Empirical Software Engineering
https://doi.org/10.1007/s10664-018-9645-2

# Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones

**Patanamon Thongtanunam**[1] · **Weiyi Shang**[2] · **Ahmed E. Hassan**[3]

Published online: 04 September 2018


**Abstract**
Code clones are created when a developer duplicates a code fragment to reuse existing functionalities. Mitigating clones by refactoring them helps ease the long-term maintenance of large software systems. However, refactoring can introduce an additional cost. Prior work also suggest that refactoring all clones can be counterproductive since clones may live in a system for a short duration. Hence, it is beneficial to determine in advance whether a newly-introduced clone will be short-lived or long-lived to plan the most effective use of resources. In this work, we perform an empirical study on six open source Java systems to better understand the life expectancy of clones. We find that a large number of clones (i.e., 30% to 87%) lived in the systems for a short duration. Moreover, we find that although short-lived clones were changed more frequently than long-lived clones throughout their lifetime, short-lived clones were consistently changed with their siblings less often than long-lived clones. Furthermore, we build random forest classifiers in order to determine the life expectancy of a newly-introduced clone (i.e., whether a clone will be short-lived or long-lived). Our empirical results show that our random forest classifiers can determine the life expectancy of a newly-introduced clone with an average AUC of 0.63 to 0.92. We also find that the churn made to the methods containing a newly-introduced clone, the complexity and size of the methods containing the newly-introduced clone are highly influential in determining whether the newly-introduced clone will be short-lived. Furthermore, the size of a newly-introduced clone shares a positive relationship with the likelihood that the newly-introduced clone will be short-lived. Our results suggest that, to improve the efficiency of clone management efforts, practitioners can leverage our classifiers and insights in order to determine whether a newly-introduced clone will be short-lived or long-lived to plan the most effective use of their clone management resources in advance.

**Keywords** Code clone · Software management · Software maintenance · Software evolution

Communicated by: Tim Menzies

✉ Patanamon Thongtanunam
patanamon.thongtanunam@unimelb.edu.au

Extended author information available on the last page of the article.

## 1 Introduction

Code clones are created when a developer duplicates a code fragment to reuse existing functionalities. Prior studies show that a sizable portion of the code in a system consists of such duplicate code fragments (e.g., 29% of JDK (Johnson 1993) and 22.7% of Linux (Li et al. 2006)). Clones can become a problem within large software systems when developers perform inconsistent changes to these clones (Baker 1995). For example, a bug fix to a clone might need to be repetitively applied to all siblings. Hence, a presence of clones may significantly increase the maintenance cost of software systems.

Mitigating clones by refactoring them will benefit practitioners for future maintenance (Fowler and Beck 1999). A recent study shows that refactoring often occurs in GitHub projects, i.e., 2,241 refactoring instances were detected in 285 of the studied GitHub projects (Silva et al. 2016). Moreover, Silva et al. (2016) reported that one of the popular reasons for refactoring is to reduce code repetitiveness (i.e., code clones). For instance, one developer in the study of Silva et al. (2016) argued that *"The reason for me to do the refactoring was: Don't repeat yourself (DRY)"*.

On the other hand, refactoring all clones can be counterproductive (Cordy 2003; Kapser and Godfrey 2006; 2008; Kim et al. 2012; Wang and Godfrey 2014). For instance, recent work finds that some patterns of clones are not considered harmful and refactoring them is unnecessary and counter productive (Kapser and Godfrey 2008). Kim et al. (2012) report that refactoring in general is considered by developers at Microsoft as a task with substantial cost and risks. Wang and Godfrey (2014) also argue that clones should be prioritized for refactoring based on the benefits, costs and potential risks.

To better manage clones, Kim et al. (2005) use clone genealogies to understand the clone evolution of clones. They suggest that refactoring all clones may not be worthwhile since 37% to 41% of clones in their studied systems only lived for a short-duration, and many of the long-lived clones cannot be removed using standard refactoring techniques. Such findings indicate that it is important to determine in advance whether a clone will be short-lived or long-lived to manage clones more efficiently. Hence, we perform an empirical study on six open source Java systems (i.e., Ant, Camel, Jackrabbit, Maven, Pig, and Tomcat) to address the following two preliminary questions:

(PQ1) **How long do clones live in a software system?**
We find that 30% to 87% of clones in the studied systems lived for a short duration. In particular, these clones disappeared within two to nine released versions which account for less than 17% of all the released versions.

(PQ2) **How were short-lived and long-lived clones changed throughout their lifetime?**
We find that many of the long-lived clones (25% to 37%) were consistently changed with their siblings, while a smaller proportion of short-lived clones (9% to 19%) were consistently changed with their siblings. This result suggests that short-lived clones may not require change consistency as much as long-lived clones do.

The findings of our preliminary study suggest that the maintenance effort that is associated with such short-lived clones is relatively smaller than the effort for long-lived clones, supporting our intuition that avoiding refactoring of short-lived clones will be beneficial for the clone management efforts. Hence, we build random forest classifiers in order to determine the life expectancy of a newly-introduced clone (i.e., whether a clone will be short-lived or long-lived). Leveraging our classifiers, practitioners can plan the most effective use of their resources. Moreover, we analyze our classifiers in order to identify the

characteristics that influence the life expectancy of a clone. Knowledge of such influential characteristics can help practitioners chart better clone management plans. Hence, we address the following research questions:

(RQ1) **How well can we determine whether an introduced clone will be short-lived versus long-lived?**

Our random forest classifiers achieve an average AUC of 0.63 to 0.92. Practitioners can leverage our classifiers to determine the life expectancy of a newly-introduced clone in order to plan for efficient clone management.

(RQ2) **What are the most influential characteristics for determining the life expectancy of an introduced clone?**

We find that the performed development activity that was made to the method containing a newly-introduced clone during the version in which the clone is introduced is highly influential in determining the life expectancy of a clone. For example, we find that the more added lines into the method containing a clone during the development cycle of the version in which the clone is introduced, the more likely the clone will be short-lived. In other words, clones, that are introduced in versions with a large amount of churn, are more likely to be short-lived. We also find that the size and complexity of the method containing a newly-introduced clone shares a positive relationship with the likelihood of the clone being short-lived. Furthermore, the size of a newly-introduced clone also shares a positive relationship with the likelihood that the clone will be short-lived. The immediate refactoring of clones with these characteristics may not be beneficial since such clones will disappear within a short period of time.

### 1.1 Paper Organization

The remainder of the paper is organized as follows. Section 2 describes our case study design. Section 3 discusses our preliminary study to understand clone genealogies. Section 4 motivates our research questions and discusses the results of our study. Section 5 surveys related work, while Section 6 discusses the threats to the validity of our study. Section 7 draws conclusions.

## 2 Case Study Design

In this work, we define *a clone* as a group of duplicate code fragments that are nearly-identical in terms of their structure and/or semantics. Each of these duplicate code fragment

**Table 1** An overview of the studied systems

| System | Age (Yrs) | #Released Versions | LOC | #Classes | #Methods | #Commits |
|---|---|---|---|---|---|---|
| Ant | 17 | 22 | 138,388 | 1,705 | 12,727 | 11,373 |
| Camel | 10 | 35 | 615,118 | 17,280 | 62,840 | 25,897 |
| Jackrabbit | 13 | 66 | 342,116 | 3,971 | 27,492 | 8,209 |
| Maven | 14 | 36 | 80,149 | 899 | 6,575 | 10,264 |
| Pig | 8 | 15 | 252,925 | 2,536 | 16,111 | 2,852 |
| Tomcat | 11 | 43 | 317,506 | 3,328 | 27,629 | 15,416 |

is defined as *a clone sibling*. These clone siblings can be introduced by copying one piece of existing code into another one, or the clone siblings are introduced into the source code at the same time. We consider the "birth" of a clone as the time when we start to detect a group of duplicate code fragments. Such a birth of clones can be due to either when a sibling of the clone is introduced, or when a group of duplicate code fragments is created all at once.

Below, we describe our studied systems, data preparation process and model construction approach in our study of clone genealogies and the life expectancy of clones.

### 2.1 Studied Systems

We perform our empirical study on six well-known open source Java systems of the Apache software foundation, i.e., Ant, Camel, Jackrabbit, Maven, Pig, and Tomcat.[1] Table 1 shows an overview of our six studied systems. Each of these systems are Java systems has an approximate age of 8 to 17 years.[2] Moreover, these systems have many released versions from the main branch (excluding release candidates).

Ant is a command-line tool for automating the software build process of Java code bases.[3] Camel is a tool that provides a Java object-based implementation of the Enterprise Integration Patterns using an application programming interface to configure routing and mediation rules.[4] Jackrabbit is a hierarchical content store with support for structured and unstructured content of Java Technology API.[5] Maven is an advanced build automation tool for Java code bases.[6] Pig is a platform for complex MapReduced transformations that are used to analyze large data sets.[7] Tomcat is a web container for Java Servlet, JavaServer Pages (JSP), Java EL, and WebSocket, and provides a Java HTTP web server environment in which Java code can run.[8]

### 2.2 Data Preparation

In this section, we describe our data preparation process. Figure 1 provides an overview of our data preparation process which consists of four steps. Below, we describe each step in detail.

#### 2.2.1 (DP1) Extract Sequentially Developed Versions

Similar to prior work (Lague et al. 1997; Bettenburg et al. 2009; Göde and Koschke 2011), we study the clone genealogies at the released versions of our studied systems in order to examine the life expectancy of clones in a long term. We mine the official Git repositories of Ant, Camel, and Maven. For Jackrabbit, Pig, and Tomcat, we mine the Git repositories that are a mirror Version Control System (VCS) containing full version histories (including

[1] Tomcat6.0, 7.0, and 8.0 are asynchronously developed in different repositories. We select Tomcat8.0 for this study since it is a long lived version.

[2] The version control systems of the systems were accessed on April, 2017

[3] http://ant.apache.org/

[4] http://camel.apache.org/

[5] http://jackrabbit.apache.org/jcr/index.html

[6] https://maven.apache.org/

[7] https://pig.apache.org

[8] http://tomcat.apache.org/index.html

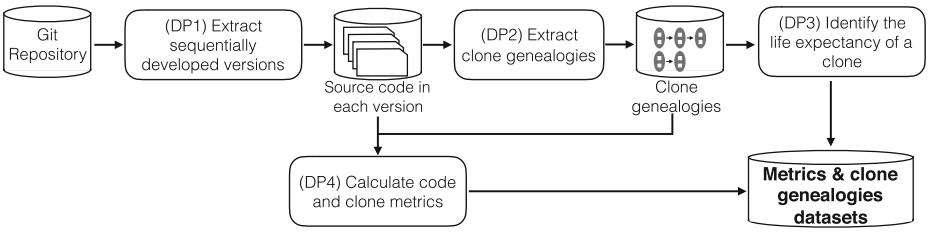


**Fig. 1** An overview of our data preparation process

branches and tags) from the respective source trees of the official Subversion repositories of these systems.[9]

Since we extract clone genealogies based on the list of released versions, we need to ensure that those released versions are sequentially developed. It is possible that some of the released versions were asynchronously developed on an isolated branch in the VCS of the system (e.g., maintenance versions). For example, Fig. 2 shows that Camel v2.17.1 to v2.17.4 are asynchronously developed on an isolated branch in the VCS. Then, a clone that is introduced in v2.17.4 will be misidentified as short-lived since the development history of v2.17.x has discontinued. Therefore, for this example, we only consider the clones that are introduced in the released versions in the main branch (i.e., Camel v2.16.0, v2.17.0, and v2.18.1 to v2.18.3).

To extract the released versions that were sequentially developed, we use the `git log` command on the most recently released version of each studied system[10] in order to retrieve a full list of commits that (1) occurred on the same branch as the recent version or (2) that originated on other branches, but have been merged into the same branch as the recent version. Then, we use the `git name-rev` command to identify the corresponding tags of the released versions of each commit. In this study, we do not consider release candidates (e.g., Maven 3.0-RC) as a released version since such release candidates are trial versions which are available for a short duration and their code is likely to change for the final released version. Therefore, we consider the commits that are associated with release candidates as a part of the development history of the following version of these release candidates.

