SAND2012-1251C

Clones: Underlying Patterns throughout the Software Lifecycle

Submission Version

***Abstract*—While there has been marked interest in identifying redundant patterns in software and their effect on software maintainability and stability, the ability to efficiently and effectively identify this redundancy is still an open area of research. During the study of a large, industrial project in which we analyzed the applicability of using design metrics as predictors of change-proneness in UML designs and traced the UML design objects to final Java code representation, patterns of identical metric values for classes and their implementation became evident. We label our technique for identifying these *metric clones* as *latent metric clone analysis*. Analyzing additional industrial study data, we found that as much as 50% of both the design modules and the implementation modules were metric clones. Additionally, due to the fact we consider multiple phases of the software lifecycle, we can efficiently identify design clones throughout development and match them to their implementation. As part of our evaluation, we were able to compare software module stability via change orders to the co-occurrence of metric clones. Our analysis suggests that module stability is fundamentally coupled to a clone and non-clone pattern. We find that clones are not inherently detrimental as previous research has conjectured, but knowledge of their existence is extremely useful to practitioners as they redesign or maintain large-scale software systems.**

***Keywords*-Clones, Metrics, Latent Metric Clone Analysis, Patterns in Software Lifecycle**

I. Introduction

Duplicate or similar code, often referred to as code clones, is a common occurrence in commercial software systems. Depending on their context, these clones may or may not adversely impact the development and stability of software engineering projects. Multiple studies have confirmed this duplicitous nature of the cloning phenomenon [1]–[9]. Clones can also be introduced unintentionally or intentionally through independent development efforts. Clones may be intentionally created to improve program reliability or development speed. The same functionality may be intentionally duplicated to minimize dependencies to deliver individual control. On the other hand, the process of applying design patterns (e.g., pattern-driven development on component platforms such as JEE and .NET) increases the possibility of unintentional cloning. A survey of clone research by Koschke lists many other root causes for software clones [10].

Since many of the previous studies demonstrate that clones are present in higher-level languages such as Java and C++, it is not surprising that clones exist in other development products, such as UML models. It is generally believed that the use of modeling tools will increase over time and the industrial standard for object-oriented software analysis and design will be UML [11]. Development will evolve into a design activity as new, high-level languages emerge and generative techniques improve. A poor design with clones will result in rework, slower development productivity and a decrease in the software's future extendibility. These trends indicate the necessity of improved modeling skills and continuous assessment of the design, including tracking clones throughout the software lifecycle.

Independent of the reason for their existence, the identification of clones in the design and implementation phase is important. Clones resulting from bad designs impact quality and maintainability [7]. It is well known that correcting an error encountered by an end-user is an order of magnitude more expensive than when finding it in earlier phases [12]. In this same light, identifying design clones and tracking them through implementation offers valuable insights into their behavior and the consequential development actions.

Approaches to identifying clones are an active research topic. Most of techniques involve comparison of the identified artifact expressed as text, tokens, an abstract syntax tree, a program dependence graph, metrics and other direct or abstracted structures or measures. The distance between each clone comparison can range from exact to various degrees of precision. Each technique holds both benefits and drawbacks in identifying exact copies, syntactically identical copies and copies with modifications. The metrics-identified clones are rated as the best choice in identifying all types of clones and can scale up for evaluating large systems [10].

In our research, we have developed a metrics-based approach for analyzing software designs (during both design and implementation) that can help designers engineer quality into the product. From our analysis of large industrial projects, we have discovered that within a software system are hidden relationships and structures that can be illuminated by evaluating and measuring software development artifacts. These relationships can be used to answer questions about the stability of the software product and perhaps, more importantly, guide software development techniques. Software development is itself a pattern-selecting and pattern-making process and the development pattern is inherent in the structure and execution of the software. The coherent patterns or arrangements of modules that are gradually realized become effective and ontologically significant by virtue of their development. To uncover these patterns, we calculate, collect, and analyze a multitude of metrics. For example, we collect 155 UML and 113 Java primitive and composite metrics. In analyzing the design (UML) and code (Java) of two large industrial projects, modules within each respective set were identified

that possessed exactly the same values for all of the calculated metrics. We label our technique for identifying these metric clones as latent metric clone analysis. One class or method is a metric clone of another if both possess exactly the same values for all of the metrics considered. When matching the model data to the code data, entire class patterns emerged. To distinguish individual model clones and code clones, a label of class implementation clone was given to the set of modules. Our contributions include:

