Conflict-aware Optimal Scheduling of Code Clone Refactoring: A Constraint Programming Approach

Minhaz F. Zibran Chanchal K. Roy
Department of Computer Science, University of Saskatchewan, Saskatoon, SK, Canada S7N 5C9
Email: {minhaz.zibran, chanchal.roy}@usask.ca

***Abstract*—Duplicated code, also known as code clones, are one of the malicious 'code smells' that often need to be removed through refactoring for enhancing maintainability. Among all the potential refactoring opportunities, the choice and order of a set of refactoring activities may have distinguishable effect on the design/code quality. Moreover, there may be dependencies and conflicts among those refactorings. The organization may also impose priorities on certain refactoring activities. Addressing all these conflicts, priorities, and dependencies, manual formulation of an optimal refactoring schedule is very expensive, if not impossible. Therefore, an automated refactoring scheduler is necessary, which will maximize benefit and minimize refactoring effort. In this paper, we present a refactoring effort model, and propose a constraint programming approach for conflict-aware optimal scheduling of code clone refactoring.**

I. Introduction

Duplicated code, or code clone is a well-known code smell. From the maintenance perspective, existence of code clones may increase maintenance effort and so, the amount of code clones should be minimized by applying active refactoring. There are many refactoring patterns, all of which are not directly applicable to code clone refactoring. The applicability of certain refactoring activities largely depends on the context. So, for code clones, refactoring activities and the relevant contexts need to be identified in the first place. The consequence of clone refactoring should also be taken into account. The effort required for applying certain refactoring on the underlying code clones should also be minimized to keep the maintenance cost within reach. The application of a subset of refactoring from a set of applicable refactoring activities may result in distinguishable impact on the overall code quality. Moreover, there may be sequential dependencies and conflicts among the refactoring activities. Hence, it is also necessary that, from all refactoring candidates a subset of non-conflicting refactoring activities be selected and ordered (for application) such that the quality of the code base is maximized while the required effort is minimized [24].

Software refactoring is often performed with the aid of graph transformation tools [17], where the available refactorings are applied in random, without having been scheduled [14]. Usually, the application order of the semi-automated refactorings is determined implicitly by human practitioners. But this is inefficient and error-prone. While experienced engineers may do it well, inexperienced engineers may lead to poor/infeasible schedule. The challenge is likely to be more severe for refactoring legacy systems, or when a practitioner new to the code base has to devise the refactoring schedule. Therefore, automated (or semi-automated) scheduling for performing selection and ordering of refactorings from a set of all refactoring candidates is a justified need.

In this regard, this paper makes two contributions. First, for estimating the refactoring effort, we introduce an *effort model* for refactoring code clones in object-oriented (OO) code base. Second, taking into account the effort model and a wide variety of possible hard and soft constraints, we formulate clone refactoring scheduling as a constraint satisfaction optimization problem (CSOP), and solve it applying constraint programming (CP) technique that aims to maximize benefit while minimizing the refactoring effort. To the best of our knowledge, ours is the first refactoring effort model for OO systems, and we are the first to apply the CP technique in software refactoring scheduling.

The remaining of the paper is organized as follows. Section II identifies the refactoring patterns that are suitable for code clone refactoring. In Section III, we describe our clone refactoring effort model. Section IV discusses how the effect of refactoring can be estimated. In Section V, we describe the possible constraints on refactorings, and Section VI presents our CSOP formulation of the refactoring scheduling problem. Section VII contains discussion on the related work, and finally Section VIII concludes the paper with our directions to future research.

II. Clone Refactoring Operations

Among the software refactoring patterns [6], we find that *extract method (EM), pull-up method (PM), extract class (EC)* and *rename refactor (RR)* are suitable for clone refactoring, and we refer to them as refactoring *operators*.

For code clone refactoring, these refactoring operators will operate on groups of clone fragments having two or more members. We refer to such clone-groups as the refactoring *operands*. Thus, a *refactoring activity* (or simply, refactoring) r can be formalized as $r = \langle op_r, g_r \rangle$, where $op_r \in \{EM, PM, EC, RR\}$ and g_r is the clone-group, which the refactoring operator op_r operates on.

