Empir Software Eng (2016) 21:517–564
DOI 10.1007/s10664-015-9368-6

Development nature matters: An empirical study of code clones in JavaScript applications

Wai Ting Cheung · Sukyoung Ryu · Sunghun Kim

Published online: 24 March 2015

Abstract Code cloning is one of the active research areas in the software engineering community. Specifically, researchers have conducted numerous empirical studies on code cloning and reported that 7 % to 23 % of the code in a typical software system has been cloned. However, there was less awareness of code clones in dynamically-typed languages and most studies are limited to statically-typed languages such as Java, C, and C++. In addition, most previous studies did not consider different application domains such as standalone projects or web applications. As a result, very little is known about clones in dynamically-typed languages, such as JavaScript, in different application domains. In this paper, we report a large-scale clone detection experiment in a dynamically-typed programming language, JavaScript, for different application domains: web pages and standalone projects. Our experimental results showed that unlike JavaScript standalone projects, JavaScript web applications have 95 % of inter-file clones and 91–97 % of widely scattered clones. We observed that web application developers created clones intentionally and such clones may not be as risky as claimed in previous studies. Understanding the risks of cloning in web applications requires further studies, as cloning may be due to either good or bad intentions. Also, we identified unique development practices such as including browser-dependent or device-specific code in code clones of JavaScript web applications. This indicates that features of programming languages and technologies affect how developers duplicate code.

Communicated by: Andrea De Lucia

W. T. Cheung · S. Ryu (✉)
Department of Computer Science, KAIST, Daejeon, South Korea
e-mail: sryu.cs@kaist.ac.kr

W. T. Cheung
e-mail: kencwt@kaist.ac.kr

S. Kim
Department of Computer Science and Engineering,
The Hong Kong University of Science and Technology, Hong Kong, China
e-mail: hunkim@cse.ust.hk

Keywords JavaScript · Code clones · Web applications · Clone properties · Software metrics · Cloning patterns

1 Introduction

Code clones are pervasive. Previous research reported that 7 % to 23 % of the code in a typical software system has been cloned (Bakerm 1995; Roy and Cordy 2008). Code cloning has become an active research area. Roy et al. (2014) studied 353 scholarly articles published between 1994 and 2013 and categorized them into detection, analysis, management, and tool evaluation. Among various clone-related research areas, empirical studies of code clones have investigated different properties of clones (Cai and Kim 2011; Kim et al. 2005; Roy and Cordy 2008).

In the literature, most clone-related research involved experiments on subjects of statically-typed languages such as Java, C, and C++. Indeed, the nature of statically-typed and dynamically-typed languages influences how developers duplicate code. Figure 1 shows an example of a code clone in Java, a statically-typed language, from the application JHotDraw to support function overloading. Function overloading enables declarations of similar methods with different parameter types. JHotDraw contains 12 such fragments in the system. In this example, developers cloned the function and changed only the parameter types. Such usage makes the code more understandable than abstracting it by a template (Kapser and Godfrey 2008). However, dynamically-typed languages such as JavaScript do not provide any overloading mechanisms. Function declarations and expressions do not specify parameter types and hence they are less likely to be subject to function-level clones. Despite the different nature of statically- and dynamically-typed languages, only a few studies compared the code clones between them (Jang et al. 2012; Roy and Cordy 2010a).

In addition, code clone studies on dynamically-typed languages have limitations: they do not consider clones in different application domains, such as standalone projects or web applications, and most clone-related studies of web applications are in HTML, ASP, and PHP. Studies on code clones in JavaScript are limited. Given that JavaScript has extremely dynamic features such as code generation at runtime, it has no module system, and its main uses are scripts embedded in HTML, comparing JavaScript code clones in standalone projects and web applications helps to understand how different features of technologies affect code cloning.

Also, studies of code clones on web applications mainly concentrate on quantitative differences (Di Lucca et al. 2002; Lanubile and Mallardo 2003; Rajapakse and Jarzabek 2005). Only a limited amount of research has included qualitative analysis on the differences.

```java
public void addAttribute(
    String name, float value,
    float defaultValue) {
 if (value != defaultValue) {
  addAttribute(name, value);
 }
}
```

JavaxDOMOutput.java:284–288

```java
public void addAttribute(
    String name, int value,
    int defaultValue) {
 if (value != defaultValue) {
  addAttribute(name, value);
 }
}
```

NanoXMLDOMOutput.java:296–300

Fig. 1 Java code clone from JHotDraw to overload functions

Analyzing code clones qualitatively helps to better understand the motivations for code cloning.

While researchers have found that different motivations for cloning may positively or negatively impact the code quality, only a few have attempted to measure whether certain clone properties are associated with any software metrics (Kozlov et al. 2010; Monden et al. 2002). Clone properties are characteristics of code clones that are often measured in terms of clone metrics. Several studies in the literature have examined the relationship between clone properties and software metrics to investigate costs and benefits of clone removal (Koschke 2007; Monden et al. 2002; Roy and Cordy 2007). However, these studies focused on only a single language and did not identify whether the relationship would be very different in other languages or application domains.

In this paper, we study clones in three different types of subjects: JavaScript web applications, JavaScript, and Java standalone projects. Throughout the paper, we denote them as `JSweb`, `JSproj`, and `Java`, respectively. Table 1 shows the languages, definitions, and study targets of the three subject types. JavaScript web applications are web applications that involve embedded JavaScript in development. JavaScript and Java standalone projects are systems that are developed as individual applications or as libraries of other applications using JavaScript and Java as the main language of development, respectively. We exclude external JavaScript source files in our study because it is not easy to automatically distinguish which of those files are from developers themselves, external libraries, or a mixture of both.

We chose JavaScript because it has become one of the most widely-used dynamically-typed languages, and it shows an increasing trend of usages: researchers reported that 98 out of the 100 most popular websites use JavaScript (Guarnieri et al. 2011), and the use of dynamic features is evident in websites (Ratanaworabhan et al. 2010; Richards et al. 2010, 2011). Given the different nature of statically- and dynamically-typed languages as well as the extremely dynamic features of JavaScript, we would like to know if the findings in the

Table 1 Definitions and study targets of the three subject types

Type	Language	Definition	Study Target
`JSweb`	JavaScript	Web applications that involve embedded *JavaScript* in development	Embedded JavaScript in the script tags of HTML files, excluding external JavaScript source files
`JSproj`	JavaScript	Systems that are developed as individual applications or as libraries of other applications using *JavaScript* as the main language of development	Files with extension .js
`Java`	Java	Systems that are developed as individual applications or as libraries of other applications using *Java* as the main language of development	Files with extension .java

literature on statically-typed languages still apply to JavaScript and whether the findings on other scripting languages, such as Python, are similar to JavaScript.

We utilize clone metrics from the literature and compare the clone properties among JavaScript web applications, JavaScript, and Java standalone projects. We also measure the correlation between the clone properties and software metrics, and manually inspect the clones to understand how code clones are different between different languages and application domains.

In our study, we found that JavaScript web applications showed different clone properties and cloning practices to those of JavaScript and Java standalone projects. We found that features of programming languages and technologies affect how developers duplicate code. We also noticed that developers of JavaScript web applications duplicate code intentionally. Existing clone tracking techniques allow developers to manage the clones efficiently, and hence, they may not be as risky as claimed in previous studies. Understanding the risks of cloning in JavaScript web applications requires further studies, as cloning may be due to either good or bad intentions. Our findings suggest that when designing tools to manage code clones, developers have to take into account different features of programming languages and technologies of individual systems.

Overall, this paper makes the following contributions:

- **An empirical study of clone properties on** `JSweb`**,** `JSproj`**, and** `Java`**:** We found unique clone properties of `JSweb`, `JSproj`, and `Java` by analyzing clones detected by `JSCD`, our custom-built JavaScript clone detector, and Deckard on 10 websites and 20 standalone projects.
- **An analysis of correlation between clone properties and software metrics on** `JSweb`**,** `JSproj`**, and** `Java`**:** We observed stronger correlations between clone properties and software metrics in `JSproj` and `Java` than `JSweb`.
- **A qualitative analysis of cloning patterns on** `JSweb`**,** `JSproj`**, and** `Java`**:** We manually inspected the code clones and identified unique development practices in `JSweb`.

The remainder of this paper is structured as follows. After discussing related work in Section 2, we describe our clone detector, `JSCD`, in Section 3. Section 4 and Section 5 present our study design and findings, respectively. Section 6 discusses the implications of our findings, lessons learned, and threats to validity. We conclude our study in Section 7.

2 Related Work

While many researchers actively studied code cloning of statically-typed languages, only a few researchers have studied the code cloning of dynamically-typed languages. A language is statically-typed if the type of variable is known at compile time. A language is dynamically-typed if the type of variable is interpreted at runtime.

2.1 Empirical Studies of Code Clones

In the literature, researchers have conducted various empirical studies on clones to study different clone properties.

Hotta et al. (2010) studied modification frequencies of clones and found that the presence of duplicate code does not always have a negative impact on software evolution.

Thummalapenta et al. (2010) studied the clone evolution of four Java and C software systems and found that more than 70 % of the clones were consistently changed and less than 16 % had a late propagation. They also found that the code in clone evolution is more bug-prone and that cloning is usually used as a templating mechanism. They reported that developers are aware of clone locations and automatic clone tracing tools can be beneficial to developers. Our study differs from theirs in that their work studied the relationship between clones and bugs in clone evolution while we study how features of programming languages and technologies affect the ways developers clone.

Zibran et al. (2011) studied the evolution of near-miss clones in terms of the clone density and Pearson coefficient, and they forecasted the amount of clones in future releases of software systems. They found that with the increase in the number of functions, the number of cloned fragments also increases in all systems. However, they identified a very weak positive correlation between clone density and the number of functions, and the average clone density for larger (in terms of LOC) systems was found to be less than that for smaller systems.

Mondal et al. (2012) studied the stability of clones in terms of types, methods, languages, and systems. They found that Type-1 and Type-2 clones are unstable, but Type-3 clones are stable. They also noticed that clones in Java and C systems are not as stable as clones in C# systems.

All the above studies were on subjects of statically-typed languages such as Java, C, C++, and C#. To the best of our knowledge, there is only a limited amount of empirical studies on code clones of dynamically-typed languages or both statically-typed and dynamically-typed languages.

2.2 Studies of Clones in both Statically-Typed and Dynamically-Typed Languages

Some researchers studied clones in both statically-typed and dynamically-typed languages, but they did not consider application domains.

Roy and Cordy (2010a) studied clones in both statically-typed and dynamically-typed languages, namely, C, Java, and Python, in terms of various clone properties: cloning level (number of methods and lines of code), clone-associated files, cloning density, and cloning localization. They studied the changes of the clone properties by varying the similarity in clone detection and observed that these properties change consistently in different languages when the similarity varies. Our work is similar to theirs in that we share three of the above clone properties in our experiments with theirs, and we compare our findings with theirs in detail in Section 5. However, their work lacks qualitative analysis to understand why they observed such findings.

Jang et al. (2012) detected 15,546 unpatched code clones on statically-typed and dynamically-typed languages—Java, C, Perl, and Python—and confirmed 145 real bugs in the latest version of Debian Squeeze packages.

Our work differs from the work mentioned above in that they have little or no qualitative analysis to gain insight on their findings between statically-typed and dynamically-typed languages. In this work, we manually investigate the clones to understand how different features of programming languages and technologies lead to different ways of cloning.

2.3 Language-Independent Approaches for Clone Detection

Researchers have proposed language-independent approaches for clone detection, but the precision and recall of these approaches vary from language to language. A language-independent clone detector is a tool that applies the same approach to detect clones in different languages with no special treatment to individual languages. The correctness and completeness of clone detection results are usually evaluated in terms of *precision* and *recall*.

Cordy et al. (2004) detected near-miss clones and conducted experiments on two HTML websites. They identified approximately 77–95 % of clones in multiple locations, and the clones tended to be medium- to large-sized, typically exceeding 20 lines of code. However, they did not measure recall and precision in their work.

Ducasse et al. (2004, 2006) detected clones on both statically-typed and dynamically-typed languages such as Java, C, and Python. They experimented with different normalization techniques and found that normalizing function names does not improve recall much but lowers precision slightly. Also, they found that setting a maximum gap size of one improves precision by 10 % while impacting recall by no more than 5 %. They defined the gap as follows: "A gap in a sequence of matching lines occurs when the corresponding lines fail to match." They reported an overall recall of 64–78 % and precision of 42–94 % without normalization and an overall recall of 85–95 % and precision of 11.5–70 % with normalization.

Bulychev and Minea (2009) detected clones in Python, Java, and Lua and reported that their approach can detect such clones that existing token-based and AST-based approaches miss. However, they compared mainly the percentages of detected clones among different tools and did not study other clone properties. They found that their tool had a recall of 31.7 %–77.3 % as well as 25–40 % higher precision than that of the clone detector CloneDR.

Brixtel et al. (2010) applied clone detection to plagiarism detection and experimented with students' assignments in both statically-typed and dynamically-typed languages such as Java, Haskell, Python, and PHP. They reported that their approach works best for Haskell but they achieved a precision of only 20 % for Python.

The above work shows that achieving high recall and precision is a challenge for language-independent clone detectors. In Section 3, we evaluate our clone detector with two language-independent clone detectors in detecting JavaScript code clones. Our evaluation results show that our clone detector has an overall recall of 87 % and precision of 96 %, which represent a 53–58 % improvement in recall and comparable precision to other two clone detectors. In Section 3, we will discuss the reasons for such a difference in detail.

2.4 Studies of Clones in Web Applications

Boldyreff and Kewish (2001) analyzed existing websites by parsing HTML documents, storing the HTML data categorized by file types in a database, and extracting duplicated contents and styles. They analyzed five websites with their mechanism and found that small sites tend to have fewer clones (11 %) because the maintainers of small sites might remember higher proportions of the sites than the maintainers of large sites; thus, the former might recognize clones more often than the latter when they have repeated some code.

Di Lucca et al. (2002) detected duplicated pages based on similarity metrics in HTML and ASP pages from four web applications. They noticed that using a frequency-based method for preliminary identification followed by the Levenshtein distance method for actual detection may be good for detecting clones in HTML pages because the computation time of the frequency-based method is much lower than the actual detection method, while it produces less precise results. In addition, they noticed that the Levenshtein distance method works better at detecting clones in HTML pages than ASP pages. With manual verification, they observed that, even when two ASP pages include some code in common, the remaining codes are very different. They reported that the percentages of files having clones are 9–74 % for HTML and 21–56 % for ASP.

Lanubile and Mallardo (2003) identified cloned functions within JavaScript and VBScript of web applications and reported that the function clones are between 39 % and 50 % of the total number of script functions in the web applications and between 32 % and 46 % of the total amount of code inside script functions. They also reported that in one of the subjects, IBIS, the refactoring opportunities for client-side and server-side function clones are 39 % and 50 %, respectively. However, their approach to detect cloned script functions reports only potential function clones, and it requires visual inspection of selected script functions.

Calefato et al. (2004) identified cloned functions within scripts in web applications and evaluated recall, precision, and refactoring opportunities of their approach. They found that most of the server-side scripts in VBScript functions in two of the web applications that they had studied, QuickAuction (57 %) and Web Wiz Forums (80 %), can be refactored. They also found that more than one-third (36 %) of the script functions in JavaScript in the IBIS application could be improved by means of refactoring.

Rajapakse and Jarzabek (2005) studied clones in 17 anonymous web applications in multiple statically-typed and dynamically-typed languages such as Java, JSP, and Python in terms of total cloned tokens, file similarity, and qualifying file counts. They observed cloning rates of 17–63 % in both newly developed and already maintained web applications. However, they did not compare clones in statically-typed and dynamically-typed languages because they did not provide the properties such as names and lines of code for each web application. In addition, their underlying clone detector, CCFinder (Kamiya et al. 2002), does not support all the languages in their subjects. Therefore, they treated all input files as plain texts and detected only exact clones. They also ignored clones in the same files to reduce the number of reported clones.