- We present a scalable metric-based technique for identifying clones in both software design objects (classes) and the final code representation. Not only can we identify clones in a single stage of the software lifecycle, we can track them as they evolve during the development process.
- Our work, based on the examination of large-scale industrial systems and complete change order histories, lend further credence to the belief that the impact of software clones is dependent on their context and not uniformly harmful. In fact, we find that many clones are more stable than non-clone code.

The rest of the paper is organized as follows: We discuss related work in Section II and provide an overview of our metrics collection and analysis process in Section III. In Section IV, we present experimental results demonstrating the effectiveness of our technique at identifying clones, discuss the validity of our results in Section V, and conclude in Section VI.

II. Related Work

The recent interest in code clones and software redundancy has led to a significant amount of research in related areas. In this section, we review previous work in three related areas: the effect of clones on software projects, methods for clone detection, and change tracking through the software lifecycle.

A. The Effects of Software Redundancy

As the prevalence of cloned code has been demonstrated by multiple studies on a variety of systems [10], [13], a large number of studies have been conducted in order to understand the effects of this redundancy on large-scale software systems [1]–[8]. These works fall primarily in two categories, with the first being those that demonstrate clones are inherently bad for software stability and maintainability and the second showing clones are essentially neutral and do not adversely affect a software system.

In the work by Lozano *et al*. [3], they developed a lexical-based clone tracking tool called CloneTracker to determine the frequencies of changes to both methods that have clones and those that do not. The study demonstrated that cloned methods change much more frequently than the methods without clones, supporting the argument that clones are change inconsistently and are harder to maintain than non-clone code. Other work by Geiger *et al*. [2] studied the relation of clones (at the file level) to the changes related to these files and found that change to clones are inconsistently applied and may at a later time need to be propagated, incurring extra maintenance costs. Kim *et al*. [1] investigated the evolution of code clones and showed that consistent changes to the code clones of a group are fewer than anticipated, leading to a higher maintenance cost. A similar study by Krinke [5] looking at five different open source projects demonstrated that that identified clones are inconsistently modified and prone to potential instability. A recent study that works on industrial code shows that inconsistent changes to code duplicates are frequent and lead to severe unexpected behavior [7].

In contrast to the above studies, recent studies have cast doubts on the fact that clones are always harmful, but instead posit that developers and testers need to be aware of the relationships in the code. A study by Aversano *et al*. [4] found that "the majority of clone classes are always maintained consistently" with the caveat that in some cases the lack of consistent changes can indeed leads to bugs and higher maintenance costs. Bettenburg *et al*. [6] studied the inconsistent changes of clones at the release level. They noted that the number of defects due to inconsistent changes in clones is substantially lower at the release level than at the revision level and over time cloned code appeared to be more stable. Rahman *et al*. [8] analyzed several large open source projects and state that cloned code is not inherently "smelly" or difficult to maintain.

Unlike previous works which use small code bases [3] or open source projects [2], our work used large-scale industrial software projects to verify our results. Additionally, we are able to leverage software lifecycle artifacts and true change orders as opposed to data mined from versioning software or manual annotations [6].

B. Software Clone and Redundancy Detection

Previous work in the detection of software redundancy and clones generally falls into one of three areas: lexical-based [14], [15], graph-based [16]–[19], and metrics-based detection [20], [21]. CCFinder [14] and CP-Miner [15] identify clones by transforming the software program according to language-specific rules into tokens and matching subsequences of tokens. While lexical techniques are easy to adapt to various languages, they often have trouble detecting clones produced by natural editing patterns [26]. Graph- based detection techniques such as those by Komondoor and Horwitz [16] and Krinke [17] transform the program into dependency graphs (PDG) using the control flow and data flow information of a program and use isomorphic subgraph matching algorithms to detect clones. Similar to PDG-based algorithms, CloneDR [19] and the work by Jiang *et al*. [18] create abstract syntax trees which are compared for similarity. While graph-based clone detection techniques have high detection rates, they generally are not scalable to large systems. The work by Mayrand *et al*. [21] and Merlo *et al*. [20] identifies redundant code by computing and comparing metrics based on intermediate representations of source code. These metrics capture information about the architecture, data types, control flow, and data flow. While our work is similar to the last two, we

collect metrics from both the design model and code base and able to track the clones and changes through the software lifecycle.

