e. *(see #method)*) that lets developers specify from which method they want to reuse a documentation whether for the whole bloc or a sub-part.

— **Include** is a reuse mechanism that allows developers to reuse a documentation that is written in another file. Therefore, they simply let you import the content of a whole file directly as a documentation, or even specific parts of documentation written in external files such as a parameter's description that we can fetch based on the parameter name. It is the most frequent reuse mechanism that we've seen in the 19 tools that we've analysed.

We do notice that there's actually only one tool (i.e. Yard) that could handle all causes of duplicates that we've identified in Section 3.4.2, as it provides the possibility to reuse any block or sub-part of documentation from any method's documentation anywhere in the source code.

It is also important to note that 7 documentation tools out of 19 don't provide any reuse mechanism at all. When we ask developers: *Do you use documentation tools functionalities to avoid duplicates?*, 94,9% (37 out of 39) stated they don't use any reuse mechanisms provided by documentation tool. When asking them why, they state that's because their tools have limited reuse mechanisms, that they don't want to use multiple tools just to avoid duplicates and that they don't know the tools that have such reuse mechanisms.

Summary
We manually analyse 19 documentation tools. Only one tool (i.e. Yard) could handle all causes of duplicates that we've identified. 7 documentation tools out of 19 don't provide any reuse mechanism at all. 94,9% of developers don't use any reuse mechanisms provided by documentation tool because of limited reuse mechanisms.

3.5 Threats to validity

We discuss here the threats to validity of our study following the guidelines provided by Wohlin et al. [Wohlin et al., 2012].

Construct validity. In Section 3.3.3, while our tool managed to identify identical duplicates in our corpus, measurement errors could possibly have been introduced by our duplicates detection tool. A mitigation to this is the fact that we chose to identify only identical clones (equivalent to type-I code clones) which are less prone to not be detected.

Internal validity. In Section 3.4.2, while the sample size is consequent (200 duplicates), it isn't large enough to be representative of the whole duplicates corpus. Further, duplicates' reasons tagging is done by the three study's authors who aren't experts of the projects containing the duplicates. While the three authors tried to mitigate the misidentification threat by concerting each other, performing a inter-rater agreement measurement on external developers should be taken in a future study replication work.

In Section 3.4.3, the identification of reuse mechanisms provided by documentation tools is performed by only one author who could have missed some information. However, the reuse mechanisms aren't that similar, meaning less subjective to identify by a single person.

External validity. This study focuses only on open-source projects with the largest number of stars on Github, corresponding to the 100 projects in our repository for two programming languages Java and Ruby. This is on purpose, as these projects are considered as being well-known and largely used by the open-source community. Therefore, the projects

may be more documented and maintained than more common projects meaning that the results we got may not be representative of all projects having a documentation. Also, our results are limited to two programming languages Java and Ruby which have different language constructs that can differ with other languages. It would be better to replicate the study with other APIs whether open-source or not and implemented in other programming languages that Java and Ruby.

In Section 3.3.2, the survey gathered 39 responses from developers. Out of our 39 respondents, 22 stated that they use Javadoc as their documentation tool, the other respondents using other documentation tools: Doxygen, JSDoc and Yard. Therefore, our findings might not be generalizable.

Conclusion validity. A threat to our study concerns the reliability of the metrics computed through our experiments. While we did follow an automated process for all measurements in RQ1 for both programming languages, which makes our results reliable, we did however follow a manual approach in RQ2 and RQ3.

Another threat to our study is the fact that we didn't statistically validate our results through non-parametric statistics (e.g. Mann-Whitney test).

3.6 Conclusion

Code documentation is a crucial part of software development. Like it is the case with source code, developers should reuse documentation as much as possible to simplify its maintenance.

Through this study on 100 repositories (50 using Java, 50 using Ruby) maintaining API documentation, we show that duplicates of documentation tags are unfortunately too abundant. By analysing these duplicates, we've identified that they are caused by five different kinds of relationships in the underlying source code: delegation, sub-typing, code clone, similar intent and similar use.

We also perform a survey on 39 developers and show that they are frequently facing duplicates in their documentation. They often have to make them co-evolve, and find such practices as being annoying and error-prone, and would love that their documentation tools can help them avoid such duplicates.

Finally, our study pinpoints the fact that it is common for documentation tools to not provide reuse mechanism to cope with these causes. However, one tool (i.e. Yard) provides a mechanism that handles all the duplicates causes that we've identified in our study.

CHAPTER

4 Duplicates in Dockerfiles

Docker is becoming a popular tool used by developers and end-users to deploy and run software applications. Dockerfiles are now found alongside projects' source code. Several projects are even starting to maintain families of Dockerfiles, like the Python project that maintains a family of 43 Dockerfiles, each for a specific Python version on a specific Linux distribution. In some situations, Dockerfiles family maintainers have to propagate a change to several, if not all, Dockerfiles of the family (for instance a bug-fix applying on all Dockerfiles targeting Python 2.7). In this chapter, our goal is to provide practitioners a clear explanation for why Dockerfile duplicates arise in projects, and what are the different means to handle duplicates with their pros and cons. We observe the practices of expert Dockerfile maintainers of Official Docker projects. We show that duplicates in Dockerfiles are frequent in our corpus, that maintainers are aware of their existence, are frequently facing them and have a mixed opinion regarding them. Finally, we describe and analyse the tools used by maintainers to handle duplicates.

Contents

4.1 Introduction

Docker simplifies the deployment of software applications by enabling developers to package their applications with all their required dependencies in a single binary unit, called a *Docker image*[1]. To build an image, developers have to write a *Dockerfile*, that will be executed by Docker. Such a Dockerfile is written in a Domain Specific Language (DSL) that can be compared to a very limited shell scripting language.

Software projects using Docker manage their Dockerfiles in their Software Configuration Management tool (e.g. GitHub), as any other software artifacts. Some projects even have to manage a family of Dockerfiles, especially when they have to support and maintain several versions in parallel with several sets of dependencies. For instance, the Python project[2] provides a family of 43 Dockerfiles, each one targeting a specific version of Python (e.g. 2.7, 3.8-rc) with a specific set of distributions (e.g. Debian, Alpine, Windows).

LISTING 4.1 – A bugfix of a same duplicate across the Alpine family of Dockerfiles in the official Python project.

```
1 RUN ...
2 && apk add --no-cache --virtual .build-deps  \
3 bzip2-dev \
4 ...
5 && make -j "$(nproc)" EXTRA_CFLAGS="-DTHREAD_STACK_SIZE=0x100000" \
6 ...
```

Looking at such projects, it appears that sometimes a patch targeting a Dockerfile has to be propagated to some (if not all) Dockerfiles of the family. As an example, Listing 4.1 shows a patch encountered in the official Python project[3]. This patch fixes an unexpected segmentation fault that appears only on Dockerfiles using the Alpine Linux distribution. This bug is due to a too small stack size and is resolved by adding a new flag in a *make* command as shown in Listing 4.1 (line 5). This bug-fix was propagated to the 7 Dockerfiles of the family that are based on the Alpine distribution.

The previous example highlights a case of patch propagation among a family of Dockerfiles. Such situations arise because the impacted Dockerfiles share duplicated instructions. In our example, all the Dockerfiles that are based on the Alpine distribution share the same *make* command with the same arguments (see Listing 4.1 (line 5)). As a consequence, for a patch to be safely propagated, all the impacted duplicates have hence to be identified and then equally modified, which is time consuming and may be error-prone. For example,

1. https://www.docker.com/
2. https://hub.docker.com/_/python/
3. https://github.com/docker-library/python/commit/8717dc2523c8093990cb

this patch applied in 2015 [4] has required another commit the same day to propagate the modification to other Dockerfiles [5].

In this chapter, we therefore tackle the question of duplicates in Dockerfiles by answering the following research questions:

— RQ1: Do Docker official projects maintain families of Dockerfiles, and why?

— RQ2: Do duplicates arise in Dockerfiles families and why?

— RQ3: What are the pros and cons of tools used by experts to manage Dockerfiles?

This chapter is then organized as follows: Section 4.2 presents background regarding Docker. Section 4.3 presents the 99 repositories composing our corpus, our survey methodology and duplicates detection technique. Section 4.4 presents the methodology and results for our three research questions. Section 4.5 presents the threats to validity regarding our experiments. Finally, Section 4.6 concludes this chapter.

4.2 Background

Since its first release in 2013, Docker is now largely adopted by the community of developers with more than 105 billion container downloads [6]. Docker simplifies the deployment of software applications by enabling developers to package their applications with all their required dependencies in a single binary unit, called a *Docker image* [7]. Once created, a Docker image can be downloaded and run on any computer that has Docker installed. Docker images are frequently used by developers of a same team to locally deploy, run and test their application in a controlled running environment. Furthermore, images can even be used as deployment units for end-users that want to run an application without the burden of following a complex installation process on their computer.

A *Docker container* is a "lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings" [8]. Docker containers are instantiated from a *Docker image*. Usually a *Docker image* is maintained by project developers or by Docker developers.

To build an image, developers have to write a *Dockerfile*. A "Dockerfile" is a script that will be executed by Docker to build an *image*. Such a script is written in a Domain Specific Language (DSL) that can be compared to a very limited shell scripting language. The `Dockerfile` language is a DSL maintained by Docker. A Dockerfile is composed of a sequence of *instructions* that are executed in order to produce an image. Listing 4.2 shows a sample Dockerfile that builds an image for a Python application that needs *python* and

4. https://github.com/docker-library/python/commit/f9739c6da575c450aaed8628c1e0bfa97bf1ba18
5. https://github.com/docker-library/python/commit/00c226b82eee61c6c68adf813d9f7177d2efa52a
6. https://www.docker.com/company
7. https://www.docker.com/
8. https://www.docker.com/what-container

apache2 in order to run. When interpreted, this Dockerfile performs the following steps: imports the Python image and uses it as a base, copies the application into the image to be built, installs all required dependencies, configures Docker to open port 80 and execute *app.py* script when launching the image. Docker images are built on top of other images. Therefore, Dockerfiles start by a mandatory `FROM` instruction, that takes as input a *base image*, which is an image name (`python`) and tag (`2.7-slim`).

LISTING 4.2 – A sample Dockerfile for a hypothetical Python application

```
1  # Use the official python base image
2  FROM python:2.7-slim
3  # Set the working directory to /app
4  WORKDIR /app
5  # Copy current directory content into the container at /app
6  ADD . /app
7  # Install dependencies
8  RUN apt-get update && apt-get install -y apache2 \
9  && rm -rf /var/lib/apt/lists/*
10 # Make port 80 available to the world outside this container
11 EXPOSE 80
12 # Define environment variable
13 ENV NAME Hello World
14 # Run "python app.py" when the container is executed
15 CMD ["python", "app.py"]
```

Besides `FROM`, the most common instructions are:

— `WORKDIR` to move the present working directory of the image,

— `COPY` and `ADD` to copy files into the image,

— `RUN` to perform system commands inside the image,

— `EXPOSE` to make available port of the image to the outside world,

— `ENV` to set some environment variables in the image,

— `CMD` and `ENTRYPOINT` to provide the start command runned by the image when instantiated as a container.

Due to the way Docker images are built, each Dockerfile instruction creates a layer (persistent state) inside the image. While some instructions do not have any impact in terms of image size, other instructions such as `RUN` instructions can have a very large impact in the final image size. For these reasons, Docker best practices recommend to use as few RUN instructions as possible. To meet this objective, developers very frequently combine multiple shell commands inside a single `RUN` instruction (as depicted in line 8 of Listing 4.2) using shell delimiters (such as '`&&`' and '`;`'). Such example is depicted in Figure 4.1, we can see that if we follow the recommended best practice, we can save up to 16MB in our yet very simple Dockerfile (from 231MB to 215MB).