De Lucia et al. (2007) represented web pages as strings of HTML tags and clustered them by an Artificial Neural Network clustering algorithm. However, their approach produced a set of results with different trade-offs of precision and recall, and they needed to choose, among the results, the one with the best F-measure. Our study differs from theirs in that we do not have to choose the best result among multiple ones since our evaluation showed that our clone detector has better recall than and comparable precision to existing tools in detecting JavaScript clones. Also, their study targets code clones of HTML tags while we target JavaScript embedded in web pages.

Selamat and Wahid (2007) combined frequent subgraph mining and string-based matching techniques to detect clones in web applications containing HTML, ASP, and PHP code. They noticed that the recall of their approach is low and most of the detected clones are identical clones.

Ramage et al. (2009) compared the clustering algorithm of web pages by k-means clustering with tags and page text as well as a generative clustering algorithm based on latent Dirichlet allocation with tags and page text. They found that including tags alone performs better than including page text alone. They also found that the generative model has 8 % higher F-measure than that of k-means. Our work differs from theirs in that their work did not consider any programming languages and just treated the web pages as documents. Also, they performed clustering on the texts and tags while we focus on identifying code clones in embedded JavaScript in web pages.

De Lucia et al. (2009) compared three clustering algorithms — a divisive clustering algorithm, a variant of the k-means partitioning clustering algorithm, and a competitive clustering algorithm — in identifying similar web pages at a structural level by Levenshtein edit distance and at a content (semantic) level by Latent Semantic Indexing. They evaluated the algorithms in two web applications and found that precision and recall are similar at both the structural and content levels. They also found that, at the content level, using k-means with the no-single cluster configuration has similar F-measure to that of the trade-off configuration. Our work differs from theirs in that they identify similar web pages in HTML while we detect JavaScript clones in web pages. Also, they identified similar web pages at both the structural and content levels while we detect only structural clones. It is well known that semantic clones are, in general, hard to be detected (Bellon et al. 2007; Kim et al. 2011; Koschke et al. 2012) and that only a few tools can detect them. In the worst case, single-link-based agglomerative hierarchical clustering algorithms achieved a precision of only 21.8 % in their results. Also, while their work relies on human experts in evaluating the precision and recall of the clustering algorithms, human experts may be biased in judging clones. However, the mutation technique we applied allows us to objectively evaluate the precision and recall of clone detectors.

Blanco et al. (2011) clustered web pages by their URLs and reported that they can cluster a website with 700,000 pages in 26 seconds. Using URLs as a means of clustering is too coarse-grained that it cannot distinguish slight differences between web pages. However, clone detection can identify similar web applications in a fine-grained manner. Specifically, this work focuses on detecting code clones of embedded JavaScript in web pages.

Islam et al. (2011) detected code clones in server pages of ASP.NET web applications and found that they have cloning rates of 10-23 %. They also found that applications developed in the classic ASP.NET framework and the ASP.NET MVC framework have significant quantitative differences in cloning. Our work differs from theirs in that they detected clones in server-side ASP.net web applications while we detect JavaScript clones in client-side web pages. Also, they analyzed the clones only quantitatively but did not further investigate why they observed such differences in their findings.

Jones (2011) conducted a large-scale clone detection study on 6,190 PHP applications and noticed that almost 90 % of projects contain code clones.

Martin and Cordy (2011) detected contextualized clones to identify similar web services in Web Service Description Language (WSDL). They defined contextualized clones as clones of modified or expanded code fragments that include information referenced elsewhere. However, they reported only the number of clones detected and did not conduct any further analysis.

Kou and Lou (2012) used hierarchical clustering to identify similar web pages on a collection of clickstream data and reported that the average coherence and utility scores in a 5-point scale are 4.46 and 4.90, respectively. However, their work did not target any particular programming languages, and hence, their clustering of web pages is coarse-grained. Such coarse-grained clustering would not be able to identify slight differences between web pages as clone detection does.

Muhammad et al. (2013) studied exact and near-miss PHP clones in two industrial dynamic web applications. They noticed that the PHP code in both web applications has a significant number of code clones and the system developed by the traditional page-based approach has more scattered clones than the system developed by a modular implementation following the MVC pattern.

Negara et al. (2013) clustered Ajax-enabled web applications with a DOM tree similarity metric and found that their proposed distance metric is more accurate than other traditional distance metrics in classifying the features of the applications. Our work differs from theirs in that they removed the `script` and `style` tags in web applications and clustered the remaining `HTML` tags while we focus on detecting code clones of JavaScript in the `script` tags.

Li et al. (2014) proposed a variation of the algorithm in Deckard (Jiang et al. 2007a) to detect code clones in Java applications and PHP web applications. Their approach is based on randomized kd-trees with dimensionality reduction to cluster characteristic vectors. They found that their approach could largely reduce the dimension of characteristic vectors and computation time in JDK. They also found that PHP web applications have cloning rates of 5-82 %. Our work is similar to theirs in that the underlying clone detection methodologies are similar because the algorithm of our clone detector is based on that of Deckard. Although the focus of this study is not on clone detection algorithms, it would be desirable to implement their algorithm to compare it with our clone detector since their clone detector is not publicly available. Also, both studies found that web applications have higher percentages of code clones than traditional applications. Our work differs from theirs in that their focus is on clone detection methodology; therefore, they reported only the lines and percentages of cloned code without further study. This study focuses on the motivation of cloning and, hence, conducts further investigations on how developers clone.

The above studies have the following limitations. Despite the prevalence of JavaScript in web applications, only a limited amount of research has examined JavaScript. In addition, such research has mainly focused on languages in web applications, but has seldom compared the properties of statically-typed and dynamically-typed languages, as well as different application domains. Our study differs from theirs in that we study the clone properties of JavaScript in web applications and also compare them with those in JavaScript and Java standalone projects.

2.5 Applications of Clone Detection in Scripting Languages and Web Applications

Researchers have applied clone detection for scripting languages and web applications to other areas such as identifying cloning patterns and achieving clone-free web applications.

Kienle et al. (2003) discussed the usefulness of generated clones for the web domain using a website that they developed for illustration.

De Lucia et al. (2005) identified and analyzed cloning patterns in web applications using clone analysis and clustering of static and dynamic web pages. They experimented with a conference website and visualized its navigation schema. They observed that large clones mainly came from pages related to the conference registration, hotel reservation, paper submission, and visualization of keynote speakers and conference sessions.

Rajapakse and Jarzabek (2007) conducted a case study to explore how far the PHP server page technique can be pushed to achieve clone-free web applications. Their study showed that the page generation time of clone unification using server pages becomes three times slower when the cloning level decreases.

The above studies show various applications of detecting clones in web applications and identifying special features in web applications. In the last part of our study, we identify common cloning patterns among `JSweb`, `JSproj`, and `Java`, and also qualitatively analyze the development practices specific in web applications.

2.6 Relationship Between Clone Properties and Software Metrics

Understanding the relationship between clone properties and software metrics helps to investigate costs and benefits of clone removal (Koschke 2007). While some researchers have studied the impacts of code clones to code quality (Bettenburg et al. 2012; Krinke 2007; Lozano et al. 2008), only a limited amount of research has examined whether certain clone properties are closely related to any software metrics.

Monden et al. (2002) studied the relationship between code clones and the software reliability and maintainability of a 20-year-old software. They found that on average modules having code clones are more reliable than modules having no clones. They also found that clone-included modules are less maintainable than modules containing no clones, and modules having more code clones are less maintainable than modules having fewer code clone.

Kozlov et al. (2010) studied the relationship between code clone metrics and internal quality attributes on 117 releases of eight eMule software project forks. They identified a number of important relationships between the metrics; e.g., the number of statements correlates positively with inter-file clones and number of files having at least one code clone.

Hegedűs et al. (2011) studied the correlation between software metrics, but they did not find any strong correlation between clone coverage and other software metrics.

While the above research studies the relationship between code clones and software metrics, they focus on a single programming language. To the best of our knowledge, there is no prior work examining the differences of relationship in different languages and application domains. In this study, we found that the correlation in `JSproj` is more similar to that in `Java` than in `JSweb`.

2.7 Summary of Literature Review

Our literature review shows that previous code clone studies have the following limitations: (1) since most of the studies were conducted on statically-typed languages such as Java and C/C++, there is a lack of studies on dynamically-typed languages or comparison of both kinds; (2) previous studies did not study code clones of the same language in different application domains, such as web applications and standalone projects, given that they are of different development natures; (3) the precision and recall of language-independent clone detectors vary from language to language; (4) while the languages HTML, ASP, and PHP have mostly been studied in web applications, studies on JavaScript are limited despite its prevalence; and (5) many studies have focused on comparing quantitative differences, and only a limited number of them have qualitatively analyzed the factors that may affect cloning.

This study addresses the above limitations by (1) studying code clones in both statically-typed and dynamically-typed languages, i.e., Java and JavaScript, (2) comparing the differences of code clones between JavaScript web applications and standalone projects, (3) developing and evaluating a clone detector targeted for JavaScript with high precision and recall, and (4) both quantitatively and qualitatively analyzing the factors that may affect the ways developers duplicate code.

3 JSCD: JavaScript Clone Detector

Many clone detection approaches have been proposed in the literature. Among different programming languages, Java and C/C++ are the languages that most clone detectors support (Roy et al. 2009). The number of clone detectors that can detect JavaScript code clones is limited. While several clone detectors are available for JavaScript (Baxter et al. 1998; Harris 2013; PMD 2013; SAFE Corporation 2012; Schleimer et al. 2003), most of them are either commercial software or have a low recall in detecting clones (Burd and Bailey 2002; Krinke 2007, 2008; Roy et al. 2009).

Among tools that can detect duplication of JavaScript code, MOSS (Schleimer et al. 2003), Simian (Harris 2013), and PMD (PMD 2013) are widely studied in the literature (Burd and Bailey 2002; Hill and Rideout 2004; Hotta et al. 2010; Krinke 2007, 2008, 2011; Li and Ernst 2012; Roy et al. 2009; Wang et al. 2013). MOSS is a general plagiarism detection tool for multiple languages that supports JavaScript. Previous studies have revealed that the recall of MOSS on Java is only 10 % (Burd and Bailey 2002). Also, researchers suggested that MOSS is very coarse-grained and is not suitable for clone detection (Jiang et al. 2007a). In addition, an extensive comparison of clone detection tools on 17 Java and C systems revealed that both Simian and PMD (Roy et al. 2009) are good at detecting identical clones but not renamed and gapped clones. The literature has shown that the above tools have low recall in detecting clones in Java and C. However, it is not known that whether their findings also apply to JavaScript because previous evaluations of these tools were conducted on other languages, but not JavaScript. The limited number of existing clone detectors for JavaScript and the types of clones that they can detect suggest the need for a clone detector for JavaScript with better recall and precision. The lack of clone detector evaluation on JavaScript also suggests the need to check whether the findings in the literature also apply to JavaScript.

In this study, we developed JSCD, a clone detector for JavaScript inspired by Deckard, a state-of-the-art C/Java/PHP clone detector (Jiang et al. 2007a). As with Deckard, JSCD first parses JavaScript source files into Abstract Syntax Trees (ASTs). It then approximates the structural information of ASTs as characteristic vectors, which are a set of fixed-dimension integer vectors representing the occurrences of the nodes in the ASTs. It uses Locality Sensitive Hashing (LSH) (Datar et al. 2004) to cluster similar vectors by their Euclidean distances into clone groups. LSH is a technique that hashes two similar vectors to the same hash value with high probability and hashes two distant vectors to the same hash value with low probability.

Our clone detector is integrated into SAFE (Lee et al. 2012; PLRG@KAIST 2012), an open-source scalable analysis framework for ECMAScript. SAFE provides a formal specification and implementation of a general framework for analyzing JavaScript programs. We used SAFE for parsing JavaScript programs and designed the characteristic vectors specifically for JavaScript, utilizing the AST nodes generated by SAFE. SAFE is an analysis framework specially designed for JavaScript, and it supports parsing both JavaScript files and embedded JavaScript code in web applications. The parser of SAFE complies with the 5th edition of the ECMAScript specification, the standardized specification that most JavaScript programs use nowadays, while most parsers for JavaScript comply with only the 3rd edition (Lee et al. 2012). Therefore, our clone detector can capture the AST nodes of JavaScript code in parsing more accurately than other language-independent clone detectors.

While most clone detectors detect identical clones well, tree-based clone detection techniques can detect renamed and gapped clones better than text-based (Simian) and token-based (PMD) techniques. In the comparison of clone detection techniques, Roy et al. (2009) discussed the drawbacks of text-based and token-based techniques in detecting renamed and gapped clones. Text-based techniques are not good at detecting renamed clones because, without normalization or transformation, they cannot identify the differences between identifiers and literals. Token-based techniques in detecting renamed clones are likely to have many false positives due to spurious clones such as sequences of assignments and very large initializer lists for arrays (Koschke et al. 2006). Tree-based techniques can detect renamed clones well because they usually ignore identifiers and literals in comparison. Both text-based and token-based techniques are not suitable in detecting gapped clones unless they apply threshold-based comparison or combine smaller identical and renamed clones in a post-processing phase. However, the use of characteristic vectors in clone detection can detect gapped clones well because the vectors approximate the structural information of ASTs in the Euclidean space. To enable other researchers to replicate our study, we made our clone detector publicly available.[1]

Since both JSCD and Deckard detect clones by measuring the similarity of characteristic vectors, we can use the same configuration for both tools. Thus, we use the same settings for `similarity` (the editing distance of two ASTs, which is smaller than a certain threshold), `minT` (the minimum number of tokens required for clones), and `stride` (the size of the sliding window).

In the rest of this section, we first discuss the challenges of evaluating clone detectors and briefly introduce an automatic evaluation technique applied. We then describe the subjects studied and the experimental setup. Lastly we present our results of evaluating JSCD with other two clone detectors by automatic and manual evaluations.

3.1 Evaluation

The evaluation of clone detectors has long been a challenge to the code clone community. There are several obstacles to overcome such as the absence of a reliable reference set (Rysselberghe and Demeyer 2004), and the fact that the union of all the results by the tools does not guarantee the detection of all the clones (Bellon et al. 2007; Burd and Bailey 2002; Falke et al. 2008). Also, oracling a system requires a great deal of human effort (Bellon et al. 2007).

Given the limitations of existing clone detection techniques, we would like to reliably evaluate the recall and precision of our clone detector while minimizing the amount of human effort involved. We applied the technique from Roy and Cordy (2009), which is widely used in other studies (Roy and Cordy 2010b; Saha et al. 2011; Stephan et al. 2013, 2014; Zibran and Roy 2012), to evaluate JSCD and other tools. Their approach mainly comprises two phases: generation and evaluation. The generation phase creates benchmarks by mutating code fragments to artificially create and inject known code clones. The evaluation phase uses these benchmarks to measure the recall and precision of clone detection tools. Since the locations of the injected code clones are known, their approach enables us to evaluate recall without manual intervention. Also, their approach is aware of

[1] http://safe.kaist.ac.kr

the types (Type-1, -2, -3) of injected clones, and such information allows us to evaluate precision automatically by pairwise validation of clone pairs. For instance, they suggested that Type-1 clones can be verified by pretty-printing while Type-2 and -3 clones can be verified by normalization and a dissimilarity threshold, respectively.

We evaluate JSCD with two clone detectors, PMD (2013), and Simian (Harris 2013), which are capable of detecting code clones in JavaScript. PMD is a token-based clone detector that detects Type-1 and -2 clones while Simian is a text-based clone detector that detects only Type-1 clones.

3.2 Generation Phase

We create benchmarks for evaluation utilizing the technique of Roy and Cordy (2009). The generation phase first selects any desired number of code fragments from the subject code base automatically and randomly for clone mutation. Then, it applies mutant operators to mutate the selected code fragments. For each of the mutated code fragments, it creates a mutated code base by injecting the fragment into a random location in a random file of the original code base. It records the locations of the injected clones in a database. A mutated code base consists of the original code base and the injected randomly mutated code fragment from the original code base.

