On the use of Developers' Context for Automatic Refactoring of Software Anti-patterns

Rodrigo Morales[a], Zéphyrin Soh[a], Foutse Khomh[a], Giuliano Antoniol[b], Francisco Chicano[c]

[a]*SWAT Lab., Polytechnique Montréal, Canada*
[b]*Soccer Lab., Polytechnique Montréal, Canada*
[c]*Departamento de Lenguajes y Ciencias de la Computación, University of Málaga*

Abstract

Anti-patterns are poor solutions to design problems that make software systems hard to understand and extend. Entities involved in anti-patterns are reported to be consistently related to high change and fault rates. Refactorings, which are behavior preserving changes are often performed to remove anti-patterns from software systems. Developers are advised to interleave refactoring activities with their regular coding tasks to remove anti-patterns, and consequently improve software design quality. However, because the number of anti-patterns in a software system can be very large, and their interactions can require a solution in a set of conflicting objectives, the process of manual refactoring can be overwhelming. To automate this process, previous works have modeled anti-patterns refactoring as a batch process where a program provides a solution for the total number of classes in a system, and the developer has to examine a long list of refactorings, which is not feasible in most situations. Moreover, these proposed solutions often require that developers modify classes on which they never worked before (*i.e.,* classes on which they have little or no knowledge). To improve on these limitations, this paper proposes an automated refactoring approach, ReCon (Refactoring approach based on task Context), that leverages

Email addresses: rodrigomorales2@acm.org (Rodrigo Morales), zephyrin.soh@polymtl.ca (Zéphyrin Soh), foutse.khomh@polymtl.ca (Foutse Khomh), giuliano.antoniol@polymtl.ca (Giuliano Antoniol), chicano@lcc.uma.es (Francisco Chicano)

information about a developer's task (*i.e.,* the list of code entities relevant to the developer's task) and metaheuristics techniques to compute the best sequence of refactorings that affects only entities in the developer's context. We mine 1,705 task contexts (collected using the Eclipse plug-in Mylyn) and 1,013 code snapshots from three open-source software projects (Mylyn, PDE, Eclipse Platform) to assess the performance of our proposed approach. Results show that ReCon can remove more than 50% of anti-patterns in a software system, using fewer resources than the traditional approaches from the literature.

Keywords: Software Maintenance, Automatic Refactoring, Task Context, Interaction Traces, Anti-patterns, Metaheuristics.

1. Introduction

Software design matters. In a general sense, the design of a software system models the representation of the problem, goals, constraints and the solution proposed by the system. Studies suggest that defects related to design are orders of magnitude more expensive to fix than those introduced during the implementation [1, 2, 3, 4]. Indeed, neglecting to carefully design a software system may result in a highly complex implementation that is harder to maintain.

As systems evolve, they tend to grow in complexity and degrade in effectiveness [5], unless the quality of the systems is controlled and continually improved. Even good design solutions (*e.g.,* design patterns) tends to decay into anti-patterns as systems age [6, 7]. When the design of a system is poor, new changes to the system often degrade quality instead of improving it. In addition to this, the competition in software industry that puts pressure on companies to deliver new products and features faster often leads development teams to adopt poor design choices. Poor design choices increase the risk of fault [8] and the cost of future maintenance and evolution changes; a phenomenon often referred to as *technical debt* [9]. This debt increases until developers *pay down* the debt by redesigning the system. It is noteworthy that the cost of removing defects in aged systems is high [10]. To combat design decay, developers should remove

anti-patterns, which are poor solutions to design problems [11]. They are not technically incorrect, but result in negative consequences in the long run [12].

An example of anti-pattern is the *Spaghetti Code*, which is a class without structure that declares long methods without parameters [13]. This anti-pattern depicts an abuse of procedural programming in object-oriented systems, that prevents code reuse. In a previous study, Bavota et al. [14] found that industrial developers assign a very high severity level to this anti-pattern. Another example of anti-pattern is the *Lazy class*, which is a class with few methods and a low complexity that does not "pay off" its inclusion in the system.

Abbès et al. [15] showed that anti-patterns affect the understandability of systems and Khomh et al. [8] found that there is a strong correlation between the occurrence of anti-patterns and the fault-proneness of source code files. In addition to this, anti-patterns often remain in a system for several releases [16, 17]. It is therefore essential to correct anti-patterns in a regular basis, to avoid their negative impact on future releases of the system.

To remove anti-patterns and improve design quality, developers perform refactoring, which is a technique that consists in reorganizing the code of a program without altering its original behavior.

There are several studies that assessed the benefits of refactoring in: reducing code complexity [18], inter-module dependencies and post-release faults [19]; in improving code comprehensibility [20], and application performance [21].

Even though refactoring is now a common practice in the industry [22], manual refactoring of anti-patterns is still a risky and error-prone task, especially when it is performed by inexperienced developers, or developers who are unfamiliar with the system. Moreover, assigning resources to perform refactoring is not always feasible, due to constraints in budget, shorter releases cycles and staff shortage.

