Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2015, Article ID 829389, 18 pages
http://dx.doi.org/10.1155/2015/829389

Research Article

Classifying Obstructive and Nonobstructive Code Clones of Type I Using Simplified Classification Scheme: A Case Study

Miroslaw Staron,[1] Wilhelm Meding,[2] Peter Eriksson,[3] Jimmy Nilsson,[3] Nils Lövgren,[3] and Per Österström[4]

[1]*Computer Science and Engineering, University of Gothenburg, 412 96 Gothenburg, Sweden*
[2]*Ericsson SW Research, Ericsson AB, 412 96 Gothenburg, Sweden*
[3]*Ericsson AB, 412 96 Gothenburg, Sweden*
[4]*Business Region Göteborg AB, 412 96 Gothenburg, Sweden*

Correspondence should be addressed to Miroslaw Staron; miroslaw.staron@gu.se

Received 4 September 2015; Revised 13 November 2015; Accepted 18 November 2015

Academic Editor: Nicholas A. Kraft

Code cloning is a part of many commercial and open source development products. Multiple methods for detecting code clones have been developed and finding the clones is often used in modern quality assurance tools in industry. There is no consensus whether the detected clones are negative for the product and therefore the detected clones are often left unmanaged in the product code base. In this paper we investigate how obstructive code clones of Type I (duplicated exact code fragments) are in large software systems from the perspective of the quality of the product after the release. We conduct a case study at Ericsson and three of its large products, which handle mobile data traffic. We show how to use automated analogy-based classification to decrease the classification effort required to determine whether a clone pair should be refactored or remain untouched. The automated method allows classifying 96% of Type I clones (both algorithms and data declarations) leaving the remaining 4% for the manual classification. The results show that cloning is common in the studied commercial software, but that only 1% of these clones are potentially obstructive and can jeopardize the quality of the product if left unmanaged.

1. Introduction

Given their complexity and required quality, most domain specific software products in the telecom industry represent a major long-term investment for these organizations. Organizations tend to ensure the longevity of these products by maintaining and evolving these products by adapting them to constantly changing functional and nonfunctional requirements. Code clones are one of the aspects which has a potential of deteriorating the quality and introducing inconsistencies into software designs and products [1]. They can cause inconsistent evolution of source code and thus increased defect proneness.

The phenomenon of code clones has been studied from multiple perspectives and a number of methods and tools for detecting code clones have been developed, for example, based on pattern matching, syntax parsing, or tree parsing [2]. However, the majority of research published about clones is based on open source software [3] where the quality assurance is significantly different than in the proprietary software [4]. In this paper we investigate software products that are ubiquitous in our society and are part of telecommunication networks. Compared to open source products studied in previous works we had the unique opportunity of presenting the results to the designers working with the development of these products, obtaining feedback on the reasons why cloning "happens" in the development, and assessing how obstructive each clone is for the quality of the product.

In this study we worked with large software development programs at Ericsson (a few hundred designers) who develop large telecom products. The organizations work in a distributed environment with empowered teams spread over a number of continents and continuously delivering code increments to common branch for each product. Since one

of the products also moved from one-site development to multiple-site development we had the opportunity to observe if this kind of distribution of development influences code cloning. In this paper we focus on Type I code clones [5] (exact code duplicates) as in our exploration this type of clones was the most fundamental one and allowed us to use the same methods for different programming languages (C/C++/Erlang) and platforms (Unix/Linux/Windows/Real-time OSs). We study how common the clones are and how obstructive this type of cloning is in commercial systems. We set off to investigate whether cloning takes place in the industrial systems; given the large body of research about cloning in open source software we wanted to explore cloning in commercial software. The initial results showed that cloning is common and that practitioners need support in distinguishing which clones can be problematic.

Therefore, in this paper we address the following research question:

> *RQ 1: Given the number of clones in large software systems, how to efficiently identify those which are potentially obstructive for the quality of the product?*

In order to address this research question we designed a study of large software systems at Ericsson (previously studied in [6–9]). We had this unique opportunity to work with a company with mature software development where monitoring of code quality is done continuously. The products studied in this paper are large (over 9 MLOC), developed in a distributed environment (multiple continents) and with the empowered development teams in a combination of Agile and Lean software development. In such a distributed environment, the management of the clones has become increasingly important as the code can be developed continuously and multiple teams can be working in parallel on similar functionality (a practice prone to cloning).

We were able to use methods for clone detection and triggered improvement initiatives based on the clones found during the study. We used a simple text recognition method [10] to identify clones and we used interviews with designers to assess how potentially obstructive the clones are. In this study we define *obstructive clones* as those duplicate code fragments which contain code with algorithms (or part of algorithms) which are critical for seamless operation of the product in the field. Examples of such clones are code fragments for parsing packet data units, complex algorithm implementations, or code with multiple pointers (C/C++).

In order to address the research question RQ 1 we investigated two subquestions:

> *RQ 1a: Which clones found in commercial products are obstructive?*
>
> *RQ 1b: How to efficiently distinguish between the obstructive and nonobstructive clones?*

RQ 1a was motivated by the existing body of research which does not provide a definitive answer whether code clones are negative, positive, obstructive, or not relevant. Thus leading to an assumption that the significance depends on the clone. We set to find an underlying model which would explain what kind of characteristics a clone should possess to be obstructive for the quality of the product—a clone classification scheme.

RQ 1b was motivated by the large amount of cloned code reported in literature and found in the studied systems. Designers and architects need an immediate feedback on whether they should react upon a discovered clone or not. In order to address this research question we needed to find a new method and tool for classifying clones according to how obstructive they are.

The results show that large, professionally developed software products can contain many clones, but over 90% of the clones are not obstructive. The nonobstructive clones are usually found in locations where they can be expected; for example, set-up of environment for test cases or target platform code.

What we have also found during the course of this study is that we can use analogy-based classification to almost automatically classify clones according to how obstructive they are. By using these methods we were able to reduce the need of manual classification to 4% of all clones. Using this method makes it possible to use clone detection as immediate feedback to designers and reduces the costs of detecting and filtering only the relevant clones.

The rest of the paper is structured as follows. Section 2 presents the most important related work. Section 3 presents the concept of code clones and the classification. Section 4 presents the design of the study and Section 5 presents the results. Section 6 contains the discussion of our results in the light of the existing research and Section 7 presents the conclusions.

2. Related Work

2.1. Reviews of Cloning Evidence. One of the most comprehensive works on code clones is the recent systematic review by Rattan et al. [3]. Rattan et al. investigated 213 papers on clones and were able to summarize the types of clones, clone detection methods, and types of systems studied. They were also able to summarize 12 papers where the harmfulness of clones was discussed. This systematic review was an input to our study and triggered the interviews on the reasons for cloning and the development of a classification scheme for clones. Despite the thorough work done by Rattan et al., only 7 commercial systems were found to be studied in the 213 reported investigations of cloning. Our intention with this paper is to contribute with insights on the significance of clones for the product quality, the evolution of clones over time, and the insights from designers working with this system on why the clones exist in the studied software in the first place.

Pate et al. [11] presented another review (preceding the one by Rattan et al.) where one of the research questions was concerned with the consistency of clone changes. They have found that there is evidence in literature that the clones change inconsistently over time. This formed an important input to our analysis of clones in new and old code bases. Their study of 30 papers, however, was not as extensive as the study of Rattan et al.

2.2. Cloning and Software Quality. Juergens et al. [12] investigated the consistency of cloning and concluded that inconsistent clones are the most serious type as they usually jeopardize software quality. They have also found that the location of the clones is a determinant of the seriousness of the clone. Similar study was also presented by Toomim et al. [13]. The location of the clones was also found important in classifying clones in our study.

Krinke [14] has studied 200 weeks of evolution of five open source projects from the perspective of stability of source code. The conclusion was that in these systems the cloned code was on average more stable than the noncloned code. This study has shown that the number of changes is larger in the noncloned code compared to the cloned code. In our study we wanted to expand on this and explore the number of clones and their size over a number of releases of commercial software systems.

Monden et al. [15] studied the relationship between cloning and reliability of software systems. Their results showed that the presence of clones had a positive influence on reliability. Their analysis suggested that the number of clones (coverage) decreases over time—the older the module, the lower the number of clones. Our analysis shows that it can indeed be the case that clones could be positive. The number of clones usually increases unless directed clone-removal activities are done and the clones usually are not obstructive. Their analysis does not take into consideration how obstructive the clones are.

Kapser [16] has conducted an extensive study on cloning and concluded that not all clones can be considered negative for software quality. Kapser has found that clones can even have a positive effect on the maintainability of software systems. The study in [16] was, nevertheless, conducted on open source projects. In their further work, Kapser and Godfrey [17] have found that by adding semantic to the clones found in the software one can filter out as many as 65% of the clones as false-positives. Our results show that by classifying the clones with the experts can filter out as much as 96% of clones as nonobstructive. Thus we see our work as an extension of the work of Kapser and Godfrey.

Selim et al. [18] have conducted a study of open source projects and concluded that the effect of cloning on defects depends on the software product. Their recommendation is to focus on products with long history to obtain better statistical power, an approach used in our case study of commercial product with a number of years of presence on the market.

2.3. Detection and Classification of Clones. Kapser and Godfrey [19] recognized the fact that not all clones are serious and introduced a template for the patterns of cloning (i.e., how designers introduce clones to their products). They have also identified variations (platform, hardware, and experimental), templating (boiler-plating, language idioms, and API), and customization (bug workarounds, replicating/specializing) as different cloning patterns. However, their work was not linked to empirical investigation of the significance of clones in industrial systems. Thus our work complements the work of Kapser and Godfrey [19].