We evaluated all three clone detectors on five JavaScript standalone projects: Bootstrap, Web Font Loader, script.aculo.us, Foundation, and jQuery. Table 2 presents an overview of the subjects used for mutation. It contains the subject name, version number, lines of non-comment JavaScript code, number of files, and number of mutated code bases.

For each code base, we randomly select 50 functions and apply each of the 13 mutators in Table 3 from Svajlenko et al. (2013) to each code fragment 20 times, yielding at most 13,000 code bases for each JavaScript project. Note that the total amount of code bases may not reach 13,000 in some subjects since they may not have 50 functions or some mutation operators such as mRL_N for number literals and mRL_S for string literals may not be applicable to them. The last column of Table 2 indicates the number of mutated code bases for each subject.

Table 2 Mutated subjects overview

Subject	Version	LOC	Files	Code Bases
Bootstrap[a]	3.0.0	1185	12	11948
Web Font Loader[b]	1.5.4	1267	21	11544
script.aculo.us[c]	1.9.0	2228	9	6091
Foundation[d]	4.3.1	3900	17	7178
jQuery[e]	2.0.3	4827	27	7737

[a]https://github.com/twbs/bootstrap/tree/v3.0.0

[b]https://github.com/typekit/webfontloader/tree/v1.5.4

[c]https://github.com/madrobby/scriptaculous/tree/v1.9.0

[d]https://github.com/zurb/foundation/tree/v4.3.1

[e]https://github.com/jquery/jquery/tree/2.0.3

Table 3 Mutation operators

Name	Mutation Description	Clone Type
mCW_A	Change in whitespace (addition)	1
mCW_R	Change in whitespace (removal)	1
mCF_A	Change in formatting (addition of newlines)	1
mCF_R	Change in formatting (removal of newlines)	1
mSRI	Systematic renaming of an identifier	2
mARI	Arbitrary renaming of a single identifier	2
mRL_N	Change in value of a single numeric literal	2
mRL_S	Change in value of a single string literal	2
mSIL	Small insertion within a line	3
mSDL	Small deletion within a line	3
mILs	Insertion of a line	3
mDLs	Deletion of a line	3
mMLs	Modification of a whole line	3

3.3 Evaluation Phase

The evaluation phase runs each of the clone detectors with its tunable parameter set to target the clone types on each of the mutated code bases. It measures the unit precision and recall by comparing the clone reports with the injected clone pairs from the database created by the generation phase. After computing the unit recall and precision for each tool with each mutant code base, it computes the summary of the precision and recall of each tool for individual mutation operators, each clone type, and the overall precision and recall.

Table 4 presents the results of evaluating all three clone detectors on the subject Bootstrap. For Type-1 clones overall, both Simian and PMD achieve more than 60 % recall while JSCD achieves an even better recall of 97 %. All the three tools achieve 100 % precision in detecting Type-1 clones. This shows that JSCD can detect more Type-1 clones than Simian and PMD.

Our manual inspection reveals that the design of the tools limits their ability in detecting non-pretty-print code as well as code with ill-formatted styles such as indentation, whitespace, or positioning of braces. We found that Simian does not detect any clones in the case of removal of newlines. Our investigation reveals that Simian needs to configure the minimum number of lines in clone detection. This limits its flexibility to detect clones of different sizes. We also notice that PMD has a relatively low recall in detecting Type-1 clones with changes in new lines, which are only 56 % in mCF_A and 34 % in mCF_R. Our manual analysis reveals that the clone reports of PMD assume that the clone instances in the same clone group have the same number of lines of code. The design of their tool limited their ability to detect code clones with changes in newlines.

For Type-2 clones overall, Simian achieves less than 40 % recall while PMD achieves only 20 %. However, JSCD achieves a better recall (97 %) than those of the other two tools. For precision, both Simian and JSCD achieve 100 % and PMD achieves 97 %, showing that all three tools have high precision in detecting Type-2 clones.

Table 4 Recall and precision of evaluating clone detectors on Bootstrap

Type	Mutator	Simian		PMD		JSCD	
		Recall	Precision	Recall	Precision	Recall	Precision
1	mCW_A	78	100	69	100	97	100
	mCW_R	94	100	91	100	97	100
	mCF_A	94	100	56	100	97	100
	mCF_R	0	–	34	100	97	100
Type-1 overall		66	100	62	100	97	100
2	mSRI	7	100	18	97	96	100
	mARI	19	100	38	96	97	100
	mRL_S	68	100	6	100	95	100
Type-2 overall		36	100	20	97	97	100
3	mSIL	2	100	3	100	97	97
	mSDL	3	97	3	89	92	95
	mILs	3	91	3	36	59	94
	mDLs	2	89	2	86	51	89
	mMLs	21	87	21	89	65	95
Type-3 overall		6	89	6	84	72	94
Total overall		34	96	29	98	87	96

Our manual investigation shows that PMD is able to detect Type-2 clones. However, the clones detected by PMD are only some parts (in terms of start and end line numbers) of the injected clones. According to the definition of *detected* by Roy and Cordy (2009), the detected clones should contain the injected clones, and hence, PMD misses the injected clones.

For Type-3 clones overall, both Simian and PMD achieve only 6 % recall while JSCD achieves more than 70 %, which show that JSCD detects many more Type-3 clones than Simian and PMD. For precision, JSCD detects Type-3 clones with 94 % precision, followed by 89 % in Simian, and 84 % in PMD, which represent 5–10 % lower in accuracy compared to JSCD. Both Simian and PMD are not designed for detecting Type-3 clones, and hence they can barely detect them.

Table 5 Scraperjs overview

Subject	Version	LOC	Files	Type-1	Type-2	Type-3
Scraperjs	0.3.0	573	8	1	6	20

Our evaluation results show that JSCD overall achieves 53–58 % higher recall and comparable precision in detecting JavaScript code clones than both Simian and PMD.

3.4 Clone Oracle

To reduce the bias of clone detector evaluation results toward the use of mutation, we verified whether the tools can detect real clones in a subject. We selected a small-scale JavaScript subject, Scraperjs,[2] among the most popular repositories from the trending repositories of GitHub.[3] We manually identified all Type-1, -2, and -3 clones in Scraperjs and computed the recall and precision of the tools. Such manual evaluation is feasible due to the small size of the subject. Table 5 shows the properties of Scraperjs. It includes version number, lines of code, number of files, and number of Type-1, -2, and -3 clones. Note that evaluation of such a small subject alone is not representative of the performance of individual clone detectors. This evaluation serves as a supplementary verification for the usefulness of the mutation technique in evaluating the tools.

To identify clones in the subject, we first manually collected all non-empty blocks of code from the source files. We used blocks as the clone granularity because they have predefined syntactic boundaries. A non-empty block consists of at least one code statement enclosed in brackets "{ }". A code statement is a line of code that ends with a semicolon ";". For each pair of blocks with more than three lines, we manually identified whether it belongs to Type-1, 2, or 3 clones, or is not a clone at all. We call the manually identified clones *reference clones* and the clones reported by the tools *detected clones*. To enable other researchers to replicate our results, we made the information of our manually identified clones publicly available.[4]

We evaluated the recall and precision of the clone detectors in a similar way to that of the mutation experiment. For each tool, we computed the unit recall and precision of each reference clone pair, the total recall and precision of each clone type, and the total recall and precision of the tool. We considered a reference clone pair as *detected* by a tool and computed the recall and precision in a similar way to that of Roy and Cordy (2009). A reference clone pair is detected by a tool if the detected clone pair subsumes the reference clone pair by containment. Unit recall for a reference clone pair is 1 if the tool detects it; otherwise, it is 0. Unit precision is the number of detected clone instances that are clones with the reference clone pair divided by the total number of clone instances in the detected clone group.

Table 6 presents the evaluation results of the clone detectors. It shows the recall and precision of each clone type, as well as the total recall and precision. All tools could detect Type-1 clones. For Type-2 clones, even Simian and PMD achieved 100 % precision, both of them had only 33 % recall. JSCD could detect all Type-2 clones and was 67 % higher recall than Simian and PMD. For Type-3 clones, Simian and PMD achieved 25 % and 40 % recall, respectively, and both of them achieved 100 % precision. However, JSCD achieved 80 % recall and 85 % precision, which was 40–60 % higher recall than that of Simian and

[2]https://github.com/ruipgil/scraperjs/tree/v0.3.0

[3]https://github.com/trending?l=javascript&since=monthly

[4]http://plrg.kaist.ac.kr/research/material

Table 6 Recall and precision of evaluating clone detectors on Scraperjs

Type	Simian		PMD		JSCD	
	Recall	Precision	Recall	Precision	Recall	Precision
1	100	100	100	100	100	100
2	33	100	33	100	100	100
3	25	100	40	100	80	85
Overall	30	100	41	100	85	90

PMD. Overall, the evaluation results on the actual clones of a subject are consistent with those of mutation evaluation. Therefore, we utilized JSCD to detect JavaScript code clones in our empirical study.

4 Empirical Study Design

We design our experiments to address the following research questions:

- **RQ1 (Clone properties): Are there any differences in clone properties among JSweb, JSproj, and Java?** We compared the clones from each category of subjects using various code clone properties.
- **RQ2 (Software metrics): Are there any software metrics closely related to the clone properties among JSweb, JSproj, and Java?** We computed the correlation between each pair of software metrics and clone properties and identified closely related pairs.
- **RQ3 (Cloning patterns): What kinds of code are being cloned in JSweb, JSproj, and Java?** We manually inspected the clones from each category of subjects and identified the patterns of clones.

4.1 Subjects

In this study, we analyzed JavaScript code embedded in web pages from 10 websites, 10 JavaScript standalone projects, and 10 Java projects. Table 7 shows the properties of the subjects, which include their names, types, versions, lines of code, number of files, and application types. Lines of code for JSweb represent the lines of non-comment JavaScript code extracted from the script tags of the web pages; it does not include external JavaScript files. Each web page is a single HTML file collected from the websites, and hence, the number of files in JSweb represents the number of HTML files collected. The lines of code and number of files for JSproj and Java represent the lines of non-comment JavaScript and Java code and files, respectively. We selected subjects of different application types to reduce the bias of our experimental results towards certain kinds of applications.

For JSweb, we collected the web pages from 10 English websites among the top 15 Alexa sites (Alexa Internet 2013). We obtained the web pages by manually browsing the websites on our client browser Safari and saved the pages delivered to the browser. Due to the large scale of the websites, we were not able to collect all the pages from each site. Therefore, we collected only the homepages of the services accessible from the homepage of each website and the pages under these service homepages. An example of web pages collected from `wikipedia.org` includes the homepages of `Wikitionary`, `Wikinews`, `Wikiquote` and the pages under these service homepages.

We manually verified all the collected pages for each website to find and eliminate any potential groups of cloned web pages. We identified a set of web pages as cloned pages when they were visually similar: the pages have the same layout but vary only in contents such as texts, images, and videos. Table 7 shows the number of web pages from each site used in our study after eliminating potential groups of cloned pages. To enable the replication of

Table 7 Subjects overview

Subject	Type	Version	LOC	Files	Application Type
`twitter.com`	`JSweb`	2014.05.15	282	25	Social networking
`wikipedia.org`	`JSweb`	2014.05.15	575	27	Encyclopedias
`youtube.com`	`JSweb`	2014.05.15	748	21	Video sharing
`facebook.com`	`JSweb`	2014.05.15	874	34	Social networking
`amazon.com`	`JSweb`	2014.05.15	1971	42	Shopping
`wordpress.com`	`JSweb`	2014.05.15	3286	23	Weblogs
`ebay.com`	`JSweb`	2014.05.15	3702	72	Shopping
`linkedin.com`	`JSweb`	2014.05.15	4056	51	Social networking
`google.com`	`JSweb`	2014.05.15	4290	47	Search engines
`yahoo.com`	`JSweb`	2014.05.15	21308	68	Web portals
D3.js[a]	`JSproj`	3.3.3	553	257	Visualization library
Bootstrap	`JSproj`	3.0.0	1185	12	Front-end framework
Foundation	`JSproj`	4.3.1	3900	17	Front-end framework
jQuery	`JSproj`	2.0.3	4827	27	Feature-rich library
Ionic[b]	`JSproj`	0.9.13	5171	60	Mobile app framework
three.js[c]	`JSproj`	r61	19656	156	3D library
AngularJS[d]	`JSproj`	1.1.5	37328	372	MVW framework
SproutCore[e]	`JSproj`	1.9.2	113343	925	Web app framework
Brackets[f]	`JSproj`	34	180208	987	Code editor
Closure Library[g]	`JSproj`	c8e0b2dcd892	215920	925	Web app library
ArgoUML[h]	`Java`	0.34	6196	37	UML modeling tool
EIRC[i]	`Java`	1.0.3	8269	65	IRC client
Javadoc[j]	`Java`	331	9579	101	Document generator
dnsjava[k]	`Java`	2.1.5	15873	130	DNS protocol
Ant[j]	`Java`	2002.02.15	16106	178	Build tool
cpptasks[l]	`Java`	1.0b5	16903	183	Task compiler
JHotDraw[m]	`Java`	7.0.6	32430	309	GUI framework
Plandora[n]	`Java`	1.13.0	89384	719	Project management tool

Table 7 (continued)

Subject	Type	Version	LOC	Files	Application Type
JDT Core[j]	`Java`	2002.02.15	98169	741	IDE infrastructure
JDK Swing[j]	`Java`	1.4	102836	538	GUI framework

[a]https://github.com/mbostock/d3/tree/v3.3.3

[b]https://github.com/driftyco/ionic/tree/v0.9.13-alpha

[c]https://github.com/mrdoob/three.js/tree/r61

[d]https://github.com/angular/angular.js/tree/v1.1.5

[e]https://github.com/sproutcore/sproutcore/tree/REL-1.9.2

[f]https://github.com/adobe/brackets/tree/sprint-34

[g]https://code.google.com/p/closure-library/source/browse/?r=c8e0b2dcd892

[h]http://argouml-downloads.tigris.org/argouml-0.34/

[i]http://sourceforge.net/projects/eirc/files/EIRC/1.0.3/

[j]http://www.bauhaus-stuttgart.de/clones/

[k]http://www.dnsjava.org/download/

[l]http://sourceforge.net/projects/ant-contrib/files/ant-contrib/cpptasks-1.0-beta5/

[m]http://sourceforge.net/projects/jhotdraw/files/JHotDraw/JHotDraw%207.0.x/

[n]http://sourceforge.net/projects/plandora/files/version/plandora-v.1.13.0/

our research, we archived our collected web pages and made them publicly available.[5]

For `JSproj`, we selected eight trending open-source projects ranked by GitHub from their repository (GitHub Inc 2013)—D3.js, Bootstrap, Foundation, jQuery, Ionic, three.js, AngularJS, and Brackets—and two commonly used projects in the literature—Closure Library and SproutCore.

For `Java`, we chose 10 projects—ArgoUML, EIRC, Javadoc, dnsjava, Ant, cpptasks, JHotDraw, Plandora, JDT Core, and JDK Swing—studied by Mondal et al. (2012), Roy and Cordy (2010a), Bellon et al. (2007), and Aversano et al. (2007). We selected subjects of different application types including an IRC client and a build tool.

4.2 Clone Detectors and Code Clone Metrics

We used Deckard (Jiang et al. 2007a) and `JSCD` (Section 3) to detect clones with a combination of configurations of `similarity = 0.85, 0.9, 0.95` and `minT = 15, 30, 50`, yielding nine combinations in total. Since `stride` is a Deckard-specific configuration, unlike `similarity` and `minT`, which are commonly used in other clone detectors, we used `stride = 0`.

To compare code clones in different languages and different application domains, we used the code clone metrics summarized in Table 8. They are commonly used in the literature (Livieri et al. 2007; Rieger et al. 2004; Roy and Cordy 2008; Tairas and Gray 2006).