III. Estimation of Refactoring Effort

The effort required for code clone refactoring is likely to depend on the type of refactoring operator and operand. Moreover, refactoring cloned code snippets that are scattered across different locations of the code base and/or inheritance hierarchy may require relatively more effort than that for refactoring clones residing cohesively at certain location of the code base. To address these issues, we propose a code clone *refactoring effort model* for procedural and OO software systems.

Suppose, a group of clones $g = \{c_1, c_2, c_3, \ldots, c_n\}$ is extracted as a set of refactoring candidates, where c_i $(1 \leq i \leq n)$ is a clone fragment inside method m_i, which is a member of class C_i hosted in file F_i contained in directory D_i. Mathematically,

$$\begin{aligned} c_i \frown m_i \frown C_i \frown F_i \frown D_i, &\qquad \text{for object-oriented code.} \\ c_i \frown m_i \frown F_i \frown D_i, &\qquad \text{for procedural code.} \end{aligned}$$

Here, the symbol $\frown$ indicates containment relationship. $x \frown y$ means, x is contained in y, in other words, y contains x. The relationship preserves transitive property, i.e., $x \frown y \frown z \Rightarrow x \frown z$. Thus, the set $C(g)$ of all classes hosting the clone fragments in g can be defined as $C(g) = \{C_i | \ \forall \ c_i \in g, c_i \frown C_i\}$.

A. Context Understanding Effort

Applicability of refactoring on certain code clones are largely dependent on the context. Therefore, before refactoring, the developer needs to understand the context pertaining to the refactoring candidate at hand. For understanding the context, the developer needs to examine two things: the caller-callee delegation of methods and the inheritance hierarchy.

Effort for Understanding Method Delegation. To understand the method delegation involving the concerned code clone c_i, the developer needs to understand the chain of methods that can be reached from c_i via caller-callee relationship. Let, $M_r(c_i)$ be the set of all such methods. The developer will also need to comprehend the set $M_f(c_i)$ of all the methods from which c_i can be reached via caller-callee relationship.

Then, the set of methods required to investigate for understanding the delegation effort concerning c_i, denoted as, $delegation(c_i) = M_f(c_i) \cup M_r(c_i) \cup \{m_i\}$. For understanding delegation concerning the clone fragments in g, the set of all methods required to examine, becomes $Delegation(g) = \bigcup_{c_i \in g} delegation(c_i)$. Thus, for the group g of refactoring candidates, the total effort for understanding method delegation understanding can be estimated as,
$E_d(g) = \sum_{m \in Delegation(g)} LOC(m)$.

Effort for Understanding Inheritance Hierarchy. Suppose, $C_p(g)$ be the set of all lowest/closest common superclasses of all pairs of classes in $C(g)$. The developer also needs to understand those classes in the inheritance hierarchy that have overridden or referenced to method m_i containing any code clone $c_i \in g$. Let, $C_s(g)$ be the set of all such classes. Then $C_h(g) = \{C_p(g) \cup C_s(g) \cup C(g)\}$ becomes the set of all classes required to examine for understanding the inheritance hierarchy concerning the code clones in g, and the effort $E_h(g)$ required for this can be estimated as, $E_h(g) = \sum_{C \in C_h(g)} LOC(C)$.

B. Effort for Code Modifications

To perform refactoring on the refactoring candidates, the developer usually needs to modify portions of source code.

Token Modification Effort. Developer's source code modification activities typically include modifications in the program tokens (e.g., identifier renaming). Let, $T = \{t_1, t_2, t_3, \ldots, t_k\}$ be the set of tokens such that a token $t_i \in T$ is required to be modified to t'_i, and the edit distance between t_i and t'_i is denoted as $\delta(t_i, t'_i)$. Then the total effort $E_t(g)$ for token modifications can be estimated as, $E_t(g) = \sum_{i=1}^{k} \delta(t_i, t'_i)$.

Code Relocation Effort. When the developers need to move a piece of code from one place to another, they typically select a block of adjacent statements and relocate them all at a time. Hence, the code relocation effort $E_r(g)$ can be estimated as, $E_r(g) = |\beta|$. where β is the set of all non-adjacent blocks of code that need to be relocated to perform the refactoring.