Kim et al. [20] studied how designers work with cloning by observing software designers and analyzing log files. They have used augmented Eclipse environment to log and record activities of designers and followed up on these activities in interviews, all in order to understand the reasoning behind cloning from designer's side. The reasons for cloning vary from reordering/reorganizing of code segments to reuse complex control structures in the code. They also identify underlying reasons that might cause the designers to copy-paste, for example, limitations in the expressiveness of the programming language. We used this work when designing the analogy-based classification algorithms and to decide whether a particular code clone can potentially have a negative impact on software quality or not.

Kodhai et al. [5] present a method for detecting clones based on the similarity of code patterns. They convert the source code into a pattern (declaration of variables, loops, etc.) and through that find Type I and Type II clones. Their approach can be used instead of simple textual analyses presented in our work in order to identify more clones. However, as the goal of our study was to understand the cloning practices, our industrial partners found the simple textual analyses to be sufficient as they allowed the practitioners to directly see the duplication of code without contextualizing it (e.g., by changing variables names).

There exist numerous tools for identifying and managing code clones, for example, CCFinder [21], Gemini [22], or Clone tracker [23], to name a few. Our work is intended to support clone tool research in providing empirical evidence on how to classify clones in order to highlight which ones can be prioritized for removal.

In practice there exist a large amount of work on clone detection in the mainstream programming languages. However, the evidence of cloning in domain specific programming languages like Erlang is scarce. Li and Thompson [24–26] provided evidence and tool support for removing cloning in Erlang programs. Haskell, which is a programming language close to Erlang, suffers from similar problems as described by Brown and Thompson [27].

An interesting aspect of cloning can be found in the works of LaToza et al. [28] who have found that cloning can be a nuisance for the programmers/maintainers as they may be forced to make the same change in several places of the code.

Thummalapenta et al. [29] developed clone evolution classification framework which can be used in connection with our classification scheme. Combining the complementary schemes can provide organizations with the possibility to track the obstructive clones over time in case it is not possible to remove them directly.

2.4. Cloning in Industrial Applications. Yamanaka et al. [30] studied the introduction of the clone management system at NEC corporation and showed that cloning tend to decrease over time. In our work we learn from their experiences in introducing the clone monitoring methods and tools at Ericsson in order to decrease or eliminate completely the clones which are identified to be obstructive.

Dang et al. [31] provided the evidence of cloning practices at Microsoft by studying how their tool, XIAO, is used at

the company. Their results show a viral effect of spreading of the use of the tool, indicating that the issue of code cloning is an important one for companies. Our results support the claim that this is important for software development companies working with large-scale software products. Similar claims are made by Hummel et al. [32].

Our work complements also the work on distance-based clone classification by Kapser and Godfrey [33]. Kapser and Godfrey found that distance-based clustering of clones can help in prioritizing maintenance activities. Our work could support further prioritization so that the architects and designers can focus only on the top 10 most problematic cloned code fragments in the multimillion lines-of-code product. We believe that our work can also help to increase the efficiency in creating vector-spaces of similar clones as advocated by Grant and Cordy [34] by providing additional dimension of semantic information about the clone.

An alternative to classifying clones is to measure their properties, which is advocated by Higo et al. [35]. Their work proposes and evaluates a number of useful metrics like distance in class hierarchy or coupling to the surrounding code. Our work can be seen as an alternative method to be used when computing the metrics is not possible, for example, in the case of multiple programming languages in the same system (like in the three studied products).

Aversano et al. [36] studied how code clones are maintained in practice and found that the majority of clones are updated consistently. However, they also describe the dangers of uncontrolled spread of cloning and lack of management of clones. The classification scheme presented in our work can provide a possibility to direct management activities towards the obstructive clones or differentiate between management strategies between obstructive and nonobstructive clones.

3. Code Clones

In this section we present the classification which we use in this paper and outline several of the methods used in the detection of clones. We start by introducing the types of clones recognized today and explaining the reason to choose one of the types for our study.

3.1. Types of Clones. We use the basic classification of clones into four types. The first three are based on the work of Kapser [16]:

> Type I: segments of code which are lexically identical.
>
> Type II: segments of code which are lexically identical if one ignores identifier names.
>
> Type III: adjacent segments of clones of Type I or II which are separated by nonisomorphic lines (so-called "gapped" clones).

Another classification of clones is presented by Roy and Cordy [2] which recognizes even a fourth type of clone based on the functionality of the code:

> Type IV: two or more parts of the code which implement the same computation but are implemented by different syntactic variants.

These different types of clones require different detection methods since their nature is quite different, from a simple "duplicate" of code to a reimplementation of the same algorithm in a different way. In this paper, however, we start with Type I clones in order to explore this cloning patterns in industrial, commercial products and in further work we intend to expand to other types of clones.

The reason for focusing on Type I clones is quite pragmatic, the ability to use tools working on different platforms (Unix, Windows, and Linux), the ability to quickly adapt the tools for different programming languages (C/C++/Erlang), and the ability to quickly start with the clone detection. The initial goal of this study was to explore how much cloning happens in commercial software systems and therefore starting with Type I clones was considered by the research team (researchers and practitioners in this study) to be a good logical first step. As our results later in the paper show, even this simple clone detection of Type I yielded interesting results.

3.2. Detection Methods. Based on the recent systematic review by Rattan et al. [3] we can summarize relevant clone detection methods relevant for our work (the original review lists 13 different methods):

(i) Source code/text: using text processing methods like string comparison, block sizes, or string comparison with distance metrics. The method can detect Type I clones and Type III clones combining Type I clones.

(ii) Regularized tokens: using statements, sets of statements, and functions. The method can detect Type I, Type II, and Type III clones.

(iii) AST/parse tree: using the abstract syntax tree to find equivalent code fragments. The method can detect Type I, Type II, and Type III clones.

In the source code/text methods we use the simple methods of string comparisons to detect clones, for example, fingerprinting (hash code) or distance metrics (Levenshtein distance, [37]) combined with the size of blocks. These detection parameters determine which clones are detected and even which kinds of clones are detected.

3.3. Obstructive and Nonobstructive Clones. We developed a definition of obstructive clones and found examples of these in the studied products, in order to address the research question *RQ 1a: Which clones found in commercial products are obstructive?* In this paper we define *obstructive clones* as those duplicate code fragments which contain code with algorithms (or part of algorithms) which are critical for seamless operation of the product in the field. Examples of such clones are

(i) code for parsing data packets,

(ii) code with multiple pointers and references (e.g., function calls) to other code fragments,

(iii) code for defining complex data types reused in multiple code modules,

```
...
    *p++ = iInt;
...
    *p++ = 0;
    Int nLength = sizeof(parameter1) + pLen;
    memcpy(p,&nLength, sizeof(nLength));
    p += sizeof(nLength)
    if (pVar)
    {
        memcpy(p, pVar, pLen);
        p += pLen;
    }
    return p;
...
```

Algorithm 1: Example clone. Comments and explanations removed for confidentiality reasons. Variable names were changed.

(iv) locally modified global declarations (used instead of polymorphism).

These clones obstruct the operation of the product long after the release, especially if they have evolved in an uncontrolled manner. For example, a duplicated code fragment which contains many pointers that has evolved separately from its original is prone to cause problems when the data type changes; to prevent it one has to use extensive testing that covers 100% of all data type values, which is unfeasible in practice. An example clone is presented in Algorithm 1 with comments and explanations removed. The code contains a part of an algorithm and has been found to be copied across two distinct modules (not between two different places in the same code module). Modifying this code in one place only can lead to problems with very specific functionality of the product in the field, hard to detect without extensive and time-consuming debugging.

We define the *nonobstructive clones* as those duplicate code fragments which do not constitute problems for the product after the release. Examples of such code clones are

(1) code templates which could be caused by design templates;

(2) local declarations which could be caused by the naming conventions and avoidance of global declarations;

(3) autogenerated code fragments which could be generated from interface descriptions or from models (e.g., skeletons of state-machines);

(4) code for handling of portability or variability which could be caused by a design decision of how to implement the portability or variability mechanisms (e.g., to keep code complexity low);

(5) repetitive declarations of data types with minor modifications.

The nonobstructive clones can cause (at worst) nuisance for the organization to maintain repetitive code fragments (as one of the designers expressed this in the study: "*the organization could be chasing ghosts*"). This kind of nuisance has been previously found by LaToza et al. [28].

An example of a code clone which is nonobstructive is a template for state machine implementation which was common for the whole module (1). This kind of clone is naturally not obstructive and is in fact necessary for the correct functioning of the software and consistency of programming at the company prescribed by coding guidelines.

The repetitive declarations (5) are nonobstructive and they could be considered as "noise" caused by the measurement instruments used, detecting only Type I clones based on pattern matching.

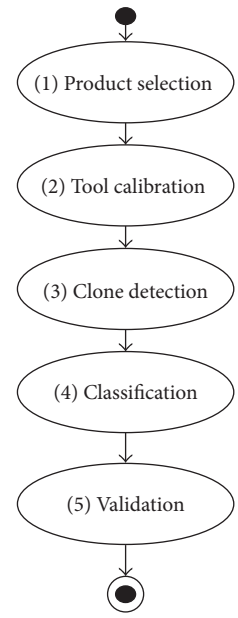

Figure 1: Research process.