C. Tracking Software Changes

Recently, there has been interest in tracking clones through a code base as it ages. Saha *et al*. [22] created a clone genealogy extractor to track changes through multiple releases of seventeen open source programs written in multiple programming languages. They found that the majority of software clones either propagate without major syntactic changes or change consistently in the subsequent releases and this was not generally affected by implementation language choice. Krinke [23] studied the evolution of code in terms of clone stability and concluded that cloned code is more stable than non-cloned code. CloneTracker is a tool by Duala-Ekoko and Robillard [24] that tracks clones during software development by creating an abstract model of different sections and notifying a developer when sections with similar abstract models are edited. In our work, we look at the beginning stages of the software lifecycle and track how clones evolve from the design documents to become realized in executable code. Additionally, we are able to track a history of changes not only from repository information, but also from customer-driven change orders.

III. Analysis System and Experimental Setup

Improvements in the software development process depend on our ability to collect and analyze data drawn from the various phases of the development lifecycle. We have developed a metrics-guided methodology for software analysis that begins with architectural design. To support this endeavor, a tool architecture as seen in Figure 1 provides a clear separation of the five subsystems needed to analyze the study data: the External Data Collection (external source and process information), the Repository System (processed representations), the Metrics Engine, the Analysis Engine and Report Generation. The architecture is both extensible and flexible, allowing for the incorporation of new analysis techniques and user-requested functionality. To concentrate on the software analysis, a lightweight incremental approach is featured to complement specific analysis techniques. Collection and storage are separated to extend future analysis to the various sources of information about a software system.

Program representation is a critical issue in software analysis and software re-engineering. It heavily relates to the portability and effectiveness of the software analysis tools that can be developed. Our approach focuses on design artifacts and source code representation in the form of an XML document. For each representation, we create a data type definition (DTD) and the source file is translated into an XML file using that document structure. The DTD for Java, C/C++, Ada, UML XMI and SDL have been created, although Figure 1 displays only a subset of these for brevity. Polylingual projects can also be analyzed. The software professional selects whether or not to merge the analysis of the supported representations in multi-language projects. The environment bridges the analysis of metrics for the design and development phases of software projects. It will trace the UML design objects (classes and methods) to the final code representation (classes and methods).

Two of the design metrics developed in our research are an external design metric D_e and an internal design metric D_i. The calculation of D_e focuses on a module's external relationships to other modules in the software system and is based on information available during architectural design. The metric D_i incorporates factors related to a module's internal structure and is calculated during the detailed design phase of software development. These metrics gauge project quality as well as design complexity during the design phase and subsequent phases thereafter. The primary objective is to employ these metrics to identify fault-prone modules, or stress points, in the structure of the software. Each metric can be used independently depending on the lifecycle phase or they can be combined to determine the stress-points and overall quality of the product. Thus, in the implementation phase, the metrics can be used to focus verification and validation activities. For testing, the metrics can assist in determining where testing effort should be focused and the types of test strategies to employ. Further, it has been shown that $D(G)$ - the linear combination D_e and D_i - can be used to predict and measure the impact of design decisions on the expected reliability of a software product during the code and test phases of development [25]. Applying $D(G)$, the concept of design significance becomes a viable and measurable attribute that helps to identify those components for which design decisions would have a significant effect on the expected reliability of the software [26]. In the twenty years of metrics validation, on a wide variety of projects ranging from missile defense, satellite, financial and telecommunications systems, to interactive games, the design metrics have identified at least 75% of the fault-prone components 100% of the time with very few false positives [12], [27]–[31].