To overcome the burden of manual refactoring, researchers have proposed the formulation of refactoring as an optimization problem, where a system finds a sequence of refactoring operations that corrects most of the anti-patterns, improving certain aspects of quality [23, 24, 25, 26]. The main problem with

current approaches is that they rely on 1) a corpus of *bad code examples* which adds a new task and responsibility to developers, to collect and manage the aforementioned corpus, or 2) a *desired design*, where the developer is expected to input the model that she wants to obtain in advance. In any case, the developers have to accept a *global solution*, which might consider classes that are not part of the scope of the maintenance task that she is performing, or in other words, *out of the context*. As a result, the developer has to deal with an long sequence of refactorings that often affect classes on which she has no prior knowledge. Yet, previous studies have shown that developers prefer approaches that do not disrupt their work flow. In fact, Murphy-Hill et al. report that developers prefer approaches that suggest refactoring operations that can be applied to the group of files that are currently active in their workspace [22].

This paper describes an automated refactoring approach that is based on task context (we refer to our proposed approach as ReCon in the rest of the paper) that has the following advantages over the state-of-the-art approaches: 1) it does not require a set of bad examples, as detection rules are derived from the literature of anti-patterns; 2) it is customizable at a high-level of abstraction, using a domain specific language, and 3) it generates a set of *local refactoring solutions*, *i.e.,* refactoring suggestions over *active classes*, that developers can apply on the fly while performing his development or maintenance task.

To evaluate the performance of ReCon, we mined 1,705 Mylyn interaction histories (IH) from three open-source projects (Mylyn, PDE, and Eclipse Platform). From the IH of each task, we computed the relevant classes and entities targeted by the developer when performing the task (this constitutes the task context). Then we download the code snapshot for each task, from the version control system (VCS) of the project and evaluate the quality of the project before and after applying ReCon, considering the removal of anti-patterns and the quality gain in terms of three desirable quality attributes: understandability, flexibility, and reusability defined in [27]. We run our approach in two scenarios, the refactoring of all classes in a system (root-canal), and the incremental refactoring of certain classes explored during maintenance sessions

(floss-refactoring). The remainder of this paper is organized as follows: Section 2 provides some background information and a description of the data used in this paper, while Section 3 describes our approach for refactoring using context. Section 4 presents and discusses the evaluation of our approach. Related work is outlined in Section 6. Section 7 discloses the threats to the validity of our study. Finally, we present our conclusions and lay out directions for future work in Section 8.

2. Background

In this section, we provide the background information about refactoring (Section 2.1) and task context (Section 2.2).

2.1. Refactoring Anti-patterns to improve design quality

The process of correcting anti-patterns can be divided in three steps: 1) determination of the classes that need to be refactored, 2) selection of appropriate refactorings to correct anti-patterns contained in the identified classes, 3) application of the refactorings and evaluation of their effectiveness with respect to some *quality criteria* [28]. Once a developer has determined the necessary steps to correct an anti-pattern, she/he has to decide the order in which refactorings will be applied. The ordering of refactoring is not a trivial problem, as the type and number of refactorings operations is typically high, and sometimes conflicting with each other.

Conflicts between refactorings occurs when the application of one refactoring cause elements of other refactorings disappear, invalidating their applicability [29]. In some cases it is possible to solve this problem changing the order in which the conflicting refactorings are applied. But in other cases it is necessary to choose in favor of one refactoring of another (mutual exclusion).

Concerning the size of the search space, if k is the number of available refactorings, the number of possible solutions (NS) is given by $NS = (k!)^k$ [30], which results in a space of feasible solutions that is too large to be explored exhaustively.

For these reasons, we believe that an *automated-refactoring approach* for refactoring suggestion could support developers to improve the design quality of their systems, without investing too much time and effort in this task.

In general, developers follow two main refactoring strategies [31]: floss refactoring and root-canal refactoring. The floss refactoring strategy consists in applying refactoring to the code while performing other development or maintenance activities, *e.g.,* adding new features, or fixing a bug. In the floss refactoring strategy, the refactoring of the code is not the main goal; developers combine different types of code changes with refactoring. In the case of root-canal refactoring, developers perform the refactoring of the code exclusively. Floss refactoring is a recommended and a common strategy followed by developers according to previous works [31, 22]. However, automating floss refactoring is challenging. So far, existing approaches perform anti-patterns detection on the whole program, and provide a final solution comprised of a set of low-level refactorings which have to be applied (sometimes on classes unknown from the developer) in order to improve the quality [23, 24, 25]. Other approaches, consider more than one objective when refactoring, *e.g.,* semantic-similarity [26] and historical information [32] but still propose long sequence of refactorings that often affect classes unknown to developers. These previous approaches are more suitable for root-canal refactoring activities. During root-canal refactoring sessions, developers can work together to improve the design quality of their system (for example by implementing the sequence of refactorings suggested by these previous approaches.

In this work, we aim to support developers during floss refactoring performed while implementing their daily development and maintenance tasks. To guide the search of refactoring opportunities in relevant artifacts (*i.e.,* classes relevant to developer's task), we leverage information provided by interaction traces (captured using the monitoring tools) and suggest refactorings that the developer can apply on the fly while performing his task.