To measure the clone properties, we implemented a tool to automatically compute the distributions of clones for each metric. In each stage of the development of the tool, we

[5]http://plrg.kaist.ac.kr/research/material

Table 8 Code clone metrics

Metric	Definition
Cloning locality (Kapser and Godfrey 2003)	Relative locations of a clone pair
Average and maximum lines of cloned code (Kapser and Godfrey 2003)	Average and maximum lines of cloned code in a system
Clone coverage (Kim et al. 2005)	Ratio of cloned code to the total lines of code
Files associated with clones (Roy and Cordy 2010a)	Proportion of files that have at least one method that forms a clone pair with another method in the same file or in a different file
Function-level clones (Calefato et al. 2004)	Clones of entire functions

verified the distribution computation in the tool by manually inspecting random samples of the clones for each metric.

5 Results

This section presents our findings in the code clone metrics, the correlation between clone properties and software metrics, and the cloning patterns. By looking at different clone metrics, we identified features of programming languages and technologies in the subjects that may affect cloning and its maintenance. We measured which pairs of clone properties and software metrics have a strong correlation, and we manually inspected the clones to identify cloning patterns.

5.1 RQ1: Clone Metrics

In this section, we investigate the main differences in code clone metrics in `JSweb`, `JSproj`, and `Java`. We measured the clone metrics for different combinations of similarity and minimum number of tokens in clone detection and plotted them as graphs. The plots show the median values of the metrics from the 10 subjects in each of `JSweb`, `JSproj`, and `Java`.

In statistical analysis, the mean is very sensitive to outliers because each value is involved in the computation of the mean. The mean is representative only when the data is normally distributed. However, the median is not very sensitive to changes in the data and it still maintains the central position in a skewed distribution. Since we cannot assume that the distribution of data is symmetric in our experiments, plotting graphs with the medians gives a better representation of the summary of data than doing so with the means.

Table 9 Comparison of findings between this work and that of Roy and Cordy (2010a)

Metric	This work			Roy and Cordy (2010a)	
	JSweb	JSproj	Java	Python	Java
Inter-file clones	95 %	40 – 80 %	60 – 90 %	70 – 90 %	70 – 80 %
Files associated with clones	91 – 97 %	33 – 87 %	36 – 83 %	25 – 60 %	45 – 75 %
Function-level clones	5 – 7 %	3 – 6 %	22 – 48 %	5 – 29 %	9 – 30 %

For inter-file clones and files associated with clones, the percentages between `JSproj` and Python, and also between `Java` in both works are similar. For function-level clones, only Java shares similar percentages.

Our literature review showed that our experiments in this section are closest to that of Roy and Cordy (2010a). Despite the fact that our experimental configurations are slightly different from theirs, we present a summary of results in Table 9 for comparison. In the following sections, we compare our findings with theirs in detail.

5.1.1 Cloning Locality

We measure the cloning locality (Kapser and Godfrey 2003) by the percentages of intra-file and inter-file clones across the file systems. Kapser and Godfrey (2003) suggested that clones across different files might lead to larger code sizes and more labor for error fixing. Developers may clone codes in different files because they are unaware of the existing clones in the system, and thus, such clones would be better refactored as a set of library functions.

Figure 2 shows the distributions of inter-file clones for different combinations of similarity and minimum number of tokens. When the similarity increases, the percentages of inter-file clones in `JSweb` remain at approximately 95 % while those in `JSproj` and `Java` decrease by 20–40 %. The percentages of inter-file clones in `JSproj` reduce from approximately 81–75 % to 40–60 %, and in `Java`, the percentages decrease from approximately 90 % to 60 %. However, different minimum number of tokens of the same similarity do not make much difference in the percentages of inter-file clones for any of the three groups.

Roy and Cordy (2010a) found that the percentages of inter-file clones in Java are approximately 70–80 %. Our finding, 60–90 %, has a slightly larger range than theirs. This can be explained by the differences between the choices of subjects and different clone detection configurations. We also compare our findings in `JSweb` and `JSproj` with Python systems since both JavaScript and Python are scripting languages. They found that Python systems have approximately 70–90 % of inter-file clones. While we found 40–80 % of inter-file clones in `JSproj`, percentages of inter-file clones for `JSweb` remain at approximately 95 % for different similarities and minimum number of tokens. This indicates that, even though JavaScript and Python are both scripting languages, they do not show similar clone localization, especially compared with `JSweb`.

Web applications differ from standalone projects in that the former fetches the source code from the server during execution while the latter requires a local copy of the source code before execution. We manually investigated clones in different files in `JSweb` and found that such a property affects the way developers duplicate code. Figure 3 shows an example of inter-file clones from `wordpress.com`. We found this duplicated code fragment in 16 out of 23 pages collected from `wordpress.com`. It detects if a user is browsing the site with a mobile device, creates a corresponding query string, and sets the source of

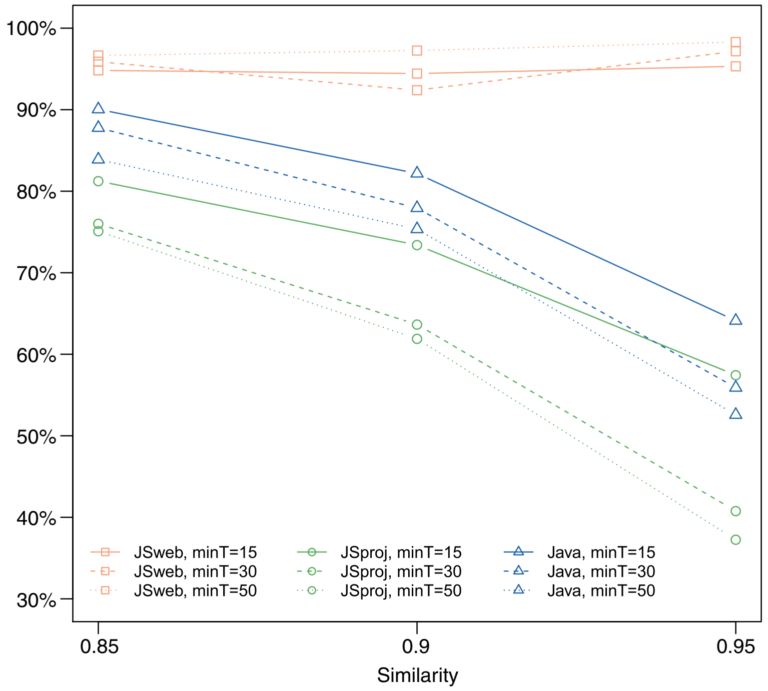

Fig. 2 Percentages of inter-file clones. `JSweb` remains at approximately 95 % for different configurations while `JSproj` and `Java` decrease with increasing similarity. The large percentages of inter-file clones in `JSweb` are intended for faster web page loading

```
if ( 'object' === typeof wpcom_mobile_user_agent_info ) {

 wpcom_mobile_user_agent_info.init();
 var mobileStatsQueryString = "";

 if( false !== wpcom_mobile_user_agent_info.matchedPlatformName )
  mobileStatsQueryString += "&x_" + 'mobile_platforms' + '=' +
    wpcom_mobile_user_agent_info.matchedPlatformName;

 if( false !== wpcom_mobile_user_agent_info.matchedUserAgentName )
  mobileStatsQueryString += "&x_" + 'mobile_devices' + '=' +
    wpcom_mobile_user_agent_info.matchedUserAgentName;

 if( wpcom_mobile_user_agent_info.isIPad() )
  mobileStatsQueryString += "&x_" + 'ipad_views' + '=' + 'views';

 if( "" != mobileStatsQueryString ) {
  new Image().src = document.location.protocol + '//stats.wordpress.
    com/g.gif?v=wpcom-no-pv' + mobileStatsQueryString + '&baba=' +
    Math.random();
 }

}
```

Fig. 3 Inter-file clones example from `wordpress.com`. It changes the image source if a user is browsing the site with a mobile browser

the image for that query string. Even though the same piece of code is used in many pages, inlined JavaScript may produce pages faster than caching external JavaScript if the script is small and the page view per session is small because external JavaScript cannot benefit from browser caching in such circumstances.[6,7] Roy and Cordy (2007) suggested that introducing clones by code inlining is a way to address the high cost of function calls in real-time programs, in case the compiler does not offer such functionality automatically. Our investigation revealed a similar motivation in that web application developers introduced clones by inlining JavaScript for faster performance.

To understand the prevalence of code inlining in web applications, we manually inspected a maximum of 10 top large (in terms of LOC) *inter-file clones* in each `JSweb` subject. We compared the number of exact clones and near-miss clones, clones that differ from each other in small details only (Cordy et al. 2004). We found that 87 % of the inter-file clones are exact clones while 13 % of them are near-miss clones. Given the large proportion of exact clones, developers could have stored them externally to reduce the code size, but they decided not to do so. Our findings agreed with the suggestion that code inlining is sometimes necessary for faster performance in web applications. Hence, we observed a large proportion of exact clones. For the remaining 13% of near-miss clones, we observed that those clones differ from others only in some attributes such as having a unique identifier.

Our manual inspection reveals the trade-off between inlining JavaScript and using external JavaScript. Although refactoring code clones as a set of library functions helps reduce maintenance efforts (Kapser and Godfrey 2003), the performance requirement of websites leads to a different design choice. Therefore, we observed a large portion of clones across different files.

5.1.2 Sizes of Code Clones

The sizes of code clone are usually measured in terms of lines of cloned code to understand the scale of code fragments being cloned in a system (Kapser and Godfrey 2003). In our experiment, the overall sizes of clones are similar for all three groups; they do not depend much on the application domains or programming languages.

Figure 4 shows the average lines of cloned code for different combinations of similarity and minimum number of tokens. For all three categories of subjects, the average number of lines changes slightly when the similarity varies and increases when the minimum number of tokens increases. Overall, the differences are minor when either the similarity or minimum number of tokens changes.

The average size of clones for most configurations falls between 5 to 10 lines, and all three different project groups, `JSweb`, `JSproj`, and `Java`, have similar values. This result is consistent with the findings in the literature. For instance, Kapser and Godfrey (2003) studied clones of the Linux system and observed that the average size of clones is approximately 12 to 14 lines.

We also measured the maximum number of lines of cloned code, and we found that the number does not change with similarity or the minimum number of tokens.

[6]https://developers.google.com/speed/pagespeed/module/filter-js-inline

[7]https://developer.yahoo.com/performance/rules.html#external

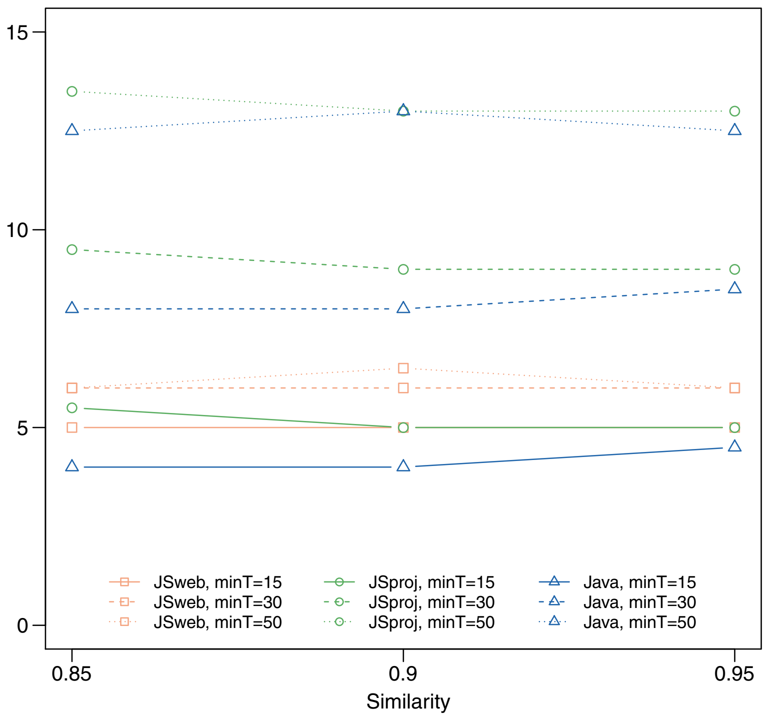

Fig. 4 Average lines of cloned code. They only vary slightly in different configurations for all three subject groups

5.1.3 Clone Coverage

Clone coverage is the ratio of the lines of cloned code to the total lines of code (Kim et al. 2005). It represents the ratio of the system affected by code duplication and, thus, estimates the probability that a change to an arbitrary program statement will require multiple similar changes (Juergens et al. 2009a).

We measured the clone coverage of different systems as shown in Fig. 5. The clone coverage of most web pages remains high at approximately 38–43 % when the similarity increases, while clone coverages in both `JSproj` and `Java` decrease from approximately 15–35 % to 5–18 %. The coverages for all three groups differ only by approximately 5–10 % for different minimum numbers of tokens of the same similarity.

The above findings suggest that the clones in `JSweb` are more similar to each other than the clones in `JSproj` and `Java`. We detect clones with at least a certain similarity, but the clone coverages of `JSweb` vary only slightly when the similarity increases. The different development nature of websites and standalone projects can explain such an observation. Because websites usually consist of pages with the same or similar layout and functionality but different contents, code clones with slight variations may occur frequently. However, because standalone projects are usually developed separately or as libraries of other applications, components with the same or similar functionalities are less likely to occur in standalone projects than in websites.

The high clone coverage in JSweb suggests that there is a higher probability that a change in one place may require the same or similar changes in multiple places (Juergens et al. 2009a). However, the slight differences of clone coverages between similarities in JSweb also suggest that code clones in JSweb share larger portions of identical fragments than in JSproj and Java. Developers can consider relatively fewer differences between code clones on any reengineering activities such as source code transformation or simple code restructuring (Balazinska et al. 1999); hence, automating reengineering activities in multiple places in JSweb requires less effort than in JSproj and Java.

5.1.4 Files Associated with Clones

We measured the percentages of files associated with clones across different systems, as shown in Fig. 6. This metric indicates whether clones are from some specific files or scattered among many files all over the system (Roy and Cordy 2010a).

Figure 6 presents the percentages of files associated with clones for different similarities and minimum numbers of tokens. In JSweb, files associated with clones remain high at approximately 91–97 % for different similarities. Both JSproj and Java show similar percentages of files associated with clones of approximately 33–87 %. For different minimum numbers of tokens of the same similarity, there are only slight differences in JSweb while there are 10–20 % differences in both JSproj and Java. This suggests that the code

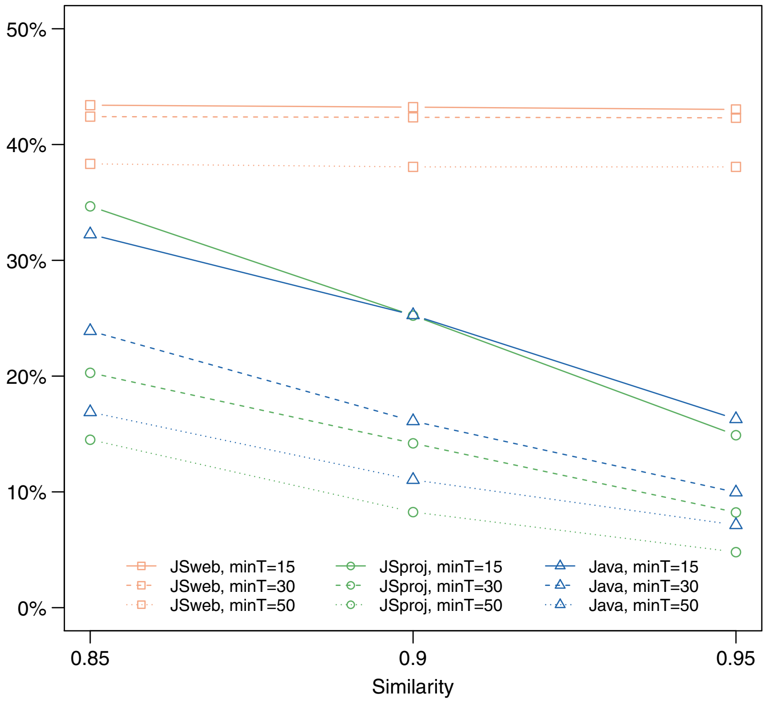

Fig. 5 Clone coverage. JSweb remains high at approximately 38 − 43 % while JSproj and Java decrease with increasing similarity. The slight difference between similarities in JSweb suggests less effort in identifying the differences between clones because they are more similar to each other

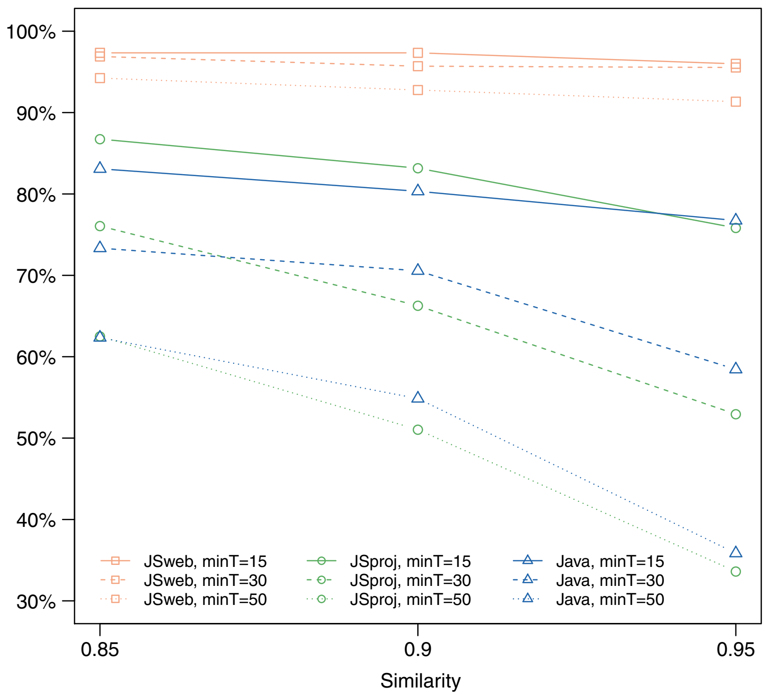

Fig. 6 Files associated with clones. `JSweb` remains high at approximately 91–97 % while `JSproj` and `Java` decrease with increasing similarity. The use of small gadgets and code generation in web applications lead to similar code fragments in multiple files

clones in `JSweb` are of at least a certain size, and hence, changing the minimum number of tokens only has a small effect on the files associated with clones.