C. Navigation Effort

Effort for source code comprehension, modification, relocation is also dependent on the number of files and directories involved, and their distribution. To capture this, our effort model includes the notion of navigation effort, $E_n(g)$, calculated as follows.

$$E_n(g) = |F_d(g) \cup F_h(g)| + |D_d(g) \cup D_h(g)| + DCH(g) + DFH(g)$$

where,

$$\begin{aligned} F_d(g) &= \{F_i| & m_i \frown F_i, m_i \in Delegation(g)\} \\ F_h(g) &= \{F_i| & C_i \frown F_i, C_i \in C_h(g)\} \\ D_d(g) &= \{D_i| & F_i \frown D_i, F_i \in F_d(g)\} \\ D_h(g) &= \{D_i| & F_i \frown D_i, F_i \in F_h(g)\} \\ DCH(g) &= & \max_{C_i, C_j \in C_h(g)} \{\partial(C_i, C_j)\} \\ DFH(g) &= & \max_{F_i, F_j \in F_d(g) \cup F_h(g)} \{\eth(F_i, F_j)\} \end{aligned}$$

Here, $DCH(g)$ refers to the *dispersion of class hierarchy* having $\partial(C_i, C_j)$ denoting the distance between class C_i and class C_j in the inheritance hierarchy. More detail about $DCH(g)$ can be found elsewhere [8]. $DFH(g)$ is a similar metric that captures the *dispersion of files*, and $\eth(F_i, F_j)$ denotes the distance between files F_i and F_j in the file-system hierarchy.

Thus, the total effort $E(g)$ needed to refactor clone-group g is estimated as,

$$\begin{aligned} E(g) = w_d \times E_d(g) + w_h \times E_h(g) \\ + w_t \times E_t(g) + w_r \times E_r(g) + w_n \times E_n(g) \end{aligned}$$

where w_d, w_h, w_t, w_r, and w_n are respectively the wights on the efforts for understanding method delegation, understanding inheritance hierarchy, token modification, code relocation, and navigation. By default, they are set to one, but the software engineer may assign different weights to penalize certain types of efforts.

IV. Prediction of Refactoring Benefits

The expected benefit from code clone refactoring is the structural improvement in the code base, which should also enhance the software design quality. Obvious expected benefits include reduced line of code (LOC), less redundant code, and so on. For procedural code, procedural metrics (e.g., LOC, Cyclomatic Complexity) as well as structural metrics (e.g., fan-in, fan-out, and information flow) can be used to estimate software quality after refactoring. For object-oriented system, these metrics can be supplemented by object-oriented design metrics suites, such as *QMOOD* [2] and *Chidamber-Kemerer* [4] metric suite. Quantitative or qualitative estimation of the effect of refactoring on quality metrics can be possible before the actual application of the refactoring [3], [15], [18], [21], [22]. For example, Higo et. al. [7] developed a tool to estimate the effect of refactoring on the *Chidamber-Kemerer* quality metric before the actual application of the refactorings.

Having chosen a suitable set of quality attributes, let, $Q = \{q_1, q_2, q_3, \ldots, q_\eta\}$ be the set of quality attribute values before refactoring, and $Q_r = \{q'_1, q'_2, q'_3, \ldots, q'_\eta\}$ be the estimated values of those quality attributes after applying refactoring r. The improvement in quality can be assessed by comparing the quality before and after refactoring. Hence, the total gain in quality $\overline{Q}_r$ from refactoring r can be estimated as, $\overline{Q}_r = \sum_{j=1}^{\eta} \vartheta_j \times (q'_j - q_j)$, where ϑ_j is the weight on the j^{th} quality attribute. By default, $\vartheta_j = 1$, but the software practitioner can assign different values to impose more or less emphasis on certain quality attributes.

V. Refactoring Constraints

Among the applicable refactoring activities, there may be conflicts and dependencies [16]. Application of one refactoring may cause elements of other refactorings disappear, and thus invalidate their applicability [3], [15], [16]. Beside such *mutual exclusion* on conflicting refactorings, the application of one refactoring may reveal new refactoring opportunities, as suggested by Lee et. al. [15]. We understand that this is largely due to the composite structure of certain refactoring patterns, where one larger refactoring is composed of several smaller core refactorings [1]. For example, when extract superclass refactoring is applied, pull-up method is also applied as a part of it. In other words, pull-up method, at times, may require extraction of new superclass.