2.2. Task context

This section presents in details the concept of context used in this paper. By *task context*, we refer to the program entities that the developers used when resolving a development or maintenance task. In fact, during development or maintenance sessions, developers usually interact with program entities through their IDE (Integrated Development Environment). The task context is accessible using monitoring tools, such as Mylyn [33], or MimEc [34]. These tools log all developers' interaction with the program entities (*e.g.,* interaction trace). In the following we use Mylyn as an example of a monitoring tool.

Mylyn is an Eclipse plug-in for task management, which introduces the concept of task-focused interface [35]. When developers activate a task, Mylyn automatically build the task context by monitoring developer's activities. Task context is a graph of program elements and their relationships that a developer uses to perform a task. Mylyn builds the task context based on a degree-of-interest model that consist of weighing the relevance of elements to the task.

A developer can create a Mylyn task to track the code changes when handling a change request. The developer's programming activities are monitored by Mylyn to create a task context and predict relevant artifacts for the task. The programming activities monitored by Mylyn include selection and edition of files. In Mylyn, each activity is recorded as an interaction event between a developer and the IDE. There are eight types of interaction events in Mylyn, as described in Table 1. Three types of interaction events are triggered by a developer, *i.e.,* *Command*, *Edit* and *Selection* events.

Each Mylyn log has a task identifier, which often contains the change request ID. A Mylyn log is stored in an XML format. Its basic element is *InteractionEvent* that describes the event. The descriptions include: a starting date (*i.e., StartDate*), an end date (*i.e., EndDate*), an event type (*i.e., Kind*), the identifier of the UI affordance that tracks the event (*i.e., OriginId*), and the names of the files involved in the event (*i.e., StructureHandle*). Figure 1 presents an example of *InteractionEvent* that was recorded during the correction of the bug

#311966 [1].

Table 1: Event Types from Mylyn.

Event Type	Description	Developer-Initiated ?
Command	Click buttons, menus, and type in keyboard shortcuts.	Yes
Edit	Select any text in an editor.	Yes
Selection	Select a file in the explorer.	Yes
Attention	Update the meta-context of a task activity.	No
Manipulation	Directly manipulate the degree of interest (DOI) value through Mylyn' user interface.	No
Prediction	Predict relevant files based on search results.	No
Preference	Change workbench preferences.	No
Propagation	Predict relevant files based on structural relationships (*e.g.*, the parent chain in a containment hierarchy).	No

```
<?xml version="1.0" encoding="UTF-8"?>
<InteractionHistory
    Id="https://bugs.eclipse.org/bugs-311966"
    Version="1">
    <InteractionEvent
        StartDate="2010-06-25 11:27:23.935 EDT"
        EndDate="2010-06-25 11:27:27.777 EDT"
        Kind="edit"
        OriginId="org.eclipse.jdt.ui.CompilationUnitEditor"
        StructureHandle="/org.eclipse.mylyn.bugzilla.ui/src/org/eclipse/mylyn/
                         internal/bugzilla/ui/tasklist/BugzillaConnectorUi.java"
        StructureKind="resource"
        Interest="2.0"
        Delta="null"
        Navigation="null"
    />
    ...
    <InteractionEvent
        ...
    />
</InteractionHistory>
```

Figure 1: Structure of the Mylyn log of bug #311966.

In the rest of the paper, we refer to the set of program entities relevant to a developer's task as the *context*.

2.3. Metaheuristics techniques

One key component of our approach is a meta-heuristic technique which goes through the set of alternative designs in the search of the optimal solution, *i.e.*,

[1] `https://bugs.eclipse.org/bugs/show\_bug.cgi?id=311966`

the sequence of refactorings that corrects more anti-patterns. Depending on the number of parameters, the scope (local or global search) and convergence time, the results may vary and can have an impact on the execution time and the solution's quality. Hence, to provide an insight into which meta-heuristics are most effective using automated refactoring, we implement three well known techniques that are described below.

Simulated Annealing (SA). It is a meta-heuristic technique [36] that imitates the process of metal annealing, by allowing movements of worse quality than the current solution, with a probability that decreases during the search process (when the temperature goes down), until only good quality solutions are accepted. In the first step, the probability toward improvement is low, allowing the exploration of the search space (consequently escaping from local optima), but this behavior changes gradually according to the cooling schedule which is crucial for the performance of the algorithm. The movements between designs are achieved by perturbing the initial solution, generally a random one.

Genetic Algorithm (GA). It is an evolutionary metaheuristic [37, 38], where a group of candidate solutions, called individuals or chromosomes, are recombined through some variation operators, *i.e.,* crossover, and mutation, in order to select the best solutions of each iteration (generation). The process of selection and recombination is guided by an evaluation function, *a.k.a.,* fitness function, which ensures that the best individuals have greater chances to be selected in each generation. GA is a population-based algorithm, because it works with several solutions at the same time, contrary to trajectory-based methods like hill-climbing and simulated annealing that work with only one solution at a time.