### 2.2.2 (DP2) Extract Clone Genealogies

To extract the studied clone genealogies, we perform an incremental clone detection on the methods across all versions using iClones (Göde and Koschke 2009). Unlike traditional genealogy extractors that compare clones in each version with the first version, the incremental detection algorithm of iClones provides a more comprehensive version-by-version view of the evolution of clones. Broadly speaking, the incremental detection algorithm of iClones performs two main steps. First, it extracts clones in each version of the studied systems using generalized suffix trees and Baker's `pdup` algorithm (Baker 1997). Then, iClones uses the change history between two consecutive versions (i.e., the current version $n$ and the previous version $n - 1$) to generate a clone genealogy. We configure iClones in the incremental mode to create clone genealogies. Similar to prior work (Kim et al. 2005),

[9] A list of Git repositories of Apache projects is available at https://git.apache.org/

[10] The most recent version of our studied systems are Ant v1.10.0, Camel v2.18.3, Jackrabbit v2.14.0, Maven v3.5.0, Pig v0.16.0, and Tomcat v8.0.43

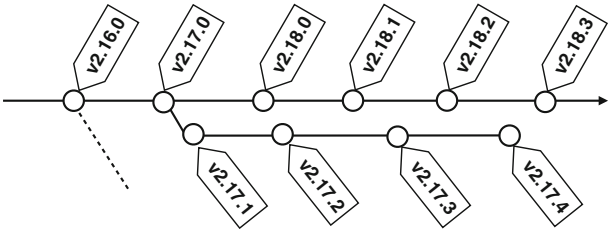


**Fig. 2** An example of released versions that are asynchronously developed on an isolated branch in the VCS of Camel

we configure iClones to remove *syntactic templates*, i.e., code comments, array initialization, import, and package statements by setting the option of `transformers` as 'all'. In addition, we configure iClones to detect all three types of clones, i.e., Type-1 (exact clone), Type-2 (parameter-substituted clone), and Type-3 (near-miss clone) by setting the `mapping` option as 'bazrafshan' (Göde and Koschke 2009). We also configure iClones to consider late change propagation (Thummalapenta et al. 2010; Barbour et al. 2011) when identifying a changing consistency in clone genealogies. Broadly speaking, iClones keeps all the dead clone genealogies in memory as "ghost fragments", in order to identify late change propagation (Göde 2011). This setting allows us to ensure that the changing consistency in clone genealogies will be correctly identified even in the cases where a developer might at a later time resynchronize changes after forgetting apply a consistent change to all clone siblings. For all other configuration options, we use the default setting of the tool. For example, the minimum length of identical token sequences that are used to merge near-miss clones (`minblock`) is set to 20 and the minimum length of clones measured in tokens (`minclone`) is set to 100. These settings allow iClones to detect less false positive clones (Göde and Koschke 2011).

### 2.2.3 (DP3) Identify the Life Expectancy of a Clone

We measure the life expectancy of a clone (i.e., the clone genealogy length) by counting the number of versions across which a clone genealogy spans. Similar to prior work (Kim et al. 2005), we study only the life expectancy of volatile clones (i.e., clones that disappeared before the last studied version). This is because we cannot determine the life expectancy of non-volatile clones as these clones still live in the system.

Once we measure the life expectancy, we identify the short-lived and long-lived clones. There can be various definitions of short-lived clones. For example, a short-lived clone can be defined in a more domain-oriented or project-specific fashion based on the internal knowledge of a system or based on the interest of a project team (e.g., choosing one version as a short-lived clone, or using the number of days or commits to determine the life expectancy of clones). To ease the replication of our work, we opt to identify short-lived clones using an automated rule. Therefore, we use the K-means clustering technique to partition the clones based on the number of versions in which a clone lived. The K-means clustering technique is widely used to partition instances in a dataset of interest (Khanchouch et al. 2015; Wagstaff et al. 2001). Moreover, we also use the `NbClust` function of the `NbClust` R package (Charrad et al. 2014) to determine the optimal number of clusters. In particular, we set the `method` option in the `NbClust` function as 'kmeans' in order to use the the K-means clustering technique for partitioning clones while determining the

optimal number of clusters. For each studied system, we apply the K-means clustering technique by providing the life expectancy of clones (i.e., the number of versions across which a clone genealogy spans) as an input to the `NbClust` function. We then consider the clones in the cluster with the clone genealogies that have the shortest lifetime as *short-lived* clones, while the clones in the remaining clusters are considered as *long-lived* clones.

#### 2.2.4 (DP4) Calculate Code and Clone Metrics

The life expectancy of a clone can be correlated to many factors. However, some factors are not quantifiable, e.g., a clone is short lived due to refactoring. In this work, we opt to examine the relationship between the measurable factors and the life expectancy of a newly-introduced clone. Therefore, we derive clone, product, and process focused metrics. Since one clone can have multiple clone siblings, we first compute metrics for each sibling of a clone. Then, we abstract the metrics of the clone siblings to the clone level by calculating an average and difference between the respective metrics of the siblings of each clone. For a clone $C$, the *average* and the *difference* of its siblings for a metric $M$ are calculated as follow:

$$\text{Avg\_}M(C) = \frac{\sum_{s \in S(C)} M(s)}{|S(C)|} \tag{1}$$

$$\text{Diff\_}M(C) = \max(\{\text{abs}(M(s_1) - M(s_2)) | s_1, s_2 \in S(C)\}) \tag{2}$$

where $S(C)$ is the set of siblings of the clone $C$, and $M(s)$ is the value of the metric $M$ for the clone sibling $s$.

Table 2 describes our 38 metrics for the clone, product, and process dimensions. We describe below each dimension and the motivation of our metrics.

**Clone Dimension** Clone metrics measure the characteristics of a clone when it was injected into the source code. In particular, we count how many siblings does a clone have at the version in which the clone is introduced (i.e., *SiblingCount*), measure the size of a clone (i.e., *CloneLineCount* and *TokenCount*), the difference between clone siblings (i.e., *DirectoryDistance* and *EditDistance*), and also identify the clone type (i.e., *CloneType*). *CloneLineCount* and *TokenCount* are calculated at the clone sibling level. We abstract these metrics to the clone level using Eqs. 1 and 2. On the other hand, *DirectoryDistance* and *EditDistance* measure the difference between a pair of clone siblings. Hence, we abstract these metrics to the clone level by calculating an average using Eq. 1 and by calculating a maximum distance of those pairs of clone siblings using the following equation.

$$\text{Max\_Dist}(C) = \max(\{(\text{Dist}(s_1, s_2) | s_1, s_2 \in S(C)\}) \tag{3}$$

**Product Dimension** Product metrics measure the code characteristics of the method that contains the clone at the version during which the clone is introduced. We use the *Understand* tool[11] to compute the product metrics for methods that contain clones. The Understand tool is a reverse engineering tool that is specifically designed to extract product metrics from the source code of software systems. Similar to the clone metrics, we first compute the product metrics at the clone sibling level. Then, we abstract these metrics to the clone level by calculating an average and a difference of the metrics using Eqs. 1 and 2.

[11] https://scitools.com/

**Table 2** An overview of the code and clone metrics

| Category | Metric | Description |
|---|---|---|
| Clone Dimension | | |
| Size | SiblingCount | Number of clone siblings in the clone. |
| | CloneLineCount | Number of physical lines in the clone sibling. |
| | TokenCount | Number of tokens in the clone sibling. |
| | DirectoryDistance | Number of directories that are traversed to reach from the method containing one sibling to the method containing another sibling of the clone. |
| | EditDistance | The string dissimilarity in source code between one sibling and another sibling of the clone. |
| | CloneType | Type of clone class to which the clone belongs. |
| Product Dimension | | |
| Size | LineCount | Number of lines in the method that contains the clone sibling. |
| | LineCodeCount | Number of source code lines in the method that contains the clone sibling. |
| | LineCodeDeclCount | Number of declarative source code lines in the method that contains the clone sibling. |
| | LineCodeExeCount | Number of executable source code lines in the method that contains the clone sibling. |
| | StmtCount | Number of declarative plus executable statements in the method that contains the clone sibling. |
| | StmtDeclCount | Number of declarative statements in the method that contains the clone sibling. |
| | StmtExeCount | Number of executable statements in the method that contains the clone sibling. |
| Relative Size | RatioLineCount | Ratio of *LineCount* to *CloneLineCount*. |
| | RatioLineCodeCount | Ratio of *LineCodeCount* to *CloneLineCount*. |
| | RatioLineCodeDeclCount | Ratio of *CodeDeclCount* to *CloneLineCount*. |
| | RatioLineCodeExeCount | Ratio of *CodeExeCount* to *CloneLineCount*. |
| | RatioStmtCount | Ratio of *StmtCount* to *CloneLineCount*. |
| | RatioStmtDeclCount | Ratio of *StmtDeclCount* to *CloneLineCount*. |
| | RatioStmtExeCount | Ratio of *StmtExeCount* to *CloneLineCount*. |

**Table 2** (continued)

| Category | Metric | Description |
|---|---|---|
| Complexity | FanOut | Number of unique methods that are called by the method containing the clone sibling. |
| | FanIn | Number of unique methods that call the method containing the clone sibling. |
| | Cyclomatic | McCabe Cyclomatic complexity of the method that contains the clone sibling. |
| | CyclomaticModified | Modified McCabe Cyclomatic complexity of the method that contains the clone sibling. |
| | CyclomaticStrict | Strict McCabe Cyclomatic complexity of the method that contains the clone sibling. |
| | Essential | Numerical measure of structuredness of the method that contains the clone sibling. |
| | MaxNesting | Maximum nesting level of control constructs in the method that contains the clone sibling. |
| | CommentLineCount | Number of comment lines in the method that contains the clone sibling. |
| | RatioCommentToCode | Ratio of *CommentLineCount* to *LineCodeCount*. |
| Process Dimension | | |
| Change | Churn | Sum of lines added into and deleted from the method that contains the clone sibling. |
| | LineAdded | Number of lines added into the method that contains the clone sibling. |
| | LineDeleted | Number of lines deleted from the method that contains the clone sibling. |
| | CommitCount | Number of commits that impact the method containing the clone sibling. |
| Human factor | DeveloperCount | Number of distinct developers who modified the method that contains the clone sibling. |
| | MajorDeveloperCount | Number of developers who contributed more than 5% of the commits that impact the method containing the clone sibling. |
| Quality | FixCommitCount | Number of commits with a description of fixing bugs and that impact the method containing the clone sibling. |
| | NewFeatureCommitCount | Number of commits that introduce a new feature and that impact the method containing the clone sibling. |
| | ImproveCommitCount | Number of commits that are other than bug fixing or new feature introduction commits, and that impact the method containing the clone sibling. |

The metrics are computed for each sibling of a clone when the clone is introduced, then abstracted to the clone level using Equations 1 and 2

We measure the code characteristics of the method that contains a clone in terms of size and complexity. Below, we describe each category in the product dimension.

- *Size*: Our intuition is that the size of the methods that contain the clone may influence its life expectancy. For example, refactoring smaller methods that contain clones might take less effort and have a reduced risk compared to the large methods that contain clones.
- *Relative Size*: Although we considered the size of each method, similarly-sized methods may contain clones with different size. Moreover, prior work reports that small clones in a method tend to have a lower number of faults per line than large clones (Monden et al. 2002). Therefore, we normalize the size of each method by the size of the cloned code that it contains.
- *Complexity*: Complexity metrics measure the complexity of the method that contains the clone. Prior studies showed that code complexity is one of the most common characteristics of clones (Kim et al. 2004; Kapser and Godfrey 2006; Saini et al. 2016). It is possible that such duplicated complex code will be short-lived as it will be modified afterwards. Therefore, we measure the complexity of the method that contains the clone in order to better understand whether the complexity of methods containing the clone can be linked to the life expectancy of a clone.

**Process Dimension** Process metrics capture the development activity of the method containing a clone before the release of the version in which that clone is introduced. Our intuition is that the development characteristics of the method that contains the clone (e.g., amount of code churn in a method) can influence the life expectancy of a clone. For example, clones in methods that are frequently changed in the past are less likely to live in the system for a long duration.

To extract the development activity of each method that contains a clone, we use *Kenja*[12] which is a maintained version of *Historage* (Hata et al. 2011). Kenja tracks the evolution of software systems at the method level. Similar to the other dimensions, we compute the process metrics at the clone sibling level (i.e., for each code fragment in the clone) and abstract these metrics to the clone level (i.e., a group of duplicate code fragments) using Eqs. 1 and 2. We measure the development characteristics in terms of change, human factor, and quality. Below, we describe each category of process metrics.