4. Design of the Study

We used a flexible research design advocated by Robson [38], because part of the research method needed to evolve during the study—the clone classification method. The goal was set a priori; research questions regarding the significance of the clones were presented in the introduction. The researcher was present at the company on a weekly basis over a period of ca. 6 months. We classify this type of research as a case study with multiple units of analysis, three different products. Since the study was conducted over a number of releases and over a longer period of time, we treat it as a longitudinal study. Our prepositions when designing the case study was that *combining simple clone identification methods with simple classification methods can reduce the number of clones to be evaluated manually by over 80%.*

4.1. Research Process. In this study we followed a research process defined a priori, presented in Figure 1.

In step 1 we started by identifying three products based on a number of criteria: size of the product (both large and

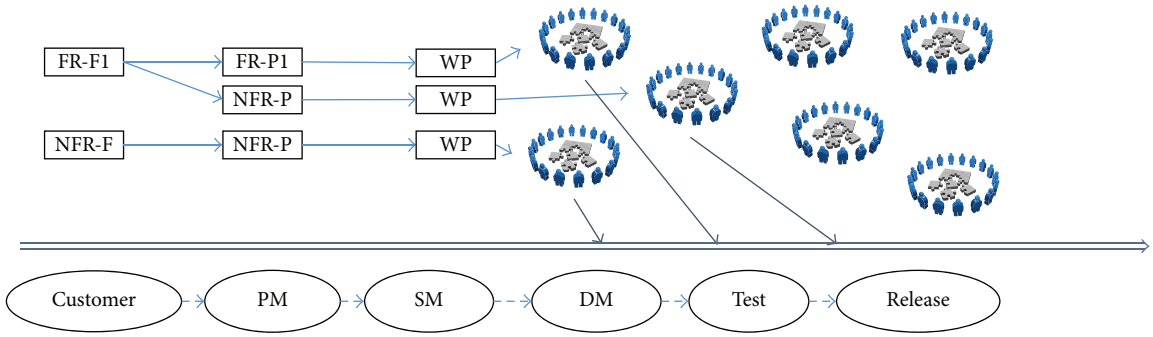

Figure 2: Feature development in Lean/Agile methods.

medium code base), diversity of programming languages (C, C++, and Erlang), maturity (multiple releases), development method (distributed, parallel development, and Streamline development), and availability of experts for the classification and calibration. After identifying the products we proceeded to calibrating the tool (step 2); the results of the calibration can be found in Section 4.3. In the next step (3) we collected the clones from all products. In the proceeding step we classified the clones (as described in Section 4.4) and finally we validated the classification together with the practitioners (described in Section 4.5).

4.2. Product Selection. Ericsson develops large products for the mobile telephony network. At the time of the study, the size of the organization was several hundred engineers and the size of the projects was up to a few hundreds. (The exact size of the unit cannot be provided due to confidentiality reasons.) Projects were increasingly often executed according to the principles of Agile software development and Lean production system, referred to as Streamline development (SD) within Ericsson [39]. In this environment various disciplines were responsible for larger parts of the process compared to traditional processes: design teams (cross-functional teams responsible for complete analysis, design, implementation, and testing of particular features of the product) and integration testing. As Agile development teams became self-organized the software development work became more distributed and harder to control centrally [40]. The difficulties stemmed from the fact that Agile teams valued independence and creativity [41] whereas architecture development required stability, control, transparency, and proactivity [42].

Figure 2 presents an overview on how the functional requirements (FR) and nonfunctional requirements (NFR) were packaged into work packages and developed as features by the teams.

The requirements came from the customers and were prioritized and packaged into features by product management (PM) who communicated with the system management (SM) on the technical aspects of how the features affected the architecture of the product. The system management communicated with the teams (Design Management/DM, Test) who designed, implemented, and tested (functional testing) the feature before delivering to the main branch. The code in the main branch was tested thoroughly by dedicated test units before they were able to release it [8].

In the context of this distributed software development code, cloning could become an issue as technical decision-making was distributed. For example, refactoring initiatives could be prioritized and taken up differently by multiple teams. In addition to the distributed decision-making, in this distributed environment, knowledge-sharing might potentially be inefficient (teams might not always know which other teams were working on the same code segments in parallel). Therefore monitoring the evolution of clones and assessment of how obstructive they are were of high importance for architects, design owners, and quality managers at the company.

4.2.1. Products. The development projects at the studied organization involved over 100 designers and lasted over a number of years. The three products examined in this study were independent from one another and had a number of releases in field (mature products and processes).

Product A. It was a large telecommunication product for handling mobile data traffic at the application layers of the ISO/OSI model. The product had ca. 1 MLOC and had been under development for a number of releases under more than 5 years. The process was waterfall in the first releases and Streamline development in the last years. The programming language was C/C++.

Product B. It was a large telecommunication product for handling mobile data traffic on the transport level of the ISO/OSI model. The product had over 9 MLOC and had been under development for a number of releases under more than 7 years. The process was waterfall in the first releases and Streamline development in the last years. The programming language was Erlang.

Product C. It was a member of a product line of a range of base station controllers and other telecommunication infrastructure components for an international market. The studied

product was developed using traditional, waterfall model for the first releases and according to Streamline development in the later releases. The programming language was C/C++.

All products are developed with development teams at multiple sites. However, Product A was developed at one location in the first releases (as indicated in Section 5).

4.3. Tool Calibration and Clone Detection. Type I clone detection methods employed in our study were based on text similarity and we used Duplo open source software (https://github.com/dlidstrom/Duplo). We used two parameters when calibrating the method—the minimum block size and the minimum line size. Before conducting the analyses we checked whether there is a major difference between the block size of 5 lines (recommended in literature) and 10 lines (as suggested by the practitioners). We calibrated these parameters by collecting clones from a subset of the product with different parameters, presenting them to the architects and designers and discussing the results with them.

For Products A and B after inspecting a subset of clones we concluded that using 5 lines as the parameter gives many false-positives, that is, duplicated code fragments which cannot be considered as clones. Examples of such code fragments are common variable declarations. Therefore the minimum number of lines was set to 10 for both Products A and B.

The other parameter, minimum length of the line in the code, was calibrated to be 5 characters. Using longer lines, for example, 10 characters, resulted in missing important clones such as state machine implementations common to different modules. The calibration was done by comparing the results for different parameters on randomly chosen set of clones.

For Product C, however, we calibrated the methods to 5 lines and 5 characters as recommended in the literature. During the discussions with the practitioners we found that the 10 lines were too restrictive and several of obstructive clones contained less than 10 lines.

4.4. Classification. As presented in Figure 3 we started the classification by introducing the classification framework (4a) to the practitioners and discussing its content—its completeness, semantics of the attributes, and discussed examples. After the introduction we randomly selected 10 clones to classify (4b) together with the practitioners during an hour-long session (4c). The practitioners were asked to explain their rationale and teach us about how to recognize the different attributes of the clones (in our classification scheme presented in Table 1). In each of the sessions, however, we managed to classify between 20 and 30 clones with explanation and towards the end of the session the practitioners examined us on the ability to correctly recognize the clones (4d). This allowed us to move forward and create a script to recognize the attributes based on what we were taught by the practitioners (4e) and automatically classify the majority of the clones (4f) as described in Section 5.4. The clones which were not classified automatically were classified manually by us, which was later cross-checked by the practitioners (4g).

In our study we worked directly with four architects and indirectly (through workshops) with over 20 design architects. The practitioners involved in the study had more than 10 years of experience and more than 5 years of experience with

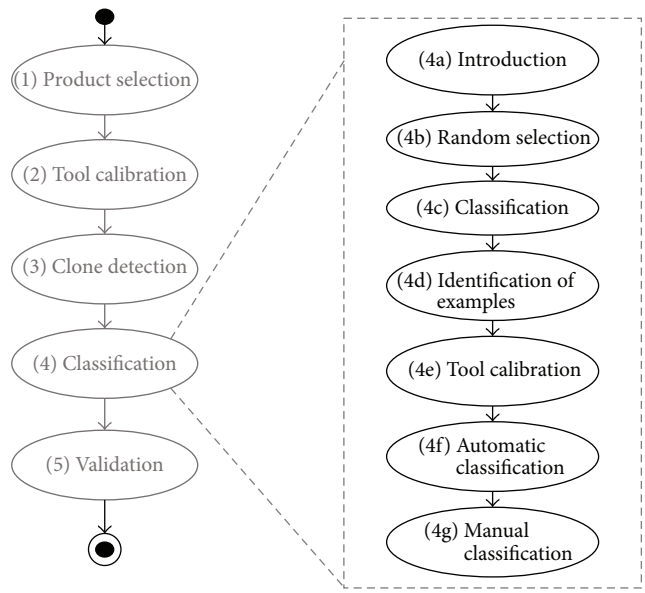

Figure 3: The process of classifying clones.

the product. Their role was software architects for the entire product (Product C) or subsystem architects (Products A and B). Their role was lead designers responsible for subareas of the product or architects. All researchers involved in the study had a background as professional programmers. In step (4d) (text recognition) we collected examples of the most common clones, for example, state-machine implementations, code with many pointers, and declarations of variables.

The results of the classification were validated during workshops with the same architects who classified the clones in the first steps of the classification (4a–4d). During the workshops (one per product) we showed the results of the classification and emphasized "new" cases which were not completely aligned with the examples discussed in step (4d) (examples identification). Since the architects agreed with our classification of these "new" cases, we were confident in the results. We also discussed 2-3 randomly chosen clones aligned with the examples and there we still had a consensus, which reduces the risk of internal validity of subjective classification of clones.

4.5. Design of Product Cloning Evaluation

4.5.1. Measures and Units of Analysis. One of the important aspects of assessing the significance of the clones was their classification in terms of time (release) and location (type of the code). In order to characterize cloning in the products in terms of both time and location we used a set of measures and their visualizations:

(i) Number of clones with granularity per file, per product, or per release.

(ii) Number of cloned LOC with granularity per file, per product, or per release.

(iii) Number of cloned files (number of files which include cloned code) with granularity per product or per release.

Table 1: Code clone classification attributes.