Variable Neighborhood Search (VNS). It consists of dynamically changing the neighborhood structures defined at the beginning of the search [39], which expands until a stopping condition is met. In its first step, a solution in the kth neighborhood is randomly selected and altered (shaking phase). Then, a process of local search starts from this point independently of the neighborhood structures. If the outcome solution of the local search is better than the current

solution, the first one is replaced by the new solution and the process restarts at the first neighborhood, otherwise k is incremented and a new shaking phase is started from a new neighborhood. The advantage of this metaheuristic is that (1) it provides diversification when changing neighborhoods in case of no improvement, (2) choosing a solution in the neighborhood of the best solution yields to preserving good features of the current one. For our particular case, the shaking phase consists of modifying i refactoring operations from the end of the sequence, until we reach the starting point. The local search mechanism is responsible to apply all the possible variations to the candidate solution and select the best local optima. In our case, local search operates at the level of the last j refactoring operations.

We selected these metaheuristics because SA and GA are well known techniques that have been used in previous works on refactoring [40, 41, 42, 24, 43], and VNS, has been applied in combinatorial optimization problems [44, 45, 46]. VNS is particular attractive to study for the reasons explained above.

3. Approach

This section presents the foundations of our proposed approach ReCon that aims to improve the design quality of object-oriented systems in an incremental way, relying on task context information produced by monitoring tools. We use the occurrence of anti-patterns as a proxy for software design quality.

3.1. Approach overview

In Figure 2 we present the workflow of ReCon. ReCon takes as an input an interaction trace generated by a monitoring tool, and a software system. We generate an abstract model by performing a static analysis of the software system. We identified the relevant classes from the interaction trace (task context), and build a map of anti-patterns based in the anti-patterns detection results. Next, we generate a list of candidate refactorings to correct the anti-patterns detected. Next, we use a search algorithm to find the best combination of refactorings that remove the largest number of anti-patterns. The search algorithm

can be any metaheuristic technique such as GA, SA, or VNS, *cf.* Section 2.3. It can be argued that in tasks comprised of a few classes, a greedy algorithm, or even applying all the candidate refactorings is more effective than running a metaheuristic. However, the real difficulty, besides the number of refactoring operations, is the refactorings that are conflicted, *i.e.,* the ones that cannot be applied together. For that reason, if we find that there is no conflict between the refactoring operations proposed, we simply apply all the refactoring operations. Otherwise a search algorithm is used.

The search algorithm generates a set of candidate sequences in a non-deterministic way. The candidate sequences are evaluated using a blackbox approach, where each candidate sequence is applied to a copy of the abstract model, and the resultant model is evaluated in terms of anti-patterns occurrences. This process continues until it finds a sequence that removes all anti-patterns, or the search algorithm reaches the maximum number of iterations. The final output is a valid (non-conflicted) sequence of refactorings to improve the design quality of the code entities in the task context.

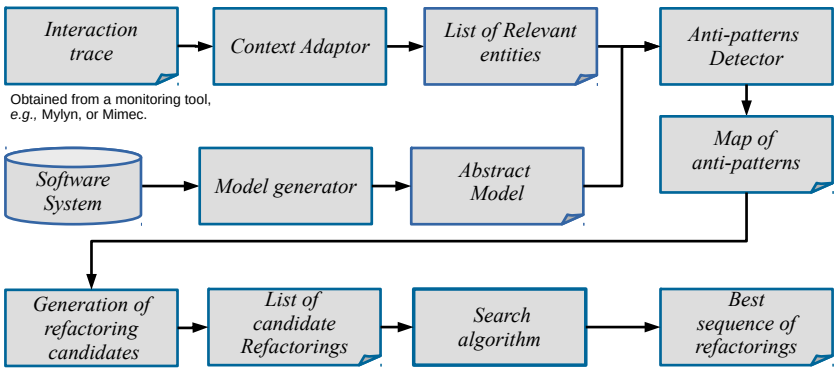

Figure 2: Workflow of ReCon.

3.2. Automated-Refactoring using task context

ReCon, our novel approach, formulates the correction of anti-patterns as a combinatorial optimization problem [47]; given a software system, with the

help of an objective function, it selects and applies refactorings to a light-weight representation of the system in order to move through the space of alternative designs, until we find the design with higher quality, measured in terms of anti-patterns correction.

The objective function that we use to guide the search for refactoring sequences is presented in Equation (1) [30]. It measures the number of anti-patterns removed in comparison with the maximum number of anti-patterns that can exist in a system. We choose this fitness function because it is easy to implement and it is an inexpensive way to measure the effectiveness of our approach with respect to the correction of anti-patterns.

$$Quality = 1 - \frac{NDC}{NC \times NAT}, \tag{1}$$

where NDC is the number of classes that contain anti-patterns, NC is the number of classes, and NAT is the number of anti-pattern's types. This objective function increases when the number of anti-patterns in the system is reduced after applying the proposed refactoring sequence. The output value of $Quality$ is normalized between 0 and 1.