Roy and Cordy (2010a) found that the percentages of files associated with clones in Java systems are approximately 45–75 %. Our findings on Java systems are consistent with theirs, as ours are approximately 36–83 %. They also found that Python systems have 25–60 % of files associated with clones. Although Python and JavaScript are both scripting languages, our findings indicate that the percentages of files associated with clones in `JSproj` are more similar to Python than to `JSweb`. This shows that different development nature of JavaScript can affect the percentages of files associated with clones. We manually investigated the clones of `JSweb` and discuss the reasons behind such an observation.

One characteristic of web applications is the use of small gadgets to connect to different services such as website traffic analysis and social networks. Our manual investigations revealed that such usage is one of the cloning targets across many files. Figure 7 shows an example of scattered clones in `JSweb` from `linkedin.com`. This is a sequence of code for web tracking from Google Analytics,[8] a service to measure how users interact with the website. We found the same piece of code in the code clones of other studied websites. This example shows that developers insert this piece of code into the pages they desire to track, and hence, many files include it.

[8]https://developers.google.com/analytics/devguides/collection/gajs/

```
(function() {
  var ga = document.createElement('script');
  ga.type = 'text/javascript';
  ga.async = true;
  ga.src = ('https:' == document.location.protocol ? 'https://ssl'
   : 'http://www') + '.google-analytics.com/ga.js';
  var s = document.getElementsByTagName('script')[0];
  s.parentNode.insertBefore(ga, s);
})();
```

Fig. 7 Google Analytics example from `linkedin.com`. It inserts an external JavaScript file to track user interactions on that page

Another characteristic of web applications is the use of code generation to enable rapid development. Code generation allows developers to quickly create web pages with similar layouts but different contents. We observed in our inspections that code generation is another target for cloning across many files. Figure 8 shows another example of scattered clones from `yahoo.com`. This is a piece of generated code to enable the same functionality for different DOM elements. To the best of our knowledge, there is no formal definition of generated code in web pages and techniques to detect them automatically. Therefore, we identify a piece of code as highly likely to be generated when it differs from other cloned fragments only in string or number literals and the literals contain words not in a dictionary after decomposition. For example, the code in Fig. 8 differs from other cloned fragments only in the `LDRB` part of the literals, `"yom-ad-LDRB"`, `"yom-ad-LDRM-iframe"`, and `"loc=LDRB noad"`, but `LDRB` is not a word in a dictionary that may be an abbreviation of a long phrase. This example indicates that the use of code generation enables the development of a web page rapidly but also causes scattered clones among different files.

One common characteristic between the above two examples is that they are sequences of code that have to be used in a certain order to provide the desired functionalities. Kapser and Godfrey (2008) described such cloning patterns as API/Library Protocols. Such uses of particular APIs often require developers to intentionally duplicate these sequences so that they can parameterize the sequences for particular problems. It reduces development

```
(function ()
{
  var wrap = document.getElementById("yom-ad-LDRB");
  if(null == wrap)
  {
    wrap = document.getElementById("yom-ad-LDRB-iframe") || {

    };
  }
  var content = wrap.innerHTML || "";
  if(content && content.substr(0, content.lastIndexOf("<script>"))
  .indexOf("loc=LDRB noad") !== - 1)
  {
    wrap.style.display = "none";
  }
})();
```

Fig. 8 Generated code example from `yahoo.com`. It differs from other code only in literals. The literals include `LDRB` that is not in a dictionary, which indicates code generation

Table 10 Distributions of files associated with clones

Code sequences	Minimized code	Function-level clones	Attributes manipulation
33 %	29 %	27 %	11 %

time because it allows developers to quickly duplicate and modify the code for their needs (Kapser and Godfrey 2008). This explains the finding that changing the minimum number of tokens in `JSweb` has only few effects on the files associated with clones. Applications developed with Java libraries also have similar usages. For example, creating a user interface with `Java Swing` requires creating a `JFrame` first before adding other components such as menus and buttons. Our findings suggest that such usages are more prevalent in `JSweb` than `JSproj` and `Java`. This shows that the different development natures of programming languages affect the percentages of files associated with clones in a system. Roy and Cordy (2010a) argued that a system with a smaller number of files associated with clones is easier to maintain since the clones are localized to certain specific files. However, even when the clones are scattered among many files all over the system, if they are of certain known sequences of code such as those found in `JSweb`, developers can benefit from identifying reusable candidates from frequently used sequences because developers are aware of those sequences.

To understand the prevalence of different motivations for scattered cloning in web applications, we inspected a maximum of 10 top large (in terms of LOC) *clones scattered among*

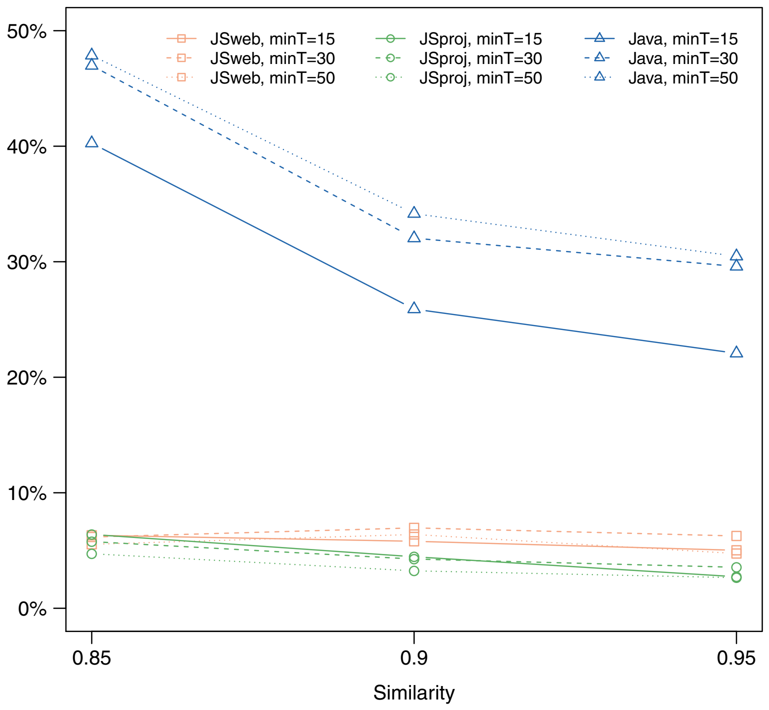

Fig. 9 Percentages of function-level clones. `Java` has relatively larger percentages of function-level clones than `JSproj` and `Java` due to the overloading mechanism in Java

more than 3 files in each JSweb subject. Table 10 shows the summary of distributions. Our inspection revealed that the use of code sequences is the most prevalent in scattered clones, which accounts for 33 %. This is then followed by minimized code and function-level clones, which account for 29 % and 27 % respectively. The least prevalent usage is manipulation of attributes, which accounts for 11 %. Such clones mainly contain codes that assign values to properties of objects. We found that such clones are common in shopping web applications such as amazon.com and ebay.com because they usually have many attributes for each shopping item to manipulate. The findings of our inspection suggest that developers can benefit from identifying reusable candidates from one-third of the duplicated code sequences.

5.1.5 Function-Level Clones

Previous literature has studied function-level clones widely. Mayrand et al. (1996) proposed a list of metrics for detecting function-level clones in software systems, and Lague et al. (1997) reported that a high number of function-level clones in a software system could increase the maintenance cost.

Figure 9 shows the percentages of function-level clones for different similarities and minimum number of tokens. For both JSweb and JSproj, the percentages of function-level clones remain low at approximately 3–7 % for different similarities and minimum number of tokens. However, the percentages in Java decrease from approximately 40–48 % to 22–30 % when the similarity increases. Even so, the percentages in Java are still higher than those in JSweb and JSproj. Our findings suggest that the cloning of functions in Java occurs more often than in JSweb and JSproj.

Roy and Cordy (2010a) found that Python and Java systems have similar percentages of function-level clones of approximately 5–30 %. In Java systems, we observed 20 % more function-level clones than their findings in all configurations. This can be explained by the differences between the selection of subjects, clone detection technologies, and clone detection configurations. While they found that Python has similar percentages of function-level clones as Java, we found that the percentages of function-level clones in both JSweb and JSproj are much lower than Java. This suggests that, although both JavaScript and Python are scripting languages, they may not show similar percentages of function-level clones.

One difference between statically-typed and dynamically-typed languages is that the former determines the types of variables at compile time while the latter determines the types at runtime. Our manual inspections revealed that such difference leads to different cloning patterns of function-level clones between the two kinds of languages. Figure 10 shows an example of function-level clones from JHotDraw. We found 12 similar fragments in both the same file and other files with different parameter types. Kapser and Godfrey (2008)

```
public void addAttribute(String name,float value,float defaultValue)
{
   if (value != defaultValue) {
    addAttribute(name, value);
   }
}
```

Fig. 10 Function-level clones example from JHotDraw. A usage of code clone to provide function overloading

described such a cloning pattern as Parameterized Code, and they explained that developers modify a solution of a common problem to create a new solution by changing only a few identifiers or literals in the code to improve comprehensibility. This example indicates that, since Java is a statically-typed language, duplicating and customizing the same function makes the code more understandable than abstracting it by a template. In particular, the overloading mechanism in Java provides a set of similar method declarations with different parameter types, which are subject to function-level clones. However, because JavaScript is a dynamically-typed language with no overloading, function declarations and function expressions do not specify parameter types, and hence, they are less likely to be subject to function-level clones. Another cause of the high percentages of function-level clones in Java systems is the use of a large number of accessor and utility methods in Java (Roy and Cordy 2010b).

To understand the prevalence of different motivations for cloning in the function-level clones of Java systems, we inspected a maximum of 10 top large (in terms of LOC) *function-level clones* in each Java subject. Among the clones we inspected, 28 % of them duplicated the exact functionality. An example of such a clone is `getLibraryPath()` in both `gcc/GccLinker.java` and `gcc/cross/GccLinker.java` of the subject cpptasks. The developers needed the same functionality in different platforms; hence, they performed such cloning. The remaining 72 % are clones of similar functionality. One such example is the code in Fig. 10.

The experimental results indicate that function-level clones in `Java` are more commonly found than those in `JSweb` and `JSproj`. Maintaining multiple similar function declarations in a system increases the total code size, and it requires extra work to keep the multiple declarations in sync in case of any changes. Therefore, the cost of maintenance would also be relatively higher (Lague et al. 1997). However, researchers found that function-level clones are often easily eliminated by refactoring (Calefato et al. 2004).

5.2 RQ2: Software Metrics

In this section, we measure how the clone properties relate to software metrics. We surveyed the software metrics used in the metrics-based clone detectors from the literature (Abd-El-Hafiz 2012; Antoniol et al. 2001, 2002; Kamei et al. 2011; Kontogiannis 1997; Kontogiannis et al. 1996; Mayrand et al. 1996; Merlo et al. 2002, 2004; Patenaude et al. 1999; Shawky and Ali 2010) and found the most commonly used ones. Among the software metrics developed in the software engineering community, the Chidamber and Kemerer metrics suite (Chidamber and Kemerer 1994) and MOOD metrics suite (Brito e Abreu 1995) are commonly used. However, many of these metrics were designed for static object-oriented languages. They are not always applicable to dynamically-typed languages, such as JavaScript, because static type features, such as classes, may be missing in dynamically-typed languages.

We collected 48 software metrics from the 11 studies mentioned above and sorted them by the frequencies of their appearance in those 11 studies. We identified 7 metrics that are measurable in both JavaScript and Java and are used in at least 6 studies. They are number of files, lines of code, number of statements, number of functions, cyclomatic complexity per function, cyclomatic complexity per file, number of parameters, number of variables, and number of function calls.

For each pair of clone properties from Section 4 and each software metric, we computed their Pearson product-moment correlation coefficient. Tables 11, 12, and 13 show the correlation for `JSweb`, `JSproj`, and `Java`, respectively. Among the software metrics, **LOC**

Table 11 Correlation of clone properties and software metrics in `JSweb`

	clones	intra-file clones	inter-file clones	avg lines of code clone	max lines of code clone	clone coverage	files associated with clones	func-level clones
files	0.41	0.33	0.42	0.51	0.44	0.61	**0.93**	0.14
LOC	0.51	0.41	0.53	**0.64**	0.41	**1.00**	0.56	0.17
statements	0.27	0.03	0.31	−0.11	0.24	0.27	0.03	−0.09
functions	0.27	0.02	0.30	−0.12	0.23	0.25	0.01	−0.09
cpl/func	0.10	0.18	0.09	0.06	0.08	−0.19	−0.27	0.25
cpl/file	0.22	−0.02	0.25	−0.21	0.19	0.15	−0.08	−0.12
parameters	0.50	0.31	0.53	0.01	0.51	0.07	−0.04	0.26
variables	**0.71**	0.55	**0.73**	0.23	**0.71**	0.18	0.09	0.49
func calls	**0.76**	0.62	**0.78**	0.30	**0.77**	0.21	0.13	0.55

9 pairs (13 %) indicate strong correlation

denotes lines of code, **cpl/func** denotes cyclomatic complexity per function, and **cpl/file** denotes cyclomatic complexity per file. We highlight the entries that have *p-value* smaller than 0.05, which indicate that they are statistically significant and have a strong correlation in the pair.

Table 11 shows the correlation between the pairs of clone properties and software metrics. Among 72 pairs of correlations, 9 of them (13 %) have a strong correlation. The remaining pairs of clone properties and software metrics do not reveal any strong correlation.