There may also be *sequential dependencies* between refactoring activities [15], [16]. Constraints of *mutual inclusion* may also arise when the application of one refactoring will necessitate operation of certain other refactorings [23]. Moreover, the organization's management may also impose *priorities* on certain refactoring activities [3], for example, lower priorities on refactoring clones in the critical parts of the system. We identify such priorities as soft constraints beside the following three types of hard constraints.

Definition 1 (Sequential dependency): Two refactorings r_i and r_j is said to have sequential dependency, if r_i cannot be applied after r_j. This is denoted as, $r_j \nrightarrow r_i$.

Definition 2 (Mutual exclusion): Given two refactorings r_i and r_j is said to be mutually exclusive, if both $r_i \nrightarrow r_j$ and $r_i \nleftarrow r_j$ holds. The mutual exclusion between r_i and r_j is denoted as, $r_i \nleftrightarrow r_j$ or $r_j \nleftrightarrow r_i$.

Definition 3 (Mutual inclusion): Two refactorings r_i and r_j are said to be mutually inclusive, if r_i is applied, r_j must also be applied before or after r_i, and vice versa. This is denoted as $r_i \leftrightarrow r_j$ or $r_j \leftrightarrow r_i$.
The complete independence of r_i and r_j is expressed as $r_i \perp r_j$.

VI. Formulation of Refactoring Schedule

Addressing all the hard and soft constraints, it becomes very difficult to compute an optimal refactoring schedule aiming to maximize code quality while minimizing efforts. Such a problem is known to be NP-hard [3], [14], [15]. Finding the optimum solution for such problems practically becomes too expensive (time consuming), and so, a feasible optimal solution is desired. However, the problem is by nature a CSOP. Therefore, we model the problem as a CSOP and solve it by applying *constraint programming* technique, which no one tried before.

Having identified the set $\mathcal{R}$ of all potential refactoring activities, we define two decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$, such that,

$$\mathcal{X}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ 1 & \text{if } r \in \mathcal{R} \text{ is chosen} \end{cases}$$

$$\mathcal{Y}_r = \begin{cases} 0 & \text{if } r \in \mathcal{R} \text{ is not chosen} \\ k & \text{if } r \in \mathcal{R} \text{ is chosen as the } k^{th} \text{ activity} \end{cases}$$

where $1 \le k \le |\mathcal{R}|$.

We also define a $|\mathcal{R}| \times |\mathcal{R}|$ constraint matrix $\mathcal{Z}$ to capture the sequential dependencies between refactorings r_i and r_j, such that,

$$\mathcal{Z}_{ij} = \begin{cases} 0 & \text{if } r_i \perp r_j \\ 1 & \text{if } r_i \nleftrightarrow r_j \\ +2 & \text{if } r_j \nrightarrow r_i \text{ and } r_i \leftrightarrow r_j \\ -2 & \text{if } r_i \nrightarrow r_j \text{ and } r_i \leftrightarrow r_j \\ +3 & \text{if } r_j \nrightarrow r_i\text{, but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \\ -3 & \text{if } r_i \nrightarrow r_j\text{, but neither } r_i \leftrightarrow r_j \text{ nor } r_i \nleftrightarrow r_j \end{cases}$$

Also note that, $\mathcal{Z}_{ij} = -\mathcal{Z}_{ji}$ or $\mathcal{Z}_{ij} = \mathcal{Z}_{ji} = 1$, for all $\langle i, j \rangle$.