Note that the objective function (the number of anti-patterns removed) depends in a non-trivial way on the code of the original and the refactored version. This fact makes it difficult to model the problem using a closed algebraic expression and, thus, limits the kind of algorithms and techniques we can use to solve the problem. In particular, mathematical programming techniques are difficult to apply in this case, since it requires constraints and objective functions given as closed algebraic expressions. Any algebraic model of the objective function should probably take into account too much detail of the source code, which would increase the number of variables and constraints of the potential mathematical program up to the point that it is too large to be solved in a reasonable time. For this reason, we follow a black box optimization approach, where the quality of the solutions proposed is given by an objective function which detects on-the-fly anti-patterns in the refactored code. Metaheuristic algorithms [48]

are among the most successful techniques to apply in the context of black box optimization. In the next paragraph we will detail the representation of the solutions, and the parameters of the algorithms used in this paper to find the best sequence of refactorings.

3.2.1. Solution representation

To represent a candidate solution, we use a vector representation where each element represents a refactoring operation (**RO**). In Table 2 we present a synthetic example. We include a *Id* field, which is an integer number assigned to each refactoring operation in our generated list of refactoring opportunities. The optimization algorithm uses this Id to know which refactorings have been applied in the sequence and what ROs can be applied (valid movements in the search space). We also include the anti-pattern's source class, and the type of refactoring. The type of refactoring is used for determining if there is any conflict with any previous RO in the sequence. In addition to this, and according to the refactoring type, we can have more fields providing additional information, *e.g.,* qualified name of long-parameter-list methods, in the case of LP class; children class for a class a class containing Speculative generality anti-pattern, etc.

Table 2: Representation of a refactoring sequence.

ID	Source class	Type	Other fields
9	ExtWindowsMenuUI	Introduce Parameter Object	List of long-parameter-list methods
26	RangeSearchFromKey	Inline class	Target class
45	ProjectResource	CollapseHierarchy	Children class
16	ActivityContextManager	Spaghetti code	Long method(s) name

3.2.2. Variation operators

Simulated annealing. It employs one variation operator, *a.k.a.,* perturbation operator, which consists of choosing a random point in a sequence, then we remove the refactoring operations from that point to the end, and finally we regenerate the sequence until we cannot add more refactoring operations. To

ilustrate this procedure consider the example show in Figure 3. We define this strategy, because an arbitrary transformation of a refactoring operation in the sequence, like the one implemented in binary strings, will lead to semantic inconsistencies given that in refactoring the order is important, *e.g.,* one refactoring could block further refactorings. Moreover, it is cheaper to add operations from a starting point (in the worst case the first refactoring operation) than verifying semantic correctness backwards, and finally, it brings more diversity which is the ultimate goal of perturbing a sequence.

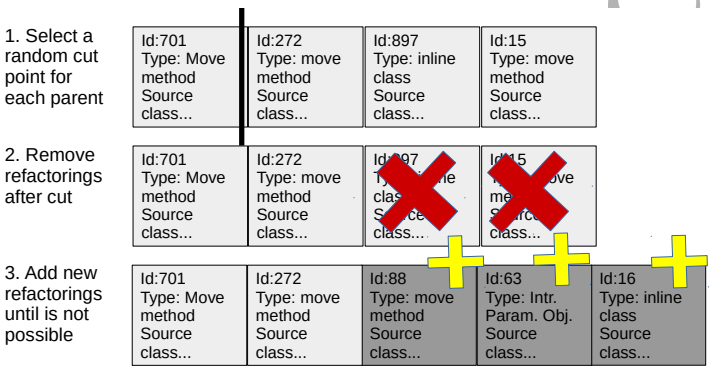

Figure 3: Example of perturbation operator

Genetic algorithm. It employs two variation operators, crossover and mutation. To select the best individuals to perform the crossover, we use the "binary tournament" technique. The crossover operator is "Cut and splice" [24, 49] technique, which consists in randomly setting a *cut point* for each parent, and recombining with the rest of elements of the other parent's cut point and vice-versa, resulting in two individuals with different lengths. An example of this operator is shown Figure 4. Note that when a refactoring operation is conflicted with a previous one in the sequence, we just drop it.

The mutation operator follows the same strategy than the perturbation process implemented in our version of simulated annealing, rather than the one proposed in in [24, 49] because we found that the former one was unable to find complete solutions, *i.e.,* the ones that removes all anti-patterns in a reasonable

amount of time.

Variable neighborhood search. It uses the same perturbation operator of SA to alter a solution in the *kth* neighborhood.