- *Change*: Change metrics measure the frequency and amount of churn that are made to the methods that contain the clone. Prior work reports that many of the methods with clones are less maintainable due to the high churn frequency of the methods (Monden et al. 2002). Therefore, we investigate whether the frequency of churn of a method can be linked to the life expectancy of a clone.
- *Human Factor*: Human factor metrics measure the number of developers and the number of major developers (i.e., owners) who modified the method that contains the clone. Zhang et al. (2012) find that developers duplicate code due to the lack of ownership (e.g., a code cannot be changed because it is owned by different developers or teams). Göde (2010) find that the number of developers who are involved in a system shares a relationship with the likelihood that clones will be removed. Therefore, we compute the human factor metrics in order to determine whether a lack of code ownership (e.g., many involved developers or few major developers) has an impact on the life expectancy of a clone.

[12] https://github.com/niyaton/kenja

– *Quality*: We measure the quality of the methods that contain clones by counting the number of commits that fix bugs, add new features, and improve these methods. Baxter et al. (1998) report that new code is more likely to contain clones than old code. Moreover, Monden et al. (2002) report that to reduce the risk of fault injection, developers tend to clone trusted code snippet rather than writing code from scratch. Therefore, we want to determine whether the number of commits for various purposes (i.e., fixing bugs, adding new feature, and improving code) can have an impact on the life expectancy of a clone. We identify the purpose of a commit using a keyword-based approach. A commit where its description contains the "fix", "bug", or "defect" words is identified as bug fixing, while a commit where its description contains the "add" or "new" words is identified as adding a new feature. The remaining commits are identified as improving code. A similar approach was used to identify commits in prior studies (Kim et al. 2008; Hassan 2008; Thongtanunam et al. 2017; Mockus and Votta 2000).

## 2.3 Model Construction

We build a classifier using the random forest technique (Breiman 2001) to determine the life expectancy of a newly-introduced clone and the characteristics that can influence the life expectancy. To help practitioners practically plan their clone management activities, we build a binary classifier to identify whether a newly-introduced clone will be short-lived or long-lived instead of determining the length of the clone lifespan. We use our clone, product and process metrics as inputs for the classifier. Figure 3 provides an overview of the three steps in our model construction approach. We describe each step in our approach below.

### 2.3.1 (MC1) Analyze Correlation

Prior to building a classifier, we check the correlation between the studied variables. Highly-correlated variables may interfere with the model analysis and lead to the reporting of

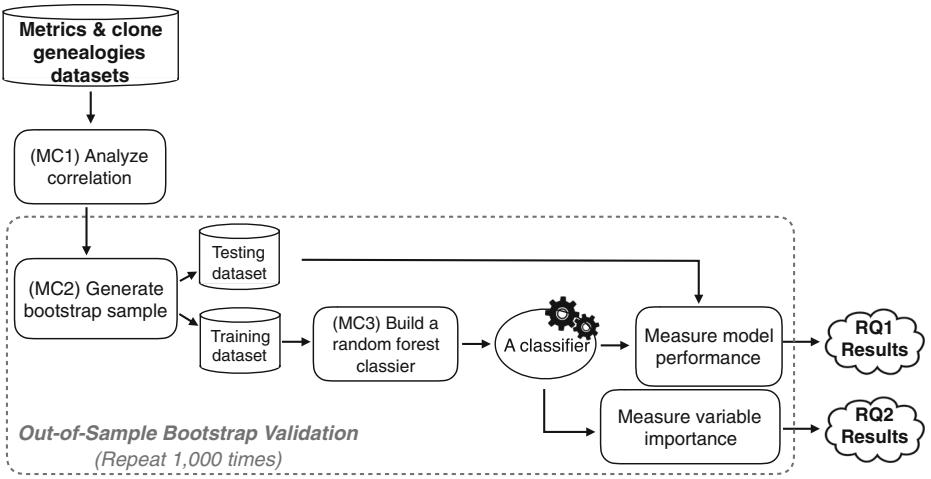


**Fig. 3** An overview of our model construction and analysis approaches

spurious influences for some of the studied characteristics (Jiarpakdee et al. 2018; Nicodemus et al. 2010; Harrell FE 2002; Tantithamthavorn and Hassan 2018). Hence, we compute the pairwise correlation between the input variables using the Spearman rank correlation test ($\rho$). For a pair of variables with high correlation ($|\rho| > 0.7$), we remove one of the two variables when building the classifier. To ease the interpretation, we keep the more straightforward variable among the highly correlated variables. For example, if the correlation between *LineCodeCount* and *CyclomaticModified* is greater than 0.7, we would keep *LineCodeCount* and remove *CyclomaticModified* since *LineCodeCount* is simpler to compute and interpret.

#### 2.3.2 (MC2) Generate Bootstrap Sample

To validate our classifiers, we use the out-of-sample bootstrap validation technique (Efron 1983), which leverages aspects of statistical inference (Efron and Tibshirani 1994). Prior work also shows that the bootstrap technique provides less biased performance estimates than the commonly-used validation techniques, e.g., 10-fold cross validation (Efron 1983; Harrell FE 2002; Tantithamthavorn et al. 2017).

Broadly speaking, the out-of-sample bootstrap technique first generates a bootstrap sample that has the same population size as the original dataset. A bootstrap sample is a dataset sampled with replacement from the original dataset. On average, 36.8% of the data points will not appear in the bootstrap sample, since it is sampled with replacement (Efron 1983). Then, we use the bootstrap sample as our training dataset and the remaining data points that do not appear in the bootstrap sample as our testing dataset. In this work, we do not re-balance nor re-sample the training data since prior studies report that such practices can have an impact on the results of variable importance (Jiarpakdee et al. 2018; Tantithamthavorn et al. 2018). Finally, the out-of-sample bootstrap process is repeated 1,000 times, and the average out-of-sample performance is reported as the performance estimate.

#### 2.3.3 (MC3) Build a Random Forest Classifier

We build a random forest classifier using the set of metrics that remain from our correlation analysis. The random forest technique is known to have a good overall accuracy and to be robust to outliers as well as noisy data. A random forest classifier generates a large number of decision trees with tree voting for the class of the life expectancy (either short-lived or long-lived) for a given clone. Then, the most popular voted class of the life expectancy for a given clone is the outcome of the random forest classifier for that clone (Breiman 2001). In this study, we use the random forest implementation provided by the `randomForest` R package (Breiman and Cutler 2015).

## 3 Preliminary Study on Clone Genealogies

In this section, we perform a preliminary study in order to better understand the clone genealogies in our studied systems. A clone genealogy tracks the life expectancy and the evolution of a clone along with its sibling. To perform our preliminary study, we use the clone genealogies data and clone data that we prepared according to our data preparation process (See Section 2.2). Below, we discuss our study and present empirical observations with respect to two preliminary questions.

**Table 3** An overview of the studied clone genealogies in the studied systems

| System | Total Clones | Clone Type | | | Clone Genealogy Length | | |
|---|---|---|---|---|---|---|---|
| | | Type-1 | Type-2 | Type-3 | Min | Median | Max |
| Ant | 165 | 25% | 14% | 61% | 1 | 1 | 14 |
| Camel | 1,145 | 36% | 24% | 40% | 1 | 3 | 25 |
| Jackrabbit | 647 | 40% | 18% | 42% | 1 | 2 | 56 |
| Maven | 74 | 38% | 24% | 38% | 1 | 5 | 20 |
| Pig | 3,837 | 29% | 24% | 48% | 1 | 5 | 14 |
| Tomcat | 182 | 29% | 19% | 52% | 1 | 9 | 38 |

### 3.1 PQ1: How long do clones live in a software system?

To better understand the life expectancy of clones in our studied systems, we examine the length of clone genealogies. Table 3 presents a statistical summary of the length of the studied clone genealogies. We find that clones lived in the studied systems for a median of one (Ant) to nine (Tomcat) versions. Then, for each studied system, we examine the number of clones that are identified as short-lived and long-lived clones when using the K-means clustering technique where the optimal number of clusters is automatically determined by the NbClust function (Charrad et al. 2014) (see DP3 in Section 2.2). In particular, the NbClust function determines the optimal number of clusters based on six clustering performance indices, i.e., the index of KL (Krzanowski and Lai 1988), CH (Caliński and Harabasz 1974), Cubic Clustering Criterion (CCC) (WS 1983), Scott (Scott and Symons 1971), and Marriott (Marriott 1971). The larger the value of these indices, the better the clustering performance. Table 4 shows the optimal number of clusters that is automatically determined by the NbClust function and the values of these six indices. Figure 4 shows the number of clones in each cluster partitioned by the K-means clustering technique.

We also further examine the evolution of clone genealogies. In particular, we examine the number of siblings of a clone at the time of its introduction. Similar to prior work (Kim et al. 2005), we identify two evolution patterns of clones: (1) add, i.e., the number of clone siblings expand, and (2) subtract, i.e., the number of clone siblings decreases.

**Table 4** Clustering performance of the K-means technique, where the optimal number of clusters are automatically determined by NbClust

| | Optimal | Clustering Performance Index | | | | | |
|---|---|---|---|---|---|---|---|
| | #clusters | KL | CH | Hartigan | CCC | Scott | Marriot |
| Ant | 6 | 3.74 | 1606.43 | -72.08 | 10.99 | 799.30 | 1127.80 |
| Camel | 2 | 0.33 | 2232.45 | 749.02 | 23.24 | 2378.61 | 25479.19 |
| Jackrabbit | 3 | 0.82 | 1726.89 | 212.75 | 55.78 | 6019.94 | 257637.8 |
| Maven | 6 | 146.00 | 743.95 | 25.50 | 11.12 | 411.27 | 1020.37 |
| Pig | 6 | 6.20 | 18012.14 | 539.32 | 66.63 | 19414.53 | 28774.21 |
| Tomcat | 3 | 8.69 | 500.88 | 170.84 | 10.26 | 575.82 | 14282.93 |

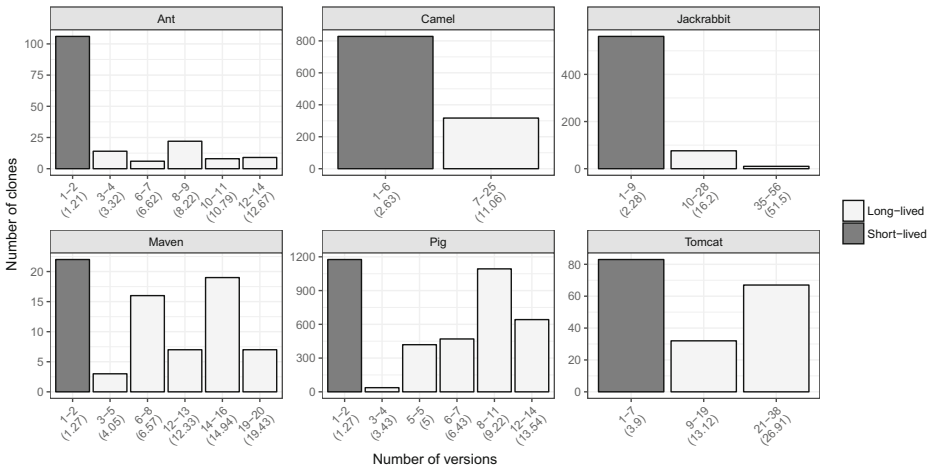


**Fig. 4** The number of clones in each cluster where the x-axis shows a range of clone life expectancy (i.e., the number of versions across which a clone genealogy spans) and a centroid of the cluster shown in parenthesis

**Observation 1: 30-87% of the clones are short-lived clones.** Table 5 shows the results of identifying clone life expectancy when using the K-means clustering technique. The longest genealogy length of the short-lived clones ranges from two (Ant, Maven, and Pig) to nine (Jackrabbit) versions, which account for only 5%($\frac{2}{36}$ in Maven) to 17%($\frac{6}{35}$ in Camel) of the studied versions.