Let, ρ_r be the priority on the refactoring r that operates on clone-group g_r. The CSOP formulation of the refactoring scheduling problem can be as follows.

$$\text{maximize} \quad \sum_{r \in \mathcal{R}} \mathcal{X}_r(\rho_r \overline{Q}_r - E(g_r)) \tag{1}$$

subject to,

$$\mathcal{X}_r + \mathcal{Y}_r \neq 1, \qquad \forall\, r \in \mathcal{R} \tag{2}$$
$$\mathcal{X}_{r_i} + \mathcal{X}_{r_j} = 2 \Rightarrow \mathcal{Y}_{r_i} \neq \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{3}$$
$$\mathcal{Z}_{ij} - \mathcal{Z}_{ji} > 0 \Rightarrow \mathcal{Y}_{r_i} < \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{4}$$
$$\mathcal{Z}_{ij} - \mathcal{Z}_{ji} < 0 \Rightarrow \mathcal{Y}_{r_i} > \mathcal{Y}_{r_j}, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{5}$$
$$|\mathcal{Z}_{ij}| = 1 \Rightarrow \mathcal{X}_{r_i} + \mathcal{X}_{r_j} \le 1, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{6}$$
$$|\mathcal{Z}_{ij}| = 2 \Rightarrow \mathcal{X}_{r_i} + \mathcal{X}_{r_j} = 2, \qquad \forall\, r_i, r_j \in \mathcal{R} \tag{7}$$

Here, Equation 1 is the objective function for maximizing the code quality and minimizing the refactoring effort while rewarding refactoring activities having higher priorities. Equation 2 ensures that the decision variables $\mathcal{X}_r$ and $\mathcal{Y}_r$

are kept consistent as their values are assigned. Equation 3 enforces that no two refactorings are scheduled at the same point in the sequence. Equation 4 and Equation 5 impose the sequential dependency constraints (i.e., $r_i \nrightarrow r_j$) on feasible schedules. Mutual exclusion (i.e., $r_i \nleftrightarrow r_j$) and mutual inclusion (i.e., $r_i \leftrightarrow r_j$) constraints are enforced by Equation 6 and Equation 7 respectively.

We have implemented the CSOP model applying constraint programming using *OPL* (Optimization Programming Language) in `IBM ILOG CPLEX Optimization Studio 12.2`.

VII. Related Work

The work of Bouktif et. al. [3], Lee et. al. [15], and Liu et. al. [14] closely relate to ours. Bouktif et. al. [3] formulated the refactoring problem as a constrained *Knapsack problem* and applied a metaheuristic *genetic algorithm* (GA) to obtain an optimal solution. However, they ignored both the constraints and consequences of the refactorings. Lee et. al. [15] applied *ordering messy GA (OmeGA)*, whereas, Liu et. al. [14] applied a heuristic algorithm to schedule code clone refactoring activities. Both of those work took into account the conflict and sequential dependencies among the refactoring activities, but missed the constraints of mutual inclusion and refactoring effort. Our work significantly differs from all those work in two ways. First, for computing the refactoring schedule, we applied constraint programming approach, which is different from theirs. Second, we took into account a wide category of refactoring constraints, some of which they ignored. Although Bouktif et. al. [3] proposed a clone refactoring effort model, their model was for procedural code only, whereas, our effort model is applicable to not only procedural but also OO source code.

O'Keeffe et. al. [13] conducted an empirical comparison of *simulated annealing* (SA), *GA* and *multiple ascent hill-climbing* techniques in scheduling refactoring activities in five software systems written in Java. They reported that the hill-climbing approach outperformed both GA and SA techniques. One of our immediate future work is to compare our technique with those approaches.

A number of methodologies [5], [12], [19], [20], [23] and metric based tools such as `CCShaper` [9] and `Aries` [8] have been proposed for semi-automated extraction of code clones as refactoring candidates. Several tools, such as `Libra` [10] and `CnP` [11], have been developed for providing support for simultaneous modification of code clones. Our work is not on finding potential clones for refactoring or providing editing support to apply refactorings. Rather, we focus on efficient scheduling of those refactoring candidates, which is missing in those tools. However, the metrics used in those tools can be exploited to estimate the refactoring effort and expected gain.

VIII. Conclusion and Future Work

In this paper, we present our ongoing work towards conflict-aware optimal scheduling of code clone refactorings. To estimate the refactoring effort, we have proposed an effort model for refactoring code clones in OO source code. Considering diverse categories of refactoring constraints, we have modelled the clone refactoring problem as a CSOP, and implemented the model using the CP technique. To the best of our knowledge, ours is the first refactoring effort model for OO code corpus, and our CP approach is a unique technique that no one in the past reported to have applied in this regard.