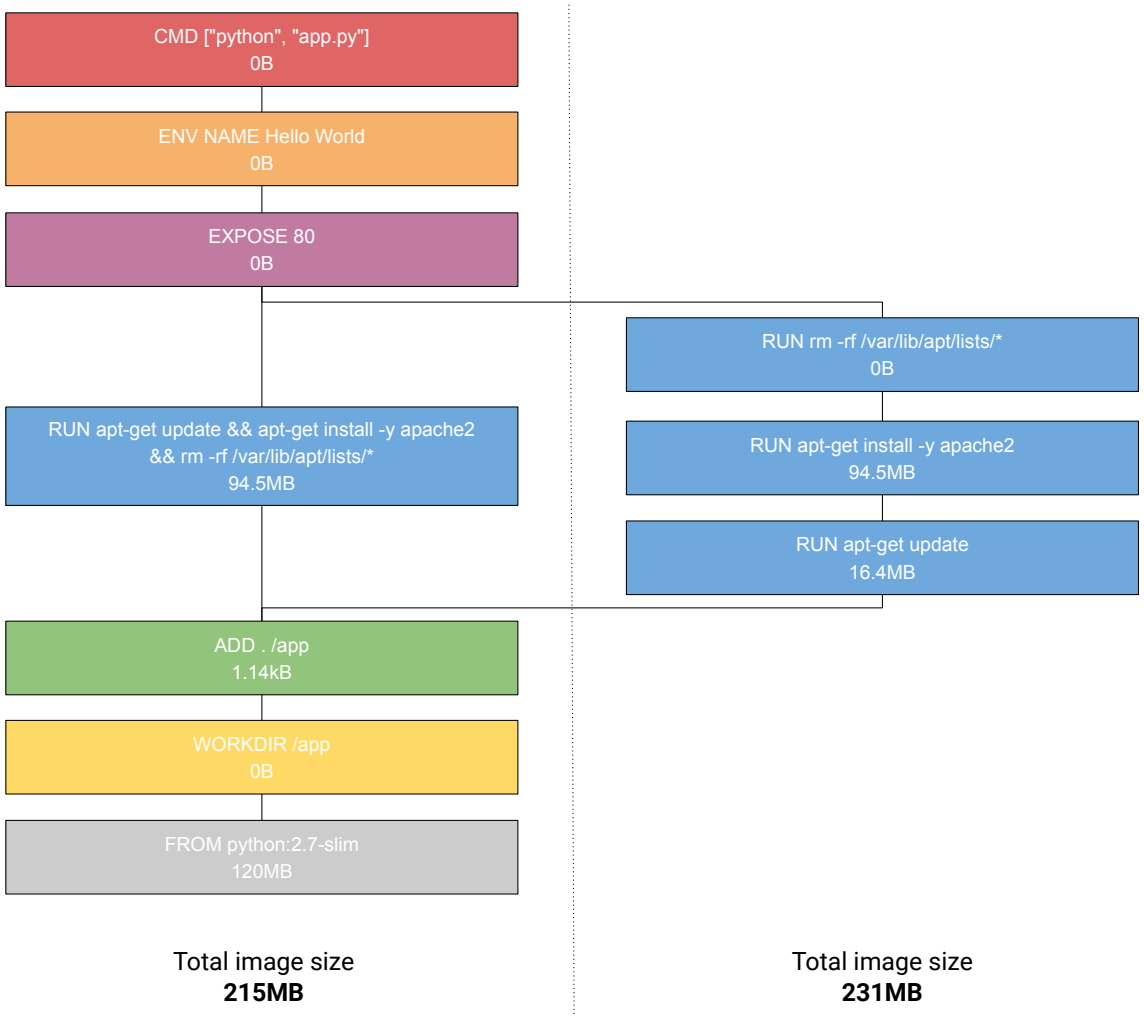

Figure 4.1 – The stack of layers built from the Dockerfile with the corresponding final image size.

In order to help developers, there exist multiple Dockerfile linters that are open-source and implement rules for all best practices recommended by Docker and by the community, such as: fromlatest [9], hadolint [10] and dockerfilelint [11]. There exist event tools that let developers automatically minify their Docker images by statically and dynamically analysing the Dockerfile and generating a new one that uses less space (i.e. docker-slim [12]).

Once a Dockerfile is written, the process to build its corresponding image is straightforward. Developers simply have to execute the Dockerfile using a dedicated Docker command: *docker build . -t image_name*. This command executes the file named `Dockerfile` (and of course written in the Dockerfile language) inside the current shell directory, and creates an image named *image_name*. Developers can then share their images to potential users using a DockerHub repository (or a private Docker registry [13])by performing the following Docker command: *docker push repository/image_name*. DockerHub is a central public *DockerHub* that stores public images [14]. Developers can register to DockerHub and publish their own images, while users can freely access DockerHub to download and use any images. Finally, the users use Docker images to create a running container by performing the following command on their computer: *docker run image_name*.

4.3 Data collection

In this section, we present our three main data sources that are used in the remainder of this study: a list of 99 GitHub repositories each corresponding to a project that manages a Dockerfile family, an on-line survey performed on 25 Dockerfile family maintainers from our 99 repositories and 877 duplicates among the Dockerfile family of our repositories. All repositories, survey questions, duplicates are available on-line [15].

4.3.1 Repositories

We choose in this study to only look at official Docker projects. This choice is driven by the fact that these projects are real-world popular projects that manage medium to large families of Dockerfiles, but most importantly because as it is written in the Docker documentation, these projects promote the best practices to maintain Dockerfiles [16] and may

9. https://www.fromlatest.io
10. https://github.com/hadolint/hadolint
11. https://github.com/projectatomic/dockerfile_lint
12. https://dockersl.im
13. https://docs.docker.com/registry
14. https://hub.docker.com/
15. https://se.labri.fr/a/ICSME19-docker-oumaziz
16. https://docs.docker.com/docker-hub/official_images/

be considered as a reference for any Dockerfile maintainer. The list of these official projects is available on GitHub, with 138 official projects, as of March 2019. [17]

Some of these projects share a same Github repository. For example, InfluxDB and Chronograf are maintained in the same GitHub repository. We therefore remove these 9 projects from our corpus to preserve a dataset uniformity. We also discard the *hello world* project as it is only intended to provide examples for beginners. At this stage, our corpus includes 128 projects with their associated GitHub repositories.

We then count the number of files named `Dockerfile` in each of these 128 projects, as it is the most common name for Dockerfiles (see Section 4.5 for potential threats). We identify 1300 Dockerfiles. Figure 4.2 depicts the number of Dockerfiles maintained by each of these 128 projects. We note that half of the projects maintain at least 4 Dockerfiles with 25% maintaining more than 10 Dockerfiles. As we are interested by Dockerfiles families, we discard 29 projects maintaining a single Dockerfile. Finally, our corpus contains 99 official Docker projects.

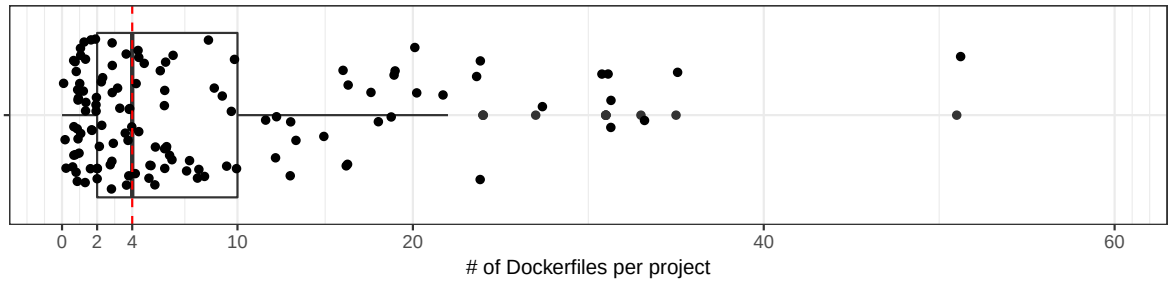

Figure 4.2 – Boxplot for the number of Dockerfiles maintained by each project.

4.3.2 Survey

Our study involves a survey of official Dockerfile maintainers (Dockerfile family maintainers) of projects in our corpus to get their opinions about duplicates and understand their practices. [18] Therefore, we create a survey composed of two main sections asking questions about 1) duplicates and 2) management tools to handle duplicates. In order to retrieve whose contact information, we analysed all commits from all Github repositories of projects in our corpus and extracted the commit author's e-mail address. Using this process, we retrieved about 900 e-mail addresses. Finally, we send a link to our online survey using these addresses and gathered 29 responses, leading to a response rate of about 3% which may indicate that a large set of emails were actually from no longer active official Dockerfile family maintainers or considered our e-mail as spam. Out of these 29 responses,

17. https://github.com/docker-library/official-images/tree/master/library
18. https://forms.gle/sDMkcvSQxnMJwXaA6

```
1  RUN apt-get update && apt-get install -y apache2
```

↓

```
1  RUN apt-get update
2  RUN apt-get install -y apache2
```

Figure 4.3 – RUN instruction with multiple shell commands split into two RUN instructions, one for each shell command.

25 maintainers were actually maintaining more than a single Dockerfile as we address in our study. The 4 remaining maintainers were ignored in these results (they left the survey directly after the first question).

4.3.3 Duplicates detection tool

We consider a Dockerfile duplicate as a sequence of instructions that is duplicated across several Dockerfiles of a same project. We call *size* the number of instructions contained in the duplicate and *owners* the number of files containing the sequence. We provide a tool that inputs a list of git repositories and outputs a list of maximal duplicates found among each repository's Dockerfiles.

As a first step, our tool starts by scanning repositories looking for Dockerfiles and parses them. During the parsing process, we ignore all comments, we identify instructions' types, we extract the instructions' argument's text and normalize their white-spaces (replacing tabs by spaces, removing carriage returns and keeping only one space between two words) to avoid missing duplicates because of indentation differences. As said in Section 4.2, `RUN` instructions often have arguments that are composed of many shell commands. Therefore, to also detect duplicates inside these shell commands, we choose to split these instructions, producing one `RUN` instruction for each contained shell command as seen in Figure 4.3. Instructions are split using the classical `&&` and `;` shell command delimiters and keep the same appearance order as in the original shell command.

Once all Dockerfiles are parsed, we extract the maximum-sized duplicates among each project's Dockerfiles. We use the *index-based duplicate detection* technique [Hummel et al., 2010]. First, for each Dockerfile in our corpus, we extract and index the hashes of all possible chunks of instructions for a given size, as shown in Figure 4.4. The chunks' size goes from the maximal number of instructions contained in the Dockerfile to one which is the smallest granularity: one instruction.

Figure 4.4 shows an example of hashes extracted with the chunk size set at 6. The tuple *(DF, 6, 0, AC4EF..)* indicates respectively: the filename, chunk size, chunk's first instruction location in the file, the chunk's hash. After having indexed all hashes, we apply the algorithm described in [Hummel et al., 2010] to extract the maximum-sized duplicated chunks

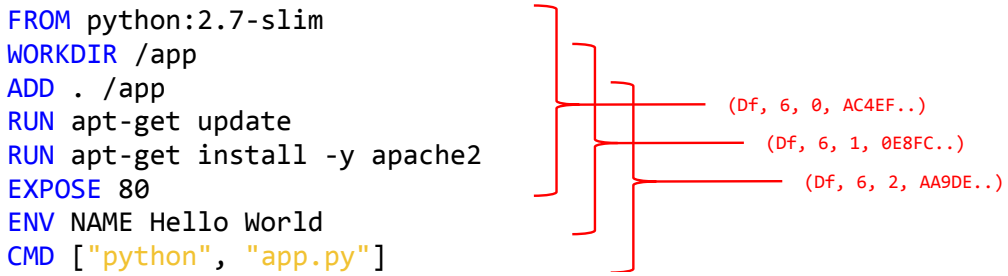

Figure 4.4 – Dockerfile presenting an example of duplicate index with chunk size set to 6.

```
1 FROM alpine:3.6
2 ENV _BASH_GPG_KEY 7C0135FB088AAF6 \
3 C66C650B9BB5869F064EA74AB
4 ENV _BASH_VERSION 3.1
5 ENV _BASH_PATCH_LEVEL 0
6 ENV _BASH_LATEST_PATCH 23
```

Figure 4.5 – Extract of real Dockerfile duplicate from Bash shell v3.1

```
1 FROM alpine:3.6
2 ENV _BASH_GPG_KEY 7C0135FB088AAF6 \
3 C66C650B9BB5869F064EA74AB
4 ENV _BASH_VERSION 4.0
5 ENV _BASH_PATCH_LEVEL 0
6 ENV _BASH_LATEST_PATCH 44
```

Figure 4.6 – Extract of real Dockerfile duplicate from Bash shell v4.0

(duplicated chunks that are not contained in another duplicated chunk). Also, contrary to [Hummel et al., 2010], we don't apply any code normalisation except what was described previously.

Figure 4.5 and Figure 4.6 show two extracts of Dockerfiles from our corpus. They are part of Dockerfiles used to build images for the well-known bash shell (versions 3.1 and 4.0). After applying our technique, we identify two duplicates with two different sizes. The first duplicate has a size of 3, ranging from line 1 to 3, while the second duplicate has the smallest possible size which is 1 corresponding to line 5.

Finally, we apply our tool on each repository from our list, and identify a total of 877 duplicates across all projects. These duplicates will constitute our duplicates dataset for the remainder of this chapter.