A discussion of model-level metrics is incomplete without considering the relationship between similar metrics at the corresponding code level. In practice, the UML model represents the design stage of a system. As implementation continues, the model is updated. In this context, the difference between the design metrics extracted from the UML model and design metrics extracted from the source code will reflect properties of the evolution of the system, rather than exclusively the properties of the design. Even when models and implementation are synchronized, there will be a difference between metric values. The UML design metrics are calculated from the UML class diagram and, therefore, are collected at the class level of abstraction.

Framed by the development process, the UML model design metrics and the source design metrics can provide developers with a comprehensive analysis of the delivery excellence of their projects. By using the design metrics as the common scale, an overall delivery excellence rating can be defined and performance consistently analyzed and trended. Monitoring

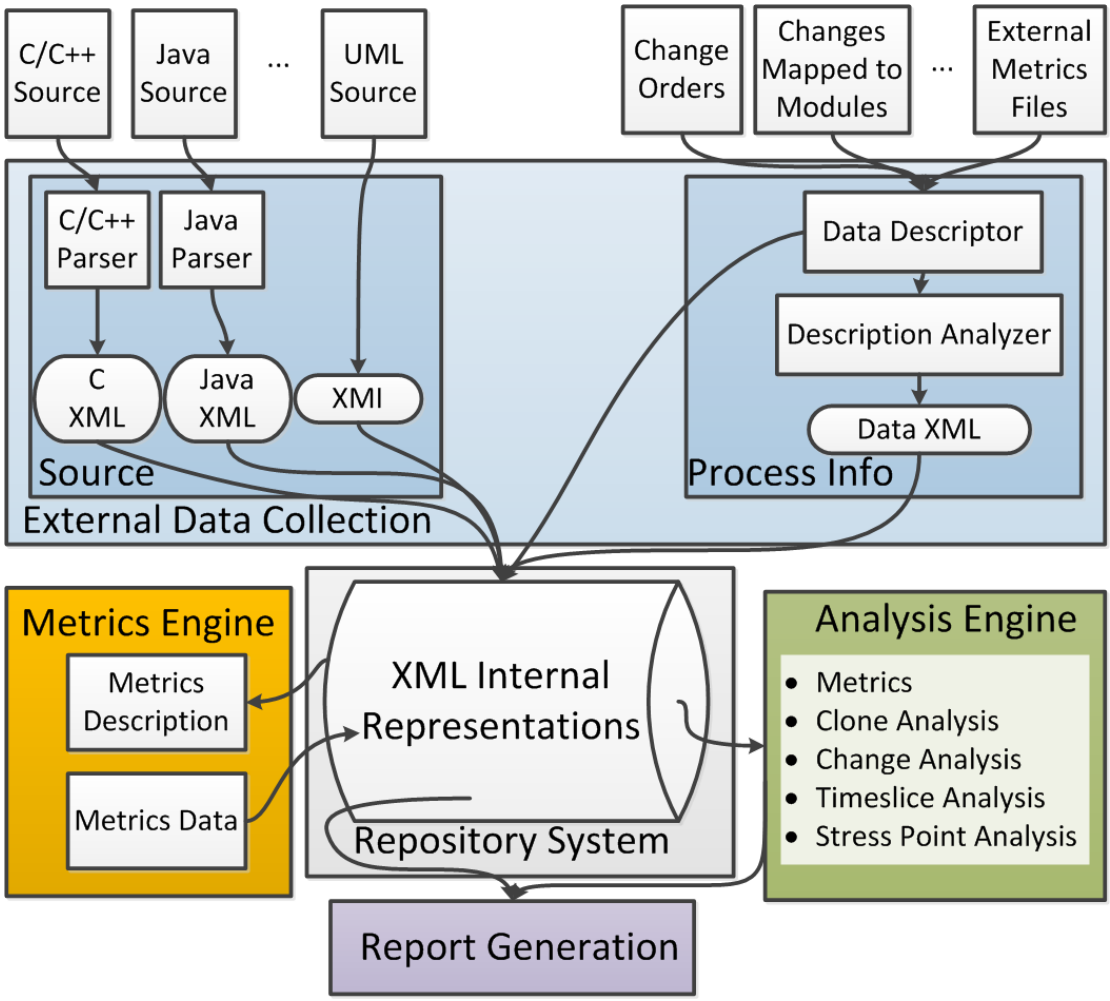


Fig. 1. Tool Architecture to Analyze Software Systems

these trends from the design phase of development to the coding phase, a final product of enhanced quality and reduced costs, due to problems being exposed earlier in the project, can be delivered. This matching provides the missing link between the artifacts of design and implementation of a software product. To our knowledge, there has been little work exploring the relationship between the model of a software system and the corresponding code through the evaluation of similar metrics (design metrics) at each level of abstraction.