Table 12 Correlation of clone properties and software metrics in `JSproj`

	clones	intra-file clones	inter-file clones	avg lines of code clone	max lines of code clone	clone coverage	files associated with clones	func-level clones
files	**0.92**	**0.92**	**0.89**	0.16	0.53	**0.81**	**0.97**	**0.73**
LOC	**0.92**	**0.83**	**0.91**	0.08	0.53	**0.80**	**0.94**	**0.79**
statements	**0.97**	**0.90**	**0.95**	0.02	**0.61**	**0.84**	**0.94**	**0.82**
functions	**0.99**	**0.89**	**0.98**	0.09	**0.71**	**0.90**	**0.92**	**0.88**
cpl/func	0.00	−0.06	0.01	−0.01	0.09	0.02	−0.09	0.06
cpl/file	0.20	0.05	0.23	−0.24	−0.28	0.17	−0.01	0.27
parameters	**0.98**	**0.78**	**0.99**	0.14	**0.78**	**0.94**	**0.86**	**0.94**
variables	**0.99**	**0.91**	**0.97**	0.08	**0.75**	**0.91**	**0.88**	**0.87**
func calls	**0.98**	**0.95**	**0.94**	0.02	**0.63**	**0.84**	**0.93**	**0.79**

47 pairs (65 %) indicate strong correlation

Table 13 Correlation of clone properties and software metrics in `Java`

	clones	intra-file clones	inter-file clones	avg lines of code clone	max lines of code clone	clone cov-erage	files asso-ciated with clones	func-level clones
files	**0.77**	**0.73**	**0.77**	−0.08	0.45	**0.81**	**0.99**	**0.68**
LOC	**0.88**	**0.81**	**0.89**	−0.18	0.36	**0.86**	**0.97**	**0.84**
statements	**0.93**	**0.86**	**0.93**	−0.26	0.30	**0.90**	**0.96**	**0.89**
functions	**0.97**	**0.90**	**0.98**	−0.34	0.25	**0.90**	**0.84**	**0.98**
cpl/func	0.28	−0.29	0.28	0.02	0.01	0.38	0.39	0.16
cpl/file	0.32	0.46	0.28	−0.06	−0.02	0.47	0.21	0.17
parameters	**0.66**	0.59	**0.67**	0.04	0.48	**0.68**	**0.95**	0.60
variables	**0.89**	**0.80**	**0.90**	−0.22	0.30	**0.85**	**0.96**	**0.86**
func calls	**0.81**	**0.72**	**0.83**	−0.12	0.38	**0.78**	**0.96**	**0.78**

40 pairs (56 %) indicate strong correlation

Table 12 shows the correlation in `JSproj`. There are 47 pairs (65 %) having a strong correlation. It indicates a stronger correlation between the clone properties and software metrics than in `JSweb`. Except the average and maximum lines of code in clone properties and complexity per function and per file in software metrics, other pairs exhibit strong correlations.

Table 13 presents the correlation in `Java`. There are 40 pairs (56 %) having a strong correlation. It shows a similar pattern of correlations to `JSproj`. Similar to `JSproj`, except for the average and maximum lines of code in clone properties and complexity per function and per file in software metrics, other pairs exhibit strong correlations.

Our findings above indicate that `JSweb` has different cloning patterns compared to `JSproj` and `Java`. While we found that most clone properties in `JSproj` and `Java` have strong correlations with software metrics, only a few clone properties in `JSweb` strongly correlate to software metrics.

5.3 RQ3: Cloning Patterns

5.3.1 Common Cloning Patterns

Our quantitative findings indicate that `JSweb` has different cloning patterns to those of `JSproj` and `Java`. In this section, we qualitatively analyze what kinds of cloning patterns exist in `JSweb`, `JSproj`, and `Java`. A cloning pattern is a cloning strategy that repeatedly appears in the code clones.

In classifying cloning patterns, code clones are classified by either their physical structures (Kapser and Godfrey 2003) or motivations for cloning (Kapser and Godfrey 2008). We use the cloning patterns by Kapser and Godfrey (2008) in our qualitative analysis by manually inspecting a maximum of 50 top large clones (in terms of LOC) in each subject and see whether certain cloning patterns exist in a category of subjects or not.

The purpose of finding cloning patterns is not to measure the usage frequencies between different groups of subjects, but to verify whether certain cloning patterns were used in any subjects. Therefore, we measured only the number of subjects having such cloning patterns. Throughout the inspection, we expected to identify cloned code fragments specific to one of the subject groups. We inspected a maximum of 50 clones in each subject to have enough samples to identify not statistical, but motivational differences among the groups.

Figure 11 shows common cloning patterns in `JSweb`, `JSproj`, and `Java`. For each category of subjects, we count how many subjects were using certain cloning patterns. Among 11 cloning patterns, General Language or Algorithmic Idioms, Boiler-plating Due to Language Inexpressiveness, and Parameterized Code were the most commonly used cloning patterns, which were used in 7 to 10 subjects. While more than 9 subjects used the pattern Replicate and Specialize in `JSproj` and `Java`, only 2 subjects used it in `JSweb`. One possible reason why `JSweb` seldom customizes code fragments is because web applications often provide similar functionalities with different contents. Another commonly used cloning pattern by all three subjects is Verbatim Snippets, which is unavoidable in software development. For Cross-cutting Concerns and API/Library Protocols, `Java` shows a slightly higher usage in the subjects than `JSweb` and `JSproj`.

While we were manually investigating the clones, we observed that developers use different cloning practices in different types of systems. Some cloning practices found in `JSweb`

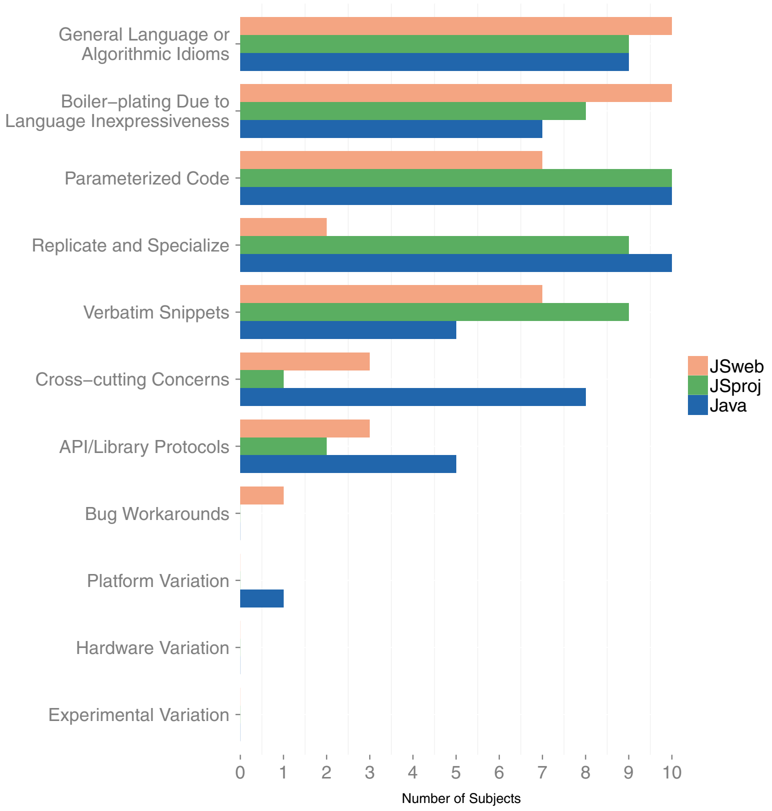

Fig. 11 Cloning patterns. General Language or Algorithmic Idioms, Boiler-plating Due to Language Inexpressiveness, and Parameterized Code are the most commonly used cloning patterns in the three subject groups

```
if (f) {
 var agt = navigator.userAgent.toLowerCase();
 var is_ie = (agt.indexOf("msie") != -1);
 if (f.Email && (f.Email.value == null || f.Email.value == '' ||
    is_ie)
     && (f.Email.type != 'hidden') && f.Email.focus) {
   f.Email.focus();
   if (f.Email.value) {

     f.Email.value = f.Email.value;
   }
 } else if (f.Passwd) {
   f.Passwd.focus();
 }
}
```

Fig. 12 Browser-dependent or device-specific code in `google.com`. It executes a customized code for Internet Explorer to ensure consistent behaviors and experiences across different platforms

do not exist in `JSproj` or `Java`. We present several case studies of the development practices found in the code clones of `JSweb` and discuss their motivations, benefits, drawbacks, and how the features of web application technologies lead to such code clones.

5.3.2 Case Study 1:`google.com`

The first development practice we discuss is the use of browser-dependent or device-specific code in `google.com`. This is a common practice in web applications to treat certain browsers or devices specially due to compatibility issues. One reason for code cloning is to reuse the functionality and logic in similar systems (Roy and Cordy 2007). After duplicating some code, developers may customize the code to satisfy the needs of different systems.

Figure 12 shows an example of such a practice, which checks the running browser and executes a customized code for Internet Explorer. Using browser-dependent or device-specific code allows web applications to deliver consistent behaviors and experiences across different platforms. However, it increases testing efforts due to the large variety of browsers and devices.

This case study shows a feature of web application technologies used to implement different compatibility requirements between web applications and standalone projects, which introduces code clones specific to web applications. One limitation of web applications is that only a single version of a web application should support different browsers and platforms of various users, and hence, one version should include customized code. On the contrary, developers of standalone projects often maintain different branches for different platforms so that they can test individual platforms with less testing effort than would be needed for web applications.

5.3.3 Case Study 2:`yahoo.com`

Another development practice we present is the manipulation of DOM or CSS in `yahoo.com`. This is a common practice in web applications to directly manipulate DOM or to change CSS styles of elements without reloading web applications. One reason for code

```
initImages : function(id) {
    var el = Y.one('#'+id);
    if (!el) return;
    var images = Y.all('#'+id+' img[realimg]');
    Y.NodeList.each(images, function(node) {
        var realimg = node.getAttribute('realimg');
        if (realimg && node.src != realimg ) {
           node.src = realimg;
        }
    });
}
```

Fig. 13 DOM or CSS manipulation in `yahoo.com`. It allows web applications to display new elements faster without reloading the applications

cloning is to have better performance in real-time programs (Roy and Cordy 2007). However, we noticed that different ways of manipulating DOM or CSS have different effects on the performance of web applications.

Figure 13 shows such an example, which iterates each of the nodes in a list and assigns the source of the node to the node with the attribute `realimg`. Such a practice allows web applications to display new elements faster without reloading the applications. However, a guideline from Google Developers[9] suggests that inappropriate manipulation of DOM or CSS will trigger multiple reflows of browsers, and hence, increase the time to render the new contents. Reflow is the process for re-calculating the positions and geometries of elements in HTML documents to re-render some part or all of the documents.

This case study shows that a unique feature of web applications that manipulates DOM or CSS leads to a specific practice. While standalone projects are usually downloaded entirely before execution, web pages are incrementally loaded and even reloaded with user interactions, which make performance enhancement more important. Because inappropriate manipulation of DOM or CSS will trigger multiple browser reflows, web application developers should manipulate DOM or CSS carefully.

5.3.4 Case Study 3:`twitter.com`

The last development practice we discuss is the use of minimized code in `twitter.com`. This is a common practice in web applications to reduce the amount of data being transferred to browsers. Another reason for code clones is system development with generative programming approach (Roy and Cordy 2007). Developers generate minimized code from original code by removing unnecessary characters including white space characters, new line characters, comments, and block delimiters. The minimized code preserves the semantics of the original code, and it forms a clone pair with the original one.

Figure 14 shows an example of such a practice. Using minimized code reduces the time for browsers to load web applications. However, it increases the effort needed for maintenance since any future change should apply to both the original code and the minimized code. Although web applications rarely use the original code and the minimized code at the same time and they generate minimized code automatically, since minimized code forms a clone pair with its original code, developers should keep both codes in sync. If developers forget to generate minimized code after changing its original code, it may cause inconsistent

[9]https://developers.google.com/speed/articles/javascript-dom

```
if (current_domain == 'dev.twitter.com' && !current_url.match(/\/(
    apps|admin)/)) {_gaq.push(['_trackPageLoadTime']); }_gaq.push(["
    _trackPageview"]);(function() {varga = document.createElement("
    script");ga.type = "text/javascript";ga.async = true;ga.src = ("
    https:" == document.location.protocol ? "https://ssl" : "http://
    www") + ".google-analytics.com/ga.js";var s = document.
    getElementsByTagName("script")[0];s.parentNode.insertBefore(ga,
    s);})();
```

Fig. 14 Minimized code in `twitter.com`. The elimination of unnecessary characters including whitespace characters, newline characters, comments, and block delimiters reduces the time for browsers to load web applications

behaviors between web applications using the original code and those using the minimized code.

This case study shows a feature of web application technologies used to implement a different performance requirement of web applications from standalone projects, which introduces code clones that exist in only web applications. Because users of web applications, especially on mobile devices, require high performance, developers often use minimized code to reduce the time for downloading web applications. On the contrary, standalone projects are usually larger, sometimes in MB or GB, and their users do not demand fast downloading.

6 Discussion

This section discusses the findings and implications of our empirical study, lessons learned, and threats to the validity of our work.

6.1 Findings and Implications

- **A feature of web application technologies used to implement the performance requirement of `JSweb` leads to 95 % of inter-file clones in `JSweb`**

  We observed that `JSweb` contains 95 % of clones across different files. Our manual inspection of inter-files clones of `JSweb` revealed that caching external JavaScript is more beneficial than inlining JavaScript only when the page views per session are sufficiently numerous. Roy and Cordy (2007) suggested that developers may introduce clones by code inlining due to the high cost of function calls in real time. Our inspection showed similar motivation in that web application developers introduced clones by inlining JavaScript for faster performance. We found that 87 % of the inlined codes are exact clones. The performance trade-off between inlined JavaScript and caching external JavaScript leads to such a significant amount of inter-file clones. However, we did not consider external JavaScript caching in our study. A possible extension of the experiments would be comparing the proportion of inlined and external JavaScript and identifying other development requirements of websites that lead to the choice between inlined and external JavaScript.
- **Slight variations of clone coverage in `JSweb` for different similarities suggest that automating reengineering activities in multiple places in `JSweb` requires less effort than `JSproj` and `Java`**

While we found that clone coverages of `JSproj` and `Java` decrease with higher similarities, clone coverages of `JSweb` remain at approximately 38–43 % and are higher than those of `JSproj` and `Java`. Although the literature suggested that higher clone coverage has a higher probability that a change in one place may require the same or similar changes in multiple places (Juergens et al. 2009a), the slight differences of clone coverages between similarities in `JSweb` also suggest that automating reengineering activities in multiple places in `JSweb` requires less effort. Because such a finding indicates that code clones in `JSweb` share larger portion of identical fragments, developers can consider a relatively smaller number of differences between code clones in any reengineering activities (Balazinska et al. 1999).

- **The use of sequences of code in a certain order in `JSweb` suggests reusable candidates**
  We observed that `JSweb` has approximately 91–97 % of files associated with clones, 25–60 % higher than those in `JSproj` and `Java`. Our inspection revealed that 33 % of the inspected clones are sequences of code in a certain order. Such use of particular sequences to achieve desired functionalities often requires developers to duplicate them intentionally so that they can parameterize them for particular problems (Kapser and Godfrey 2008). Even though the literature suggested that scattered clones are harder to maintain (Roy and Cordy 2010a), if they are of certain known sequences of code, developers can benefit from identifying reusable candidates because they created the clones intentionally and are aware of the sequences.
- **`Java` has more function-level clones and hence requires more maintenance efforts**
  We found that `Java` has 10–30 % more function-level clones than `JSweb` and `JSproj`. This is mainly because of the overloading mechanism in Java. Statically-typed languages, such as Java, require the type of variable to be determined at compile time, and hence, reusing the same functionality for different types leads to code clones. Maintaining multiple similar function declarations in a system increases the total code size, and it requires extra work to keep the multiple declarations in sync in case of any changes. This increases the effort needed to maintain function-level clones in `Java`.
- **More clone properties in `JSproj` and `Java` have strong correlation with software metrics than in `JSweb`**
  We noticed that the 56–65 % of pairs of clone properties and software metrics in `JSproj` and `Java` have a strong correlation while only 13 % of pairs in `JSweb` have a strong correlation. This implies that it is harder to estimate the costs and benefits of clone removal in web applications than in standalone projects because fewer clone properties have a strong correlation with software metrics in web applications.
- **`JSweb` has unique development practices**
  In our manual inspection of clones, we found unique development practices of `JSweb` due to different development requirements in web applications from standalone projects. This implies that it may not be easy to adapt existing clone management techniques in standalone projects to web applications because some clones such as those that involve DOM or CSS manipulation, are specific to web applications, and tool developers have to consider those practices when they design tools to manage the code clones in web applications.
- **Unlike what the literature suggests, the clones in `JSweb` may not be that risky to system management**

The literature suggests that systems having a high percentage in certain clone properties such as inter-file clones and files associated with clones may lead to more labor for error fixing (Kapser and Godfrey 2003) and increase the difficulty in maintenance (Roy and Cordy 2010a) because developers may not be aware of the existing clones in the system and manually tracking clones across multiple files requires much effort. Our manual investigation regarding the clone properties and cloning practices revealed that this might not be the case for `JSweb`. The features of JavaScript web application technologies lead to different ways of cloning from those of standalone projects. Developers of `JSweb` were aware of those features and, hence, created the clones intentionally. Even though the developers or maintainers are not those who created the clones anymore, a recent study on the industrial development process showed that a clone change notification system is able to identify both changed code clones that industrial developers are not aware of and stable clone sets in the system (Yamanaka et al. 2013). Therefore, unlike what previous studies suggest, the code clones found in `JSweb` may not be that risky to system management. Understanding the risks of cloning to system management in `JSweb` requires further specific studies.