4.4 Contributions

In this section, we then present the three main contributions of this study:

4.4.1 RQ1: Do official projects maintain families of Dockerfiles, and why?

Methodology

While looking at Dockerfiles when building our dataset, we note that the location path of a Dockerfile contains information about its purpose. For instance, the `rabbitmq` project has a Dockerfile located in `3.6/alpine/Dockerfile`. Based on this path, we can easily understand that this Dockerfile targets the version `3.6` of `rabbitmq` and is based upon Alpine Linux, a lightweight Linux distribution. Therefore, by analysing all Dockerfiles' paths, we see that `rabbitmq` maintains Dockerfiles for various versions and with various base images.

To perform this analysis on all of our repositories, we follow a semi-automated process. First for each project, we retrieve the path of every Dockerfile. Then, we split all paths using the directory separator symbol. For a given nesting level `i` in the path, we establish the list of all values seen at this level. For instance for the following set of paths: `3.6/alpine/Dockerfile`, `3.7/alpine/Dockerfile` we extract the following set of names: *Level 1: `3.6`*, `3.7`, *Level 2: `alpine`*. For each project, we go through the whole list of extracted names and, for each of them, either creates a new category and associates the name to it, or associates the name to a category that was created for a previous name. Finally, we count the number of elements in each category.

Results

Based on our methodology, we identify four categories. The most represented category is *version* that contains several versions of the project, similarly to the `rabbitmq` example. The second category by order of importance is *base image* that contains the different base images used to produce the project images (e.g. Alpine Linux and Debian Linux). The third category is *flavour* that contains different variants of the project (e.g. normal or multi-threaded flavour for the Perl project [19]). The most seldom category is *platform*, that contains the different Docker platforms supported by the project (e.g. ARM, x86).

Figure 4.7 is an *UpSet* plot [Lex et al., 2014] showing the relationships between versions, flavours, base images and platforms across the projects in our repositories. We first notice that the majority of projects (41) are actually maintaining images for multiple base images and versions at the same time. We also notice that maintaining multiple images only because of multiple versions is very common with 28 projects doing it, while 13 projects

19. https://github.com/Perl/docker-perl/

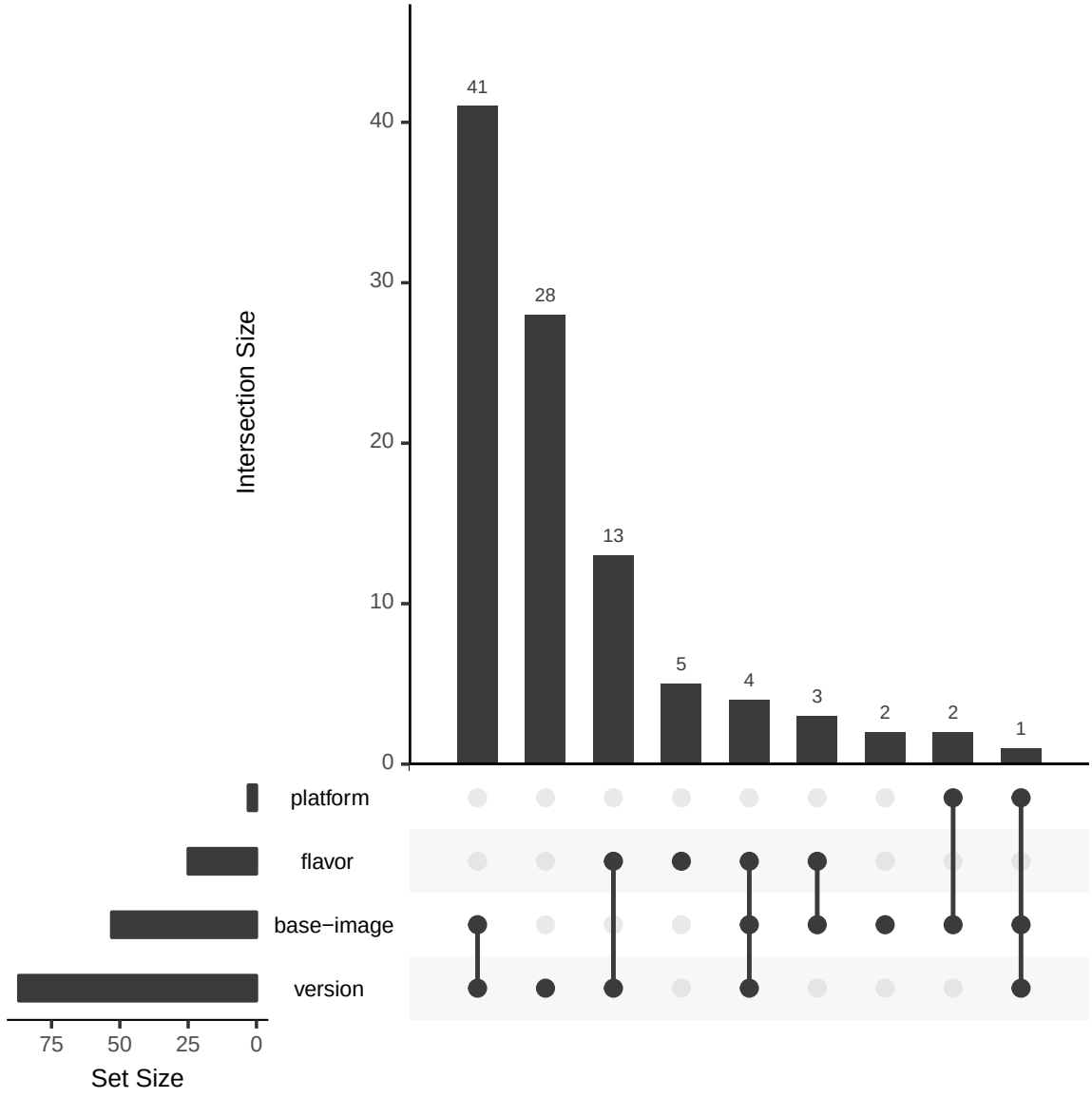

Figure 4.7 – UpSet plot showing the relationships between versions, flavours, base images and platforms across our repositories.

maintain images for multiple versions and flavours.

Summary
The reasons for maintaining several Dockerfiles are to support multiple versions, base-images, flavours and platforms. The most common combinations are version/base-image and version/flavour.

4.4.2 RQ2: Do duplicates arise in Dockerfiles families and why?

To answer RQ2, we report statistics regarding duplicates we've identified with our detection tool and confront them to our survey results. We also take a closer look at the maintenance surrounding these duplicates through a co-evolution analysis. Finally, we manually examine a random subset of duplicates and analyse the reasons behind their existence (Section 4.4.2).

Methodology

As we said in Section 4.3.3, our tool identified 877 duplicates in our repositories. Based on this, we compute several statistics about the characteristics of these duplicates. We also take a closer look at the Dockerfile DSL instructions composing duplicates, especially, which instructions are most commonly concerned by duplicates.

We then evaluate if these duplicates have an impact on Dockerfile family maintenance by analysing if it's common practice for Dockerfile family maintainers to perform a same modification on several Dockerfiles of a same Dockerfile family. To that extent, we look at every commit of our repositories seeking for commits that had two Dockerfiles or more being edited with the exact same modifications (a modification being a sequence of removed code and a sequence of added code, as computed by *diff*).

Duplicates

Figure 4.8 depicts a violin plot for respectively: the percentage of duplicate instructions per project (upper-left) and the number of instructions per project (upper-right). We notice that 75% of projects have more than 50 instructions (upper-right) and nearly half of these instructions that are duplicated (upper-left). Half the projects have more than 83% of duplicate instructions (half the projects having at least 117 instructions). All of this indicates how frequent duplicates are in Dockerfile family. When asking Dockerfile family maintainers of our repositories if they have faced duplicates in the past: 68% (17 out of 25) said that they did, while 20% said they've never faced duplicates and 12% didn't know.

Figure 4.8 (bottom) presents a violin plot for the number of instructions for each identified duplicate. We notice that duplicates can be quite small, with half of them having less

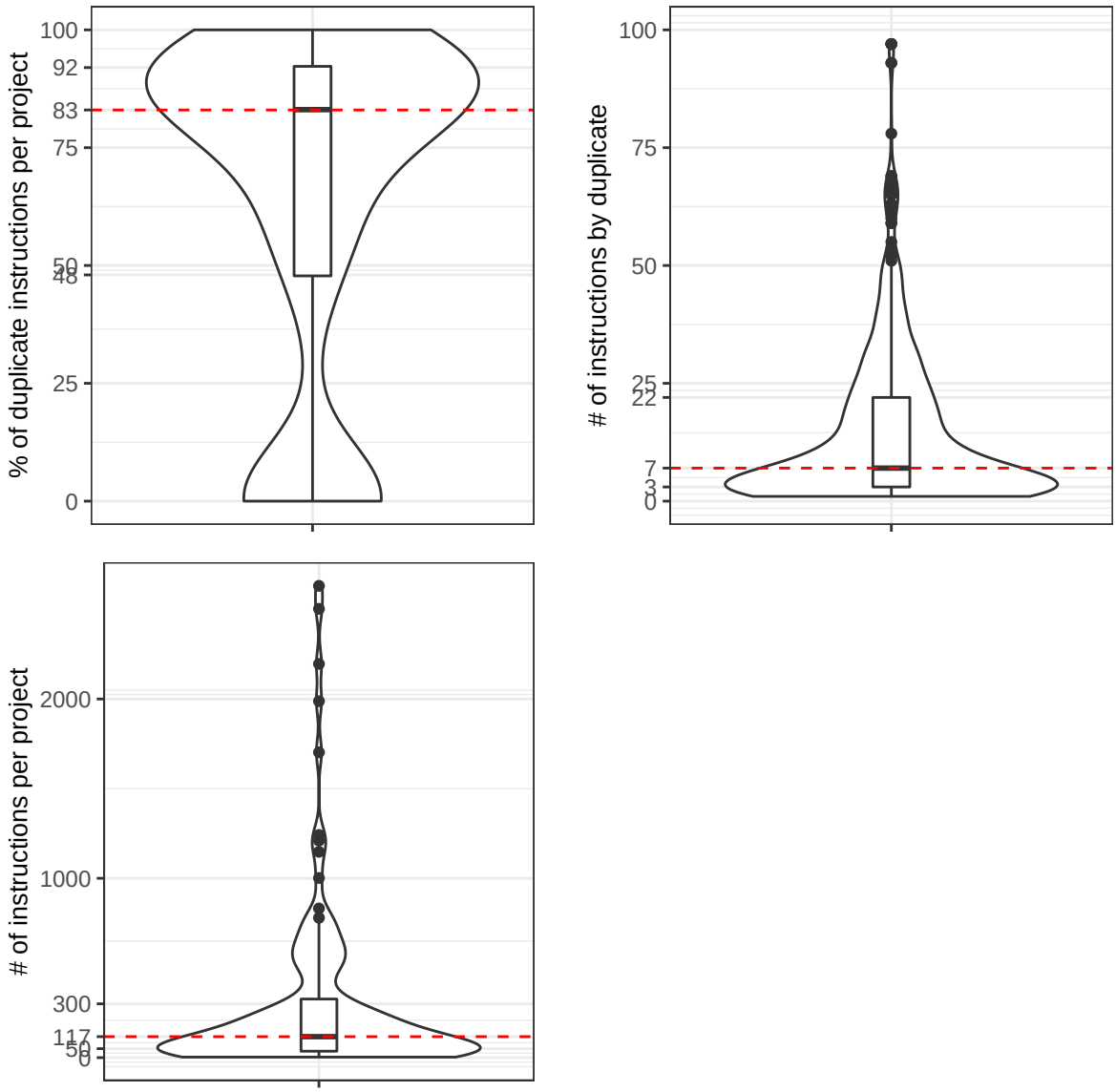

Figure 4.8 – Upper-left plot: Violin plot for the percentage of duplicate instructions per project. Upper-right plot: Violin plot for the number of instructions per project. Bottom plot: Violin plot for the number of instructions by duplicate.

than 7 instructions. However, we also note that 25% of duplicates we've identified have more than 22 instructions. Finally, we remind that we chose to split *RUN* instructions as described in Section 4.3.3, which could artificially increase the number of instructions in blocks of duplicates. Also, when we ask all Dockerfile family maintainers: *What should be*

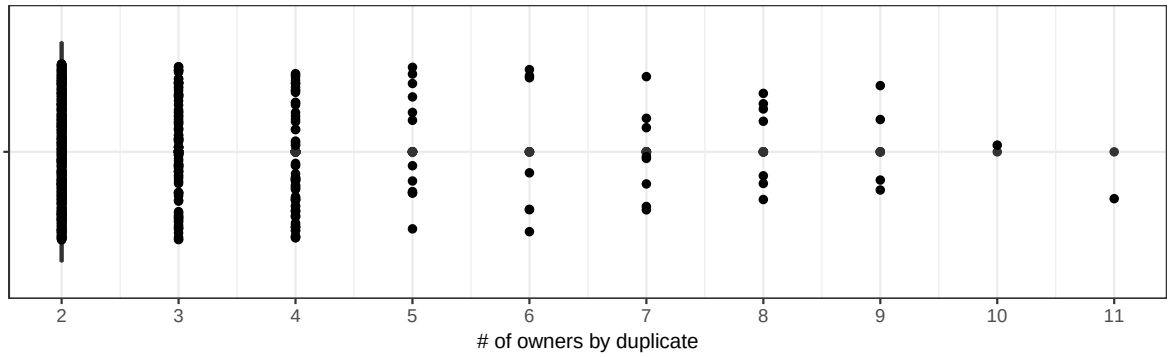

Figure 4.9 – Stripplot of the number of owners of every duplicate in our corpus.

the minimal size for a duplicate bloc of instructions to be detected?, the majority of Dockerfile family maintainers (15 out of 25) stated it should be between 1 and 5 instructions which indicates that even duplicates composed of a single instruction shouldn't be ignored.