Attribute	Values	Mapping to classification by Kapser and Godfrey [19]
Severity	Obstructive/ not obstructive	N/A
Location/type of code	Product/ Test/ Target environment/ Simulation environment/ Platform specific code	Customization Hardware variations (type of forking) Experimental variations (type of forking) Platform variations (type of forking)
Source	Coding guidelines/ Templates/ Unknown	General language or algorithmic idioms (templating)
Target action	Remove/ Leave as-is/ Refactor[1]	N/A

[1]The target action "Refactor" was recognized after the study was completed and therefore it is not part of the results presented in the paper.

We explicitly excluded the percentage of cloned code per module or per subsystem from the analysis as it did not provide insights into development practices leading to cloning. It did not provide the possibility of pinpointing whether cloning is a problem or not.

In the study we also investigated how the clones spread over the system, that is, whether cloning happens within the same subsystem or within different subsystems. The clones which spread between different subsystems could potentially lead to more serious problems than clones within the same subsystem (e.g., with respect to inconsistent updates of code clones during corrective system maintenance). This analysis was done by visualizing the patterns using heatmaps and discussing their right-to-be with the designers and architects. Heatmaps were used in our previous research [7, 43] to visualize code changes and thus were familiar to the designers and architects in the organization. The use of heatmaps for clone visualization was inspired by the visualization used by CCFinder [44].

4.5.2. Analysis Methods. In order to address the research question we used the following analyses in the study: total number of clones, cloned files, and cloned lines of code per release. This analysis provided us with the insight into how cloning evolved in the software lifecycle.

Cloning across subsystems. We investigated whether the cloned code was spread among subsystems of the product or it was part of only one subsystem. This analysis provided us with the insight into whether cloning is a phenomenon which occurs in "own" code or whether it also occurs in other teams' code.

5. Results

When presenting the results from the study, for confidentiality reasons, we cannot provide the absolute number of clones. (As we studied commercial products which are on the market, due to confidentiality reasons, we are not able to provide such data as percentage of the cloned code, the total number of clones, the size of the clones, or the timescale of the product development.) We use percentage and normalized values wherever applicable to show the trends. The names of the components are also changed due to the same confidentiality agreement.

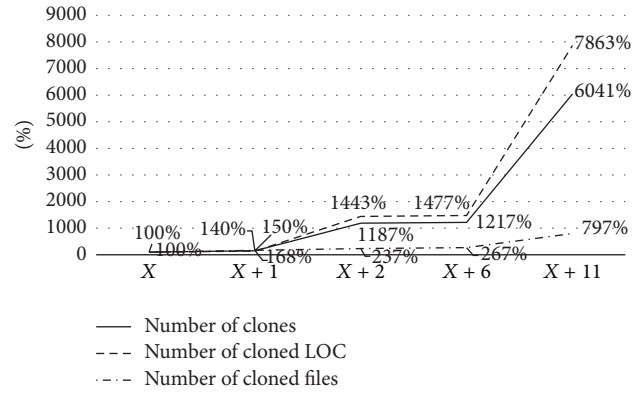

Figure 4: Number of clones, cloned LOC, and files with clones per release for Product A. The trend shows Type I clones. The rate between the obstructive and nonobstructive clones is consistent across releases.

This section is structured as follows. In Sections 5.1 and 5.2, we present the background results of how often cloning happens in the commercial software in the studied company. In Section 5.3, by applying the classification scheme from Section 4 to the three commercial systems, we address the research question RQ 1a: Which clones are obstructive and which are not? In Section 5.4, we address the research question RQ 1b: How to efficiently distinguish which clones are obstructive? In Section 5.5, we summarize the analyses and in Section 5.6 we present the feedback from architects about our results.

5.1. Evolution of Clones over Releases. Figure 4 shows the evolution of clones in Product A over a number of releases.

The figure shows a large increase between releases $X + 6$ and $X + 11$ (almost 5 times) which might seem serious. This increase could lead to wrong conclusions if not followed up with the architects and designers of the system. After

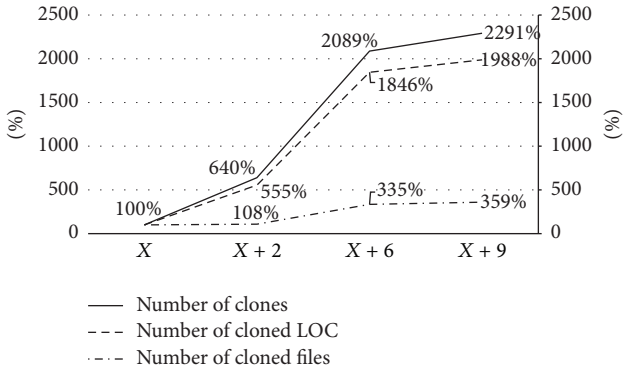

Figure 5: Number of clones, cloned LOC, and files with clones in Product B. The trend shows Type I clones. The rate between the obstructive and nonobstructive clones is consistent across releases.

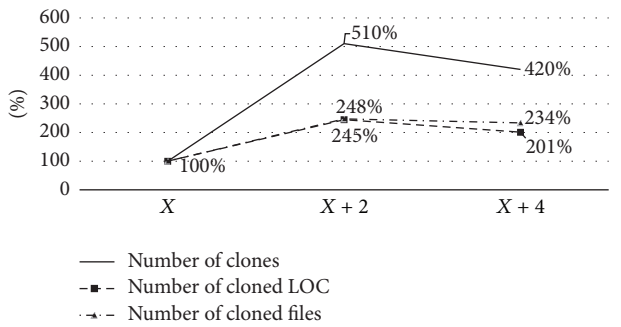

Figure 6: Number of clones and cloned LOC for Product C. The trend shows Type I clones. The rate between the obstructive and nonobstructive clones is consistent across releases.

the initial investigation with the designers we found that the large increase was caused by

(i) new platforms introduced, which required duplication of code to handle portability,

(ii) merging with another product (and thus another code base),

(iii) transition to distributed and cross-functional development.

During a workshop with design architects at the company it was found that there can be good reasons for cloning which needed to be investigated further; for example, coding guidelines need to create functions with similar functionality to existing ones but with slight modifications based on platforms. These are reported in Section 5.4.

Figure 5 shows the increase of number of clones in Product B. The trend is similar to the trend in Product A where the last releases have more clones in the existing files than the previous releases.

Our investigations with the product showed that the trend and reasons for the existence of clones were similar to Product B.

Figure 6 shows the increase of the number of clones and number of cloned lines (LOC) per release for Product C.

The trends show that there is an initial increase of the number of clones and cloned LOC (releases X and $X + 2$).

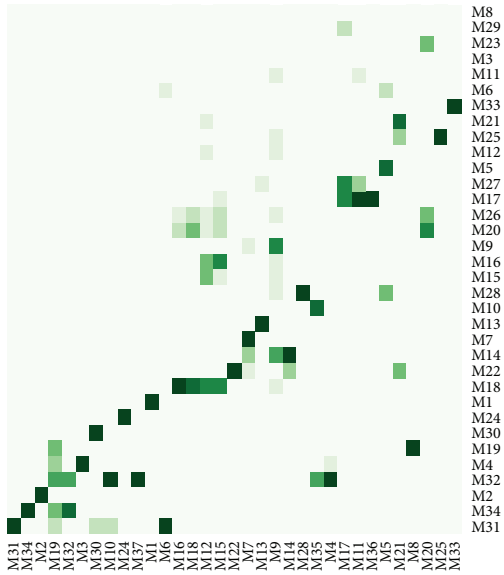

Figure 7: Map of clones per source code component for Product A. Darker dots indicate large number of clones shared between components.

In release $X + 4$ the number of clones decreased nonproportionally to the number of files with clones. It was explained by the architects that the project refactored the code in certain component during that time. The total number of cloned LOC decreased compared to release $X + 2$. This showed that the refactoring activities resulted in reducing the amount of cloning, but there were still a large number of clones left in the product. These results are in line with the results of Göde who found that clones of Type I need to be explicitly managed to be refactored [45].

No refactoring activities were reported for Product A and Product B, which could explain the fact that no decrease in number of clones was observed for Product A and Product B.

5.2. Detected Clones. Figure 7 shows how clones spread over the product; the more intensive the color, the more the clones shared between the two components. The grey scale represents the number of clones shared between each component or between modules in the case component. The background represents 0 (no common clones). Given that number of all clones is 100 (the real number of clones is, however, much higher), the scale is as follows: (i) black: 4 or more clones, (ii) dark grey: 3 clones, (iii) medium grey: 2 clones, and (iv) light grey: 1 clone.

As it is shown in the figure a number of clones are located within the same source code component, denoted by the dark color around the diagonal in the diagram (although they may be between different modules within the component). This can be seen in line M29 which contains clones only within the same component.

However there are clones which are not within the same components; for example, the fourth line from the top (M11) shows that component 11 (M11) shares clones with component

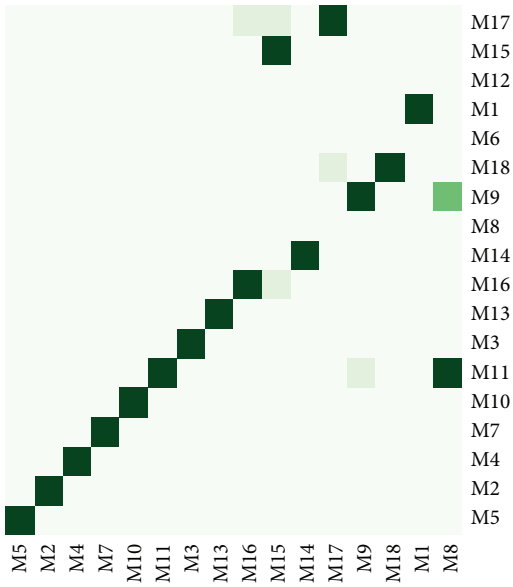

Figure 8: Map of clones per component for Product B.