Table 5 shows that 106 out of the 165 clones (64%) in Ant are identified as short-lived clones. We also find that many of the clones in the other four systems are short-lived, i.e., the proportion of short-lived clones ranges from 30% (Maven) to 87% (Jackrabbit). Moreover, Table 5 shows that 84 out of the 165 clones (51%) in Ant lived for one version. We also find a relatively large proportion of clones that lived for one version for Camel (26%), Jackrabbit (36%), Maven (22%), and Pig (23%).

In addition, we further examine the proportion of short-lived clones at the package level for the studied systems. We find that in Camel, 74% of the short-lived clones appear in the

**Table 5** The number of short-lived and long-lived clones that are identified using the K-means clustering technique

| System | Length of the longest genealogy of short-lived clones | #Short-lived clones | #Long-lived clones | #Clones that lived for 1 version |
|---|---|---|---|---|
| Ant | 2 | 106 (64%) | 59 (36%) | 84 (51%) |
| Camel | 6 | 828 (72%) | 317 (28%) | 299 (26%) |
| Jackrabbit | 9 | 561 (87%) | 86 (13%) | 230 (36%) |
| Maven | 2 | 22 (30%) | 52 (70%) | 16 (22%) |
| Pig | 2 | 1,176 (31%) | 2,661 (69%) | 864 (23%) |
| Tomcat | 7 | 83 (46%) | 99 (54%) | 3 (2%) |

Camel component Java packages. We also find that in Jackrabbit, 50% and 43% of the short-lived clones appear in the Jackrabbit Core and API Java packages, respectively. One possible explanation is that these Java packages are evolving as we find that these Java packages have more code changes than other packages. However, for the other studied systems, the short-lived clones tend to be distributed across all Java packages.

We find that 80-93% of the clones are introduced into the source code with one sibling. In particular, 151 out of the 165 clones (92%) in Ant are introduced into the source code with one sibling. We also find a large proportion of clones that are introduced with one sibling for the other studied systems, i.e., 80%($\frac{911}{1145}$) in Camel, 85%($\frac{553}{647}$) in Jackrabbit, 93%($\frac{69}{74}$), and 90%($\frac{163}{182}$) in Tomcat. Furthermore, we observe that there is no association between the length of a clone genealogy and the number of siblings. In other words, the number of clones that are introduced with one sibling account for a similar proportion in each group of life expectancy. For example, in Ant, we find that 90% of short-lived clones and 93% of long-lived clones are introduced with one sibling. We observe a similar proportion of the number of clones that are introduced with one sibling in the other studied systems, i.e. 81%-100% of short-lived clones and 76%-93% of long-lived clones. This result suggests that developers often make a single copy of an original code and introduce it as a clone into the source code.

We also find that few clones have a change (either add or subtract patterns) in the number of sibling across their lifetime. In particular, none of the clones in Maven have a change in the number of sibling across their lifetime. We find a small proportion of clones that have a change in the number of siblings for the other studied systems, i.e., 1% ($\frac{1}{165}$) in Ant, 7% ($\frac{82}{1145}$) in Camel, 2% ($\frac{11}{74}$) in Jackrabbit, and 2% ($\frac{3}{182}$) in Tomcat. Moreover, we observe that most of these clones have only a single change in the number of siblings across their lifetime. For example, 64 out of the 82 clones in Camel and 9 out of the 11 clones in Jackrabbit have only a single change with either an add or subtract pattern. Similarly, the number of siblings changes only once for the one clone in Ant and the three clones in Tomcat that have a change in the number of siblings. These results suggest that once clones are injected into the source code, the clones tend to remain a constant number of siblings throughout their lifetime.

### 3.2 PQ2: How were Short-lived and Long-lived Clones Changed Throughout their Lifetime?

We address this preliminary question in order to better understand the development activity of short-lived and long-lived clones after they are introduced into the systems. Therefore, we determine how often clones change consistently with their siblings. Similar to prior work (Kim et al. 2005), we identify that a clone genealogy includes a consistently changing pattern if the clone genealogy includes at least one "consistent change" pattern, i.e., all clone siblings in the current version have changed consistently from the previous version. We use iClones to identify the consistent change pattern while considering the possibility of late change propagation. We then count the number of clone genealogies that include a consistently changing pattern for short-lived and long-lived clones.

In addition, we observe the change frequency of short-lived clones and long-lived clones during their lifetime. Note that we did not distinguish between consistent and inconsistent changes when we analyzed the difference in change frequency between short-lived and long-lived clones. We count the number of changes that impacted the methods containing a clone and the ones that occurred in the development cycles of the versions in which the clone was alive. We only observe the changes that occurred in the development cycles of

**Table 6** A proportion of clone genealogies that include consistently changing pattern

| | Ant | Camel | Jackrabbit | Maven | Pig | Tomcat |
|---|---|---|---|---|---|---|
| Short-lived clones | 10% | 16% | 15% | 14% | 9% | 19% |
| Long-lived clones | 31% | 37% | 26% | 25% | 25% | 35% |

the versions in which the clone was alive. Since the number of changes can be related with the clone life expectancy (i.e., the longer the lifetime of a clone in the system, the more the changes will occur), we normalize the number of changes by the number of versions in which the clone has lived (i.e., $\frac{\#Changes}{\#Versions}$). To determine the difference in change frequency between short-lived and long-lived clones, we measure the effect size, i.e., the magnitude of the difference using Cliff's $\delta$ (Macbeth et al. 2011). Cliff's $\delta$ is considered as trivial for $|\delta| < 0.147$, small for $0.147 \leq |\delta| < 0.33$, medium for $0.33 \leq |\delta| < 0.474$, and large for $|\delta| \geq 0.474$ (Romano et al. 2006). We also use Mann-Whitney U tests to determine the statistical significance of the difference ($\alpha = 0.05$).

**Observation 2: Consistent changes appear in the genealogies of long-lived clones 1.7 to 3 times more often than the genealogies of short-lived clones.** Table 6 shows that only 9% to 19% of the genealogies of short-lived clones include a consistently changing pattern. On the other hand, 25% to 37% of the genealogies of long-lived clones include a consistently changing pattern. In other words, the proportion of containing consistent changes in short-lived clones is 1.7 ($\frac{26\%}{15\%}$ in Jackrabbit) to 3 ($\frac{31\%}{10\%}$ in Ant) times more than the proportion in long-lived clones. This result indicates that while many of long-lived clones were consistently updated, there is a smaller number of short-lived clones that were consistently updated, suggesting that the associated maintenance effort with short-lived clones is smaller than the associated maintenance effort with long-lived clones.

Figure 5 shows distributions of change frequency in the short-lived and long-lived clones. Table 7 shows that there is a large difference in the change frequency between short-lived and long-lived clones for all of the six studied systems. The Mann-Whitney U tests also

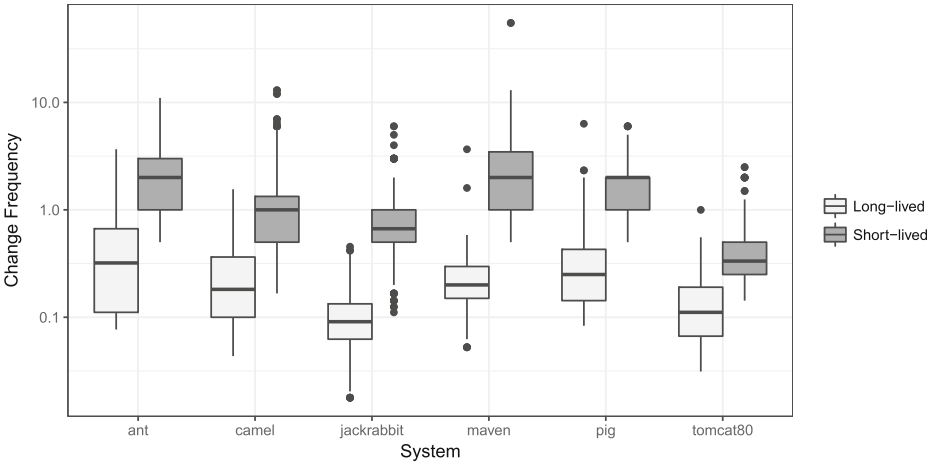


**Fig. 5** The distributions of change frequency in the six studied systems

**Table 7** A statistical summary and the effect size (i.e., the magnitude of the difference) of the change frequency between short-lived and long-lived clones

| System | Change Frequency | | | Stat. Significance |
|---|---|---|---|---|
| | Short-lived (Med.±SD) | Long-lived (Med.±SD) | Effect Size | |
| Ant | 2.00±2.23 | 0.32±0.78 | Large | $***$ |
| Camel | 1.00±1.36 | 0.18±0.21 | Large | $***$ |
| Jackrabbit | 0.67±0.75 | 0.09±0.09 | Large | $***$ |
| Maven | 2.00±18.70 | 0.20±0.57 | Large | $***$ |
| Pig | 2.00±0.94 | 0.25±0.33 | Large | $***$ |
| Tomcat | 0.33±0.59 | 0.11±0.13 | Large | $***$ |

Statistical significance: $\circ p \geq 0.05$, $*p < 0.05$, $**p < 0.01$, $***p < 0.005$, and $****p < 0.001$

confirm that the differences are statistically significant. Moreover, short-lived clones have a larger median of change frequency than long-lived clones do, indicating that short-lived clones were changed more often than long-lived clones did during their lifetime. One possible interpretation is that code fragments in short-lived clones are evolving while code fragments in long-lived clones tend to be stable code. The results of the co-change consistency and change frequency analysis suggest that although short-lived clones were changed more often than long-lived clones, short-lived clones did not require co-change consistency as much as long-lived clones did. These findings also support our intuition that avoiding refactoring of short-lived clones will be beneficial for the clone management efforts.

***Findings:*** *30-87% of the clones are short-lived clones. Moreover, only 9-19% of the short-lived clones were consistently changed, while 25-37% of the long-lived clones were consistently changed. (Observations 1 and 2)*
***Implication:*** *Many clones lived in the studied systems for a short duration. However, the maintenance effort that is associated with such short-lived clones is smaller than the maintenance effort associated with long-lived clones.*

## 4 Case Study Results

In this section, we present the results of our case study with respect to our two research questions. For each research question, we present our (a) motivation, (b) approach, and (c) results, then followed by our general conclusion.

### 4.1 RQ1: How Well can we Determine Whether an Introduced Clone will be Short-lived Versus Long-lived?

#### 4.1.1 (RQ1-a) Motivation

Our preliminary study shows that 30% to 87% of the clones lived for a short duration. Furthermore, we also find that only 9-19% of the short-lived clones were consistently changed throughout their lifetime, suggesting that the maintenance effort that is associated with such

**Table 8** The studied variables that remain after our correlation analysis (MC-1)

| Category | Variable | Ant | Camel | Jackrabbit | Maven | Pig | Tomcat |
|---|---|---|---|---|---|---|---|
| Clone | SiblingCount | • | • | • | • | • | • |
| | Avg_CloneLineCount | • | • | • | • | • | • |
| | Diff_CloneLineCount | • | • | • | • | • | • |
| | Avg_TokenCount | | | | • | | |
| | Diff_TokenCount | • | • | • | • | • | • |
| | Max_DirectoryDistance | • | • | • | • | • | • |
| | Max_EditDistance | • | • | • | | • | |
| | CloneType | • | • | • | • | • | • |
| Size | Diff_LineCount | • | • | • | • | • | • |
| | Avg_LineCodeCount | • | • | • | • | • | • |
| | Diff_LineCodeCount | • | | | | | |
| | Diff_LineCodeDeclCount | • | • | • | • | | • |
| | Avg_LineCodeExeCount | • | | • | | | • |
| | Diff_LineCodeExeCount | | • | | | | |
| Relative Size | Avg_RatioLineCount | • | | • | • | | |
| | Diff_RatioLineCount | | • | | | • | |
| Complexity | Diff_FanOut | • | • | | • | | • |
| | Avg_FanIn | | | | | • | |
| | Diff_FanIn | • | • | • | • | • | |
| | Avg_Cyclomatic | • | • | • | • | • | • |
| | Diff_Cyclomatic | | • | • | • | • | |
| | Avg_CyclomaticModified | | • | • | | • | |
| | Diff_Essential | • | • | • | • | • | |
| | Avg_MaxNesting | • | • | | • | • | • |
| | Diff_MaxNesting | • | | • | | | |
| | Avg_CommentLineCount | • | • | • | • | • | • |
| | Diff_CommentLineCount | • | • | | • | • | |
| | Avg_RatioCommentToCode | | • | | • | • | |
| Change | Avg_Churn | • | | | | | |
| | Diff_Churn | • | • | • | | | |
| | Avg_LineAdded | • | • | • | • | • | • |
| | Diff_LineDeleted | • | • | • | • | • | |
| | Diff_CommitCount | • | • | • | | • | |
| Human factor | Avg_DeveloperCount | | | • | | | |
| | Diff_DeveloperCount | • | • | • | • | | • |
| | Avg_MajorDeveloperCount | | | | | | • |
| Quality | Diff_FixCommitCount | • | • | • | • | • | |
| | Diff_NewFeatureCommitCount | • | • | • | • | • | • |
| | Avg_ImproveCommitCount | | • | • | • | | • |
| | Diff_ImproveCommitCount | | • | • | • | | |

The bullet symbol (•) indicates that the variable is used in the model

short-lived clones is relatively smaller than the effort for long-lived clones. Hence, immediate refactoring of short-lived clones may not be beneficial for the clone management efforts. On the other hand, it is worthwhile to manage long-lived clones by either refactoring them or tracking them using efficient bookkeeping techniques to reduce maintenance efforts. Hence, to help practitioners efficiently manage these clones, we build a classifier to determine the life expectancy of a newly-introduced clone.