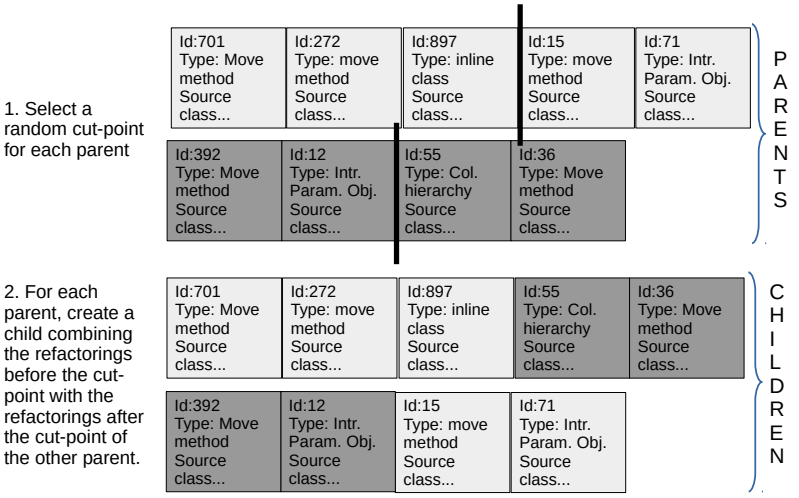

Figure 4: Example of crossover operator

Parameters of the metaheuristics. We use three metaheuristic techniques in our case study. As we mentioned before, they make use of different settings to move through the decision space in the search for an optimal solution. To determine the best parameters for the techniques employed, we run each algorithm with different configurations 30 times, following a *factorial design*.

In the case of GA we test 16 combinations of mutation probability $p_m = (1, 0.8, 0.5, 0.2)$, and crossover probability $p_c = (1, 0.8, 0.5, 0.2)$, and obtained the best results with the pair (0.8, 0.8). This is not a surprise as in [49] they found high mutation and crossover values to be the best trade for algorithm performance.

For SA, we set the initial temperature to 10,000,000, and tried three different values of cooling factor (CF), $CF = (0.990, 0.993, 0.996, 0.998)$ and found the best results with the latter one.

For VNS we define a maximum number of neighborhoods $maxK = 100$.

In the local search, we tried different values of local factor $j = (2, 4, 6, 8)$, and found the best results with the smallest value of 2.

For the specific problem of automated refactoring, setting the initial size of the refactoring sequence is crucial to find the best sequence in a reasonable time, especially when we have a huge number of candidate refactorings, because setting a low value will lead to find poor solutions in terms of anti-patterns correction. On the contrary, if the initial size is very large, we may obtain the reverse effect because applying many refactorings not necessarily implies better quality, as refactorings can improve one aspect of quality while worsen others. Hence, we experiment running the algorithms with three relative thresholds: 25%, 50%, 75% and 100%, of the total number of refactoring opportunities. We found that 50% give us the best results in terms of removal of anti-patterns.

Finally, the number of iterations for all the algorithms is set to 1000. The population size for GA is set to 100 individuals as typically used value in other refactoring works [50], and the selection operator used is binary tournament.

4. Evaluation

The *goal* of this case study is to assess the effectiveness of ReCon in correcting anti-patterns in object-oriented systems (OO) during maintenance tasks. The *quality focus* is the improvement of the design quality of OO systems. The *perspective* is that of researchers interested in developing automated refactoring tools, and developers interested in improving the design quality of their code.

The context consists of *1,705* task contexts, and *1,013* code snapshots from three open-source software systems (Mylyn, PDE, and Platform), and three metaheuristic techniques (GA, SA, VNS). In Table 3, we present relevant information about the Eclipse projects studied and the count for each type of anti-pattern.

4.1. Dependent and Independent Variables

To assess whether automatic refactoring using context improves the quality of a system, we consider the following dependent and independent variables:

Table 3: Descriptive statistics of the studied projects.

Subproject	Num. of classes	Num. of tasks	Num. of anti-patterns
Mylyn	2,365	183	167
PDE	16,045	129	3,512
Platform	20,259	213	3,558

Independent variables: The independent variables define the refactoring approaches that we performed. We use two refactoring approaches: automated root-canal refactoring and automated floss refactoring.

Dependent variables: We use the following variables to assess whether a refactoring approach (*i.e.,* automated floss refactoring or automated root-canal refactoring) improves the quality of the system.

- Number of anti-patterns removed after refactoring (#AP): For each refactoring approach, we compute the number of anti-patterns removed. The number of anti-patterns removed is an indication of the improvement of the design quality of the system. The more anti-patterns are removed, the better is the design quality of the system.

- Design quality improvement. After finding the best refactoring solution for each program using the proposed metaheuristics, we evaluate the resulting design code using 5 quality functions attributes of QMOOD hierarchical model.

4.2. Data Collection and Processing

We follow two main steps to collect and process the data of our experiment:

(1) In step one, we collect developers' interaction traces from the Eclipse bug repository [2]. Interaction traces appear as attachments to a bug report. These interaction traces contain program entities that developers interacted with (*i.e.,* the context). When collecting an interaction trace during a bug resolution, developers also perform modifications on the system. These modifications that

[2]https://bugs.eclipse.org/bugs/

change the state of the system (and which can improve or degrade the quality of the system) are essential for the completion of the developer's task. Hence, it is important to consider these developers' modifications when looking for refactoring opportunities. We consider a patch attached to a bug report as the changes performed by a developer during his working session if and only if the interaction trace and the patch are attached by the same developer at the same time [51]. In our experiment, we consider the interaction traces and patches of three Eclipse projects that have most interaction traces. Precisely, we downloaded 663, 132, and 218 couples of interaction traces and patches for Mylyn, PDE and Platform systems, respectively.