Figure 4.9 is a stripplot depicting the number of owners (i.e. the number of files containing the duplicate) of every duplicate in our repositories. We notice that the large majority of duplicates has around 2 and 4 owners. While duplicates having 5 to 11 owners are less common. When we ask Dockerfile family maintainers for the minimal number of files sharing a duplicate (owners) in order for that duplicate to be detected, 13 out of 25 Dockerfile family maintainers stated that it should be 2. These responses confirm that the minimal thresholds we had set previously in Section 4.3.3 were actually corresponding to the Dockerfile family maintainers needs.

Also, if we look at the ratio of duplicated instructions over the total number of instructions for every instruction provided by the Dockerfile DSL, we find that the high majority of instructions available in our corpus are duplicated (32,269 out of 37,319). The *RUN* instructions being the most frequent instructions in our corpus by far (26,053 instructions) where 86% of them are duplicated. The second most frequent instructions are *ENV* instructions (3,762 instructions) with more than 78% of them being duplicated. Followed by *FROM* instructions (1,372 instructions) with 79% of them being duplicated.

Finally, while our tool identifies 877 duplicates, 64% of all Dockerfile family maintainers from our survey (16 out of 25) said that they don't need a tool to detect duplicates. They state that duplicates are easy to find and that they don't want to use another tool. Therefore, the need for a detection tools for Dockerfile duplicates isn't as important as it's the case with duplicates in programming languages [Kamiya et al., 2002; Baxter et al., 1998; Bellon et al., 2007a], however, 9 maintainers are nonetheless asking for such tools to help them ease the maintenance process.

Co-evolution

Figure 4.10 depicts a violin plot for the percentage of co-evolving commits per project in our corpus. We can see that 50% of all projects have 14% of all their commits propagating a same edit across several Dockerfiles. This number can go up to more than 29% for 25% of projects in our corpus.

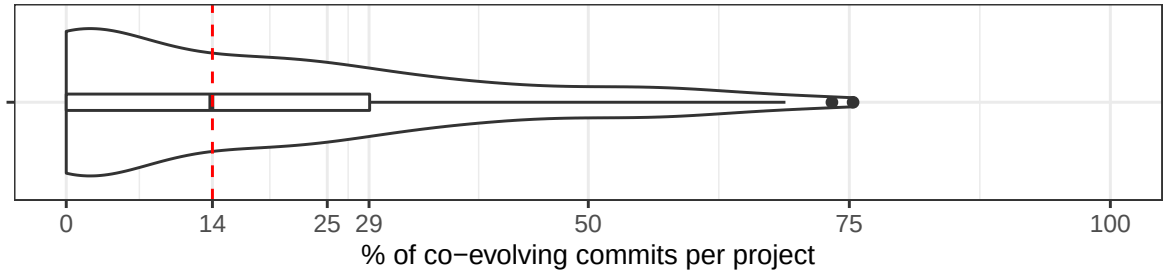

Figure 4.10 – Violin plot for the percentage of co-evolving commits per project.

Also, 94,1% of Dockerfile family maintainers who encountered duplicates (16 out of 17) stated that they had performed identical changes across multiple Dockerfiles in a single commit in the past. This confirms that keeping consistency across duplicates is a classical task performed by maintainers. Half of them (8 out of 16) qualified these consistency updates as being annoying. 4 out of 16 Dockerfile family maintainers felt that these consistency updates can be error-prone. However, others (6 out of 16) qualified them as being easy to perform since they can be automated through templates and other techniques as we'll see in Section 4.4.3.

Summary

Duplicates in Dockerfile family are frequent. While they're usually small and span across only few Dockerfiles, they can also sometimes be large and span across many Dockerfiles. Dockerfile family maintainers of our corpus frequently have to propagate identical changes to multiple Dockerfiles. They also have a mixed opinion about these changes. Dockerfile family maintainers not using any tool find them error-prone and annoying, while Dockerfile family maintainers using dedicated tools find them easy to perform.

Reasons for duplicates in Dockerfiles

In this section, we aim at understanding the underlying reasons behind duplicates we've identified earlier.

Methodology

For this study, we take a random sample of 50 duplicates from our duplicates corpus. We limited the size of this random selection to only 50 duplicates, because analysing a duplicate takes a long time. Three of the authors, external to the Dockerfile family maintainers of our corpus, review each duplicate and tag them with what they think is their corresponding reason (see Section 4.5 for potential threats). Note that a single duplicate can have multiple underlying reasons.

Results

In the remainder of this section, we describe the main reasons behind duplicates.

Software installation and configuration. The installation process of software can be identical across multiple images, leading to duplicate instructions. The process can be composed of different steps going from downloading the software and installing it, to checking its signature, etc. Once the software is installed, the next step in this process is to configure the software before running it.

LISTING 4.3 – Duplicate due to identical software installation process

```
1  ENV XWIKI_VERSION=8.4.5
2  ENV XWIKI_URL_PREFIX "http://maven.xwiki.org/.../${
       XWIKI_VERSION}"
3  ENV XWIKI_DOWNLOAD_SHA256 52ed122c44984748a729a78 \
4   4c94cb70ccf0d2fa34c2340d0fd45c75deb3b0bc9
5  RUN rm -rf /usr/local/tomcat/webapps/* && \
6   mkdir -p /usr/local/tomcat/temp && \
7   mkdir -p /usr/local/xwiki/data && \
8   curl -fSL "${XWIKI_URL_PREFIX}/xwiki-ent-web-${XWIKI_VERSION
       }.war" -o xwiki.war \&& \
9   echo "$XWIKI_DOWNLOAD_SHA256 xwiki.war" | sha256sum -c - &&
       \
10  unzip -d /usr/local/tomcat/webapps/ROOT xwiki.war && \
11  rm -f xwiki.war
```

Listing 4.3 is an extract of a whole duplicate (11 lines) that we manually analysed. This duplicate involves two files from the *XWiki* project. It first starts by defining some environment variables (lines 1 to 4), then cleans a folder, and creates other ones that are mandatory for running XWiki (5 to 7). It then downloads the software and puts it in the right folder, validates it, unzip it and deletes the downloaded file. 50% of duplicates we analysed were due to software installation and configuration.

Dependency management. Dependencies also follow an installation process. Before installing dependencies, package managers need to be configured. After having installed the dependencies, it is sometimes necessary to configure them. Finally, the list of dependencies that needs to be installed may vary across images, but most dependencies are identical.

LISTING 4.4 – Duplicate due to package manager configuration

```
1 RUN apt-key adv --keyserver pgp.mit.edu \
2 --recv-keys 1614552E5765227AEC39EFCFA7E00EF33A8F2399
3 RUN echo "deb http://download.rethinkdb.com/apt xenial main"
    \
4 > /etc/apt/sources.list.d/rethinkdb.list
```

Listing 4.4 is an extract of a duplicate caused by the package manager's configuration to be able to download the dependency. It involves three Dockerfiles from the rethinkdb project [20].

Listing 4.5 shows a duplicate due to the installation of identical dependencies. In this extract, the package manager starts by updating its list of packages (line 1). Then, it downloads a bunch of dependencies, and finally, it cleans the package list it previously downloaded in order to reduce the final image size. It involves 2 files of the bonita project [21].

It is common to see the underlying shell commands combined in a single RUN instruction as we explained in Section 4.2. We encountered this reason for 40% of duplicates we analysed.

LISTING 4.5 – Duplicate due to dependency manager installation

```
1 RUN apt-get update
2 RUN apt-get install -y mysql-client-core-5.7 openjdk-8-jre-
    headless postgresql-client unzip curl zip
3 RUN rm -rf /var/lib/apt/lists/*
```

Runtime configuration. These duplicates arise because developers set-up a same way of running the container in several images. Indeed, while writing the Dockerfile, it's possible to configure some parameters for the container runtime. For instance, the instruction *ENTRYPOINT* lets developers configure a shell command that will be run inside the image when instantiated as a container. For example, a bash image when started, will directly run the bash binary. There are also instructions such as *VOLUME* for specifying how to mount a folder from the user computer into the container, instructions for specifying the working directory, etc..

Listing 4.6 shows a duplicate due to runtime configuration. In this extract, the Dockerfile specifies where a volume should be mounted (line 1), what command will be used as

20. https://github.com/rethinkdb/rethinkdb-Dockerfiles
21. https://github.com/Bonitasoft-Community/docker_bonita

an entrypoint (line 4) and what command should be used (line 5). It involves 2 files from the *spiped*[22] project. We encountered this reason in 26% of duplicates we analysed.

LISTING 4.6 – Duplicate due to runtime configuration

```
1 VOLUME /spiped
2 WORKDIR /spiped
3 COPY *.sh /usr/local/bin/
4 ENTRYPOINT ["docker-entrypoint.sh"]
5 CMD ["spiped"]
```

Summary
Duplicate instructions are mainly due to software installation and configuration (50%), dependency management (40%) and runtime configuration (26%).

4.4.3 RQ3: What are the pros and cons of tools used by experts to manage Dockerfiles?

Methodology

To answer our third and final research question, we manually analyze each repository in our corpus to determine if what we call a *Dockerfile management tool* has been used. We define as *Dockerfile management tool* any tool that helps the maintenance of Dockerfiles. To that extent, we manually search in all 99 projects in our corpus looking for scripts or binaries that could be used as tools to manage Dockerfiles. We then cluster theses scripts and binaries and identify 3 main Dockerfile management tool categories. We perform a thorough analysis to describe each one of these categories (Section 4.4.3). We also report our survey's answers relative to the use of tools. Finally, we discuss the tools with regard to their capabilities to handle change propagation, highlighting their advantages and their limitations (Section 4.4.3).

Dockerfile Management Tools

We find that 66 projects in our repositories use tools to manage their Dockerfile family (66% of our corpus) . We notice that there is no on-the-shelf tool for this purpose: all developers maintain their own tool. However, all management tools have an *update script file* (located in the root of the repository) that is usually named `update.sh`, which is responsible for updating the whole Dockerfile family. The update script needs some input parameters to generate the Dockerfiles, such as the version number or the target base image. 75% of them receive the parameters from the command-line while 25% automatically

22. https://github.com/TimWolla/docker-spiped

fetch parameters from a website. Finally some of the latter projects even automate the execution of the `update.sh` script at regular intervals via a dedicated bot. We notice that 88% of these scripts are written in *Bash*, the remaining 12% are written in different languages such as *Go*, *PHP*, *Makefile*, *Python*, *Perl* and *Groovy*.

Further, when we ask Dockerfile family maintainers in our survey if they use tools to handle duplicates, 56% (14 out of 25) said they actually don't, because it is too much time consuming or difficult to implement. When we ask them: *Would you like to have a tool built to avoid duplicates in Dockerfiles?*, about 57% (8 out of 14) said they would like to have one, which confirms the usefulness for management tools. The remaining ones (6 out of 14) said no because they don't want to use another tool or don't see the need.

However, among the remaining 44% (11 out of 25) who are using tools, 9 replied they were using the tools that we'll present in this section, and 2 replied that they were using git branches and multi-stage builds as management tools for their Dockerfiles. 7 out of the 11 Dockerfile family maintainers using tools said that they were satisfied with their tools, the remaining 4 weren't satisfied with their tool. However, by analysing the answers, we realized that all maintainers use multiple different tools depending on the project. Therefore, we could not use the answers of Dockerfile family maintainers to directly pinpoint their favorite tools.

We now present the main categories of Dockerfiles management tools we encountered.