#### 4.1.2 (RQ1-b) Approach

We build a random forest classifier to determine whether a newly-introduced clone will be a short-lived or a long-lived clone. Table 5 shows the number of short-lived and long-lived clones. We then build a classifier as described in Section 2.3. Table 8 shows the variables that we use to build our classifiers once we remove the highly-correlated variables.

To answer our RQ1, we measure the Area Under the receiver operating characteristic Curve (AUC). The AUC is commonly used to evaluate the degree of discrimination achieved by the classifier. The AUC captures an area below the curve of the true positive rate (i.e., the proportion of short-lived clones that are correctly classified) against false positive rate (i.e., the proportion of long-lived clones that are misclassified). The value of AUC ranges between 0 (worst) and 1 (best). An AUC greater than 0.5 indicates that our classifiers outperform a random classifier. We compute the AUC using the `auc` function of the `pROC` R package (Robin et al. 2014). As described in our MC2 in Section 2.3, we run 1,000 iterations of the out-of-sample bootstrap technique to validate our results. Hence, for each iteration, we build a random forest classifier using a bootstrap sample. Then, we measure the AUC of the bootstrap classifier. Below, we present and discuss the results of our classifiers.

#### 4.1.3 (RQ1-c) Results

**Observation 3: Our random forest classifiers achieve an average AUC of 0.63 to 0.92.** Table 9 shows that our random forest classifiers achieve an average AUC of 0.65, 0.80, 0.78, 0.73, 0.92, and 0.63 for Ant, Camel, Jackrabbit, Maven, Pig, and Tomcat, respectively. These results indicate that our random forest classifiers can determine the life expectancy of a newly-introduced clone, and perform better than a random guessing. The lower AUC value of the Ant, Maven, and Tomcat classifiers may be in part due to the small number of clones in these systems. In Ant, Maven, and Tomcat, there are 165, 74, and 182 clones that were introduced. On the other hand, there are 1145, 647, and 3837 clones in Camel, Jackrabbit, and Pig, respectively. Table 9 also shows that our random forest classifiers achieve an average true positive rate of 0.31(Maven)-0.98(Jackrabbit). This result indicates that 31% to 98% of the short-lived clones can be identified by our random forest classifiers. On the other hand, the false positive rate of 0.07(Pig)-0.89(Jackrabbit) indicates that 7% to 89% of long-lived clones are misclassified as short-lived clones. The high false positive rate in

**Table 9** An average of AUC, true positive rate, and false positive rate of our six classifiers

| | Ant | Camel | Jackrabbit | Maven | Pig | Tomcat |
|---|---|---|---|---|---|---|
| AUC | 0.65 | 0.80 | 0.78 | 0.73 | 0.92 | 0.63 |
| True positive rate | 0.83 | 0.95 | 0.98 | 0.31 | 0.70 | 0.49 |
| False positive rate | 0.65 | 0.58 | 0.89 | 0.12 | 0.07 | 0.32 |

Jackrabbit may be in part due to the small number of long-lived clones, i.e., there is only 13% ($\frac{86}{647}$) of clones in Jackrabbit are long-lived clones.

Figure 6 shows the distributions of AUC estimates of our random forest classifiers when using the out-of-sample bootstrap validation technique. The narrower the distribution of the boxplot is, the more stable the performance that a classifier determines the life expectancy of a newly-introduced clones on the testing data. The standard deviation of 0.01 to 0.10 for our six classifiers suggests that our classifiers achieve stable performance in determining the life expectancy of a newly-introduced clone.

***Result:*** *Our random forest classifiers achieve an average AUC of 0.63 to 0.92. (Observation 3)*
***Implication:*** *We can determine the life expectancy of a newly-introduced clone using our clone, product, and process metrics.*

## 4.2 RQ2: What are the most influential characteristics for determining the life expectancy of an introduced clone?

### 4.2.1 (RQ2-a) Motivation

The results of our RQ1 show that we can use metrics of the clone, product, and process dimensions to determine the life expectancy of a newly-introduced clone (i.e., short-lived and long-lived). In addition to determining the life expectancy, it would be beneficial for charting clone management plans to examine the influence of the characteristics of a clone (i.e., clone metrics), the methods containing the clone at its introduction (i.e., product metrics), and the development activity of those methods before the clone is introduced (i.e., process metrics) on the life expectancy of a clone. Therefore, in this RQ2, we set out to analyze our random forest classifiers to investigate the characteristics that influence the life expectancy of a newly-introduced clone.

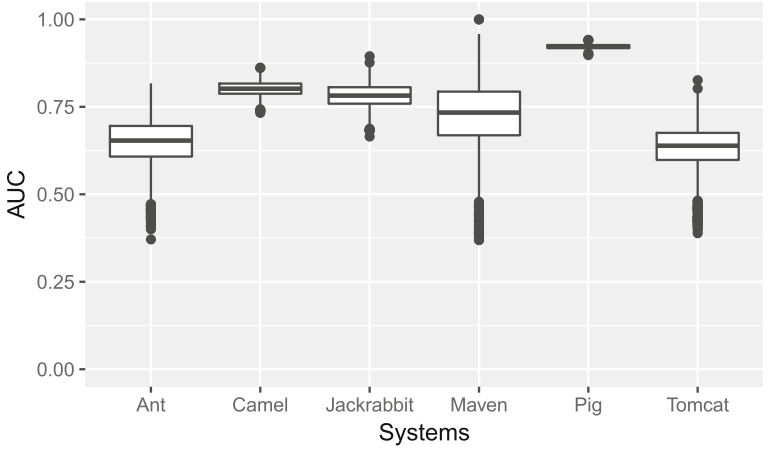


**Fig. 6** AUC estimates of our random forest classifiers when using the out-of-sample bootstrap validation technique

#### 4.2.2 (RQ2-b) Approach

To answer our RQ2, we measure (1) the influence of our metrics and (2) the direction of the relationships between our metrics and the life expectancy of a newly-introduced clone. We describe each measurement below.

**The Influence of Metrics** To evaluate the influence of each metric on our random forest classifiers, we leverage an approach for estimating variable importance in the random classification technique (Breiman 2001; Liaw and Wiener 2002). The approach computes the Mean Decrease Accuracy (MDA) for each metric. Broadly speaking, we evaluate the influence of each metric by permuting the corresponding value of one metric while keeping the values of all other metrics unchanged. Then, we compute an MDA value, i.e., an average decrease in prediction accuracy of the classifier. The larger the MDA of the metric with the permuted value is, the more influential the metric to the classifier is. To measure the influence of metrics, we use the `importance` function of the R `randomForest` package.

In each of the 1,000 iterations of the out-of-sample bootstrap technique, we measure the MDA for each metric in the classifier that was built using the bootstrap sample. To find a statistically distinct rank of our metrics, we perform a Scott-Knott test (Scott and Knott 1974). The Scott-Knott test performs the hierarchical clustering to partition the set of importance means of the metrics into statistically distinct groups. In this study, we use the enhanced Scott-Knott ESD of Tantithamthavorn et al. (2017), i.e., a Scott-Knott test that considers the magnitude of the difference (i.e., effect size) in addition to the statistical significance between groups. To do so, we use the `sk_esd` function of the ScottKnottESD package (Tantithamthavorn 2017).

**The Direction of the Relationships** In addition to quantifying the importance of the metrics, we examine the direction of the relationship between each metric and the likelihood of a newly-introduced clone being short-lived. To do so, we measure the correlation between each metric and the response (i.e., 1 if a newly-introduced clone is short-lived, and 0 otherwise) using a Spearman rank correlation ($\rho$). A positive Spearman rank correlation indicates that the metric shares a positive relationship with the likelihood of a newly-introduced clone being short-lived, whereas a negative correlation indicates an inverse relationship.

#### 4.2.3 (RQ2-c) Results

Table 10 shows the average MDA for the 5 most influential characteristics and the direction (i.e., the sign of $\rho$) of their relationships with the likelihood that a newly-introduced clone being short-lived. We now discuss our results below.

**Observation 4: Change metrics of the process dimension are highly influential in determining the life expectancy of a newly-introduced clone.** Table 10 shows that the *Avg_LineAdded* metric has the largest MDA value in the Jackrabbit, Maven, and Tomcat classifiers. This metric is also in the second rank of the most influential metrics in the Camel and Pig classifiers. Table 10 also shows that the *Avg_LineAdded* metric shares a positive relationship with the likelihood that a newly-introduced clone will be short-lived in Camel, Jackrabbit, Maven, and Tomcat. This result indicates that a newly-introduced clone tends to be short-lived if a large fragment of code that was added into the method containing that clone during the development cycle of the version in

**Table 10** The Mean Decrease Accuracy (MDA) and the Spearman rank correlation ($\rho$) of the 5 most influential metrics

| Rank | Metric (Category) | MDA±SD | $\rho$ | Rank | Metric (Category) | MDA±SD | $\rho$ |
|---|---|---|---|---|---|---|---|
| Ant | | | | Camel | | | |
| 1 | Avg_CommentLineCount (Comp.) | 0.052±0.015 | + | 1 | Avg_LineCodeCount (Size) | 0.062±0.005 | + |
| 2 | Avg_Cyclomatic (Comp.) | 0.046±0.012 | + | | Avg_LineCodeExeCount (Size) | 0.062±0.007 | + |
| 3 | Avg_CloneLineCount (Clone) | 0.041±0.011 | + | 2 | Avg_LineAdded (Change) | 0.051±0.006 | + |
| 4 | Avg_LineCodeExeCount (Size) | 0.031±0.006 | - | 3 | Avg_CloneLineCount (Clone) | 0.049±0.007 | + |
| 5 | Avg_Churn (Change) | 0.030±0.008 | - | 4 | Avg_Cyclomatic (Comp.) | 0.044±0.006 | + |
| | Avg_RatioLineCodeCount (Rel. Size) | 0.030±0.008 | - | | Avg_CyclomaticModified (Comp.) | 0.043±0.006 | + |
| | Diff_Churn (Change) | 0.030±0.008 | - | 5 | Max_EditDistance (Clone) | 0.043±0.006 | + |
| | | | | | Diff_LineDeleted (Change) | 0.042±0.005 | - |
| Jackrabbit | | | | Maven | | | |
| 1 | Avg_LineAdded (Change) | 0.046±0.007 | + | 1 | Avg_LineAdded (Change) | 0.031±0.010 | + |
| 2 | Avg_CloneLineCount (Clone) | 0.029±0.006 | + | | Avg_Cyclomatic (Comp.) | 0.030±0.011 | + |
| | Avg_LineCodeCount (Size) | 0.028±0.004 | - | 2 | Avg_MaxNesting (Comp.) | 0.026±0.013 | + |
| 3 | Avg_LineCodeExeCount (Size) | 0.027±0.004 | - | 3 | Avg_LineCodeCount (Size) | 0.025±0.008 | + |
| 4 | Max_EditDistance (Clone) | 0.026±0.005 | - | | Avg_RatioCommentToCode (Comp.) | 0.024±0.012 | + |
| 5 | Avg_CyclomaticModified (Comp.) | 0.025±0.005 | - | | Diff_LineDeleted (Change) | 0.024±0.010 | + |
| | Avg_Cyclomatic (Comp.) | 0.024±0.004 | - | 4 | Avg_CloneLineCount (Clone) | 0.022±0.009 | + |
| | | | | | Avg_NewFeatureCommitCount (Quality) | 0.022±0.014 | + |
| | | | | | Avg_RatioLineCodeCount (Rel. Size) | 0.020±0.007 | - |
| | | | | | Max_DirectoryDistance (Clone) | 0.020±0.012 | - |
| | | | | 5 | Avg_CommentLineCount (Comp.) | 0.018±0.008 | + |
| | | | | | Diff_LineCount (Size) | 0.018±0.011 | + |
| | | | | | Avg_ImproveCommitCount (Quality) | 0.017±0.007 | + |