6.2 Lessons Learned

This section discusses the lessons learned from the findings of this study

- **Language features play an important role in determining the ways developers duplicate code**
  Previous work has mostly studied code clones in statically-typed languages. However, our manual inspections revealed that the differences of language features between statically-typed and dynamically-typed languages lead to more function-level clones in Java than JavaScript in implementing the same functionality. This suggests that when comparing code clones of different languages, we have to take into account of their individual features, especially when the languages do not have many features in common.
- **Clone properties of the same language in different application domains can be different**

  There are previous studies on standalone projects and also studies on web applications. However, there is a lack of work that examines whether clone properties of the same language in different application domains are the same. Our findings on several clone properties, such as inter-file clones and clone coverage, showed that features of programming languages and technologies affect the ways developers clone and, hence, the clone properties of a system. In order to understand the motivations for cloning, talking to developers is the most direct way. However, studying code clones of the same language in different application domains also provides another means of acquiring such knowledge.
- **Code clones are not always unintentional**
  Previous studies suggested that the motivations for cloning are twofold (Roy and Cordy 2007). On one hand, developers duplicate code due to unawareness of existing code in a system or lack of understanding on the cloned code. However, there are also circumstances in which developers intentionally introduced clones due to development strategy and maintenance benefits. Our findings suggest that the latter case is more

prevalent in JSweb than in JSproj and Java. In JSweb, we observed 5–55 % more inter-file clones than JSproj and Java because web application developers created clones intentionally as one of the ways to enhance the loading speed of their sites. In addition, JSweb has 8–64 % more files associated with clones than JSproj and Java because duplicating and modifying sequences of code enable web application developers to deploy their applications in a short period of time.

- **Code clones may not always be risky**

  Some studies in the literature suggest that code clones are risky to system management and should be eliminated (Fowler and Beck 1999; Jiang et al. 2007b; Juergens et al. 2009b). However, some researchers found that code clones have positive impacts on productivity and may not be as bad as others had claimed (Kapser and Godfrey 2008; Kim et al. 2005; Thummalapenta et al. 2010). A recent study on the relationship between cloning and defect proneness found little evidence to support the conventional wisdom that clones that span across multiple files or directories are more defect prone (Rahman et al. 2012). Even though we found that JSweb contains high percentages of clones across different files, it may not be as risky as some studies have suggested. Understanding the risks of cloning in JSweb requires further studies, as developers on one hand may introduce clones with good intentions such as technology limitations and external business forces, but on the other hand cloning may be due to bad intentions such as programmers' laziness and programmers' memories (Kapser and Godfrey 2008).

- **Statistics alone do not give enough information about the clone properties of a system**

  Previous studies have mostly focused on quantitative differences of code clones between systems. However, looking at the numbers alone cannot provide sufficient insights on the factors that lead to such results. We observed that the percentages of inter-file clones and files associated with clones in JSweb vary only slightly for different clone detection configurations. Without further investigation, it is difficult to understand how such a phenomenon occurs. Therefore, qualitative analysis is essential in understanding the motivations for cloning.

6.3 Threats to Validity

We find the following threats to the validity of our experiments:

6.3.1 Internal Validity

- **Representativeness of open-source projects and web pages**

  In our experiments, we used open-source projects and the top web pages from Alexa for clone detection. Open-source projects may not be adequately representative of all kinds of projects since closed-source projects may have different clone properties due to different software development practices in the industry. However, we chose to study those projects because they were also widely studied in the literature (Aversano et al. 2007; Bellon et al. 2007; Mondal et al. 2012; Richards et al. 2010; Roy and Cordy 2010a). In addition, due to the large-scale and dynamic features of web applications, the web pages we collected are only part of the websites and the code quality may differ from site to site. Therefore, they may also not be sufficiently representative of other websites. However, we chose the web pages from Alexa because they were also widely used in other JavaScript related studies (Chugh et al. 2009; Finifter et al. 2010; Martinsen et al. 2011; Nikiforakis et al. 2012; Ocariza et al. 2011).

- **Representativeness of embedded JavaScript in web applications**
  In our experiments, we studied only embedded JavaScript in web applications. However, we observed that a feature used to implement the performance requirement of web pages leads to the choice between inlining and caching external JavaScript and, hence, affects the percentages of intra-file and inter-file clones. A possible extension of the experiments would be to compare the proportion of inlined and external JavaScript in web applications.
- **Representativeness of the clone detection technique**
  Due to the lack of reasonable clone detectors for JavaScript, we built our clone detector JScd and compared its recall and precision with those of two other clone detectors. In our experiments, we detected code clones with JScd and Deckard, which use only one kind of clone detection technique. However, our results may not be sufficiently representative of the clones detected by other clone detection techniques.
- **Representativeness of the clone detectors evaluation technique**
  We evaluated JScd by the mutation approach of Roy and Cordy (2009). We selected this technique because it is widely used (Roy and Cordy 2010b; Sahaet al. 2011; Stephan et al. 2013, 2014; Zibran and Roy 2012) and automatic, which reduces the amount of human effort in clone verification. However, their technique evaluates only the randomly injected clones and it may not cover all the clones in a system. Therefore, it may not be sufficiently representative of other clone detectors evaluation techniques. To reduce the bias of clone detectors evaluation results toward the use of mutation, we also evaluated the tools on the actual clones of a subject.
- **Representativeness of clone metrics and software metrics**
  Due to the different nature of statically-typed and dynamically-typed languages, we selected clone metrics and software metrics that are applicable to both types of languages. However, they may not be sufficiently representative of other properties of individual languages.

6.3.2 *External Validity*

- **Code generation in web development**
  Code generation is a well-known practice to enable rapid web development. In our study, we observed that code generation is one of the causes for the high percentages of files associated with clones in JSweb. However, we are not able to automatically distinguish them from manually duplicated code due to the lack of formal definitions and detection techniques. Our judgement of whether a code fragment is generated or not may be biased.
- **Potentially influential factors on the results**
  In our experiments, we looked at code clones of different languages and application domains. Our manual inspection revealed that there are different factors affecting how developers created clones. For instance, performance is essential in web applications, and we observed different practices in the web application clones to improve performance such as code inlining and code minimization. Code inlining leads to more clones

across different files because external JavaScript cannot benefit from browser caching when both the script and the page views per session are few. Code minimization generates a code that forms a clone pair with the original code of the same semantic by removing unnecessary characters such as new line characters and comments to reduce the amount of data being transferred to browsers. We also found the use of browser-dependent or device-specific code in web application clones as a means of compatibility improvement. We observed such use because, in web applications, a single version of application should support different browsers and platforms of various users while standalone projects often contain different branches for different platforms. In addition, the use of tools that support code generation (Wikipedia 2015) and the use of code sequences duplication to improve development time not only occurs in standalone projects, but also in web applications. Such uses lead to more files associated with clones in web applications because they enable web application developers to generate code or quickly duplicate and modify their code in order to deploy their applications in a short period of time. The use of interactive design patterns in web applications development may also lead to code duplication. Researchers showed that they can decompose high-level design patterns into low-level ones (Van Welie and Van der Veer 2003). Even though there can be different implementations for the same pattern, the implementations share a set of common features (Di Lucca et al. 2005), which explain the occurrences of code clones in web applications. Some language features only in statically-typed languages cause more function-level clones to improve comprehensibility. For example, the overloading mechanism in Java provides a set of similar method declarations with different parameter types, which are subject to function-level clones. However, other potentially influential factors on results may exist such as developers' experiences and application types. Our future work may examine other influential factors on the properties of code clones.

6.3.3 Statistical Conclusion Validity

In our experiments, we measured the correlations between clone metrics and software metrics. A threat to the statistical conclusion validity is concerned with the statistical power. We have a limited number of subjects in our measurement of correlation and, hence, such sample sizes increase the probability of making a Type II error (concluding that there is no effect when there actually is). Another threat is concerned with the reliability of measures and treatment implementation. Our measures and treatment implementation are considered reliable, since we detected clones on all subjects with the same approach and measured the clone metrics and software metrics with the same tool.

7 Conclusion

We developed JSCD, a JavaScript clone detector based on JavaScript characteristic vectors inspired by Deckard, a state-of-the-art C/Java clone detector. Using JSCD and Deckard, we detected clones in JSweb, JSproj, and Java. We compared their clone properties, measured their correlation, and manually inspected the clones to understand the motivations for cloning. We found several clone properties in JSweb such as 95 % of inter-file clones and 91–97 % of files associated with clones different from those of JSproj and Java. Our manual investigations revealed that language features play an important role in determining the ways developers duplicate code. Also, clone properties of the same language in different application domains can be different. In addition, code clones may not always be unintentional and risky as previous studies have claimed, and qualitative analysis is essential in understanding the motivations for cloning.

In this study, we did not consider external JavaScript source files because it is not easy to automatically distinguish whether those files are from web application developers or libraries. However, we observed that performance requirements of web applications affect how developers choose between inlined and external JavaScript source files. Therefore, our future work will study other factors of web applications that may affect the locations of JavaScript source files. Also, we studied clones only within web applications but not across web applications. We plan to study clones across web applications because it is beneficial in identifying license violations between applications.

Acknowledgments This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2014R1A2A2A01003235 and NRF-2008-0062609), Samsung Electronics, and Google.

References

Abd-El-Hafiz SK (2012) A metrics-based data mining approach for software clone detection. In: Proceedings of the 36th annual computer software and applications conference. IEEE, pp 35–41

Alexa Internet Inc (2013) Alexa top sites. http://www.alexa.com/topsites

Antoniol G, Casazza G, Di Penta M, Merlo E (2001) Modeling clones evolution through time series. In: Proceedings of international conference on software maintenance. IEEE, pp 273–280

Antoniol G, Villano U, Merlo E, Di Penta M (2002) Analyzing cloning evolution in the Linux kernel. Inf Softw Technol 44(13):755–765

Aversano L, Cerulo L, Di Penta M (2007) How clones are maintained: an empirical study. In: Proceedings of the 11th European conference on software maintenance and reengineering. IEEE, pp 81–90

Bakerm BS (1995) On finding duplication and near-duplication in large software systems. In: Proceedings of the 2nd working conference on reverse engineering. IEEE, pp 86–95

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999) Measuring clone based reengineering opportunities. In: Proceedings of the 6th international software metrics symposium. IEEE, pp 292–303

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: Proceedings of international conference on software maintenance. IEEE, pp 368–377

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E (2007) Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 33(9):577–591

Bettenburg N, Shang W, Ibrahim WM, Adams B, Zou Y, Hassan AE (2012) An empirical study on inconsistent changes to code clones at the release level. Sci Comput Program 77(6):760–776

Blanco L, Dalvi N, Machanavajjhala A (2011) Highly efficient algorithms for structural clustering of large websites. In: Proceedings of the 20th international conference on World wide web. ACM, pp 437–446

Boldyreff C, Kewish R (2001) Reverse engineering to achieve maintainable WWW sites. In: Proceedings of the 8th working conference on reverse engineering. IEEE, pp 249–257

Brito e Abreu F (1995) The MOOD metrics set. In: Proceedings of European conference on object-oriented programming, vol 95, p 267

Brixtel R, Fontaine M, Lesner B, Bazin C, Robbes R (2010) Language-independent clone detection applied to plagiarism detection. In: Proceedings of the 10th IEEE working conference on source code analysis and manipulation. IEEE, pp 77–86

Bulychev P, Minea M (2009) An evaluation of duplicate code detection using anti-unification. In: Proceedings of the 3rd international workshop on software clones. Citeseer, pp 22–27

Burd E, Bailey J (2002) Evaluating clone detection tools for use during preventative maintenance. In: Proceedings of the 2nd international workshop on source code analysis and manipulation. IEEE, pp 36–43

Cai D, Kim M (2011) An empirical study of long-lived code clones. Fundamental approaches to software engineering, pp 432–446

Calefato F, Lanubile F, Mallardo T (2004) Function clone detection in web applications: a semiautomated approach. J Web Eng 3:3–21

Chidamber SR, Kemerer CF (1994) A metrics suite for object oriented design. IEEE Trans Softw Eng 20(6):476–493

Chugh R, Meister JA, Jhala R, Lerner S (2009) Staged information flow for javascript. In: ACM Sigplan Notices, vol 44. ACM, pp 50–62

Cordy JR, Dean TR, Synytskyy N (2004) Practical language-independent detection of near-miss clones. In: Proceedings of the 2004 conference of the centre for advanced studies on collaborative research. IBM Press, pp 1–12

Datar M, Immorlica N, Indyk P, Mirrokni VS (2004) Locality-sensitive hashing scheme based on p-stable distributions. In: Proceedings of the 12th annual symposium on computational geometry. ACM, pp 253–262

De Lucia A, Francese R, Scanniello G, Tortora G (2005) Understanding cloned patterns in web applications. In: Proceedings of the 13th international workshop on program comprehension. IEEE, pp 333–336

De Lucia A, Scanniello G, Tortora G (2007) Identifying similar pages in web applications using a competitive clustering algorithm. J Softw Maint Evol Res Pract 19(5):281–296

De Lucia A, Risi M, Scanniello G, Tortora G (2009) An investigation of clustering algorithms in the identification of similar web pages. J Web Eng 8(4):346–370

Di Lucca GA, Di Penta M, Fasolino AR (2002) An approach to identify duplicated web pages. In: Proceedings of the 26th annual international computer software and applications conference. IEEE, pp 481–486

Di Lucca GA, Fasolino AR, Tramontana P (2005) Recovering interaction design patterns in web applications. In: Proceedings of the 9th European conference on software maintenance and reengineering. IEEE, pp 366–374

Ducasse S, Nierstrasz O, Rieger M (2004) Lightweight detection of duplicated code–a language-independent approach. Institute for Applied Mathematics and Computer Science. University of Berne

Ducasse S, Nierstrasz O, Rieger M (2006) On the effectiveness of clone detection by string matching. J Softw Maint Evol Res Pract 18(1):37–58

Falke R, Frenzel P, Koschke R (2008) Empirical evaluation of clone detection using syntax suffix trees. Empir Softw Eng 13(6):601–643

Finifter M, Weinberger J, Barth A (2010) Preventing capability leaks in secure javascript subsets. In: NDSS

Fowler M, Beck K (1999) Refactoring : improving the design of existing code. Addison-Wesley, Reading

GitHub Inc (2013) JavaScript Projects in GitHub. https://github.com/trending?l=javascript

Guarnieri S, Pistoia M, Tripp O, Dolby J, Teilhet S, Berg R (2011) Saving the world wide web from vulnerable JavaScript. In: Proceedings of the 20th international symposium on software testing and analysis