Template processor. They are the most frequent tools in our corpus (54%) and are used by 8 out of 11 maintainers in our survey. Template processors use a template, some input data, and a template engine. This kind of tools are also widely used to generate web pages [Parr, 2004; Tatsubori and Suzumura, 2009]. When invoked, the template engine injects the input data into the templates to generate the outputs.

The most classical templates used in our corpus are just plain Dockerfiles containing several placeholders as done in Listing 4.7. This listing presents an extract of the template used to generate the Dockerfiles of the Python project. We first notice that the base image is defined by a text replacement and a variable (line 1). The GPG key and Python version that needs to be installed use the same features (text replacement and a variable (lines 2 and 3)).

LISTING 4.7 – An extract of the Python Dockerfile template

```
1 FROM debian:%%PLACEHOLDER%%
2 ENV GPG_KEY %%PLACEHOLDER%%
3 ENV PYTHON_VERSION %%PLACEHOLDER%%
```

Only a few projects use more advanced template technologies supporting advanced features such as inclusion of sub-templates, loops and conditional statements: only two projects implement sub-templates, and only one project supports conditional statements. For instance, the *XWiki* project uses the most advanced template language we observed.

Listing 4.8 shows an extract of this project's template. Among the list of XWiki's dependencies, developers must install either *mysql* or *postgres* according to the desired image flavour In the example, we see that a conditional statement is used (coming from Groovy's templating language) to handle this case (lines 7 and 8). Regardless of the template language, projects using templates all perform the template rendering inside the `update.sh` script.

LISTING 4.8 – An extract for the XWiki Dockerfile template

```
1 RUN apt-get update && \
2 apt-get --no-install-recommends -y install \
3 curl \
4 libreoffice \
5 unzip \
6 procps \
7 <% if (db == 'mysql') print 'libmysql-java'
8 if (db == 'postgres') print 'libpostgresql-jdbc-java' %> && \
9 rm -rf /var/lib/apt/lists/*
```

Find and replace. *Find and replace* tools are also fairly common Dockerfile management tools, and are used by 36% of the projects in our corpus and by 2 out of 11 maintainers in our survey. These tools proceed to update all Dockerfiles present in the repository by directly updating some of their content using the input data. Therefore, previous Dockerfiles present in the repository are overwritten by the updated ones.

Such Dockerfile management tools mainly use regular expressions or dedicated Unix tools such as *sed*, located directly in the `update.sh` script. For instance, Listing 4.9 depicts a real extract from Kibana project's update script. We see that the values at the right of *KIBANA_MAJOR*, *KIBANA_VERSION* and *KIBANA_SHA1* are replaced by the value contained in the variables passed as input parameters to the update script.

LISTING 4.9 – An extract of the Kibana update script

```
1 sed -ri '
2 s/^(ENV KIBANA_MAJOR) .*/\1 '"$version"'/;
3 s/^(ENV KIBANA_VERSION) .*/\1 '"$fullVersion"'/;
4 s/^(ENV KIBANA_SHA1) .*/\1 '"$sha1"'/;
5 ' "$version/Dockerfile"
```

Generator. *Generators* are the least frequent Dockerfile management tool, only used by 10% of the projects in our corpus. However, they are used by 8 out of 11 Dockerfile family maintainers in our survey. They consist of a single update script that generates all Dockerfiles with their content using a shell language (bash for 4 out of 5 projects) or a general-purpose language (perl for one project). These tools leverage on features offered by their

host language and therefore provide many features (variables, loops, conditional evaluation, functions, ...).

Listing 4.10 is an extract from the OpenJDK project's update script (written in bash). In this example, we see that a loop is used to generate the Dockerfiles for all versions of OpenJDK. Additionally, a conditional evaluation is used to add some extra dependencies if they are available for the target base image. Tools using generators place all the generator's code directly in the `update.sh` script.

LISTING 4.10 – An extract for the OpenJDK Dockerfile generator

```
1 for version in "${versions[@]}"; do
2 ...
3 if [ "$addSuite" ]; then
4 cat >> "$version/Dockerfile" <<-EOD
5 RUN echo 'deb http://deb.debian.org/debian $addSuite main' >
     /etc/apt/sources.list.d/$addSuite.list
6 EOD
7 fi
8 done
```

Summary

Many projects use tools to handle duplicates. They fall into three categories: find and replace, template processors and generators. Several Dockerfile family maintainers stated that such tools can be too much time consuming or difficult to implement, and thus are not using them. However Dockerfile family maintainers using tools are mostly satisfied with them.

Discussion

In this section, we review how each kind of Dockerfile management tool enables developers to handle duplicate code. In order to discuss the pros and cons of each tool, we start by taking a closer look at why Dockerfile family co-evolve and therefore what types of changes Dockerfile family maintainers are propagated on Dockerfiles. To do so, we randomly select a sample of 50 commits having several Dockerfiles co-evolving from all co-evolving commits previously extracted in Section 4.4.2. We then manually look at each commit and determine the reason behind each co-evolution. As a result, we find two main types of changes: version update (28 out of 50) that are similar to the change shown in Listing 4.11 and other changes (22 out of 50), such as bug-fixes or refactorings, similar to the change shown in Chapter 1 (Listing 4.1). Version updates are predictable changes regularly performed at a same location, while the other changes are arbitrary and can span across multiple lines. We use these two categories of propagated changes to discuss the pros and cons of each category of tools in the remainder of this section.

LISTING 4.11 – An extract of the dicussed version number update across the whole Dockerfile family

```
1 ENV PYTHON_PIP_VERSION 19.0.2 19.0.3
```

Find and replace. *Pros.* Find and replace tools handle version update propagation very well. Since the location of changes is known in advance, it is easy to build a regular expression or a sed command that automates them. In addition, it is very easy to set-up a find and replace tool, since it only requires to write a script that does nothing more than applying the sed command or the regular expression when called.

Cons. On the other hand, find and replace tools are not adapted to other changes than version updates. Indeed, these changes are usually only applied once on the code base, therefore there is no use for defining and storing a regular expression or a sed command to perform it. In case of such changes, developers have to find all Dockerfiles containing the duplicate sequence of instructions and apply the fix manually.

Template processor. *Pros.* Template processors are capable of handling the two change propagation scenarios. For both scenarios, it is sufficient to apply the change on the templates containing the concerned sequence of instructions, and re-generate the Dockerfiles. We note that projects using template processors and maintaining Dockerfiles for only one reason, usually write only one template, thus eliminating all possibles duplicates.

Cons. Projects using template processors and maintaining a Dockerfile family for more than one reason write multiple templates (only four projects out of 26 in this case managed to write only one template). When we run the detection tool we used previously in Section 4.3.3 on the templates of projects using multiple templates (23 projects out of 35 projects using templates), we still find 95 duplicates in the templates.

Figure 4.11 presents statistics about these projects. The left figure is a Violin plot showing the percentage of duplicate instructions across Dockerfiles of projects using templates The right figure shows a Violin plot for the percentage of duplicates reduction in projects using templates. We can clearly see that the use of templates can help reduce the number of duplicates that a Dockerfile family maintainers has to manage with a median at 31% reduction. However, duplicates are still not fully eliminated even when using templates, therefore Dockerfile family maintainers still have to propagate some changes manually.

Generator. *Pros.* Similarly to the template tools, generator tools are capable of handling both change propagation scenarios. Indeed, the five projects using these tools have a single file in their repository that generates all Dockerfiles, thus eliminating all duplicates. By looking at the source code of their generator, we found out that the key features were: text replacement, loops and conditional evaluation (as we can see in Listing 4.10). To propagate a change, the Dockerfile family maintainers have to locate the concerned location in the generator's code, perform the change, and regenerate all Dockerfiles.

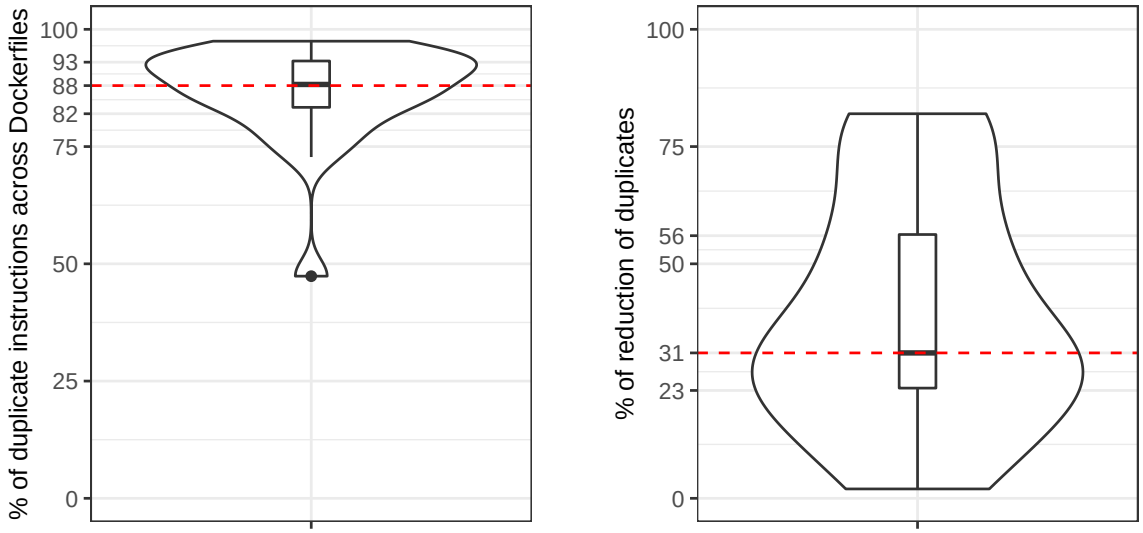

Figure 4.11 – Left plot: Violin plot for the percentage of duplicate instructions in Dockerfiles of a project using Templates. Right plot: Violin plot for the percentage of duplicates reduction in projects using templates.

Cons. The only downside we found with these tools is that the content of the generator (usually a shell script) is very cumbersome. Indeed, while Dockerfile family maintainers using templates write and read code that is very similar to a Dockerfile, Dockerfile family maintainers using generators write Dockerfiles with a totally different language. Additionally, it is far from obvious to understand the content of Dockerfiles that will be generated from it.

Summary
Find and replace tools handle very well the propagation of version updates but aren't adapted for other changes. Template tools are capable of handling the two change propagation scenarios but still don't fully eliminate duplicates. Generators are also capable of handling the two change propagation scenarios, they do fully eliminate duplicates but are much less readable than templates.

4.5 Threats to validity

We discuss here the threats to validity of our study following the guidelines provided by Wohlin et al. [Wohlin et al., 2012].

Construct validity. In Section 4.3.3, while our tool managed to identify identical duplicates in our corpus, measurement errors could possibly have been introduced by our duplicates detection tool. A mitigation to this is the fact that we chose to identify only identical clones (equivalent to type-I code clones) which are less prone to not be detected.

Internal validity. In Section 4.3.1, we look for Dockerfiles only based on their filename while supposing that the name is `Dockerfile`. We also assumed in Section 4.4.1 that the reason behind the existence of a Dockerfile is encoded in its path. Of course, it is possible that some projects use alternative naming and location schemes and this would bias our results. To ensure the validity of our hypothesis we manually inspect a random subset of 20% of projects and don't find any counter-example.

In Section 4.4.2, while the sample size is consequent, it isn't large enough to be representative of the whole duplicates corpus. Further, duplicates' reasons tagging is done by the study's authors who aren't experts of the projects containing the duplicates. Moreover, the process of tagging a duplicate is very subjective which could bias the tagging results. Finally, while the three authors tried to mitigate the misidentification threat by concerting each other, a more formal approach should be taken in a future study replication work.

In Section 4.4.3, Dockerfile management tools are identified by two authors of the study. While strategies aren't that similar, meaning less subjective to identify, the two authors concerted each other to mitigate the misidentification threat. We attempt to provide all the necessary details to replicate our study and analysis. We also provide all data involved in our study to enable replication and scrutiny of our results [23].

External validity. This study focuses only on mature and official projects, corresponding to the 99 projects in our repository. This is on purpose, as these projects are defined by Docker as implementing all best practices. Therefore, the projects may be more maintained and have larger Dockerfiles than more common projects meaning that the results we got may not be representative of the whole Docker ecosystem but to a rather more advanced category of projects.

In Section 4.3.2, the survey gathered 25 responses from Dockerfile family maintainers (3% response rate). While this number isn't large, the fact that they are experienced Dockerfile family maintainers of official Docker repositories means that their answers can be considered of high-quality. Also, since we are aiming at only official repositories Dockerfile family maintainers, the answers we gathered could be biased towards a more advanced category of Dockerfile maintainers.

Conclusion validity. A threat to our study concerns the reliability of the metrics computed through our experiments. We did follow a semi-automated process for all measurements in RQ1, however we followed a fully manual approach in RQ2 and RQ3.