To evaluate our approach, we are currently working to apply it on several open-source and proprietary code bases. Our immediate future work includes comparison of our approach with other techniques such as GA and SA. We believe that our constraint programming approach is capable of efficiently compute optimal feasible schedule of code clone refactoring activities.

Acknowledgments: This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] D. Advani, Y. Hassoun, and S. Counsell. Understanding the complexity of refactoring in software systems: a tool-based approach. *Intl. J. Gen. Sys.*, 35(3): 329–346, 2006.

[2] J. Bansiya and C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Engg.*, 28(1): 4–17, 2002.

[3] S. Bouktif, G. Antoniol, M. Neteler, and E. Merlo. A Novel Approach to Optimize Clone Refactoring Activity. In *GECCO*, July 8 –12, 2006.

[4] S. Chidamber and C. Kemerer. A metric suite for object-oriented design. *IEEE Trans. Softw. Engg.*, 25(5): 476–493, 1994.

[5] S. Ducasse, M. Rieger, and G. Golomingi. Tool Support for Refactoring Duplicated OO Code. In *WOOT*, pp. 177–178, 1999.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley Professional, 1999.

[7] Y. Higo, Y. Matsumoto, S. Kusumoto, and K. Inoue. Refactoring Effect Estimation based on Complexity Metrics. In *ASWEC*, pp. 219–228, 2008.

[8] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. ARIES: Refactoring Support Tool Code Clone. In *3-WoSQ*, pp. 1 – 4, 2005.

[9] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring Support Based on Code Clone Analysis. *PROFES, LNCS 3009*, pp. 220–233, Springer-Verlag, 2004.

[10] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue. Simultaneous Modification Support based on Code Clone Analysis. In *APSEC*. pp. 262–269, 2007.

[11] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an environment for the proactive management of copy-and-paste programming. In *ICPC*, pp. 238–242, 2009.

[12] E. Kodhai , V. Vijayakumar, G. Balabaskaran, T. Stalin, and B.Kanagaraj. Method Level Detection and Removal of Code Clones in C and Java Programs using Refactoring. In *IJJCET*, pp. 93 – 95, 2010.

[13] M. O'Keeffe and M. Ó Cinnéide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.: Res. Pract.*, 20: 345 – 364, 2008.

[14] H. Liu, G. Li, Z. Ma, and W. Shao. Conflict-aware schedule of software refactorings. *IET Softw.*, 2(5): 446–460, 2008.

[15] S. Lee, G. Bae, H. S. Chae, and D. Bae, and Yong Rae Kwon. Automated scheduling for clone-based refactoring using a competent GA. *Softw. Pract. Exper.*, Wiley Online Library, 2010.

[16] T. Mens, G. Taentzer, and O. Runge. Analysing refactoring dependencies using graph transformation. *J. Softw. and Syst. Modeling*, 6(3): 269–285, 2007.

[17] J. Pérez, Y. Crespo, B. Hoffmann, and Tom Mens. A case study to evaluate the suitability of graph transformation tools for program refactoring. *Intl. J. Softw. Tools Tech. Transfer*, 12: 183–199, 2010.

[18] H. Sahraoui, R. Godin, and T. Miceli. Can metrics help to bridge the gap between the improvement of OO design quality and its automation?. In *ICSM*, pp. 154–162, 2000.

[19] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a Refactoring Guideline Using Code Clone Classification. In *WRT*, pp. 6:1–6:4, 2008.

[20] S. Schulze and M. Kuhlemann. Advanced Analysis for Code Clone Removal. In *WSR*, 2009.

[21] F. Simon, F. Steinbrucker, and C. Lewerentz. Metrics based refactoring. In *CSMR*, pp. 30–38, 2001.

[22] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *CSMR*, pp. 183–192, 2003.

[23] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On refactoring support based on code clone dependency relation. In *METRICS*, 10 pp., 2005.

[24] M. F. Zibran and C. K. Roy. Towards Flexible Code Clone Detection, Management, and Refactoring in IDE. In *IWSC*, 2 pp., 2011 (to appear).