Harris S (2013) Simian–similarity analyser. http://www.harukizaemon.com/simian/index.html

Hegedűs P, Bakota T, Illés L, Ladányi G, Ferenc R, Gyimóthy T (2011) Source code metrics and maintainability: a case study. In: Software engineering, business continuity, and education. Springer, Berlin Heidelberg New York, pp 272–284

Hill R, Rideout J (2004) Automatic method completion. In: Proceedings of the 19th international conference on automated software engineering. IEEE, pp 228–235

Hotta K, Sano Y, Higo Y, Kusumoto S (2010) Is duplicate code more frequently modified than non-duplicate code in software evolution?: an empirical study on open source software. In: Proceedings of the joint ERCIM workshop on software evolution and international workshop on principles of software evolution, pp 73–82

Islam M, Islam M, Halim T (2011) A study of code cloning in server pages of web applications developed using classic asp. net and asp. net mvc framework. In: Proceedings of the 14th international conference on computer and information technology. IEEE, pp 497–502

Jang J, Agrawal A, Brumley D (2012) ReDeBug: finding unpatched code clones in entire os distributions. In: Proceedings of symposium on security and privacy. IEEE, pp 48–62

Jiang L, Misherghi G, Su Z, Glondu S (2007a) Deckard: scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on software engineering. IEEE, pp 96–105

Jiang L, Su Z, Chiu E (2007b) Context-based detection of clone-related bugs. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM, pp 55–64

Jones MC (2011) Remix and reuse of source code in software production. PhD thesis, Citeseer

Juergens E, Deissenboeck F, Hummel B (2009a) CloneDetective–a workbench for clone detection research. In: Proceedings of the 31st international conference on software engineering. IEEE, pp 603–606

Juergens E, Deissenboeck F, Hummel B, Wagner S (2009b) Do code clones matter? In: Proceedings of the 31st international conference on software engineering. IEEE Computer Society, pp 485–495

Kamei Y, Sato H, Monden A, Kawaguchi S, Uwano H, Nagura M, Matsumoto Ki, Ubayashi N (2011) An empirical study of fault prediction with code clone metrics. In: Proceedings of the joint conference of the 21th international workshop on software measurement and the 6th international conference on software process and product measurement. IEEE, pp 55–61

Kamiya T, Kusumoto S, Inoue K (2002) CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 28(7):654–670

Kapser C, Godfrey M (2003) Toward a taxonomy of clones in source code: a case study. In: Proceedings of the conference on evolution of large scale industrial software architectures, pp 67–78

Kapser CJ, Godfrey MW (2008) “Cloning Considered harmful” considered harmful: patterns of cloning in software. Empir Softw Eng 13(6):645–692

Kienle HM, Müller HA, Weber A (2003) In the web of generated “clones” (position paper)

Kim H, Jung Y, Kim S, Yi K (2011) Mecc: memory comparison-based clone detector. In: Proceedings of the 33rd international conference on software engineering. IEEE, pp 301–310

Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. ACM SIGSOFT Softw Eng Notes 30(5):187–196

Kontogiannis K (1997) Evaluation experiments on the detection of programming patterns using software metrics. In: Proceedings of the 4th working conference on reverse engineering. IEEE, pp 44–54

Kontogiannis K, DeMori R, Merlo E, Galler M, Bernstein M (1996) Pattern matching for clone and concept detection. Autom Softw Eng 3(1-2):77–108

Koschke R (2007) Survey of research on software clones. Duplication, redundancy, and similarity in software. http://drops.dagstuhl.de/volltexte/2007/962/

Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. In: Proceedings of the 13th working conference on reverse engineering. IEEE, pp 253–262

Koschke R, Baxter ID, Conradt M, Cordy JR (2012) Software clone management towards industrial application (dagstuhl seminar 12071). Dagstuhl Reports 2(2)

Kou G, Lou C (2012) Multiple factor hierarchical clustering algorithm for large scale web page and search engine clickstream data. Ann Oper Res 197(1):123–134

Kozlov D, Koskinen J, Sakkinen M, Markkula J (2010) Exploratory analysis of the relations between code cloning and open source software quality. In: Proceedings of the 7th international conference on the quality of information and communications technology. IEEE, pp 358–363

Krinke J (2007) A study of consistent and inconsistent changes to code clones. In: Proceedings of the 14th working conference on reverse engineering. IEEE, pp 170–178

Krinke J (2008) Is cloned code more stable than non-cloned code? In: Proceedings of the 8th international working conference on source code analysis and manipulation. IEEE, pp 57–66

Krinke J (2011) Is cloned code older than non-cloned code? In: Proceedings of the 5th international workshop on software clones. ACM, pp 28–33

Lague B, Proulx D, Mayrand J, Merlo E, Hudepohl J (1997) Assessing the benefits of incorporating function clone detection in a development process. In: Proceedings of the international conference on software maintenance, vol 97

Lanubile F, Mallardo T (2003) Finding function clones in web applications. In: Proceedings of the 7th European conference on software maintenance and reengineering. IEEE, pp 379–386

Lee H, Won S, Jin J, Cho J, Ryu S (2012) SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In: Proceedings of the 19th international workshop on foundations of object-oriented languages

Li C, Sun J, Chen H (2014) An improved method for tree-based clone detection in web applications. In: Proceedings of the 4th international conference on digital information and communication technology and it’s applications. IEEE, pp 363–367

Li J, Ernst MD (2012) Cbcd: cloned buggy code detector. In: Proceedings of the 2012 international conference on software engineering. IEEE Press, pp 310–320

Livieri S, Higo Y, Matsushita M, Inoue K (2007) Analysis of the linux kernel evolution using code clone coverage. In: Proceedings of the 4th international workshop on mining software repositories. IEEE, pp 22–22

Lozano A, Wermelinger M, Nuseibeh B (2008) Evaluating the relation between changeability decay and the characteristics of clones and methods. In: Proceedings of the 23rd international conference on automated software engineering-workshops. IEEE, pp 100–109

Martin D, Cordy JR (2011) Analyzing web service similarity using contextual clones. In: Proceedings of the 5th international workshop on software clones. ACM, pp 41–46

Martinsen JK, Grahn H, Isberg A (2011) A comparative evaluation of javascript execution behavior. In: Web engineering. Springer, Berlin Heidelberg New York, pp 399–402

Mayrand J, Leblanc C, Merlo EM (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of international conference on software maintenance. IEEE, pp 244–253

Merlo E, Antoniol G, Di Penta M, Rollo VF (2004) Linear complexity object-oriented similarity for clone detection and software evolution analyses. In: Proceedings of the 20th international conference on software maintenance. IEEE, pp 412–416

Merlo E, Dagenais M, Bachand P, Sormani J, Gradara S, Antoniol G (2002) Investigating large software system evolution: the Linux kernel. In: Proceedings of the 26th annual international computer software and applications conference. IEEE, pp 421–426

Mondal M, Roy CK, Rahman MS, Saha RK, Krinke J, Schneider KA (2012) Comparative stability of cloned and non-cloned code: an empirical study. In: Proceedings of the 27th annual symposium on applied computing. ACM, pp 1227–1234

Monden A, Nakae D, Kamiya T, Sato S, Matsumoto K (2002) Software quality analysis by code clones in industrial legacy software. In: Proceedings of the 8th symposium on software metrics. IEEE, pp 87–94

Muhammad T, Zibran MF, Yamamoto Y, Roy CK (2013) Near-miss clone patterns in web applications: an empirical study with industrial systems. In: Canadian conference on electrical and computer engineering

Negara N, Tsantalis N, Stroulia E (2013) Feature detection in ajax-enabled web applications. In: Proceedings of the 17th European conference on software maintenance and reengineering. IEEE, pp 154–163

Nikiforakis N, Invernizzi L, Kapravelos A, Van Acker S, Joosen W, Kruegel C, Piessens F, Vigna G (2012) You are what you include: large-scale evaluation of remote javascript inclusions. In: Proceedings of the 2012 ACM conference on computer and communications security. ACM, pp 736–747

Ocariza F, Pattabiraman K, Zorn B (2011) Javascript errors in the wild: an empirical study. In: Proceedings of the 22nd international symposium on software reliability engineering. IEEE, pp 100–109

Patenaude JF, Merlo E, Dagenais M, Laguë B (1999) Extending software quality assessment techniques to Java systems. In: Proceedings of the 7th international workshop on program comprehension. IEEE, pp 49–56

PLRG@KAIST (2012) SAFE: Scalable Analysis Framework for ECMAScript. http://plrg.kaist.ac.kr/redmine/projects/jsf/repository

PMD (2013) PMD's copy/paste detector. http://pmd.sourceforge.net/pmd-5.0.5/cpd-usage.html

Rahman F, Bird C, Devanbu P (2012) Clones: what is that smell Empir Softw Eng 17(4-5):503–530

Rajapakse D, Jarzabek S (2005) An investigation of cloning in web applications. Web Engineering pp 252–262

Rajapakse DC, Jarzabek S (2007) Using server pages to unify clones in web applications: a trade-off analysis. In: Proceedings of the 29th international conference on software engineering. IEEE, pp 116–126

Ramage D, Heymann P, Manning CD, Garcia-Molina H (2009) Clustering the tagged web. In: Proceedings of the 2nd ACM international conference on web search and data mining. ACM, pp 54–63

Ratanaworabhan P, Livshits B, Zorn BG (2010) JSMeter: comparing the behavior of JavaScript benchmarks with real web applications. In: Proceedings of the 2010 USENIX conference on Web application development. USENIX Association, pp 3–3

Richards G, Hammer C, Burg B, Vitek J (2011) The eval that men do. In: Proceedings of the 25th European conference on object-oriented programming. Springer, Berlin Heidelberg New York, pp 52–78

Richards G, Lebresne S, Burg B, Vitek J (2010) An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the SIGPLAN conference on programming language design and implementation, vol 45. ACM, pp 1–12

Rieger M, Ducasse S, Lanza M (2004) Insights into system-wide code duplication. In: Proceedings of the 11th working conference on reverse engineering. IEEE, pp 100–109

Roy C, Cordy J (2007) A survey on software clone detection research. Queen's School of Computing TR 541:115

Roy C, Cordy J (2010a) Are scripting languages really different? In: Proceedings of the 4th international workshop on software clones. ACM, pp 17–24

Roy C, Cordy J (2010b) Near-miss function clones in open source software: an empirical study. J Softw Maint Evol Res Pract 22(3):165–189

Roy CK, Cordy JR (2008) An empirical study of function clones in open source software. In: Proceedings of the 15th working conference on reverse engineering. IEEE, pp 81–90

Roy CK, Cordy JR (2009) A mutation/injection-based automatic framework for evaluating code clone detection tools. In: Proceedings of the international conference on software testing, verification and validation workshops. IEEE, pp 157–166

Roy CK, Cordy JR, Koschke R (2009) Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci Comput Program 74(7):470–495

Roy CK, Zibran MF, Koschke R (2014) The vision of software clone management: past, present, and future. In: Proceedings of the IEEE CSMR-18/WCRE-21 software evolution week

Rysselberghe FV, Demeyer S (2004) Evaluating clone detection techniques from a refactoring perspective. In: Proceedings of the 19th international conference on automated software engineering. IEEE Computer Society, pp 336–339

SAFE Corporation (2012) CodeMatch. http://www.safe-corp.biz/products_codematch.htm

Saha RK, Roy CK, Schneider KA (2011) An automatic framework for extracting and classifying near-miss clone genealogies. In: Proceedings of the 27th international conference on software maintenance. IEEE, pp 293–302

Schleimer S, Wilkerson DS, Aiken A (2003) Winnowing: local algorithms for document fingerprinting. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data. ACM, pp 76–85

Selamat A, Wahid N (2007) Code clone detection using string based tree matching technique. InTech

Shawky DM, Ali AF (2010) An approach for assessing similarity metrics used in metric-based clone detection techniques. In: Proceedings of the 3rd international conference on computer science and information technology, vol 1. IEEE, pp 580–584

Stephan M, Alalfi MH, Stevenson A, Cordy JR (2013) Using mutation analysis for a model-clone detector comparison framework. In: Proceedings of the 2013 international conference on software engineering. IEEE Pres, Piscataway, pp 1261–1264

Stephan M, Alalfi MH, Cordy JR (2014) Towards a taxonomy for simulink model mutations. In: Proceedings of the 7th international conference on software testing, verification and validation workshops. IEEE, pp 206–215

Svajlenko J, Roy CK, Zibran MF, Cordy JR (2013) A mutation analysis based benchmarking framework for clone detectors. In: Proceedings of short/tool papers track of the ICSE 7th international workshop on software clones

Tairas R, Gray J (2006) Phoenix-based clone detection using suffix trees. In: Proceedings of the 44th annual southeast regional conference. ACM, pp 679–684

Thummalapenta S, Cerulo L, Aversano L, Di Penta M (2010) An empirical study on the maintenance of source code clones. Empir Softw Eng 15(1):1–34

Van Welie M, Van der Veer GC (2003) Pattern languages in interaction design: structure and organization. In: Proceedings of interact, vol 3, pp 1–5

Wang T, Harman M, Jia Y, Krinke J (2013) Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering. ACM, pp 455–465

Wikipedia (2015) List of graphical user interface builders and rapid application development tools. http://en.wikipedia.org/wiki/List_of_graphical_user_interface_builders_and_rapid_application_development_tools

Yamanaka Y, Choi E, Yoshida N, Inoue K, Sano T (2013) Applying clone change notification system into an industrial development process. In: Proceedings of the 21st international conference on program comprehension. IEEE, pp 199–206

Zibran MF, Roy CK (2012) Ide-based real-time focused search for near-miss clones. In: Proceedings of the 27th annual ACM symposium on applied computing. ACM, pp 1235–1242

Zibran MF, Saha RK, Asaduzzaman M, Roy CK (2011) Analyzing and forecasting near-miss clones in evolving software: an empirical study. In: Proceedings of the 16th international conference on engineering of complex computer systems. IEEE, pp 295–304

Wai Ting Cheung received the BEng degree in computer engineering and the MPhil degree in computer science and engieering from The Hong Kong University of Science and Technology. He is currently pursuing the joint PhD degree in Computer Science at Korea Advanced Institute of Science and Technology and The Hong Kong University of Science and Technology. His research interests include code clone analysis and fault localization.

Sukyoung Ryu is an assistant professor of Computer Science at Korea Advanced Institute of Science and Technology (KAIST). Before joining KAIST, she worked at Sun Microsystems Laboratories to design and develop the new programming language, Fortress. Before that, she was a Research Associate in Computer Science at Harvard, where she worked on the Debugging Everywhere project. She is the lead author of the publicly-available Scalable Analysis Framework for ECMAScript (SAFE), which is integrated in the Tizen SDK of the Tizen Linux Foundation project. Her main research interests include programming languages, program analysis, and programming environments for debugging and testing.

Sunghun Kim is an Assistant Professor of Computer Science at the Hong Kong University of Science and Technology. He got his BS in Electrical Engineering at Daegu University, Korea in 1996. He completed his Ph.D. in the Computer Science Department at the University of California, Santa Cruz in 2006. He was a postdoctoral associate at Massachusetts Institute of Technology and a member of the Program Analysis Group. He was a Chief Technical Officer (CTO), and led a 25-person team at the Nara Vision Co. Ltd, a leading Internet software company in Korea for six years.

His core research area is Software Engineering, focusing on software evolution, program analysis and empirical studies. He publishes his work on top venues such as TSE, ICSE, FSE, AAAI, SOSP and ISSTA. He is a four-time winner of the ACM SIGSOFT Distinguished Paper Award (ICSE 2007, ASE 2012, ICSE 2013 and ISSTA 2014). Besides, he received various awards including 2010 and 2011 Microsoft Software Innovation Awards and 2011 Google Faculty Research Award. He served on a variety of program committees including FSE 2011, FSE 2013, FSE 2015, ASE 2013, ICSE 2012, ICSE 2013 and ICSE 2015. He was a program co-chair of MSR 2013 and 2014. Further information is available at http://www.cse.ust.hk/~hunkim.