23. https://se.labri.fr/a/ICSME19-docker-oumaziz

Another threat to our study is the fact that we didn't statistically validate our results through non-parametric statistics (e.g. Mann-Whitney test).

4.6 Conclusion

In this chapter, we answered questions regarding duplicates handling in Dockerfiles families by providing a grounded study based on the analysis of the official Docker projects. We showed that official Docker projects frequently maintain families of Dockerfiles and found the underlying reasons: supporting multiple versions/base-images and versions/flavours. We then showed that duplicates in Dockerfiles are abundant, and found the underlying reasons behind them: software installation and configuration, dependency management and runtime configuration. We also performed a survey on Dockerfile family maintainers of official projects and found that they are aware of their existence and are frequently facing them. However, Dockerfile family maintainers have a mixed opinion regarding them. While Dockerfile family maintainers not using tools for handling duplicates state that their handling may be error-prone, Dockerfile family maintainers using tools state that they are easy to maintain. We also found that some Dockerfile family maintainers handle duplicates by using ad-hoc tools: template processors, code generators, find and replace executors. Finally, we showed that projects using template processors and code generators manage to reduce the amount of duplicates with a median at 30% up to 100% for generators.

CHAPTER

5 Conclusion

Software maintenance and evolution represents an important part of the software's development life-cycle, making up to 80% of the overall cost and effort [Alkhatib, 1992]. During the maintenance effort, it happens that developers have to resort to copying and pasting source code fragments in order to reuse them. Such practice, seemingly harmless is more frequent than we expect. During the last two decades, many studies have tried to understand the underlying reasons for developers to resort to code clones [Baxter et al., 1998; Kapser, 2009; Rieger, 2005], and multiple code clone detectors have been released [Baxter et al., 1998; Kamiya et al., 2002; White et al., 2016a]. Through this thesis, we investigate the existence of clones beyond source code in two types of software artifacts: API documentation and deployment files (i.e. Dockerfiles). To do so, we answered the following research questions:

— RQ1: Do developers often resort to copy-pastes?

— RQ2: Why do developers resort to perform copy-pastes?

— RQ3: Could duplicates be avoided by a proper usage of existing tools?

5.1 Summary of contributions

In this section, we propose a summary of the two contributions discussed in this thesis regarding two specific types of artifacts: API documentation and Dockerfiles.

5.1.1 Duplicates in API documentation

Our first contribution, presented In Chapter 3, aims at investigating the issue of duplicates in API documentation. In order to perform such investigation, we apply an empiri-

cal research methodology on a corpus composed of 100 open-source project repositories maintaining API documentation (50 using Java programming language and 50 using Ruby programming language). We also perform a survey on 39 developers using either Java or Ruby and gather their opinions regarding duplicates in API documentation.

As a result, we first show that duplicates of documentation tags are abundant. Secondly, we then manually analyse these duplicates and identify that they are caused by five different kinds of relationships in the underlying source code: (1) delegation: when a method calls another one, (2) sub-typing: when a method performs a computation that is similar to another method, (3) code clone, (4) similar intent: when a method overrides another one that is defined in a same hierarchy and (5) similar use: when two methods or more share a common input parameter, output or exception but each method's behavior is totally different. Thirdly, we find that developers responding to our survey, often resort to make API documentation duplicates co-evolve, and find such practices as being annoying and error-prone. They also ask for documentation tools that can help them avoid such duplicates. Finally, through our study, we pinpoint the fact that it is common for documentation tools to not provide reuse mechanisms to cope with these causes. However, one tool (i.e. Yard) provides a mechanism that handles most duplicates causes that we've identified in our study.

This research work was published at the *16th International Conference on Software Reuse (ICSR 2017)* [Oumaziz et al., 2017].

5.1.2 Duplicates in Dockerfiles

Our second contribution, presented In Chapter 4, aims at providing practitioners a clear explanation for why Dockerfile duplicates arise in projects, and what are the different means to handle duplicates with their pros and cons. To do so, we apply a grounded study based on the analysis of the official Docker projects (99 open-source projects) which are real-world popular projects that manage medium to large families of Dockerfiles and which promote the best practices to maintain Dockerfiles. We also perform a survey on official Docker projects' maintainers and gather their opinions regarding duplicates on Dockerfiles.

As a result, we first show that official Docker projects frequently maintain families of Dockerfiles and find the underlying reasons: supporting multiple versions/base-images and versions/flavours. Secondly, we then show that duplicates in Dockerfiles are abundant, and find the three underlying reasons behind them: (1) Software installation and configuration, (2) dependency management and (3) runtime configuration. Thirdly, we find that official Docker project maintainers are aware of the existence of duplicates in the Dockerfiles they maintain and are frequently facing them. However, they have a mixed opinion regarding them. While maintainers not using tools for handling duplicates state that their handling may be error-prone, maintainers using tools state that they are easy to maintain. We also find that some maintainers handle duplicates by using ad-hoc tools:

template processors, code generators, find and replace executors. Finally, we show that projects using template processors and code generators manage to reduce the amount of duplicates with a median at 30% up to 100% for generators.

This research work was published at the *35th IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)* [Oumaziz et al., 2019].

5.2 Perspectives and discussion

Software artifacts beyond code contain duplicates just as the source code does. In this section, we present several perspectives for extending the research work presented in this thesis, and discuss duplicates in software artifacts a more broader perspective.

5.2.1 Duplicates in API documentation

In our first contribution, presented In Chapter 3, as we've stated in our threats to validity (Section 3.5), we applied an empirical research methodology on a corpus composed of 100 open-source project repositories maintaining API documentation (50 using Java programming language and 50 using Ruby programming language). Therefore, our study focused only on open-source projects with the largest number of stars on Github, corresponding to the 100 projects in our repository meaning that the results we got may not be representative of all projects having an API documentation. In this sense, it would be interesting to further investigate the existence and underlying causes of clones on projects which aren't open-source such as industrial projects and even projects using other programming languages to determine if the issues we've identified are actually common in other situations.

Another interesting perspective is to investigate the existence of duplicates on other types of API documentation. In our work, we focused only on code APIs documentation, one could take a closer look on Web API documentation, or in this case, specification, such as OpenAPI specifications for instance. OpenAPI specification is an API description format for REST APIs. Such specification is written in a JSON or XML file and is used to describe an entire API such as the different API endpoints, input parameters and output, HTTP method type, etc. These API descriptions, such as parameter descriptions for instance, can therefore be duplicated just as we've previously seen with methods having their input parameters being duplicated.

Finally, until now, we've only taken a closer look at exact API documentation duplicates, an interesting perspective could be to take a closer look at similar duplicates. What we mean by similar duplicates are sentences which are considered as duplicates based on their meaning rather than a textual similarity. One could use more recent Natural Language Processing (NLP) approaches relying on deep learning techniques such as word embeddings (e.g. Word2Vec) and document embeddings (e.g. Doc2Vec) to better identify sen-

tence similarity. However, such approach might have to be semi-automated, as identified duplicates might be false-positives.

5.2.2 Duplicates in Dockerfiles

In our second contribution, presented In Chapter 4, as we've stated in our threats to validity (Section 4.5), we applied a grounded study based on the analysis of the official Docker projects (99 open-source projects) which are real-world popular projects that manage medium to large families of Dockerfiles and which promote the best practices to maintain Dockerfiles. Therefore, through our study we might have focused on projects that may be more maintained and have larger Dockerfiles than more common projects meaning that the results we got may not be representative of the whole Docker ecosystem but to a rather more advanced category of projects. Thus, extending our initial study to a broader set of projects maintaining Dockerfiles can be interesting, as not all projects necessarily maintain families of Dockerfiles.

Based on the feedback that we've gathered from our survey of official Docker project maintainers, another interesting perspective is to investigate the possibility to have a tool that could out-of-the-box identify duplicates in a family of Dockerfiles and automatically provide templates avoiding as much duplicates as possible. The tool could go even further in its detection technique and also detect some similar sets of Dockerfile instructions such as detecting for instance: a set of *RUN* instructions installing a identical but differently ordered list of dependencies through a dependency manager. Such a tool could even extend the existing Dockerfile build engine (through BuildKit) by adding custom reuse mechanisms to Dockerfiles.

5.2.3 Discussion

In this dissertation, we've seen that software artifacts beyond code contain many duplicates. We think that developers as a broad audience should take a closer attention to it, just as it is the case with code clones. In his thesis, Alan Charpentier [Charpentier, 2016] proposed the idea of using more specialised code clone detection tools that could output more precise results better corresponding the users needs. He states that such specialisation is the answer to make clone detection tools useful on a daily basis to developers and maintainers.

While such specialisation is at some point necessary, we think that the community of software artifact maintainers isn't there yet. As a first step, it would be more interesting for the research community to investigate the idea of providing clone detection tools that are extremely generic, using simple techniques such as a line-based clone detection technique that could be compatible with a wider range of textual artifacts and therefore, build more awareness from the community. Such generic tools could be directly integrated with classical Integrated Development Environments (IDEs) as it's the case with the *IntelliJ IDEA*

IDE that automatically detects cloned code fragments and even proposes to automatically refactor them when possible.

Finally, as a second step, after having increased people's awareness, we think that the research community should also focus on the clone management part, just as we've seen for code clones in Chapter 2. To the best of our knowledge, except our research work (in Chapter 3 and Chapter 4) no research work has been done regarding how maintainers manage duplicates in software artifacts. Based on the feedback we've gathered from the surveys we've performed on the community of maintainers of Dockerfiles and API documentation, we do notice that software artifacts actually lack reuse mechanisms while the community asks for it. We think that the research community should investigate the idea of initially providing generic reuse mechanisms such as the use of templates for instance, rather than on a specialised case by case basis. Later, just as previously stated by Alan Charpentier [Charpentier, 2016], the community could go further and provide specialised clone detection tools which could be used by specialised linters (integrated in IDEs). These linters could help educate maintainers on the different existing reuse approaches for removing existing clones (corrective management) or even notify them when new ones are introduced (preventive management).

APPENDIX

A Résumé en Français

Lors du développement d'un logiciel, la maintenance et l'évolution représentent une part importante du cycle de vie du développement, représentant jusqu'à 80% du coût et de l'effort global [Alkhatib, 1992]. Pendant l'effort de maintenance ou de développement, il arrive que les développeurs aient à copier et coller des fragments de code source afin de les réutiliser. Communément appelés "clones" dans la littérature, ces doublons de code source sont un sujet bien connu et étudié en génie logiciel. De nombreuses études empiriques [Baker, 1995; Baxter et al., 1998; Rieger et al., 2004; Zibran et al., 2011] ont montré que les grands logiciels peuvent avoir entre 5% et 20% de leur base de code qui est clonée.

Au cours des deux dernières décennies, de nombreuses études ont tenté de comprendre les raisons sous-jacentes qui poussent les développeurs à recourir aux clones de code [Baxter et al., 1998; Kapser, 2009; Rieger, 2005]. Par exemple, Cordi [Cordy, 2003] affirme que dans le secteur financier, le clonage est une stratégie de réutilisation courante. Il explique que comme les produits financiers ne sont pas très différents les uns des autres, lorsqu'un nouveau produit doit être développé, les développeurs commencent par cloner un projet existant et l'adaptent ensuite pour produire le nouveau produit. Les développeurs doivent recourir à de telles pratiques car les risques monétaires sont si élevés en cas d'erreurs, que tout outil financier doit être fortement testé (70% des coûts des logiciels financiers sont consacrés aux tests [Cordy, 2003]). Par conséquent, il est moins cher et plus rapide de simplement réutiliser le code source existant en le clonant. Baxter et al. [Baxter et al., 1998] écrit aussi sur ce qu'on appelle les clones accidentels, qui sont inconsciemment produits par les développeurs. De tels clones peuvent se produire lors de l'utilisation de bibliothèques par exemple, où les développeurs doivent écrire du code répétitif tel qu'une séquence d'appels pour exécuter une tâche.

Cependant, le clonage de code n'est pas sans conséquence, Fowler et al. le considèrent même comme un "bad smell". Par exemple, les clones de code peuvent être dangereux

lorsque les développeurs n'en sont pas conscients. Pendant le processus de maintenance, s'il y a un bogue dans un fragment de code qui est cloné, le développeur doit propager une correction de bogue à travers tous les fragments clonés et donc augmenter les coûts de maintenance. Cependant, le développeur peut ne pas savoir que le bogue existe ailleurs dans sa base de code. De nos jours, comme les développeurs travaillent généralement en équipe et ne sont pas nécessairement conscients des clones existants, ce type d'erreurs peut devenir plus fréquent et problématique. Par conséquent, les développeurs et les responsables sont tenus de savoir si leur code est cloné et, le cas échéant, où se trouvent les autres fragments clonés.