A. *Other Clone Inferences*

For every analyzed project, changes stemming from the change report are bound to a module. Examining the process information for each module has the potential to tailor activities to individual sets of modules rather than a one-size fits all approach. In previous studies, change reports collected as early as the requirements phase were provided. Such thorough fault reporting is most helpful in determining the origin and resolution of faults in the development process.

Specifically, we have been working within the domains of UML models, large software repositories, software metrics, change order systems, social/professional feedback and intrusion detection through alerts. Each of these domains has a catalog of information to extract. A lightweight ontology mechanism was introduced, labeled as Data Descriptor in Figure 1, to manage the similarities and differences of each data set. The Description Analyzer (Figure 1) contains input/output APIs that encapsulate and separate domain objects, enabling tasks to be completed in a scalable, efficient and error-free method while permitting these tasks to be populated with extra features when necessary. The Analysis Engine shown in Figure 1 has the capability to analyze a system with respect to clones, changes, changes over time (time slices) and stress points. For this paper, we focus on clone analysis.

IV. Clone Analysis Results

The goal of our software analyses was to assess the predictive ability of the design metrics in identifying change-prone components in the UML design, plus trace the UML design objects (classes and methods) to the final code representation (Java classes and methods) to uncover relationships and clone patterns between the UML design and the Java code. By defi-

	Project 1				Project 2			
	design		code		design		code	
	#	%	#	%	#	%	#	%
# of mods including clones	4846		16124		754		1261	
# metrics collected	156		114		156		114	
# of unique modules	3119	64%	8045	50%	168	22%	445	35%
unique mods w/wo changes	668 2451	21% 74%	2802 5243	35% 65%	95 73	57% 43%	445 0	100% 0%
# of clones	1727	36%	8079	50%	586	78%	816	65%
# of clone groups	334		1255		44		113	
clone group w/wo changes	13 321	4% 96%	743 512	59% 41%	13 31	30% 70%	113 0	100% 0%
total changed modules	705	15%	10080	63%	178	24%	1261	100%
total changes	1803		11998		2907		2023	

Fig. 2. Overall Project Data

Group 21: DMA Clone ID: 2001234 Number of Clones 4

MetricValues																														
48	11	59	1	0	2	2	6	7	1	6	6	5	0	0	0	0	0	6	0	0	0	1	0	0	0	.	.	.		

Module Name/ID	Type	Changes
ProtectedAreaPanel	MEMBER_FUNCTION	2
FriendlyUnitCheckPanel	MEMBER_FUNCTION	2
TargetManager	MEMBER_FUNCTION	32
AlertsPanel	MEMBER_FUNCTION	1

Fig. 3. A Clone Group Example

nition, the design artifacts are not complete and only represent an early model of the actual system to be developed. For this reason, analyses of the metrics from both the UML model and the source code were completed to draw conclusions concerning the relationship between models of the system and the corresponding code. Our analyses consisted of three separate examinations: the metrics analysis of the XMI data; the metrics analysis of the Java data; and the matched XMI and Java metrics.

Two project analyses using the above outline have been completed. Project 1 is a complete system. Project 2 is a subset of a larger, older system. In analyzing the design (UML design) and code (Java) of both Projects 1 and 2, modules within each respective set were identified that possessed exactly the same values for all of the metrics under consideration, i.e., module clones. This redundancy does not allow one to distinguish one module from the other and, therefore, for our analysis, clones are grouped as one entity. To identify the clone group of a module, a unique number was assigned to each group by the tool. The clones are assigned numbers from our Analysis Engine (Figure 1).