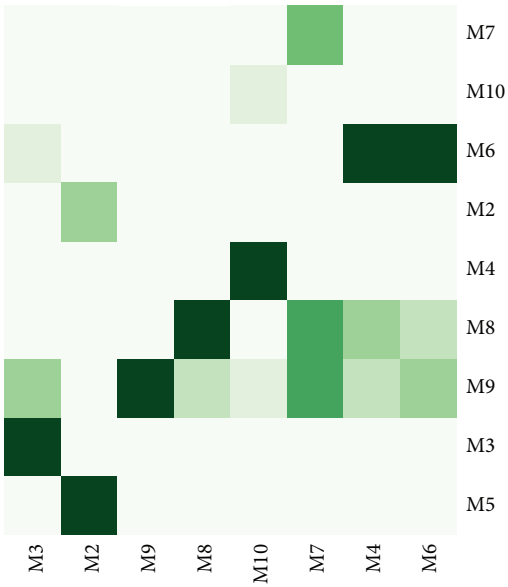

Figure 9: Map of clones per component for Product C.

M9. The right-hand side of the figure shows that cloning between components is quite common in this product.

Figure 8 presents the heatmap of clones per component for Product B.

This figure shows a pattern of cloning which is different from Product A. Most of the clones are created within the same component and only a subset of components share the same code fragments.

Figure 9 presents the heatmap of clones per component for Product C.

The heatmap for Product C shows that the clones spread over multiple components (for components M6, M8, and M9). This was validated with the architects who pinpointed that the clones between different components were indeed potentially negative for the software quality and should be addressed.

As we can observe from these diagrams, the patterns of cloning for Product A and Product C include cloning between components. Given higher number of dark-colored cells we can also observe that there are many clones. After discussions with practitioners it became even more evident that we need a simple way of almost automatically finding clones which should be removed (target-remove).

5.3. Obstructive and Nonobstructive Clones (RQ 1a)

5.3.1. Classification Scheme. The classification scheme presented in this paper was developed for the purpose of this study and addresses RQ 1b: *How to efficiently distinguish between the obstructive and nonobstructive clones?* (Although the classification scheme was developed as part of the study we present it in Section 3 in order to explain how the classification was done.) Although there exists a classification scheme developed by Kapser and Godfrey [19], a simplification of that scheme was needed in order to capture the obstructiveness of clones and at the same time require as little time as possible from the classifiers (when bootstrapping) and low manual classification rate when using analogy-based classification. In particular our classification merged source of cloning (e.g., templating, boilerplating in Kapser's classification) to a few attributes relevant for the practitioners (e.g., coding guidelines). This adaptation was done after the discussions with the practitioners who needed a tool for making decisions on changing the product (e.g., removal of the clone) or changing processes (e.g., changing the coding guidelines).

Classifying the clones with practitioners is a means of adding more semantic information to the clone and can act as an extension to automated classification using distance and grouping of the clones (e.g., as described by Kapser and Godfrey [17]). The design rationale behind the classification scheme was that it should be simple to fill-in and thus efficient, when used by designers.

In order for the scheme to be effective we needed to include attributes which would pinpoint the location of the clone (e.g., test code, product, and simulation environment) and its right-to-be (e.g., if it is ok to keep the clone or whether it should be removed). These kinds of attributes were found to be important in another study by Kapser and Godfrey [19]. Table 1 presents the attributes in the classification scheme. This new scheme was developed together with practitioners at Ericsson and can partially be mapped to the classification by Kapser and Godfrey.

One new attribute was *target action* which allowed selecting the clones which are not important for the product and can be left in the code. Examples of such clones are clones in the platform portability code which is supposed to implement the same platform specific code in multiple ways; thus the code clones are quite common and known. In theory these code clones are not serious or obstructive either. The target action means that the clone should be removed in the next release of the software. During the discussions with architects

we found that not all obstructive clones should be removed; sometimes there is a case that the code is scheduled for refactoring (thus target action is to leave it as-is) or that the clone is in the code area which should not be altered and the clone should only be monitored. From the study by Zibran et al. [46] we could conjure that the properties of the clones (e.g., size or location) do not always correlate with the decision to remove the clone and that manual assessment is needed.

We chose to use the binary scale (e.g., obstructive versus nonobstructive) to reduce the conceptual need of classifiers to understand the difference in other scales (e.g., Likert scale). We chose instead to complement the scheme with more attributes (e.g., type of code). In this choice we were inspired by such classification schemes as, for example, Orthogonal Defect Classification or LiDeC [47, 48].

5.3.2. Automatic Classification. In order to identify which clones are obstructive we collected all clones from Product A and ordered them by two attributes:

(i) Location: module and directory where the clone was located.

(ii) Type of the code cloned: whether the clone was an algorithm or data declaration.

Then we asked a design architect to read the clone and reason whether this clone was obstructive and why. We have then filled in the attributes from the classification scheme. These were used to develop a script for automated classification of clones. We developed relatively simple scripts in MS Excel VBA and Ruby to compare the similarity of clones. We then run a script which found clones in the same location and of the same type of the cloned code. The script classified the other clones (by analogy) to the same categories as the first clone. Then we have proceeded with the next nonclassified clone and repeated the procedure.

The goal of the script was to be able to classify the majority (over 80% as stated in our preposition) of clones without human intervention thus saving the time of the experienced practitioners. The resulting scripts were able to classify 95%-96% of the clones (for Product A, for the latest release, they were able to classify 96%; for Product C it was 95%). After the discussions with the practitioners we concluded that the balance between the complexity of the scripts and the ability to classify more clones was satisfactory.

5.3.3. Validation. The results of the classification were validated during workshops with the same architects who classified the clones in the first steps of the classification. During the workshops (one per product) we showed the results of the classification and emphasized "new" cases which were not completely aligned with the examples discussed in step (4d) (examples identification). Since the architects agreed with our classification of these "new" cases, we were confident in the results. We also discussed 2-3 randomly chosen clones aligned with the examples and there we still had a consensus, which reduces the risk of internal validity of subjective classification of clones.

Table 2: Percentage of obstructive/nonobstructive clones in Product A, the latest release.

Manual or automated	Obstructive	Nonobstructive	Total
M	1%	3%	4%
A	0%	96%	96%
Total	**1**%	99%	100%

In this paper we applied the automated analogy-based classification method on the latest release of Product A where the number of clones was the largest. The method could be applied across releases, but we decided to evaluate it on the largest number of clones available to us, as the method is supposed to scale up to large quantities, both in terms of performance and accuracy.

The automated classification cannot classify all clones as the variability of clones is just too large; for example, for Product A for the latest release, 96% of clones could be classified automatically and the remaining 4% were classified manually.

We found that there is a correlation between the significance of a clone and the ability of our algorithm to classify it as obstructive/nonobstructive as presented in Table 2.

Using this analogy-based classification provided us with a possibility to identify obstructive clones and therefore define which they were (the same definition as that presented in Section 4): those duplicate code fragments which contain code with algorithms (or part of algorithms) which are critical for seamless operation of the product in the field. The location and type of cloned code determined this.

5.4. Distinguishing between the Obstructive and Nonobstructive Clones (RQ 1b). After investigating the reason for this we concluded that the obstructive clones are the ones which are "odd" in the sense that they do not fall into a known category where our algorithm can easily classify them. Therefore we see this algorithm as a filter and input to decision-making for architects and designers; it addresses the research question RQ 1b: *How to efficiently distinguish between the obstructive and nonobstructive clones?*

Our industrial partners saw this as a good trade-off between the erroneous classification and manual effort.

Figure 10 shows the statistics of clones per type of cloned code.

The statistics in Figure 10 show that 92% of clones are related to platform, which in our classification means that the clone is a duplicated code with minor modifications to reflect the platform specific code, for example, memory handling for different operating systems. The clones which are part of the product are in minority (category Product).

Closer investigation of the clones showed that 91% of the clones are caused by design decisions on how the platform portability is implemented (thus deemed not obstructive by the design architect). These clones exist in the code base but not in the product; when the code is compiled there is only one version of the code.

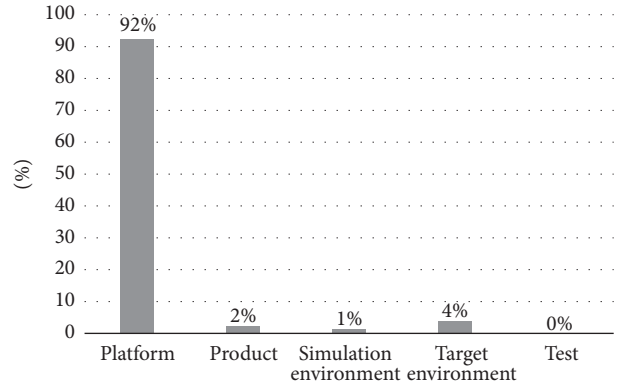

Figure 10: Types of clones in Product A.

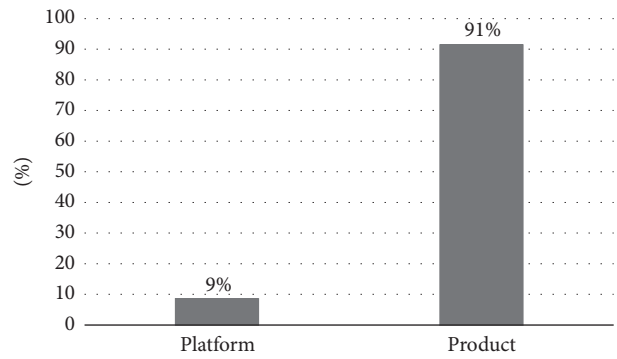

Figure 11: Types of clones in Product C.

In the studied code repository there were a few test cases (not the whole test repository) which are shown in the diagram.