Néanmoins, certaines études ont également discuté de la possibilité que les clones de code puissent être utiles [Rieger, 2005].

Au cours de la dernière décennie, de multiples détecteurs de codes clonés ont été développés. Les détecteurs de clones sont utilisés pour identifier des clones de code similaires ou identiques dans une code-base Les détecteurs de clones existants utilisent une pléthore de techniques telles que la détection de clones basée sur l'AST (Abstract Syntax Trees) [Baxter et al., 1998], la détection de clones de code par jeton [Kamiya et al., 2002] ou même les techniques d'apprentissage profond [White et al., 2016a].

Bien que la communauté des ingénieurs logiciels se soit beaucoup intéressée aux clones de code au cours des deux dernières décennies, il y a eu peu d'intérêt pour l'existence de clones autres que le code source, comme dans les artefacts logiciels produits pendant la conception de logiciels. Au cours du processus de développement d'un logiciel, un ensemble de sous-produits appelés artefacts logiciels sont également produits. Ces artefacts peuvent être de différents types : Documentation API, diagrammes de conception, spécifications des exigences, fichiers de configuration, fichiers de déploiement, etc. En raison de leur rôle important dans le processus de développement, les artefacts logiciels peuvent avoir un impact important sur le produit final et jouer un rôle important pendant le processus de développement. Cependant, on sait peu de choses sur les clones dans les artefacts logiciels dans la littérature. Quelques études ont montré qu'il est également fréquent d'avoir des doublons dans les artefacts logiciels. Par exemple, de multiples études [Liu et al., 2006b; Störrle, 2013] ont montré que les diagrammes UML sont également confrontés à des problèmes de doublons tout comme le code. Juergens, Domann et al. [Juergens et al., 2010; Domann et al., 2009] montrent que les spécifications des exigences peuvent également être confrontées à de tels problèmes de doublons. McIntosh et al. [McIntosh et al., 2014] montrent que les systèmes de compilation sont également confrontés à des problèmes de doublons.

Parmi tous les différents types d'artefacts que nous avons présentés précédemment, nous avons choisi dans cette thèse de nous concentrer sur l'existence de clones sur deux types spécifiques : les Fichiers de documentation d'API et de déploiement (i.e. Dockerfiles). À notre connaissance, aucun travail de recherche n'a étudié les clones de ces deux types d'artefacts par le passé. Nous avons choisi d'examiner de plus près la documentation d'API car elle aide les développeurs à comprendre comment utiliser et donc intégrer

une API externe qu'ils ne connaissent pas dans leur code. Nous pensons que le fait d'avoir des clones dans la documentation d'une API peut conduire à des incohérences qui peuvent amener les développeurs et les responsables à ne pas comprendre le comportement correct d'une API, augmentant ainsi les coûts et les efforts de développement logiciel.

Ensuite, nous avons également choisi d'examiner de plus près les Dockerfiles qui sont un type de fichiers de déploiement propriétaire. Les Dockerfiles sont utilisés pour regrouper une application avec toutes ses dépendances en un seul paquet qui peut ensuite être facilement distribué. Ces paquets représentent donc la dernière étape entre les développeurs et les utilisateurs d'applications. Les fichiers docker sont écrits sous la forme d'un simple fichier texte composé d'une séquence d'instructions écrites dans un langage spécifique au domaine (DSL).

Par conséquent, nous croyons que la communauté pourrait bénéficier de ces deux études empiriques étant donné l'importance de ces artefacts.

Dans cette thèse, nous étudions notre hypothèse sur la duplication dans la documentation d'API et les Dockerfiles, et répondons plus formellement aux trois questions de recherche suivantes :

— RQ1 : Les développeurs recourent-ils souvent au copier-coller ?

— RQ2 : Pourquoi les développeurs recourent-ils au copier-coller ?

— RQ3 : Peut-on éviter les copier-coller en utilisant correctement les outils existants ?

Notre première contribution, présentée dans In Chapter 3, vise à étudier la question des doublons dans la documentation d'API. Pour ce faire, nous appliquons une méthodologie de recherche empirique sur un corpus composé de 100 projets open-source maintenant une documentation d'API (50 en langage de programmation Java et 50 en langage de programmation Ruby). Nous effectuons également une enquête auprès de 39 développeurs utilisant Java ou Ruby et recueillons leurs opinions concernant les doublons dans la documentation d'API.

Notre première contribution, présentée dans In Chapter 3, vise à étudier la question des doublons dans la documentation d'API. Pour ce faire, nous appliquons une méthodologie de recherche empirique sur un corpus composé de 100 projets open-source maintenant une documentation d'API (50 en langage de programmation Java et 50 en langage de programmation Ruby). Nous effectuons également une enquête auprès de 39 développeurs utilisant Java ou Ruby et recueillons leurs opinions concernant les doublons dans la documentation d'API.

En conséquence, nous montrons d'abord que les doublons de tags de documentation, indépendamment du langage de programmation utilisé, sont malheureusement trop nombreux. Ensuite, nous analysons manuellement ces doublons et identifions qu'ils sont causés par cinq différents types de relations dans le code source sous-jacent : (1) délégation : lorsqu'une méthode en appelle une autre, (2) sous-typage : lorsqu'une méthode effectue un calcul similaire à une autre méthode, (3) clonage de code, (4) intention similaire : lorsqu'une méthode supplante une autre qui est définie dans une même hiérarchie et (5)

utilisation similaire : lorsque deux méthodes ou plus partagent un paramètre commun, une sortie ou une exception mais la tâche que chaque méthode doit faire est totalement différente. Troisièmement, nous constatons que les développeurs qui répondent à notre enquête ont souvent recours à des doublons dans la documentation d'API et trouvent ces pratiques ennuyeuses et sujettes aux erreurs. Ils aimeraient également disposer d'outils de documentation qui leur permettraient d'éviter de tels doublons. Enfin, à travers notre étude, nous soulignons le fait qu'il est courant que les outils de documentation ne fournissent pas de mécanismes de réutilisation pour faire face à ces causes. Cependant, un outil (c.-à-d. Yard) fournit un mécanisme qui traite toutes les causes de double emploi que nous avons identifiées dans notre étude.

Notre deuxième contribution, présentée dans In Chapter 4, vise à fournir aux praticiens une explication claire des raisons pour lesquelles les doublons de Dockerfile apparaissent dans les projets, et quels sont les différents moyens dont ils disposent pour les traiter, avec leurs avantages et inconvénients. Pour ce faire, nous réalisons une étude basée sur l'analyse des Official Docker Projects (99 projets open-source) qui sont des projets populaires dans le monde réel qui gèrent des familles de Dockerfiles moyennes à grandes et qui favorisent les meilleures pratiques pour maintenir les Dockerfiles. Nous effectuons également un sondage auprès des mainteneurs officiels des projets Docker et recueillons leurs opinions concernant les doublons sur les fichiers Dockerfiles.

Par conséquent, nous montrons d'abord que les projets Docker officiels maintiennent fréquemment des familles de Dockerfiles et identifions les raisons sous-jacentes : supporter plusieurs versions / images de base et versions / approche. Deuxièmement, nous montrons ensuite que les doublons dans les Dockerfiles sont abondants, et identifions les trois raisons sous-jacentes qui les expliquent : (1) Installation et configuration de logiciel, (2) gestion des dépendances et (3) configuration du runtime. Troisièmement, nous constatons que les mainteneurs officiels des projets Docker sont conscients de l'existence de doublons dans les Dockerfiles qu'ils maintiennent et sont fréquemment confrontés à ceux-ci. Cependant, ils ont une opinion mitigée à leur sujet. Alors que les mainteneurs qui n'utilisent pas d'outils pour gérer les doublons déclarent que leur gestion peut être sujette aux erreurs, les mainteneurs qui utilisent des outils déclarent qu'ils sont faciles à gérer. Nous constatons également que certains mainteneurs gèrent les doublons en utilisant des outils ad-hoc : processeurs de templates, générateurs de code, rechercher / remplacer. Enfin, nous montrons que les projets utilisant des processeurs de templates et des générateurs de code parviennent à réduire le nombre de doublons avec une médiane allant de 30% à 100% pour les générateurs.

Enfin, le Chapter 5 conclut cette thèse en résumant nos contributions, et en présentant plusieurs perspectives possibles pour étendre notre travail.

Bibliography

Ahasanuzzaman, M., Asaduzzaman, M., Roy, C. K., and Schneider, K. A. (2016). Mining duplicate questions in stack overflow. In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 402–412. ACM. Cited page 27.

Alalfi, M. H., Antony, E. P., and Cordy, J. R. (2018). An approach to clone detection in sequence diagrams and its application to security analysis. *Software & Systems Modeling*, 17(4):1287–1309. Cited page 23.

Alkhatib, G. (1992). The maintenance problem of application software: An empirical analysis. *Journal of Software Maintenance: Research and Practice*, 4(2):83–104. Cited pages 2, 81, and 87.

Antony, E. P., Alalfi, M. H., and Cordy, J. R. (2013). An approach to clone detection in behavioural models. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 472–476. IEEE. Cited page 23.

Baker, B. S. (1993). A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49. Cited page 16.

Baker, B. S. (1995). On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering*, pages 86–95. IEEE. Cited pages 2 and 87.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., and Kontogiannis, K. (1999). Measuring clone based reengineering opportunities. In *Proceedings Sixth International Software Metrics Symposium (Cat. No. PR00403)*, pages 292–303. IEEE. Cited page 10.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE. Cited pages 2, 10, 18, 68, 81, 87, and 88.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007a). Comparison and evaluation of clone detection tools. *Software Engineering, IEEE Transactions on*, 33(9):577–591. Cited pages 10, 23, and 68.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007b). Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591. Cited page 30.

Bratthall, L. and Jørgensen, M. (2002). Can you trust a single data source exploratory software engineering case study? *Empirical Software Engineering*, 7(1):9–26. Cited page 6.

Burd, E. and Bailey, J. (2002). Evaluating clone detection tools for use during preventative maintenance. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43. IEEE. Cited page 10.

Charpentier, A. (2016). *Contributions à l'usage des détecteurs de clones pour des tâches de maintenance logicielle*. PhD thesis. Cited pages 84 and 85.

Charpentier, A., Falleri, J.-R., Lo, D., and Réveillère, L. (2015). An empirical assessment of bellon's clone benchmark. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10. Cited page 14.

Charpentier, A., Falleri, J.-R., and Réveillère, L. (2016). Automated extraction of mixins in cascading style sheets. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 56–66. IEEE. Cited page 26.

Cordy, J. R. (2003). Comprehending reality-practical barriers to industrial adoption of software maintenance automation. In *11th IEEE International Workshop on Program Comprehension, 2003.*, pages 196–205. IEEE. Cited pages 2 and 87.

Cordy, J. R. and Roy, C. K. (2011). The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE. Cited pages 20 and 23.

Correia, F. F., Aguiar, A., Ferreira, H. S., and Flores, N. (2009). Patterns for consistent software documentation. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, page 12. ACM. Cited page 34.

Dagenais, B. and Robillard, M. P. (2010). Creating and evolving developer documentation: understanding the decisions of open source contributors. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 127–136. ACM. Cited page 34.

Davey, N., Barson, P., Field, S., Frank, R., and Tansley, D. (1995). The development of a software clone detector. *International Journal of Applied Software Technology*. Cited page 21.

de Souza, S. C. B., Anquetil, N., and de Oliveira, K. M. (2005). A Study of the Documentation Essential to Software Maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting &Amp; Designing for Pervasive Information*, SIGDOC '05, pages 68–75, New York, NY, USA. ACM. Cited page 34.

Deissenboeck, F., Hummel, B., Jürgens, E., Schätz, B., Wagner, S., Girard, J.-F., and Teuchert, S. (2008a). Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, pages 603–612. ACM. Cited pages 24 and 101.

Deissenboeck, F., Juergens, E., Hummel, B., Wagner, S., y Parareda, B. M., and Pizka, M. (2008b). Tool support for continuous quality control. *IEEE software*, 25(5):60–67. Cited page 24.

Di Lucca, G. A., Di Penta, M., Fasolino, A. R., and Granato, P. (2001). Clone analysis in the web era: An approach to identify cloned web pages. In *Proceedings of the 7th IEEE Workshop on Empirical Studies of Software Maintenance (WESS'99)*, pages 107–113. Cited pages 19 and 26.

Domann, C., Juergens, E., and Streit, J. (2009). The curse of Copy&Paste cloning in requirements specifications. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 443–446. IEEE Computer Society. Cited pages 3, 22, and 88.