A. Metrics Analysis of XMI Data

This analysis revealed that Project 1's UML design contained 334 groups of clones accounting for 1727 modules from the total of 4846 modules, which means that 36% of the modules were design clones (Figure 2). Change Orders (COs) were distributed to design classes. Of these 334 clone groups only 4% or 13 of the clone group had COs. The percentage of total modules changed during design for Project 1 was low at 15% for a project of this size. The 13 clone groups represent 37 changed clones and thus 705 total changed modules in design (as seen in Figure 2). In Project 2's design phase, 78% of the modules were identified as clones. With a higher percentage of clone modules in Project 2, it seems plausible that the number of clones with changes would also increase. As shown in Figure 2, design clone groups from Project 2 had 30% of its clone groups identified with changes. In both studies, design clone modules had fewer changes than non-clone (unique) modules.

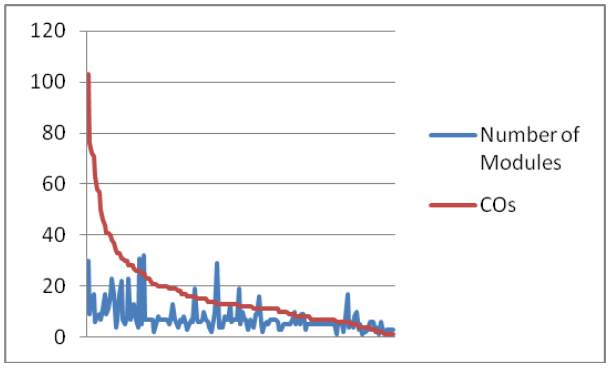
B. Metrics Analysis of Java Data

Many code modules in Projects 1 and 2 were also clones at 50% and 65% respectively. In the coding phase many more clones groups were identified with changes at 59% and 100%. For each project, the non-clone and clone modules were categorized by COs for both design and code. When observing the full spectrum of COs, the clone groups are clustered at the lower end of the CO range, while the higher end of COs are void of clones. This pattern was observed in both the design and code phases. These data suggest that both ends of the CO spectrum are fundamentally coupled to a clone or non-clone pattern.

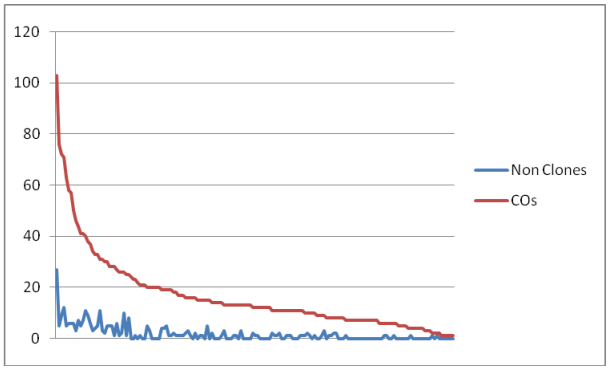
C. Other Clone Inferences

A clone group can contain modules with different CO values. For example, in Figure 3, the design clone group 2001234 containing 4 modules has COs ranging from 1 to 32. Such inconsistencies in the CO count require a review. Similar to design clones, a code module clone group can also contain modules identified with a different number of COs. Naturally, as the number of clones in a group increases, the probability that all of the modules in the clone group possess the same CO count decreases.

Clones can skew metric data resulting in poorer analysis results. Figure 4(a) displays the presence of noisy data produced by clones and Figure 4(b) displays the data when clones are



(a) All Modules



(b) Non-clone Modules

Fig. 4. Number of modules per class listed in descending order by change order value

grouped. Shown in Figure 4(a) is the relationship between the CO value of the class and the number of modules within the class. The horizontal axis lists the number of modules per class in decreasing order by the CO count. As seen in Figure 4(a), there is a mild positive correlation (.5) that classes with more modules have more COs. In Figure 4(b), removing the clone modules from the number of modules in each of the classes results in an increased positive correlation (.9) between COs and non-clone modules. The relationship between clones and COs was also evident in the UML classes where non-clone classes possessed the higher concentrations of COs versus their clone counterparts.