In Product C we used the same approach and adjusted the parameters of the algorithm (e.g., regular expressions of component names) and we were able to classify 100% of the clones automatically. Figure 11 presents the types of clones found in Product C.

The figure shows that the majority (91%) of clones in this product are part of the product. All of them were classified as obstructive.

Since the results differed between Product A and Product C we investigated the reasons for that and found that

(i) in product A we used the whole source code repository, which included much more than just the product code, whereas in Product C the architects exported the list of files that were used in the production code with some elements of the platform code (necessary for the code to execute);

(ii) in Product C the code base which we used in clone analysis was much smaller (ca. ten times) and the architects knew details of it and could directly pinpoint why the clones are obstructive.

The quantitative analyses allowed us to visualize the magnitude of the potential problem with cloning for the company which triggered improvements. However, we conducted also the qualitative interviews with architects of Products A and C in order to validate the quantitative analysis.

5.5. Summary of Analysis of Clones. The previous Sections 5.1–5.3 presented a number of analysis of clones for each of the products. In this subsection we summarize and reflect on the results of the analyses per product and identify commonalities and differences, drawing the conclusions which we used as input for discussions with designers.

5.5.1. Product A. Product A was the first one to be analyzed and the analysis methods were adjusted during the research process. The analyses were redone as we were able to calibrate the analyses methods based on discussions with the designers and product owners.

For the product we observed the largest increase of clones over time (60 times over a period of 11 releases). This kind of increase was previously observed in open source projects, so this called for further analyses. Since this product undergone a transition from waterfall development to Lean/Agile development (Streamline development [8, 49]) our initial hypothesis was that the transition to distributed and self-organized development was the cause of the clones. However, the analyses showed that the clones were caused by the increased number of supported platforms and including the platform code in the source code repository.

Since we did the analyses of the clones based on all source codes available in the repository, the number of false-positives was obstructive; 96% of the clones were not obstructive. We could consider this to be the noise which should be reduced in analyses. The remedy to this was analysis of Product C, where we involved the architects when choosing source code files in the analyses.

5.5.2. Product B. This product showed a different pattern of clones than Product A. The spread of the clones over components was lower than in Product A and the increase of the number of clones in percent was much smaller (ca. 20 times over 9 releases).

There were two factors which could potentially influence this analysis—the analyzed code was developed at one site (compared to multiple sites in Product A) and the programming language used in the implementation was Erlang (compared to C/C++ in Product A).

5.5.3. Product C. Compared to the analyses of the previous products this product was considerably smaller, written in C/C++, and the input data for clone analyses was provided to us by the two main architects of the product. As the analyses showed, the number of clones that were identified as potentially obstructive was much higher percentage-wise. We confirmed that number by discussing this with the architects and their responses were that they had done the prefiltering of files to only the relevant ones. They had removed files with test cases, simulation environment, or configuration.

Because of this prefiltering it was possible to create an algorithm which could classify all clones automatically based on the location of the file (reflecting the architectural location

of the file) and whether the code had clones within the same or different components or blocks.

5.6. Feedback from Designers and Architects. During our analyses we kept close contact with the designers and architects of the studied products in order to confirm or refute our intermediate findings and to minimize the threat of making wrong interpretation of the findings. We used their product and process knowledge and we also observed their reactions to the presented data. In this section we report the most prominent insight which both we and them found important for others who want to replicate this type of study.

One of the first elements of feedback we obtained on the study was when we presented the design of the study to the company. The initial feedback was that there would be no clones of Type I and Type II in the code due to the strict quality assurance and monitoring processes at the company. As we refuted this claim at the beginning of the analysis of Product A, we understood that we need to conduct a deeper analysis of the sources and nature of clones to understand why the practitioners were mistaken; that is, how the need for study of significance became evident.

For Product A we presented the results for the design owner and then for all component architects of the product. The set of initial answers was consistent:

(i) One of the explanations behind cloning could be that *if a code has been proved good, then we clone it in order not to "reinvent the wheel."* The analysis of the obstructiveness of the clones showed that this can indeed be true. Designers do not clone the code which can cause problems for the quality of the product but the code which has obstructively lower importance (e.g., test code).

(ii) Another explanation of cloning is that in each release one might change the design guidelines which affect the code, for example, adding a nonfunctional requirement that all modules have to implement the same state machine.

(iii) Finally one more explanation was the changed process, that is, going Lean/Agile, which could contribute to the distribution of knowledge and the implicit need of duplicating/modifying code rather than refactoring not-owned code; as described in Section 5.5 we could not find any evidence of that.

In Product C the architects asked for more fine-grained categorization of obstructive clones. In particular they suggested that certain types of cloning patterns should be considered as obstructive as they quickly become unmanageable; those were as follows:

(i) Clones within the same module should be avoided and the teams should be able to identify those clones themselves.

(ii) Clones between different modules but within the same software unit (one level up in the architectural hierarchy) should be removed as there is a risk that the modules are assigned to different teams over time and then the clones become unmanageable; thus the architects stressed the importance of the results of Kapser and Godfrey [50].

(iii) Clones between different software units or blocks are obstructive since they become unmanageable almost once they are created; finding inconsistencies of fixing problems with the related code is very difficult as they are managed by different teams; confer [6].

We observed that the designers and architects took a "defensive" position when first confronted with the results, but after a discussion their attitude changed to positive and constructive. They helped in the analyses and read the clones and made decisions about what to do with them. This was the final feedback to us that the results are important for the company and lead to making their product better, which was the goal of the collaboration from the very beginning.

For Product C, however, the architects have refactored all of the clones which were found obstructive in the study.

6. Discussion

In our study we focused on Type I clones, which are the most basic type of clones, exact code duplicates. By using the Levenshtein distance metric when comparing code fragments we allowed for some modifications in the fragments thus expanding the set of clones found.

Before setting off to study the clones there were a number of challenges which we wanted to explore based on the existing body of research on cloning in general; for example, whether there are clones in the commercial systems at all; how much cloning takes place; whether the clones are obtrusive for software designers; and whether we can be able to provide a simple method for finding the clones which should be decided upon by the designers and architects.

For the first challenge, whether there are clones in the commercial systems, we chose the systems in which we suspected that the clones would be present given the fact that there is some evidence of cloning in commercial systems as presented by Rattan et al. [3]. We also understood that the industrial set-up of large software systems is challenging for the collection of advanced cloning data for such reasons as (i) multiple language use (C/C++/Erlang), (ii) multiple environments (Windows, Unix, Linux, and Real-time OSs), and (iii) dependencies on hardware libraries, which made it difficult for the parsers to parse the code out of its context. These findings quickly resulted in the fact that using tools that can parse source code to extract clones was not feasible. Therefore we focused on extracting the clones of the simplest type—Type I, exact clones. As Roy and Cordy [51, 52] pointed out this type of clone is simplistic and the more interesting ones are the Type III clones (e.g., near-miss function clones) detected using more advanced extraction method (e.g., dedicated programming language TXL). By studying the clones of Type III presented in the literature we agree with the current finding but we perceive this as a challenge to extract them by practitioners.

This kind of need of simple usage for practitioners led us to investigating the challenge of how to provide this simple

methods to industrial applications where the focus is on quick detection of clones and their quick classification. Type III clones are the most interesting type but require extra effort from the usually busy software designers, which led us to asking the question: how can we use Type I clones with some simple postprocessing (classification) to provide similar information? This led us to the usage of distance metrics (Levenshtein distance) when extracting the clones. This approach is not as robust as the fully fledge metric-based extraction such as that presented by Kontogiannis [53] who used complexity and similar metrics as distance between the clones.

Using the metric-based approach would indeed improve the ability to identify clones that should be removed because they contribute to lower quality. When discussing the clones with the practitioners such metrics as complexity could be easily recognized as the metrics of the obstructiveness of the clones. For example the technical architects often regarded "complex" code to be more problematic when cloned (e.g., contributing to lower understanding), but they have also indicated that this code might have been copied on purpose (because it was proven to work). In our previous studies we have found a similar trend that the complexity was only a symptom that needed to be combined with other symptoms to indicate if it was a problem [54, 55].

In our further work we intend to find a method to use clones of Type III and to automatically classify them. In light of the existing body of research on cloning in open source systems, we believe that this would be a step toward automated support for refactoring. Since in Product C we observed that the architects removed all obstructive clones, one could develop an early warning system for the designers to notify them that a cloned code is about to be introduced to the main software branch. When discussing the clones and considering that code that represented an algorithm, we could often see that the context of the clone (the code before and after the clone) was only slightly modified which meant that if we used more advanced extraction of Type III clones we could increase the accuracy of the analysis, for example, automatically recognize if the cloned code was an algorithm or a data declaration. This remains to be studied further in our next studies, whether and which parameters of the clones are important for the practitioners (e.g., algorithm versus data declaration, complex versus simple code).

For the second challenge, how much cloning takes place, we could observe that Type I cloning takes place quite often. The fact that we only used Type I contributed to the large number of clones found and the large number of clones that were found to be unobtrusive. We have also found that that cloning tends to increase with the lifetime of the product unless specific initiatives take place (refactoring in Product C after this study). The same trends were observed in open source software in the study of Thummalapenta et al. [29]. The practitioners participating in the study indicated that this was an "eye-opener" regarding the development practices, understanding of what consequences of cloning are, and how to deal with them.

For the third challenge, whether the clones are obtrusive, we set off to explore the significance of the clones for practitioners. Given the body of research on cloning and no consensus on whether they are a good or a bad phenomenon we wanted to explore what practitioners consider to be problematic in cloning. Therefore instead of asking about the problems with clones we focused on whether the clones obstruct the work of the software designers and architects. The first step was to define the concepts of an obtrusive source code clone. We have found that the parameters which the practitioners considered were related to the location, type of code, and the fact whether the clone will actually make it into the compiled product. This last aspect, including the compiled code, is something that we have not found to be considered in the existing body of research. We have found that cloning takes place for test code, platform code, or another "auxiliary" code which often does not result in duplicated code fragments after compiling the specific version of the product. This can contribute to the debate on the importance of clones, namely, by asking the question what is essentially cloned?