(2) In step two, we identify the start timestamp of each interaction trace. We consider that developers checkout the system before they start to fix a bug. Thus, we checkout the snapshot of the system from the appropriate source code repository (*i.e.,* the VCS of the project on which the task was performed) on the master branch and before the start timestamp. In total we checkout 663, 132, and 218 snapshots of Mylyn, PDE and Platform projects, respectively. Snapshots provide the states of the system used by the developers and the patches contain the changes made by the developers.

4.3. ReCon implementation

We instantiate our generic approach ReCon, in Java. We start extracting relevant code entities in a task from interaction traces, generated by Mylyn, using our context adaptor. Then, we perform the static analysis of the system, using Ptidej tool suite[3]. The result is a PADL Model, which is an abstract representation of the code entities, such as classes, interfaces, methods and attributes, and their structural relationships, *e.g.,* inheritance, association, etc. Next, we detect anti-patterns in the PADL model using SAD tool, which is the implementation of DECOR [52], a well known approach to define and detect anti-patterns, also part of Ptidej tool suite. DECOR uses a set of rules defined in

[3]http://www.ptidej.net/tools/designsmells/

a domain specific language (DSL) to characterize anti-patterns. These rules are derived from metrics, structural and semantic properties. DECOR is recognized to have the highest precision in detecting anti-patterns and code smells [52].

In this work, we consider four types of anti-patterns, namely **Lazy Class (LC)**, **Long Parameter list (LP)**, **Spaghetti Code (SC)** and **Speculative Generality (SG)**. We select these anti-patterns, because (1) they are well defined in the literature, with the recommended steps to remove them [53], (2) they are easy to identify by developers [14], (3) they have been studied in previous works [8, 52, 54, 55]. In Table 4, we present a brief definition of each anti-pattern and the proposed refactoring(s) to correct them. The proposed refactorings procedures, suggested in the literature [56, 57], aim to support developers with a previous knowledge of the system functionality that they want to improve. To automatize this task, we have to adapt the aforementioned procedures, leveraging the structural information computed from the abstract model and the anti-pattern detection, defining a corresponding *refactoring strategy* for each anti-pattern, and following the recommendations from previous works for semantic preservation [58, 59].

Table 4: ReCon anti-patterns refactoring strategies.

Name	Description	Refactoring(s) strategy
LC	Small classes with low complexity that do not justify their existence in the system	Inline class
LP	A class with one or more methods having a long list of parameters	Introduce parameter object
SC	A class without structure that declares long methods without parameters	Replace method with method object
SG	An abstract class that is not actually needed, as it is not specialized by any other class	Collapse hierarchy

We describe the set of refactoring strategies that we implemented in our approach to correct classes affected by the aforementioned anti-patterns.

In the case of lazy class, the proposed refactoring is *inline class*, which consist of moving all the features of a LC to another class, and after that remove the LC

class from the system. As an example, we present in Figure 5 the UML diagram of class *XMLCleaner*, from Eclipse Mylyn Project. This class, which aims to escape "&" characters from XML files, consists of only one public method with less than 20 LOC. Hence, a candidate refactoring operation could be to inline class XMLCleaner to another class; for example, AbstractReportFactory class from the same package, that makes use of this class in the method collectResults.

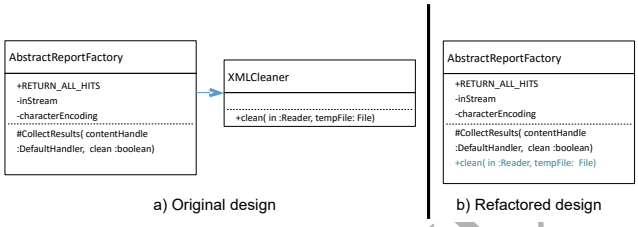

Figure 5: An example of Lazy class and its corresponding refactoring.

As we can observe, inline class refactoring is comprised of a series of *low level* refactorings that have to be applied in specific order, *e.g.,* move method(s) and/or attribute(s) to another class, update call sites, and delete LC class. Unlike previous refactoring approaches where low level refactorings are combined without targeting an specific anti-pattern, the sequence of refactorings operations generated by our approach contains all necessary steps to remove a particular type of anti-pattern. Before applying a refactoring operation, we check if it satisfies a set of pre- and post-conditions to preserve the semantic of the code. For example, one precondition is that we do not inline parent classes, as inlining such classes will introduce regression in the children. An example of post-condition is that after inlining a LC, there is no class in the system with the same signature of the LC. Other aspects of quality, such as cohesion, are also considered by our approach when applying a refactoring operation. For the inline class example, we select a destiny class that is related to the LC as much as possible. To select such a class, we iterate over all the classes in the systems, searching for methods and attributes that access the LC features directly, or by public accessors (getters or setters). From those classes we choose the one with

the large number of access to the LC.

Long parameter list classes are classes that contain one or more methods with an excessive number of parameters, in comparison with the rest of the entities. DECOR defines a threshold to detect when a method have excess of of parameters, based on the computation of boxplot statistics involving all the methods in the system. For example, class *RemoteIssue* from mylyn project (shown in Figure 6) has 21 parameters in its constructor, making it hard to understand and maintain.