D. Matched XMI and Java Metrics

The methods from Projects 1 and 2 were matched with their respective class from the XMI information. When matching the design data to the code data, entire class patterns of clones were uncovered. To distinguish individual design class clones and code module clones, a label of class implementation clone was given to a set of modules. An example of a class implementation clone is provided in Figure 5. The design clone, identified as pattern 2019, the first row of the table, was repeated 29 times within the UML design of the project. When matched with its code modules, the design pattern 2019 consisted of six virtually identical code patterns. Pattern 2019-1, displayed in Figure 5, lists the design record at the top of the table and the five Java modules contained in that class, namely Java clones with Clone IDs 1001,1002,1003,1017 and 1021.

For the remaining five patterns, one module is exchanged for another. For example, in pattern 2019-2 (not shown here) clone module 1021 is replaced with an extra copy of clone 1003 resulting in patterns 1001,1002, 1003,1017 and a extra copy of 1003. There are eight class implementation clones with the 2019-2 pattern. Pattern 2019-3 replaces a non-clone module for either clone module 1003 or clone module 1021 depending on the comparison. In the final three patterns, other minor variations are found. Observing these patterns draws attention to the variation of COs among almost identical classes. For example, in the instances where a non-clone module replaced a clone module, the COs among the variations of the patterns fluctuate more than variations where a clone is swapped for anther clone. In our analysis, we also observed that the class implementation clone pattern underscores the variation of COs among almost identical class designs.

V. Threats to Validity

In this section, we discuss the threats to validity that could adversely affect the results we present and the generalization of our technique.

A. Construct Validity

Our paper focuses on determining metric clones and only considers two modules to be clones if all of the metrics are identical, minimizing the potential for false positives. Our work has presented evidence that changes are required more often for non-clone modules. In Section IV, we show a strong positive correlation between the change orders and non-clone modules. This correlation is demonstrated for both the UML classes and the finished Java code.

B. Internal Validity

The change orders were collected from the corporate repository for software changes and represent a complete set of modifications made to the project. As each change order is well documented and directly associated with a module, we do not have to worry about errors induced by linking changes and bug fixes to potential locations in the code. Not only do we have the advantage of well-organized change orders, but also we have the completeness of change orders over the entire life of the system.

C. External Validity

In order to keep our findings and methodology as generalizable as possible, our work has examined multiple industrial software projects and found consistent results across those projects. Additionally, as the change orders are the mandated way to incorporate changes and bug fixes to the industrial systems, we claim the change orders represent the issues that are typically associated with a typical industrial project.

#	Module NAME/ID	TYPE	Clone	Clone ID	Changes
	_15_5_8ce027f_1221744687687_615554_760	CLASS NORMAL	Yes	2019	7
Methods					
1	_15_5_8ce027f_1221744687687_615554_760	MEMBER_FUNCTION	Yes	1021	
2	_15_5_8ce027f_1221744687687_615554_760	PROTOTYPE_CONSTRUCTOR	Yes	1002	
3	_15_5_8ce027f_1221744687687_615554_760	MEMBER_FUNCTION	Yes	1017	
4	_15_5_8ce027f_1221744687687_615554_760	MEMBER_FUNCTION	Yes	1003	
5	_15_5_8ce027f_1221744687687_615554_760	CLASS	Yes	1001	

Fig. 5. Class Implementation Clone: Pattern 2019-1

VI. Conclusion

Our analyses suggest that both ends of the change order spectrum, namely those modules with few or no changes and those modules with many changes, are fundamentally coupled to a clone or non-clone pattern. When matching the XMI data to the Java data, entire class patterns emerged. These patterns underscore the variation of COs among almost identical class designs. Through this study, we presented the effectiveness of the evaluation of design to implementation, allowing for continuous assessment. These results also imply that early class categorization can correctly identify problem methods later in development.

The foundation of our previous software design research is the result that the design metrics De and Di are excellent predictors of fault-prone components. Cloning can distort this design metric technology as it introduces duplicate values, throwing off stress-point thresholds leading to stress points that may not be fault-prone. We have seen in this study that clones can permeate the lifecycle phases from analysis through implementation. Awareness of these patterns will ensure the applicability of these design metrics to systems with significant cloning.