The location as a determinant of a seriousness of cloning has also been indicated in the studies of Juergens et al. [12] and Toomim et al. [13]. Thus we perceive that the industrial applications are similar to the open source ones in this aspect. Therefore we also found that the location is a good indicator for the classification and therefore we used that parameter in the classification scheme.

This distinction of the type of the code cloned was also discussed during the workshop with design architects who decided to prioritize the algorithm code for our studies. Their perception was that it is harder to ensure the quality of the algorithm code without advanced methods and the consistent modifications can be harder to capture compared to the data declarations.

7. Validity Evaluation

To evaluate the validity of the study we followed the framework presented by Wohlin et al. [56].

The main threat to the *external validity* of the results of this study is naturally the size of the sample, three products. Although the sample is small we believe that the contributions are valid as the results have been confirmed qualitatively through interviews and discussions with the practitioners with insight into the product (all interviewees had more than 10 years of experience with these products). The initial results (e.g., number of clones) were in line with the previous research on open source projects presented in the systematic review [3] which also minimizes the risk of external invalidity of the study. Using single case study has a natural threat to the external validity in terms of the specific context. However, we believe that the studied case provides insights of the industrial practices. During the study none of the practitioners mentioned the code clones to be specific to the telecom domain or the company but rather to the task of designing software (as discussed in Section 5.4). This lack of telecom specific statements together with the large body of knowledge in the open source cloning studies indicates that the results can be extrapolated to other contexts. It also shows

that the results from the open source clone studies apply to the industrial software.

The main *construct validity* of the study which we identified was the fact that we chose simple tools for detecting clones. This could result in the fact that we missed important clones or that we found too many clones that were irrelevant. We recognized this threat in the study of Product A and therefore, to minimize it, we asked the architects of Product C to preselect files for us. Since this preselection had an effect on the number of found clones, we discussed that fact with the architects. The common conclusion was that it would be interesting to compare the results with and without the prefiltering. The quality of the resulting automated classification was good enough to justify the prioritization of the architects' effort to other activities.

Another construct validity threat is the use of expert classification as a measure of significance of the clone instead of such measures as number of defects per component. We have collected the number of defects per component for Products A and B, but the data was not reliable. For Product C the data was not available with the required resolution; it was not possible to link defect reports to particular components. Introducing low quality data would jeopardize the analyses and therefore we decided not to analyze whether the existence of clones could be correlated with the number of defects reported per module.

Since this evaluation is done a posteriori, that is, historical releases were analyzed years after the release, our study has the history effect—a threat to *internal validity*. In order to minimize the threat that we used wrong baseline in the analyses, we confirmed the baselines with the official release documentation and involved the configuration management team in accessing the right baselines in the study. Another important threat is the choice of the instruments, which in our case is the classification scheme and using binary scale for each attribute (e.g., obstructive versus nonobstructive). We deliberately chose the binary scale in order to simplify the classification process. To minimize the threat of misunderstanding we cross-checked the classification results with two architects (for Product C). In all cases they both agreed with the assignment of each attribute to each clone in a randomly chosen sample. We were also clear in our study that we only consider clones of Type I.

There are two *conclusion validity* threats which we recognized as obstructive and actively worked to minimalize. The first threat is the fact that the research was conducted at a company, which brings the question of the completeness of the information we obtained. Since we had a long history of collaboration (7 years) we could openly discuss the results and the practitioners treated us as one of them, providing thorough insights into the product. We were able to analyze the source code and architecture descriptions and had unlimited access to the engineers. The other threat is the lack of inferential statistics in our analyses. We considered using the measure of number of defects as an "effect" and pose hypothesis accordingly, but that measure was not reliable enough; in particular neither we nor the practitioners could say that the location in source code where the defect was fixed is the location where the defect occurs. Therefore we decided to evaluate the results by interviews and presentations to practitioners. In total the results were presented to ca. 40 engineers and managers for feedback (summarized in Section 5.6).

8. Recommendations for Replications

In short, based on the experience from this study, we can make the following recommendations for companies willing to replicate this study:

(i) If possible, preselect software modules (files) before running the clone analyzers; by using these two strategies in Product A and Product C we could observe that the initial effort in filtering enabled fully automated analogy-based classification.

(ii) Carefully analyze the significance of clones before spreading the information throughout the organization. As our analyses in Product A showed that there could be a number of nonobstructive clones, only the obstructive ones should be spread; otherwise it could lead to quick discrediting and rejection of the results.

(iii) Analyze the clones starting from the ones which are "furthest away" from each other; clones which are within the same source code file are potentially less obstructive than the ones from two different subsystems. From the discussions with architects of Product C we could observe that these clones caused the most lively discussions that could lead to actions.

(iv) Once the clones are identified they should be monitored. Duala-Ekoko and Robillard [57] provide a tool which can be a complement to the analyses presented in this paper. This tool can allow for tracking of the clones which are not considered obstructive but should be updated in parallel.

Finally, we recommend using simple tools for the initial analyses. The simple tools produce results which are simple and help the organization to learn about the phenomena rather than get overwhelmed about the technical details of building abstract syntax trees and equivalence classes. This conclusion is also supported by the claims of Rattan et al. [3] who found that the token-based clone detection tools detect the most number of clones but are not straightforward in helping the designers to remove the clones. Their recommendation was to use tree-based approaches or text based approaches to start with.

9. Conclusions

The phenomenon of clones of source code in software products is known to the community and there exist a number of tools which can detect cloning. There is, however, no consensus whether cloning as a practice is really problematic. Reports about positive effects of code clones exist alongside reports of the negative effects. In this paper we recognized this fact and investigated whether there are different kinds of clones which are more obstructive than others. We set off to investigate the following research question:

RQ 1: Given the number of clones in large software systems, how to efficiently identify those which are potentially obstructive for the quality of the product?

The first element of our investigation was to find which clones are obstructive (RQ 1a) and in order to do that we have created a classification scheme based on the experience of the architects and designers working with us. They identified the location and the type of the code clone to be the two most important determinants of whether the clone was obstructive or not. We used that knowledge to create a classification scheme and to define an analogy-based classification for the clones (RQ 1b). One of the requirements of the answer to this research question was the principle of simplicity. The goal was to find a method which would provide a means of automatically classifying the majority of the clones (over 80%) and provide a direct feedback to the architects and designers on how to proceed with the found clones. Therefore we combined the experiences of the architects with the existing schemes (Pate et al. [11]) and designed a process of making an automated classification (described in Section 4.4). The results of applying this method showed that a limited number of example clones can be found and that we can use these examples to develop scripts to find similar examples and classify them (e.g., based on the location of the clone or the type of the code like state-machine).

We analyzed three products and found that cloning is a common practice in industrial software development (similar to the reported studies in the open source software). We have also found that most of the clones in industrial software development do not matter for the quality of the product in field. When applying clone detection on the entire code base (without prefiltering, similar to applying clone detection on entire code base of an open source product) we could observe that only 1% of the clones were obstructive (Table 2). In such a case we could also find that the simple script could classify 96% of clones leaving only 4% for manual classification.

We have also found that when spending a small amount of effort to choose the right modules (Product C, Section 5.5.3) almost no manual effort is needed and 91% of the found clones were potentially obstructive.

9.1. Further Work. In our future work we plan to expand the analyses to clones of Type II, Type III, and Type IV, introduce more advanced clone detection tools to the quality monitoring processes, and perform clone analyses on a daily basis which was shown to be successful in other projects with this organization, for example, [58, 59]. We also intend to analyze cloning patterns in test code as cloning of test code might have an important, complementary, influence on the quality of the final product.

We also plan to use more advanced distance measures (e.g., using static code analysis metrics) during the classification to increase the ratio between the clones classified automatically and the ones classified manually. We also plan to compare the performance of the automatic classification in order to investigate how much precision and recall it provides compared to a fully manual classification.

Finally, our classification scheme can be used together with mining software repositories (e.g., as advocated by Canfora et al. [60]) and automatically classify newly mined clones based on analogy to already found clones.

The classification scheme presented in our work can provide a possibility to direct management activities towards the obstructive clones or differentiate between management strategies of obstructive and nonobstructive clones. This kind of work would complement the work of Aversano et al. [36] who studied how code clones are maintained in practice and found that the majority of clones are updated consistently.

In the further work we also intend to study the practices of cloning between the project's boundaries.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

The authors would like to thank the managers and designers who helped in the study and supported their efforts. They would like to thank Ericsson for the possibility to conduct this study and for sharing the results with the community. This work has been partially supported by the Swedish Strategic Research Foundation under Grant no. SM13-0007.

References

[1] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 187–196, 2005.

[2] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," Tech. Rep. 2007-541, School of Computing, Queen's University, 2007.

[3] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: a systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[4] L. Zhao and S. Elbaum, "Quality assurance under the open source development model," *Journal of Systems and Software*, vol. 66, no. 1, pp. 65–75, 2003.

[5] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, and B. V. Saranya, "Detection of type-1 and type-2 code clones using textual analysis and metrics," in *Proceedings of the International Conference on Recent Trends in Information, Telecommunication, and Computing (ITC '10)*, pp. 241–243, IEEE, Kerala, India, March 2010.

[6] M. Staron, W. Meding, C. Hoglund, P. Eriksson, J. Nilsson, and J. Hansson, "Identifying implicit architectural dependencies using measures of source code change waves," in *Proceedings of the 39th Euromicro Conference Series on Software Engineering and Advanced Applications*, pp. 325–332, IEEE, Santander, Spain, September 2013.