**Table 10** (continued)

| Rank | Metric (Category) | MDA±SD | $\rho$ | Rank | Metric (Category) | MDA±SD | $\rho$ |
|---|---|---|---|---|---|---|---|
| Pig | | | | Tomcat | | | |
| 1 | Diff_LineDeleted (Change) | 0.127±0.005 | - | 1 | Avg_Cyclomatic (Comp.) | 0.057±0.011 | + |
| 2 | Avg_LineAdded (Change) | 0.093±0.004 | - | | Avg_LineAdded (Change) | 0.057±0.012 | + |
| 3 | Avg_FanIn (Comp.) | 0.087±0.006 | + | 2 | Avg_LineCodeCount (Size) | 0.051±0.010 | + |
| 4 | Avg_LineCodeCount (Size) | 0.085±0.005 | + | 3 | Avg_CommentLineCount (Comp.) | 0.048±0.011 | + |
| 5 | Avg_Cyclomatic (Comp.) | 0.059±0.004 | - | 4 | Avg_LineCodeExeCount (Size) | 0.042±0.008 | + |
| | | | | | Diff_LineCount (Size) | 0.041±0.010 | + |
| | | | | 5 | Avg_CloneLineCount (Clone) | 0.037±0.008 | + |

A "+" (or "-") sign of $\rho$ indicates a positive (or an inverse) relationship of the metric with the likelihood that an introduced clone being short-lived
The larger MDA that a metric has, the more influential the metric is

which that clone is introduced. For example, we observe that during the development cycle of Jackrabbit v2.2.0, 160 lines of the `testMatchesWildcardAll()` method were added into the `GlobPatternTest.java` file, and 24 lines of these 160 lines are duplicate of each other. Then, during the development cycle of Jackrabbit v2.3.0, the `testMatchesWildcardAll()` method was modified with 71 added lines and 28 deleted lines, and those 24 duplicate lines become dissimilar. Moreover, we observe that in the Jackrabbit dataset, 78% of the methods that were added with more than 37 lines during the version in which the clones are introduced (i.e., the methods of clones with *Avg_LineAdded* above the third quartile of the data) will be modified within the next two versions. On the other hand, only 44% of the methods that were added with less than 14 lines during the version in which the clones are introduced (i.e., the methods of clones with *Avg_LineAdded* below the first quartile of the data) will be modified within the next two versions. These findings suggest that in Camel, Jackrabbit, Maven, and Tomcat, immediate refactoring of a clone may not be necessary if the clone is introduced in methods that were changed with large churn since such methods will be soon modified and the clone is likely to disappear.

We also observe that several change metrics are highly influential metrics in our classifiers. For instance, the *Avg_Churn* metric is one of the 5 most influential metrics in Ant and this metric shares an inverse relationship with the likelihood of an introduced clone being short-lived. The *Diff_LineDeleted* metric also shares an inverse relationship with the likelihood in Pig. These findings also consistent with the results in our preliminary study, i.e., the short-lived clones were changed frequently than long-lived clones did during the their lifetime (see Figure 5), suggesting that code fragments in short-lived clones are still evolving.

**Observation 5: Size and complexity metrics of the product dimension are highly influential in determining the life expectancy of a newly-introduced clone.** Table 10 shows that the *Avg_LineCodeCount* metric has the largest MDA value in the Camel classifier. This *Avg_LineCodeCount* metric is also one of the 5 most influential metrics in the Ant, Jackrabbit, Maven, Pig, and Tomcat classifiers. Table 10 also shows that the *Avg_LineCodeCount* metric shares a positive relationship with the likelihood that a newly-introduced clone will be short-lived in Camel, Maven, Pig, and Tomcat. This result indicates that the larger the methods that contain a clone, the more likely the clone will be short-lived. One possible interpretation for this result is that the source code in those large methods is still evolving. We also find a Spearman rank correlation of 0.31-0.42 between the average size of the methods containing a clone at the version in which the clone is introduced (i.e., *Avg_LineCodeCount*) and the number of commits that are made to those methods during the clone lifetime in Camel, Maven, and Tomcat. This finding suggests that a clone that is introduced in large methods may live in the system for a short duration since those methods are still actively evolving.

In addition to the *Avg_LineCodeCount* metric, Table 10 shows that the *Avg_Cyclomatric* metric has the largest MDA value in the Maven and Tomcat classifiers. The *Avg_Cyclomatric* metric is also one of the 5 most influential metrics in the Ant, Camel, Jackrabbit, and Pig classifiers. Table 10 shows that the *Avg_Cyclomatric* metric shares a positive relationship with the likelihood that a newly-introduced clone will be short-lived in Ant, Camel, Maven, and Tomcat. We also check the correlation between *Avg_LineCodeCount* and *Avg_Cyclomatric* metrics since they often correlate with each other. However, we find that *Avg_Cyclomatric* shares a weak to medium correlation with *Avg_LineCodeCount* (i.e., the $|\rho|$ value ranges from 0.08 to 0.46). One

possible explanation for the high influence of *Avg_Cyclomatric* metric on our random forest classifiers is that developers may start creating a new method by reusing complex source code (Kim et al. 2004; Kapser and Godfrey 2006), then that new method is modified afterward and the source code becomes dissimilar. For example, we observe that during the development cycle of Tomcat v8.0.4, a developer copied the `actionInternal(ActionCode,Object)` method in the `Http11AprProcessor` class and paste it into the `Http11Nio2Processor` class. The average Cyclomatric complexity of the `actionInternal(ActionCode,Object)` method in these two classes (i.e., *Avg_Cyclomatric*) is 57, which is above the third quartile of the data. Then, during the development cycle of Tomcat v8.0.6, the `actionInternal(ActionCode,Object)` method of the `Http11Nio2Processor` class was modified, and the `actionInternal(ActionCode,Object)` method in the `Http11Nio2Processor` become dissimilar from the original method in the `Http11AprProcessor` class. This finding suggests that although developers reuse complex source code by copy-and-paste, the duplicate code fragments are likely to become dissimilar within a short period of time.

**Observation 6: Large clones are more likely to be short-lived than smaller clones.** Table 10 shows that the *Avg_CloneLineCount* metric is one of the 5 most influential metrics for our classifiers and it shares a positive relationship with the likelihood of a newly-introduced clone being short-lived. One possible reason for this result is that a large method was duplicated in order to make the system work with different versions of third-party systems. For example, we observe that during the development cycle of Camel 2.14.0, 473 lines of the `camel-test-spring3` component was duplicated from the `camel-test-spring` component in order to make Camel work with a third-party system named Spring versions 3 and 4, respectively. These duplicate components lived in Camel for six versions. Then, the `camel-test-spring3` component was removed since Spring version 3 is no longer supported. This finding suggests that although a large clone is introduced, it may not require a co-change consistency with its sibling. Then, such a large clone is more likely to disappear within a short period of time.

Although we observe several influential metrics that can be used as an indicator of short-lived clones, these observations may be limited Java systems. Lopes et al. (2017) shows that Java has the smallest amount of code duplication. With other programming languages, managing short-lived clones may be even more beneficial (as there may be more short-lived clones to manage). Lopes et al. (2017) also point out that the characteristics of clones in different language can be different. The software metrics that are associated with the life expectancy of clones may also be different in the other systems. Our exact findings may not be generalized to other systems, however our proposed methodology does generalize.

***Findings:*** *Change metrics (e.g., the number of added lines into the method containing a clone before the release of the version in which the clone is introduced) of the process dimension are highly influential in determining the likelihood of a newly-introduced clone being short-lived. The size and complexity metrics in the product dimension and the size of the clone are influential in determining the life expectancy of a newly-introduced clone. (Observations 4-6)*
***Implication:*** *These highly influential metrics can be used as indicators of the clone life expectancy.*

## 5 Related Work

In order to discuss the related work on code clones, we group them into: (1) understanding code clones and (2) clone management support.

### 5.1 Understanding Code Clones

Several studies investigate the presence of clones in large software systems (Ducasse et al. 1999; Kamiya et al. 2002; Lopes et al. 2017). For example, Lopes et al. (2017) find that Java systems tend to contain a small amount of code duplication while JavaScript systems tend to contain a large amount of code duplication. Koschke and Bazrafshan (2016) report that 80% of the C and C++ projects have at least one Type-2 clone. On the other hand, in this paper, we investigate a different phenomenon of clones, i.e., how long did clones survive in systems and we find that many of the clones (30-80%) are short-lived clones (Observation 1).

Due to a sizable presence of clones in large software systems, several studies investigate how clones are introduced. Kim et al. (2004) find that due to the limitations posed by a programming language, developers often clone code (e.g., lack of multiple inheritance or "enum" construct in Java leads to cloned code). Al-Ekram et al. (2005) report that there is a large number of clones that are introduced accidentally, due to precise protocols that have to be strictly followed for libraries or API interaction. Zhang et al. (2012) show that 33% to 48% of developers clone code due to technical reasons (e.g., following an existing solution, or due to complexity in reusing code), while 24% to 48% of developers clone code due to organizational reasons (e.g., time limitation of code delivery, or code ownership issues). Moreover, they find that 43% of developers introduce clones for learning and experimentation purposes. Such clones are temporarily added into the source code then removed afterwards. Saini et al. (2016) investigate whether cloned methods are different from non-cloned methods in terms of quality (i.e., complexity, modularity, and documentation). They report that 30% of the studied Maven projects contained cloned methods where the values of quality metrics are significantly different from the non-cloned methods. Zibran et al. (2013) also report that majority of the volatile clones in ArgoUML, JabRef, ZABBIX, Conky, and ZedGraph were removed within the initial five to ten releases. Moreover, they find that the size of clones is not associated with the clone removal practice. Saha et al. (2010) examine clone genealogies and volatile clones. They find that on average, 4-75% of the clones in ArgoUML, Linux Kernel, and iTextSharp disappear within five releases. Interestingly, they find that the dead genealogies are removed in a very short period of time, i.e., there exists short-lived clones which we further study in this paper. Yet, the factors that may relate to the phenomenon of short-lived clones have not been investigated in the study of Saha et al. (2010). Complementary to the findings of these prior studies, we find that clones in complex methods and in methods that were changed in releases with large amount of churn are more likely to be short-lived (Observations 4 and 5).

Since a clone is involved with multiple code fragments (i.e., clone siblings), Inconsistent changes may occur when a developer is unaware of all the siblings of a clone. To better understand clones, prior work studies inconsistent changes of clones (i.e., a code change that is applied to a clone but not applied to all its siblings). Barbour et al. (2011) perform an empirical study on the late propagation of changes, i.e., a situation that a clone undergoes inconsistent co-changes then its siblings were resynchronized afterward, in the ArgoUML and Ant open source systems. They find two types of late propagation that are the most prone to faults, i.e., (1) no change propagation and (2) simultaneous propagation, i.e., clone

siblings undergo an inconsistent change followed by a consistent change that modifies both clone siblings. On the other hand, Göde and Koschke (2011) study the evolution of clones in three open source systems. They find that 35% to 63% of clones are never co-changed. In our preliminary study, we find that the consistent change pattern appears in short-lived clones less often than in long-lived clones (Observation 2). Furthermore, Bettenburg et al. (2012) find that only 1% to 3% of the inconsistent changes introduced post-release software defects in Apache Mina, jEdit, and ArgoUML. Xie et al. (2014) examine how changes to clone genealogies can affect the risk of having faults in clones. In particular, they focused at clone migration (i.e., the location of a clone is changed in a version across which the clone genealogy spans) and mutation (i.e., the type of a clone is changed in a version across which the clone genealogy spans). On the other hand, our study examines a different aspect of clone genealogy, i.e., the life expectancy of clones and the factors that may influence their life expectancy.