Duala-Ekoko, E. and Robillard, M. P. (2007). Tracking code clones in evolving software. In *29th International Conference on Software Engineering (ICSE'07)*, pages 158–167. IEEE. Cited page 29.

Duala-Ekoko, E. and Robillard, M. P. (2008). Clonetracker: tool support for code clone management. In *Proceedings of the 30th international conference on Software engineering*, pages 843–846. ACM. Cited page 29.

Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 109–118. IEEE. Cited page 15.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA. ACM. Cited page 40.

Fluri, B., Würsch, M., and Gall, H. C. (2007). Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79. IEEE. Cited page 34.

Forward, A. and Lethbridge, T. C. (2002). The Relevance of Software Documentation, Tools and Technologies: A Survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering*, DocEng '02, pages 26–33, New York, NY, USA. ACM. Cited page 34.

Gabel, M., Jiang, L., and Su, Z. (2008). Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330. ACM. Cited page 20.

Giesecke, S. (2007). Generic modelling of code clones. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik. Cited page 28.

Gitchell, D. and Tran, N. (1999). Sim: a utility for detecting similarity in computer programs. In *ACM SIGCSE Bulletin*, volume 31, pages 266–270. ACM. Cited page 18.

Han, J., Pei, J., and Kamber, M. (2011). *Data mining: concepts and techniques*. Elsevier. Cited page 18.

Hermans, F., Sedee, B., Pinzger, M., and van Deursen, A. (2013). Data clone detection and visualization in spreadsheets. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 292–301. IEEE. Cited page 27.

Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. (2010). Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9. IEEE. Cited pages 17, 21, 62, and 63.

Jablonski, P. and Hou, D. (2007). Cren: a tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. In *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 16–20. ACM. Cited page 29.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society. Cited page 20.

Johnson, J. H. (1993). Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press. Cited page 15.

Juergens, E., Deissenboeck, F., Feilkas, M., Hummel, B., Schaetz, B., Wagner, S., Domann, C., and Streit, J. (2010). Can clone detection support quality assessments of requirements specifications? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 79–88. ACM. Cited pages 3, 22, and 88.

Juergens, E., Deissenboeck, F., and Hummel, B. (2009a). Clonedetective-a workbench for clone detection research. In *Proceedings of the 31st International Conference on Software Engineering*, pages 603–606. IEEE Computer Society. Cited page 22.

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009b). Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE. Cited page 2.

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009c). Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 485–495, Washington, DC, USA. IEEE Computer Society. Cited page 34.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670. Cited pages 2, 10, 17, 68, 81, and 88.

Kapser, C. (2009). Toward an understanding of software code cloning as a development practice. Cited pages 2, 81, and 87.

Kapser, C. and Godfrey, M. W. (2006). "Cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. IEEE. Cited page 2.

Karp, R. M. and Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM journal of research and development*, 31(2):249–260. Cited page 15.

Kawaguchi, S., Yamashina, T., Uwano, H., Fushida, K., Kamei, Y., Nagura, M., and Iida, H. (2009). Shinobi: A tool for automatic code clone detection in the ide. In *2009 16th Working Conference on Reverse Engineering*, pages 313–314. IEEE. Cited page 29.

Komondoor, R. and Horwitz, S. (2001). Using slicing to identify duplication in source code. In *International Static Analysis Symposium*, pages 40–56. Springer. Cited page 20.

Kontogiannis, K. A., DeMori, R., Merlo, E., Galler, M., and Bernstein, M. (1996). Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1-2):77–108. Cited page 19.

Koschke, R., Falke, R., and Frenzel, P. (2006). Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE. Cited page 20.

Kramer, D. (1999). API documentation from source code comments: a case study of Javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153. ACM. Cited page 34.

Krinke, J. (2001). Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE. Cited page 20.

Lague, B., Proulx, D., Mayrand, J., Merlo, E. M., and Hudepohl, J. (1997). Assessing the benefits of incorporating function clone detection in a development process. In *1997 Proceedings International Conference on Software Maintenance*, pages 314–321. IEEE. Cited page 28.

Lakhotia, A. (1993). Understanding Someone else's Code: Analysis of Experiences. *J. Syst. Softw.*, 23(3):269–275. Cited page 34.

Lanubile, F. and Mallardo, T. (2003). Finding function clones in web applications. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.*, pages 379–386. IEEE. Cited page 19.

Lethbridge, T. C., Singer, J., and Forward, A. (2003). How Software Engineers Use Documentation: The State of the Practice. *IEEE Softw.*, 20(6):35–39. Cited page 34.

Lex, A., Gehlenborg, N., Strobelt, H., Vuillemot, R., and Pfister, H. (2014). UpSet: visualization of intersecting sets. *IEEE transactions on visualization and computer graphics*, 20(12):1983–1992. Cited page 64.

Li, L., Feng, H., Zhuang, W., Meng, N., and Ryder, B. (2017a). Cclearner: A deep learning-based clone detection approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 249–260. IEEE. Cited page 21.

Li, Z., Lu, S., Myagmar, S., and Zhou, Y. (2006). Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192. Cited page 17.

Li, Z., Yin, G., Yu, Y., Wang, T., and Wang, H. (2017b). Detecting duplicate pull-requests in github. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*, page 20. ACM. Cited page 26.

Liu, C., Chen, C., Han, J., and Yu, P. S. (2006a). Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881. ACM. Cited page 20.

Liu, H., Ma, Z., Zhang, L., and Shao, W. (2006b). Detecting duplications in sequence diagrams based on suffix trees. In *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, pages 269–276. IEEE. Cited pages 3, 22, and 88.

Martin, D. and Cordy, J. R. (2011). Analyzing web service similarity using contextual clones. In *Proceedings of the 5th International Workshop on Software Clones*, pages 41–46. ACM. Cited page 25.

Mayrand, J. (1996). Evaluating the benefits of clone detection in the software maintenance activities in large scale systems. *WESS'96*. Cited page 28.

Mayrand, J., Leblanc, C., and Merlo, E. (1996). Experiment on the automatic detection of function clones in a software system using metrics. In *icsm*, volume 96, page 244. Cited pages 10 and 19.

McIntosh, S., Poehlmann, M., Juergens, E., Mockus, A., Adams, B., Hassan, A. E., Haupt, B., and Wagner, C. (2014). Collecting and leveraging a benchmark of build system clones to aid in quality assessments. In *Companion proceedings of the 36th international conference on software engineering*, pages 145–154. ACM. Cited pages 3, 24, and 88.

Miller, J. (2008). Triangulation as a basis for knowledge discovery in software engineering. *Empirical Software Engineering*, 13(2):223–228. Cited page 6.

Monperrus, M., Eichberg, M., Tekes, E., and Mezini, M. (2012). What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation. *Empirical Software Engineering*, 17(6):703–737. Cited page 34.

Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., and Zave, P. (2007). Matching and merging of statecharts specifications. In *Proceedings of the 29th international conference on Software Engineering*, pages 54–64. IEEE Computer Society. Cited page 23.

Nguyen, H. A., Nguyen, T. T., Pham, N. H., Al-Kofahi, J., and Nguyen, T. N. (2011). Clone management for evolving software. *IEEE transactions on software engineering*, 38(5):1008–1026. Cited page 30.

Oumaziz, M. A., Charpentier, A., Falleri, J.-R., and Blanc, X. (2017). Documentation reuse: Hot or not? An empirical study. In *International Conference on Software Reuse*, pages 12–27. Springer. Cited pages 7, 38, and 82.

Oumaziz, M. A., Falleri, J.-R., Blanc, X., Bissyandé, T. F., and Klein, J. (2019). Handling duplicates in dockerfiles families: Learning from experts. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 524–535. IEEE. Cited pages 7 and 83.

Parnas, D. L. (1972). A Technique for Software Module Specification with Examples. *Commun. ACM*, 15(5):330–336. Cited page 34.

Parr, T. J. (2004). Enforcing Strict Model-view Separation in Template Engines. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 224–233, New York, NY, USA. ACM. event-place: New York, NY, USA. Cited page 73.

Pérez-Agüera, J. R., Arroyo, J., Greenberg, J., Iglesias, J. P., and Fresno, V. (2010). Using bm25f for semantic search. In *Proceedings of the 3rd international semantic search workshop*, page 2. ACM. Cited page 26.

Pollack, M. (2000). Code generation using javadoc. *JavaWorld, http://www. javaworld. com/javaworld/jw-08-2000/jw-0818-javadoc. html*. Cited pages 34 and 35.

Rajapakse, D. C. and Jarzabek, S. (2007). Using server pages to unify clones in web applications: A trade-off analysis. In *29th International Conference on Software Engineering (ICSE'07)*, pages 116–126. IEEE. Cited page 25.

Raza, A., Vogel, G., and Plödereder, E. (2006). Bauhaus–a tool suite for program analysis and reverse engineering. In *International Conference on Reliable Software Technologies*, pages 71–82. Springer. Cited page 18.

Rieger, M. (2005). *Effective clone detection without language barriers*. PhD thesis, Verlag nicht ermittelbar. Cited pages 2, 81, 87, and 88.

Rieger, M., Ducasse, S., and Lanza, M. (2004). Insights into system-wide code duplication. In *11th Working Conference on Reverse Engineering*, pages 100–109. IEEE. Cited pages 2 and 87.

Roy, C. K. and Cordy, J. R. (2007). A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68. Cited page 14.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495. Cited page 10.

Roy, C. K., Zibran, M. F., and Koschke, R. (2014). The vision of software clone management: Past, present, and future (keynote paper). In *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 18–33. IEEE. Cited page 28.

Sæbjørnsen, A., Willcock, J., Panas, T., Quinlan, D., and Su, Z. (2009). Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM. Cited page 27.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. IEEE. Cited page 17.

Seaman, C. B. (1999). Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572. Cited page 6.

Sharma, T., Fragkoulis, M., and Spinellis, D. (2016). Does your configuration code smell? In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 189–200. IEEE. Cited page 25.

Störrle, H. (2013). Towards clone detection in UML domain models. *Software & Systems Modeling*, 12(2):307–329. Cited pages 3, 23, and 88.

Sun, C., Lo, D., Khoo, S.-C., and Jiang, J. (2011). Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 253–262. IEEE Computer Society. Cited page 26.

Sun, C., Lo, D., Wang, X., Jiang, J., and Khoo, S.-C. (2010). A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 45–54. ACM. Cited page 26.

Sureka, A. and Jalote, P. (2010). Detecting duplicate bug report using character n-gram-based features. In *2010 Asia Pacific Software Engineering Conference*, pages 366–374. IEEE. Cited page 26.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering*, pages 492–497. IEEE Computer Society Press. Cited page 28.

Synytskyy, N., Cordy, J. R., and Dean, T. (2003). Resolution of static clones in dynamic web pages. In *Fifth IEEE International Workshop on Web Site Evolution, 2003. Theme: Architecture. Proceedings.*, pages 49–56. IEEE. Cited page 29.

Tairas, R. and Gray, J. (2006). Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, pages 679–684. ACM. Cited page 20.

Tatsubori, M. and Suzumura, T. (2009). HTML Templates That Fly: A Template Engine Approach to Automated Offloading from Server to Client. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 951–960, New York, NY, USA. ACM. event-place: Madrid, Spain. Cited page 73.

Van Heesch, D. (2004). *Doxygen*. Cited pages 34 and 35.

Vanter, M. L. V. D. (2002). The documentary structure of source code. *Information and Software Technology*, 44(13):767 – 782. Cited page 34.

Wahler, V., Seipel, D., Wolff, J., and Fischer, G. (2004). Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, Fourth IEEE International Workshop on*, pages 128–135. IEEE. Cited page 18.

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016a). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM. Cited pages 2, 81, and 88.

White, M., Tufano, M., Vendome, C., and Poshyvanyk, D. (2016b). Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 87–98. ACM. Cited page 21.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media. Cited pages 52 and 77.

Wood, M., Daly, J., Miller, J., and Roper, M. (1999). Multi-method research: An empirical investigation of object-oriented technology. *Journal of Systems and Software*, 48(1):13–26. Cited page 6.

Yan, X., Han, J., and Afshar, R. (2003). Clospan: Mining: Closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM international conference on data mining*, pages 166–177. SIAM. Cited page 17.

Yang, W. (1991). Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755. Cited page 18.

Zhang, G., Peng, X., Xing, Z., Jiang, S., Wang, H., and Zhao, W. (2013). Towards contextual and on-demand code clone management by continuous monitoring. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 497–507. IEEE. Cited page 28.

Zhou, J. and Zhang, H. (2012). Learning to rank duplicate bug reports. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 852–861. ACM. Cited page 26.

Zibran, M. F., Saha, R. K., Asaduzzaman, M., and Roy, C. K. (2011). Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, pages 295–304. IEEE. Cited pages 2 and 87.

List of Figures

List of Tables