[7] M. Staron, J. Hansson, R. Feldt et al., "Measuring and visualizing code stability—a case study at three companies," in *Proceedings of the Joint Conference of the 23rd International Workshop on Software Measurement and the 2013 8th International Conference on Software Process and Product Measurement (IWSM-MENSURA '13)*, pp. 191–200, Ankara, Turkey, October 2013.

[8] M. Staron and W. Meding, "Monitoring bottlenecks in agile and lean software development projects—a method and its industrial use," in *Product-Focused Software Process Improvement: 12th International Conference, PROFES 2011, Torre Canne, Italy, June 20–22, 2011. Proceedings*, vol. 6759 of *Lecture Notes in Computer Science*, pp. 3–16, Springer, Berlin, Germany, 2011.

[9] M. Staron and W. Meding, "Using models to develop measurement systems: a method and its industrial use," in *Software Process and Product Measurement: International Conferences IWSM 2009 and Mensura 2009 Amsterdam, The Netherlands, November 4–6, 2009. Proceedings*, vol. 5891 of *Lecture Notes in Computer Science*, pp. 212–226, Springer, Berlin, Germany, 2009.

[10] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proceedings of the 2nd Working Conference on Reverse Engineering*, pp. 86–95, Toronto, Canada, July 1995.

[11] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: a systematic review," *Journal of software: Evolution and Process*, vol. 25, no. 3, pp. 261–283, 2013.

[12] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 485–495, Vancouver, Canada, May 2009.

[13] M. Toomim, A. Begel, and S. L. Graham, "Managing duplicated code with linked editing," in *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, pp. 173–180, IEEE, Rome, Italy, September 2004.

[14] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 57–66, IEEE, Beijing, China, September 2008.

[15] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto, "Software quality analysis by code clones in industrial legacy software," in *Proceedings of the 18th IEEE Symposium on Software Metrics*, pp. 87–94, Ottawa, Canada, 2002.

[16] C. J. Kapser, *Toward an Understanding of Software Code Cloning as a Development Practice*, University of Waterloo, 2009.

[17] C. Kapser and M. W. Godfrey, "Aiding comprehension of cloning through categorization," in *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE '04)*, pp. 85–94, IEEE Computer Society, 2004.

[18] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, "Studying the impact of clones on software defects," in *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE '10)*, pp. 13–21, IEEE, Beverly, Mass, USA, October 2010.

[19] C. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful," in *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE '06)*, pp. 19–28, IEEE, Benevento, Italy, October 2006.

[20] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Proceedings of the International Symposium on Empirical Software Engineering (ISESE '04)*, pp. 83–92, August 2004.

[21] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 106–115, Minneapolis, Minn, USA, May 2007.

[22] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: maintenance support environment based on code clone analysis," in *Proceedings of the 8th IEEE Symposium on Software Metrics*, pp. 67–76, 2002.

[23] E. Duala-Ekoko and M. P. Robillard, "Clone tracker: tool support for code clone management," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 843–846, Leipzig, Germany, May 2008.

[24] H. Li and S. Thompson, "Clone detection and removal for erlang/OTP within a refactoring environment," in *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Program Manipulation*, pp. 169–178, ACM, January 2009.

[25] H. Li and S. Thompson, "Similar code detection and elimination for erlang programs," in *Practical Aspects of Declarative Languages*, vol. 5937 of *Lecture Notes in Computer Science*, pp. 104–118, Springer, Berlin, Germany, 2010.

[26] H. Li and S. Thompson, "Incremental clone detection and elimination for erlang programs," in *Fundamental Approaches to Software Engineering: 14th International Conference, FASE 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, vol. 6603 of *Lecture Notes in Computer Science*, pp. 356–370, Springer, Berlin, Germany, 2011.

[27] C. Brown and S. Thompson, "Clone detection and elimination for Haskell," in *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '10)*, pp. 111–120, Madrid, Spain, January 2010.

[28] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: a study of developer work habits," in *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pp. 492–501, ACM, Shanghai, China, May 2006.

[29] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones," *Empirical Software Engineering*, vol. 15, no. 1, pp. 1–34, 2010.

[30] Y. Yamanaka, E. Choi, N. Yoshida, K. Inoue, and T. Sano, "Applying clone change notification system into an industrial development process," in *Proceedings of the 21st International Conference on Program Comprehension (ICPC '13)*, pp. 199–206, IEEE, San Francisco, Calif, USA, May 2013.

[31] Y. Dang, S. Ge, R. Huang, and D. Zhang, "Code clone detection experience at Microsoft," in *Proceedings of the 5th International Workshop on Software Clones (IWSC '11)*, pp. 63–64, Honolulu, Hawaii, USA, May 2011.

[32] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '10)*, pp. 1–9, IEEE, Timişoara, Romania, September 2010.

[33] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems: a case study," *Journal of Software Maintenance and Evolution*, vol. 18, no. 2, pp. 61–82, 2006.

[34] S. Grant and J. R. Cordy, "Vector space analysis of software clones," in *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*, pp. 233–237, Vancouver, Canada, May 2009.

[35] Y. Higo, S. Kusumoto, and K. Inoue, "A metric–based approach to identifying refactoring opportunities for merging code clones in a Java software system," *Journal of Software Maintenance and Evolution*, vol. 20, no. 6, pp. 435–461, 2008.

[36] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: an empirical study," in *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR*

'07), pp. 81–90, IEEE, Amsterdam, The Netherlands, March 2007.

[37] L. Yujian and L. Bo, "A normalized Levenshtein distance metric," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.

[38] C. Robson, *Real World Research*, Blackwell Publishing, Oxford, UK, 2nd edition, 2002.

[39] P. Tomaszewski, P. Berander, and L.-O. Damm, "From traditional to streamline development—opportunities and challenges," *Software Process Improvement and Practice*, vol. 13, no. 2, pp. 195–212, 2008.

[40] E. Richardson, "What an agile architect can learn from a hurricane meteorologist," *IEEE Software*, vol. 28, no. 6, pp. 9–12, 2011.

[41] H. Sharp, N. Baddoo, S. Beecham, T. Hall, and H. Robinson, "Models of motivation in software engineering," *Information and Software Technology*, vol. 51, no. 1, pp. 219–233, 2009.

[42] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, 1992.

[43] R. Feldt, M. Staron, E. Hult, and T. Liljegren, "Supporting software decision meetings: heatmaps for visualising test and code measurements," in *Proceedings of the 39th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '13)*, pp. 62–69, IEEE, Santander, Spain, September 2013.

[44] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[45] N. Gode, "Evolution of type-1 clones," in *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*, pp. 77–86, IEEE, Edmonton, Canada, September 2009.

[46] M. F. Zibran, R. K. Saha, C. K. Roy, and K. A. Schneider, "Evaluating the conventional wisdom in clone removal: a genealogy-based empirical study," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC '13)*, pp. 1123–1130, ACM, Coimbra, Portugal, March 2013.

[47] N. Mellegård, M. Staron, and F. Törner, "A light-weight defect classification scheme for embedded automotive software and its initial evaluation," in *Proceedings of the IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12)*, pp. 261–270, Dallas, Tex, USA, November 2012.

[48] M. Staron and C. Wohlin, "An industrial case study on the choice between language customization mechanisms," in *Product-Focused Software Process Improvement: 7th International Conference, PROFES 2006, Amsterdam, The Netherlands, June 12–14, 2006. Proceedings*, vol. 4034 of *Lecture Notes in Computer Science*, pp. 177–191, Springer, Berlin, Germany, 2006.

[49] M. Staron, W. Meding, and K. Palm, "Release readiness indicator for mature agile and lean software development projects," in *Agile Processes in Software Engineering and Extreme Programming*, vol. 111 of *Lecture Notes in Business Information Processing*, pp. 93–107, Springer, Berlin, Germany, 2012.

[50] C. Kapser and M. W. Godfrey, "Improved tool support for the investigation of duplication in software," in *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 305–314, Budapest, Hungary, September 2005.

[51] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC '08)*, pp. 172–181, Amsterdam, The Netherlands, June 2008.

[52] C. K. Roy and J. R. Cordy, "Near-miss function clones in open source software: an empirical study," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.

[53] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," in *Proceedings of the 4th Working Conference on Reverse Engineering*, pp. 44–54, IEEE, Amsterdam, The Netherlands, October 1997.

[54] V. Antinyan, M. Staron, W. Meding et al., "Identifying risky areas of software code in Agile/Lean software development: an industrial experience report," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering, Software Evolution Week (CSMR-WCRE '14)*, pp. 154–163, IEEE, Antwerp, Belgium, February 2014.

[55] V. Antinyan, M. Staron, W. Meding et al., "Monitoring evolution of code complexity in Agile/Lean software development—a case study at two companies," in *Proceedings of the 13th Symposium on Programming Languages and Software Tools (SPLST '13)*, pp. 1–15, Szeged, Hungary, August 2013.

[56] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslèn, *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, Boston, Mass, USA, 2000.

[57] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*, pp. 158–167, Minneapolis, Minn, USA, May 2007.

[58] M. Staron, W. Meding, and C. Nilsson, "A framework for developing measurement systems and its industrial evaluation," *Information and Software Technology*, vol. 51, no. 4, pp. 721–737, 2009.

[59] M. Staron, W. Meding, G. Karlsson, and C. Nilsson, "Developing measurement systems: an industrial case study," *Journal of Software Maintenance and Evolution*, vol. 23, no. 2, pp. 89–107, 2011.

[60] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories," in *Proceedings of the IEEE 4th International Workshop in Mining Software Repositories (MSR '07)*, p. 14, Minneapolis, Minn, USA, May 2007.