### 5.2 Clone Management Support

To prevent clones, prior work proposes approaches to identify the introduction of newly-introduced clones into the source code or soon after clones are introduced into the source code. Hou et al. (2009) proposed a proactive copy-and-paste support tool by observing all possible development activities that can lead to a clone throughout its life cycle. Zhang et al. (2013) proposed CCEvents (Code Cloning Events) that provides timely code cloning notifications by continuous monitoring of the code base. However, such a proactive clone management is considered as an ideal way to deal with clones (Roy et al. 2014). On the other hand, researchers have developed tools to detect clones in large software systems. A recent survey by Rattan et al. (2013) shows that over 70 clone detection tools are available for various programming languages. Despite the success of detecting code clones, managing clones in large and evolving software systems remains challenging due to the sizable presence of clones.

In clone management, refactoring all clones can be a task with substantial cost and effort. Several studies have been performed to better understand characteristics of clones. Kim et al. (2005) proposed a clone genealogy extractor to investigate clone evolution structurally and semantically and find that refactoring may not be required for volatile clones (i.e., short-lived clones). Moreover, they find that the proposed techniques in literature for refactoring are ineffective for many long-lived clones where the clone siblings are consistently changing. Prior work identified several cloning patterns that are often used (Kapser and Godfrey 2006; 2008). From their observations, they suggest that refactoring may not be the best solution for all cloning patterns. Consistent to prior work, our preliminary study shows that many of clones lived in the systems for a short duration (Observation 1).

Prior work proposed various approaches to aid in clone management. Göde (2010) performed a case study to understand the characteristics of clones that encourage developers to refactor them out. They find that the number of developers who are involved in a system shares an inverse relationship with the likelihood that clones will be removed. Dang et al. (2012) developed a Microsoft Visual Studio 2012 plug-in to help developers inspect clones within a development environment. Duala-Ekoko and Robillard (2007) proposed CloneTracker for bookkeeping and monitoring clones in evolving software systems. Yun et al. (2014) developed a tool to help developer identify the differences among clone siblings. Wang and Godfrey (2014) propose an approach to recommend clones that should be refactored based on the benefit, cost, and risks of refactoring existing clones. In this work, we

built random forest classifiers which can determine the life expectancy of clones at the time of clone introduction into a software system (Observation 3). Our classifiers can complement the tools of prior work to help practitioners better allocate their resource towards the managing of long-lived clones.

## 6 Threats to Validity

In this section, we discuss the limitations and the threats to the validity of our findings.

### 6.1 Construct Validity

Threats to construct validity describe concerns regarding the validity of our case study design and measurement. In this work, the studied clone genealogies are extracted from iClones. Using different tools or techniques may produce different set of clone genealogies. For example, gCad is one of the tools that can also extract a clone genealogy (Saha et al. 2011). However, the study of Saha et al. (2011) shows that both tools have similar performance. Moreover, prior studies advocate that iClones has a better performance in detecting clones compared to many other available clone detection tools (e.g., Deckard, NiCad) (Svajlenko and Roy 2014). Moreover, the findings of our preliminary study arrive at similar findings as the study of Kim et al. (2005) who use a different clone detection technique. Nevertheless, validating the performance of iClones with other approaches to extract clone genealogies may further strengthen our findings.

Parameter settings may play an important role in detecting clones of iClones. Ragkhitwetsagul et al. (2017) also find that the performance of clone detection tools can vary based on the parameter settings and the data set. Hence, using an optimal parameter settings may allow iClones to detect more clones with less false positive in the studied systems. However, every technique and tool turned out to be extremely sensitive to its own con- figurations consisting of several parameter settings and a similarity threshold. Moreover, for some tools the optimal configurations turned out to be very different to the default configuration, showing one cannot just reuse (default) configurations.

We use the the K-means clustering technique for partitioning our clones based on the number versions in which a clone lives. Although our approach of using the NbClust function and the K-means clustering technique in this study is widely used to determine the most appropriate number of clusters for a dataset of interest (Khanchouch et al. 2015; Wagstaff et al. 2001), a different clustering technique may yield different results. Finally, we wish to reiterate that there are many ways to define the concept of short-lived clones. Our work simply proposes one possible and reasonable definition. Nevertheless, we expect that practitioners would customize this definition based on the peculiarities of their project and system.

The selection of classification techniques may have an impact on our results. Although we use the random forest technique which is known to have a good overall accuracy, different classification techniques may yield a better classification performance. Hence, we check for this threat by building classifiers using the logistic regression and Support Vector Machine (SVM) techniques in addition to the random forest technique. We find that our random forest classifiers achieve a higher AUC value than the logistic regression and SVM classifiers. In particular, the random forest classifiers achieve AUC values that are 12-38% higher than the logistic regression classifiers and that are 2-16% higher than the SVM classifiers.

We assume that a clone and all its siblings lived in the systems for the same duration. However, there are likely cases during the clone lifetime where a clone may change the

number of its siblings. Then, the life expectancy of the siblings of the clone can be miscalculated. To mitigate this concern, we check for the number of siblings of a clone at the time of its introduction and count how many times the number of sibling changes for each clone. We find that 80% (Camel) to 93% (Maven) of the clones are introduced into the source code with one sibling. Moreover, only 0% (Maven) to 7% (Camel) of the clones have a change in the number of sibling across the lifetime of any of their clones. These results suggest that once clones are injected into the source code, the clones tend to maintain a constant number of siblings throughout their lifetime. Therefore, it is unlikely that a clone and its all siblings lived in the systems for different duration.

We extract the released versions that are sequentially developed using the `git log` command. It is possible that the Git history was modified (i.e., the reference of commits were changed) by the `git rebase` command. However, in this work, we did not find any changing commit references in the repositories of our studied systems when we use the `git reflog` command to check for the rewriting of history. Nevertheless, future studies should be aware of such a practice of rewriting history in the VCSs.

### 6.2 Internal Validity

The internal threat to validity is concerned with our ability to draw conclusions based on the relation between the outcome and the set of software metrics that are used as independent variables. Although we use a selected set of metrics from three different dimensions (i.e., clone, product, and process metrics), the addition of other metrics may improve the performance of the classifier. Nonetheless, our random forest classifiers achieve an average AUC 0.63-0.92, which is better than random guessing. Moreover, our metrics (which are derived and motivated from prior studies) help us understand the impact that prior findings can also have on the life expectancy of a clone.

Our random forest classifiers can correctly determine the life expectancy of a clone (i.e., short-lived or long-lived) using a set of software metrics. Yet, the observed relationship in our classifiers does not represent the causal effects of these metrics on the likelihood of a clone to be short-lived or long-lived. The real causes of short-lived clones are hard to determine using an automated clone detection technique. Therefore, future studies are needed to verify the causal effect of our observations.

The life expectancy of a clone can be correlated to many factors other than the metrics that we used. Clones might be short lived due to various reasons such as (1) the clone is refactored, (2) the method containing a clone is removed, and (3) the difference in the code changes applied on the siblings of a clone. We further check whether clones are short-lived due to refactoring or not. We use a tool by Tsantalis et al. (2018) to identify commits that refactor cloned methods during the development cycle of the version during which the clone disappears. We first run the tool on all commits that impact methods containing short-lived clones. Then, we collect the commits that are identified as refactoring changes by the tool. Finally, for these commits, we manually examine each of them to check whether the detected refactoring actually caused any short-lived clones to disappear. We find that 6%, 3%, 18%, 24%, and 7% of short-lived clones disappear due to refactoring in Ant, Camel, Jackrabbit, Maven, Pig, and Tomcat, respectively. A relatively large proportion in Pig is in part due to the refactoring of clones in test files, i.e., one commit made an abstract test class and removed 71 clones in the test files.[13]

[13]The commit hash is `a760df0b20425eef2820b2526baa617c81358ce4`.

Prior work noted the varying life expectancy of a clone (Saha et al. 2010; Zibran et al. 2013). We believe this is an interesting and intricate phenomenon that is worthy of a deeper investigation. However, the notion of life expectancy can be further divided in order to reveal other interesting phenomena around clones. For example, long-lived clones can be classified into: (1) long-lived clones that change and (2) long-lived clones that never change throughout their lifetime. Developers might be interested to refactor or monitor only the long-lived clones that change. In particular, the changing long-lived clones that often co-change with their siblings as such clones are the most costly to maintain.

### 6.3 External Validity

External threats are concerned with our ability to generalize our findings. The influence of the studied metrics on our classifiers may vary across the project characteristics. In this work, we make an observation for the category of software metrics that share the common rank of the variable importance, i.e., the category of software metrics that appears as the 5 most influential metrics. However, some projects may not have the exact same metrics that are highly influential. Nevertheless, we believe that these observations are still of value to inform practitioners what kind of software metrics can be used as an indicator of short-lived clones.

We focus our study on six long-lived and popular open source Java systems. The number of studied systems may limit the generalizability of our results. However, the goal of this work is not to define a wide ranging theory that holds for every system. Instead, the main contribution of our work is to show that in some systems, clones can be short-lived and that we can determine such clones using machine learning classifiers. Our proposed approach can be used and customized for each system based on the needs and expectations of the developers of these systems (for instance, the definition of short-lived will vary between systems and projects). Nevertheless, we provide our replication package to aid future work.[14]

## 7 Conclusion

To manage clones, refactoring is the state-of-the-art method to "fix" clones in order to reduce the maintenance cost of a software system. We briefly highlight the key observations for the studied systems (i.e., Ant, Camel, Jackrabbit, Maven, Pig, and Tomcat). However, refactoring all clones may not be either productive (Kim et al. 2005). Our preliminary study shows that:

- 30% to 87% of clones lived for a short duration when we use the K-mean clustering technique to determine the life expectancy of clones (i.e., short-lived versus long-lived) (Observation 1).
- 25% to 37% of the long-lived clones were consistently changed with their siblings, while there is a smaller proportion of short-lived clones (9% to 19%) that were consistently changed with their siblings (Observation 2).

Instead of managing and monitoring all clones, it will be beneficial for practitioners to determine in advance whether a newly-introduced clone will be short-lived or long-lived to

[14] https://github.com/SAILResearch/clone-life-expectancy

plan the most effective use of the resources. Hence, we build random forest classifiers using a set of selected software metrics in order to determine the life expectancy of a clone at the version when the clone is introduced. Our case study results of the five studied systems show that:

– Our random forest classifiers achieve an average AUC of 0.63 to 0.92, suggesting that our classifiers can be used to determine the life expectancy of a newly-introduced clone. (Observation 3)
– The number of lines that were added into the methods containing clones of the process dimension, the size and complexity of the methods containing clones of the product dimension, and the size of a newly-introduced clone share a positive relationship with the likelihood of a newly-introduced being short-lived. These results suggest that immediate refactoring of a newly-introduced clone with these characteristics may not be necessary (Observations 4-6).