We believe as stated in the introduction that there is an abundance of information that can be gathered from each of the products produced during software development (specifications, design, code, test cases, etc.). Unlike traditional metrics tools that tell you specific information about the current product and essentially provide a record of the past, our direction is to use latent metric clone analysis as predictive in a proactive process. This study emphasizes that much can be learned from metrics and also much can be expected from their use.

VII. Acknowledgments

This research is based upon work supported by the National Science Foundation under Grant No. 0968959.

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] R. Geiger, B. Fluri, H. Gall, and M. Pinzger, "Relation of code clones and change couplings," *Fundamental Approaches to Software Engineering*, vol. 3922, pp. 411–425, 2006.

[3] A. Lozano, M. Wermelinger, and B. Nuseibeh, "Evaluating the harmfulness of cloning: A change based experiment," in *4th International Workshop on Mining Software Repositories*, 2007.

[4] L. Aversano, L. Cerulo, and M. Di Penta, "How clones are maintained: An empirical study," in *11th European Conference on Software Maintenance and Reengineering*, 2007.

[5] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *14th Working Conference on Reverse Engineering*, 2007.

[6] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *16th Working Conference on Reverse Engineering*, 2009.

[7] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *31st International Conference on Software Engineering*, 2009.

[8] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *7th IEEE Working Conference on Mining Software Repositories*, 2010.

[9] S. Jarzabek and Y. Xue, "Are clones harmful for maintenance?" in *4th International Workshop on Software Clones*, 2010.

[10] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings on Duplication, Redundancy, and Similarity in Software*, 2007.

[11] H. Störrle, "Towards clone detection in uml domain models," in *4th European Conference on Software Architecture: Companion Volume*, 2010.

[12] D. Zage and W. Zage, "An analysis of the fault correction process in a large-scale sdl production model," in *25th International Conference on Software Engineering*, 2003.

[13] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, pp. 470–495, 2009.

[14] 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, pp. 654–670, 2002.

[15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code," in *6th Conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[16] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," *Static Analysis*, vol. 2126, pp. 40–56, 2001.

[17] J. Krinke, "Identifying similar code with program dependence graphs," in *8th Working Conference on Reverse Engineering*, 2001.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering*, 2007.

[19] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *International Conference on Software Maintenance*, 1998.

[20] E. Merlo, G. Antoniol, M. Di Penta, and V. Rollo, "Linear complexity object-oriented similarity for clone detection and software evolution analyses," in *IEEE International Conference on Software Maintenance*. IEEE, 2004, pp. 412–416.

[21] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *International Conference on Software Maintenance*, 1996.

[22] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider, "Evaluating code clone genealogies at release level: An empirical study," in *10th IEEE Working Conference on Source Code Analysis and Manipulation*, 2010.

[23] J. Krinke, "Is cloned code more stable than non-cloned code?" in *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2008.

[24] E. Duala-Ekoko and M. Robillard, "Tracking code clones in evolving software," in *29th International Conference on Software Engineering*, 2007.

[25] W. Zage and D. Zage, "Smart: Security measurements and assuring reliability through metrics technology," in *21st Annual Systems and Software Technology Conference*, 2009.

[26] J. Stineburg, W. Zage, and D. Zage, "Measuring the effect of design decisions on software reliability," in *International Society of Software Reliability Engineers (ISSRE)*, 2005.

[27] W. Wong, J. Horgan, M. Syring, W. Zage, and D. Zage, "Applying design metrics to predict fault-proneness: a case study on a large-scale software system," *Software: Practice and Experience*, vol. 30, pp. 1587–1608, 2000.

[28] W. Zage and D. Zage, "Relating design metrics to software quality: Some empirical results," SERC Technical Report 74-P, Tech. Rep., 1990.

[29] ——, "Evaluating design metrics on large-scale software," *IEEE Software*, vol. 10, pp. 75–81, 1993.

[30] W. Zage, D. Zage, J. McGrew, and N. Sood, "Using design metrics to identify error-prone components of sdl designs," in *9th SDL Forum*, 1999.

[31] W. Zage and D. Zage, "Metrics directed verification of uml designs," SERC Technical Report 281, Tech. Rep., 2006.