Our study sheds light into the life expectancy of clones. Practitioners can leverage our classifiers to determine whether a newly-introduced clone will be short-lived or long-lived to plan the most effective use of their resources in advance. Moreover, our work provides a good insight of the software metrics that can be used as an indicator of the life expectancy of a clone in a software system. Yet, the goal of this work is not to define a wide ranging theory that holds for every system, every type of clone detection tools, and every definition of "short-lived" clones. The findings may vary as one changes some of the experimental settings (e.g., using different systems or clone detection tools). Instead, our key contribution is to highlight that for some definition of "short-lived" clones, we can flag such clones which are not worthwhile to immediately refactor. We expect the definition of "short-lived" clones to vary from project to project based on the need of the project and expertise of the team. Nevertheless, our approach would be still of value to teams in helping them reduce and prioritize their clone management and maintenance efforts. To facilitate future work, we provide online access to our patch data and example R scripts for model our construction and analysis approaches.[15]



## References

Al-Ekram R, Kapser C, Holt R, Godfrey M (2005) Cloning by accident: an empirical study of source code cloning across software systems. In: Proceedings of the 4th international symposium on empirical software engineering (ISESE), pp 376–385

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

Baker BS (1997) Parameterized duplication in strings: algorithms and an application to software maintenance. Journal of Society for Industrial and Applied Mathematics (SIAM) 26(5):1343–1362

Barbour L, Khomh F, Zou Y (2011) Late propagation in software clones. In: Proceedings of the 27th international conference on software maintenance (ICSM), pp 273–282

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: Proceedings of the 14th international conference on software maintenance (ICSM), pp 368–377

[15] ln:appendix

Bettenburg N, Shang W, Ibrahim W, Adams B, Zou Y, Hassan AE (2009) An empirical study on inconsistent changes to code clones at release level. In: Proceedings of the 16th working conference on reverse engineering (WCRE), pp 85–94

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. J Sci Comput Program 77(6):760–776

Breiman L (2001) Random forests. Mach Learn 45(1):5–32

Breiman L, Cutler A (2015) Breiman and cutler's random forests for classification and regression. https://www.stat.berkeley.edu/~breiman/RandomForests/

Caliński T, Harabasz J (1974) A dendrite method for cluster analysis. Commun Stat 3(1):1–27

Charrad M, Ghazzali N, Boiteau V, Niknafs A (2014) Nbclust : An R package for determining the relevant number of clusters in a data set. J Stat Softw 61(6):1–36

Cordy JR (2003) Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In: Proceedings of the 11th international workshop on program comprehension, pp 196–205

Dang Y, Zhang D, Ge S, Chu C, Qiu Y, Xie T (2012) XIAO: tuning code clones at hands of engineers in practice. In: Proceedings of the 28th annual computer security applications conference (ACSAC), pp 369–378

Duala-Ekoko E, Robillard MP (2007) Tracking code clones in evolving software. In: Proceedings of the 29th international conference on software engineering (ICSE), pp 158–167

Ducasse S, Rieger M, Demeyer S (1999) A language independent approach for detecting duplicated code. In: Proceedings of the 15th international conference on software maintenance (ICSM), pp 109–118

Efron B (1983) Estimating the error rate of a prediction rule: Improvement on cross-validation. J Am Stat Assoc 78(382):316–331

Efron B, Tibshirani RJ (1994) An introduction to the bootstrap. CRC Press, Boca Raton

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

Göde N (2010) Clone removal: fact or fiction? In: Proceedings of the 4th international workshop on software clones (IWSC), pp 33–40

Göde N (2011) Clone evolution. PhD Thesis, The Universitat Bremen, Bremen

Göde N, Koschke R (2009) Incremental clone detection. In: Proceedings of the 13th conference on software maintenance and reengineering (CSMR), pp 219–228

Göde N, Koschke R (2011) Frequency and risks of changes to clones. In: Proceeding of the 33rd international conference on software engineering (ICSE), pp 311–320

Harrell FE Jr (2002) Regression modeling strategies: with application to linear models, logistic regression, and survival analysis, 1st edn. Springer, New York

Hassan AE (2008) Automated classification of change messages in open source projects. In: Proceedings of the 23rd symposium on applied computing (SAC), pp 837–841

Hata H, Mizuno O, Kikuno T (2011) Historage: fine-grained version control system for java. In: Proceedings of the 12th international workshop principles on software evolution and the 7th annual ERCIM workshop on software evolution (IWPSE-EVOL), pp 96–100

Hou D, Jablonski P, Jacob F (2009) CnP: towards an environment for the proactive management of copy-and-paste programming. In: Proceedings of the 17th international conference on program comprehension (ICPC), pp 238–242

Jiarpakdee J, Tantithamthavorn C, Hassan AE (2018) The impact of correlated metrics on defect models. arXiv:1801.10271

Johnson JH (1993) Identifying redundancy in source code using fingerprints. In: Proceedings of the conference of the centre for Advanced studies on collaborative research (CASCON), pp 171–183

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

Kapser C, Godfrey MW (2006) "Cloning considered harmful" considered harmful. In: Proceedings of the 13th working conference on reverse engineering (WCRE), pp 19–28

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

Khanchouch I, Charrad M, Limam M (2015) An improved multi-SOM algorithm for determining the optimal number of clusters. J Future Comput Commun 4(3):198–202

Kim M, Bergman L, Lau T, Notkin D (2004) An ethnographic study of copy and paste programming practices in OOPL. In: Proceedings of the international symposium of empirical software engineering (ISESE), pp 83–92

Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. In: Proceedings of the 10th joint meeting of the European software engineering conference and the international symposium on the foundations of software engineering (ESEC/FSE), pp 187–196

Kim M, Zimmermann T, Nagappan N (2012) A field study of refactoring challenges and benefit. In: Proceedings of the 20th international symposium on the foundations of software engineering (FSE), p Article No. 50

Kim S, Whitehead J Jr, Zhang Y (2008) Classifying software changes: clean or buggy? Trans Softw Eng (TSE) 34(2):181–196

Koschke R, Bazrafshan S (2016) Software-clone rates in open-source programs written in C or C++. In: Proceedings of the 23rd international conference on software analysis, evolution, and reengineering (SANER), pp 1–7

Krzanowski WJ, Lai YT (1988) A criterion for determining the number of groups in a data set using sum-of-squares clustering. J Biometrics 44(1):23–34

Lague B, Proulx D, Merlo EM, Mayrand J, Hudepohl J (1997) Assessing the benefits of incorporating function clone detection in a development process. In: Proceedings international conference on software maintenance (ICSM), pp 314–321

Li Z, Lu S, Myagmar S, Zhou Y (2006) CP- miner: finding copy-paste and related bugs in large-scale software code. Trans Softw Eng 32(3):176–192

Liaw A, Wiener M (2002) Classification and regression by randomForest. R News 2(3):18–22

Lopes CV, Maj P, Martins P, Saini V, Yang D, Zitny J, Sajnani H, Vitek J (2017) DéjàVu: A map of code duplicates on GitHub. In: Proceedings of the object-oriented programming, systems, languages & applications (OOPSLA), pp 1–28

Macbeth G, Razumiejczyk E, Ledesma RD (2011) Cliff's delta calculator: a non-parametric effect size program for two groups of observations. Universitas Psychologica 10:545–555

Marriott FHC (1971) Practical problems in a method of cluster analysis. Biometrics 27(3):501–514

Mockus A, Votta LG (2000) Identifying reasons for software changes using historic databases. In: Proceedings of the 16th international conference on software maintainance (ICSM), pp 120–130

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

Nicodemus KK, Malley JD, Strobl C, Ziegler A (2010) The behaviour of random forest permutation-based variable importance measures under predictor correlation. BMC Bioinforma 11(1):110–124

Ragkhitwetsagul C, Krinke J, Clark D (2017) A comparison of code similarity analysers. J Empir Softw Eng (EMSE) 23(4):2464–2519

Rattan D, Bhatia R, Singh M (2013) Software clone detection: a systematic review, vol 55

Robin X, Turck N, Hainard A, Tiberti N, Lisacek F, Sanchez JC, Müller M, Siegert S (2014) Display and Analyze ROC Curves

Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Appropriate statistics for ordinal level data: should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys? In: The annual meeting of the Florida association of institutional research (FAIR), pp 1–33

Roy CK, Zibran MF, Koschke R (2014) The vision of software clone management: past, present, and future. In: Proceedings of the joint European conference on software maintenance and reengineering and the working conference on reverse engineering (CSMR-WCRE), pp 18–33

Saha RK, Asaduzzaman M, Zibran MF, Roy CK, Schneider KA (2010) Evaluating code clone genealogies at release level: an empirical study. In: Proceedings of the 10th source code analysis and manipulation (SCAM), pp 87–96

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 (ICSM), pp 293–302

Saini V, Sajnani H, Lopes C (2016) Comparing quality metrics for cloned and non cloned java methods : A large scale empirical study. In: Proceedings of the international conference on software maintenance and evolution (ICSME), pp 256–266

Scott AJ, Knott M (1974) A cluster analysis method for grouping means in the analysis of variance. Biometrics 30(3):507–512

Scott AJ, Symons MJ (1971) Clustering methods based on likelihood ratio criteria. Biometrics 27(2):387–397

Silva D, Tsantalis N, Valente MT (2016) Why we refactor? Confessions of github contributors. In: Proceedings of the 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 858–870

Svajlenko J, Roy CK (2014) Evaluating modern clone detection tools. In: Proceedings of the 30th international conference on software maintenance and evolution (ICSME), pp 321–330

Tantithamthavorn C (2017) The Scott-Knott Effect Size Difference (ESD) Test version 2.0.2. https://cran.r-project.org/web/packages/ScottKnottESD/ScottKnottESD.pdf

Tantithamthavorn C, Hassan AE (2018) An experience report on defect modelling in practice: Pitfalls and challenges. In: Proceedings of the international conference on software engineering: software engineering in practice track (ICSE-SEIP'18), p To Appear

Tantithamthavorn C, McIntosh S, Hassan AE, Ki Matsumoto (2017) An empirical comparison of model validation techniques for defect prediction models. Trans Softw Eng (TSE) 43(1):1–18

Tantithamthavorn C, Hassan AE, Matsumoto K (2018) The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. arXiv:1801.10269

Thongtanunam P, McIntosh S, Hassan AE, Iida H (2017) Review participation in modern code review. Empir Softw Eng (EMSE) 22(2):768–817

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

Tsantalis N, Mansouri M, M-Eshkevari L, Mazinanian D, Dig D (2018) Accurate and efficient refactoring detection in commit history. In: Proceedings of the 40th international conference on software engineering (ICSE), p to appear

Wagstaff K, Cardie C, Rogers S, Schroedl S (2001) Constrained K-means clustering with background knowledge. In: Proceedings of the 8th international conference on machine learning, pp 577–584

Wang W, Godfrey MW (2014) Recommending clones for refactoring using design, context, and history. In: Proceedings of the 30th international conference on software maintenance and evolution (ICSME), pp 331–340

WS S (1983) SAS technical report A-108, cubic clustering criterion. Tech. Rep. SAS Institute Inc, Cary

Xie S, Khomh F, Zou Y, Keivanloo I (2014) An empirical study on the fault-proneness of clone migration in clone genealogies. In: Proceedings of the international conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE), pp 94–103

Yun Lin, Xing Z, Xue Y, Liu Y, Peng X, Sun J, Zhao W (2014) Detecting differences across multiple instances of code clones. In: Proceedings of the 36th international conference on software engineering (ICSE), pp 164–174

Zhang G, Peng X, Xing Z, Zhao W (2012) Cloning practices: why developers clone and what can be changed. In: Proceedings of the 28th international conference on software maintenance (ICSM), pp 285–294

Zhang G, Peng X, Xing Z, Jiang S, Wang H, Zhao W (2013) Towards contextual and on-demand code clone management by continuous monitoring. In: Proceedings of the 28th international conference on automated software engineering (ASE), pp 497–507

Zibran MF, Saha RK, Roy CK, Schneider KA (2013) Genealogical insights into the facts and fictions of clone removal. SIGAPP Applied Computing Review 13(4):30–42

**Patanamon Thongtanunam** is a lecturer at the School of Computing and Information System, the University of Melbourne, Australia. Prior to that, she was a lecturer at the School of Computer Science, the University of Adelaide, a research fellow of Japan Society for the Promotion of Science (JSPS). She received PhD in Information Science from Nara Institute of Science and Technology, Japan. Her research interests include empirical software engineering, mining software repositories, software quality, and human aspect. Her research has been published at top-tier software engineering venues like International Conference on Software Engineering (ICSE) and Journal of Empirical Software Engineering (EMSE). More about Patanamon and her work is available online at http://patanamon.com.

**Weiyi Shang** is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queen's University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve the quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang.

**Ahmed E. Hassan** is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queens University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo, Canada. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. More about Ahmed and his work is available online at http://sail.cs.queensu.ca/.

## Affiliations

**Patanamon Thongtanunam[1] · Weiyi Shang[2] · Ahmed E. Hassan[3]**

Weiyi Shang
shang@encs.concordia.ca

Ahmed E. Hassan
ahmed@cs.queensu.ca

1 School of Computing and Information System, The University of Melbourne, Melbourne, Australia

2 Department of Computer Science and Software Engineering, Concordia University, Montréal, Canada

3 Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Canada