The refactoring strategy consists in (1) extracting a new class for each long-parameter-list-method, that will encapsulate a group of parameters that are often passed together, and that can be used by more than one method or classes (improving the readability of the code); (2) updating the signature of each method to remove the migrated parameters, and update the callers and method body in the LP class, to instantiate and replace the parameter with the new parameter object.

```
public RemoteIssue(java.lang.String id,
        org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteVersion[]
             affectsVersions,
        java.lang.String assignee,
        java.lang.String[] attachmentNames,
        org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteComponent[] components
             ,
        java.    util.Calendar created,
        org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteCustomFieldValue[]
             customFieldValues,
        java.lang.String description,
        java.util.Calendar duedate,
        java.lang.String environment,
        org.eclipse.mylyn.internal.jira.core.wsdl.beans.RemoteVersion[] fixVersions,
        java.lang.String key,
        java.lang.String priority,
        java.lang.String project,
        java.lang.String reporter,
        java.lang.String resolution,
        java.lang.String status,
        java.lang.String summary,
        java.lang.String type,
        java.util.Calendar updated,
        java.lang.Long votes) { ... }
```

Figure 6: An example of Long Parameter list constructor detected in Mylyn.

Spaghetti code classes are those classes that implement long methods with no parameters at all, abusing of old procedural programming paradigm, and neglecting the advantages of object-oriented programming. Hence, the proposed refactoring strategy includes the extraction of one or more long methods as new objects. This requires creating a new class for each long method, where the local variables become fields, and a constructor that takes as a parameter a reference to the SC class; the body of the original method is copied to a new method *compute*, and any invocation of the methods in the original class will be referenced through the parameter (stored as final field) to the SC class. Finally, the *original* long method is replaced in the SC class by the creation of the new object, and a call to the *compute* method. Note that we updated the detection rule of spaghetti code defined in SAD to better reflect the definition in the litterature [57], where is stated that spaghetti code is a class with no hierarchy that declares long methods with no parameters. However, the detection condition for *method with many parameters* in SAD is set to number of parameters *inferior* to five. We modified the condition to methods with number of parameters equal to zero, to avoid detecting false positives of this anti-pattern. Note that we did not find instances of this anti-pattern in any of the projects studied, using neither the original nor the suggested fix, and for that reason we cannot provide any example.

In the case of classes affected by speculative generality, the definition states that there is an *abstract class* that is specialized *only by one* class, mainly for handling future enhancements that are not currently required, and thus it is not worthy to keep both classes in the system. We can observe this anti-pattern when we find a subclass and superclass that look-alike. For example, considering the classes *AbstractHandler* from packages org.eclipse.core.commands and org.eclipse.ui.commands in Platform project depicted in Figure 7. We can observe that these two classes are practically the same. In addition, there is no other class that inherits from the parent class AbstractHandler (core.commands), hence this case is candidate to apply collapse hierarchy refactoring.

To collapse hierarchy, we first pull up the methods and attributes from the

child class to the parent class, update the constructor, remove the children class from the system, remove the abstract modifier from the parent class and update the call sites, and types to point to the parent class. There is one case where we omit the application of this strategy, and it is when the child class is defined as inner class inside another class. Inner classes are an integral part of the event-handling mechanism in user interfaces events [60], which is far different from the definition and application of SG anti-pattern, and moving the features of those classes to another entities, may introduce a regression in the system, or deviate from the designer intention.

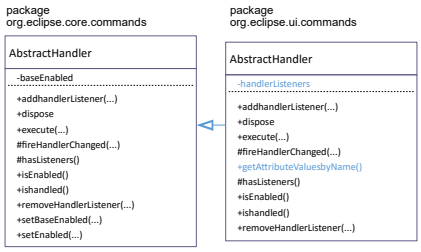

Figure 7: An example of Speculative Generality anti-pattern.

With this information, the map of anti-patterns and the relevant code entities, we automatically generate a list of candidate refactorings. The list of candidate refactorings, and the abstract model are the input of the search algorithm. The search algorithm generate a set of refactoring sequences. The refactoring sequences are evolved using the corresponding variation operators. All the candidate sequences are applied to a copy of the PADL model. Then the number of anti-patterns in the resulting model are computed, and sequence is evaluated using the objective function. The process finish when the stop condition is met. The final output is the best refactoring sequence for the current execution.

4.4. Analysis Method

We examine two scenarios: 1) developers perform a dedicated refactoring session after the completion of the task (*i.e.,* root-canal refactoring) and 2) de-

velopers intersperse refactorings among other changes during the task activity (*i.e.,* floss refactoring). In the first scenario, we generate all refactoring candidates in the system, while in the second scenario, we generate only refactorings that are relevant for the classes in the developer's context. The generated refactorings aim to remove the studied four anti-patterns. Note that we apply the corresponding patch to each snapshot to ensure that the refactoring opportunities generated are valid (*i.e.,* they do not remove changes essential to the successful completion of the task).

Due to the random nature of metaheuristic techniques employed in this paper, it is necessary to perform several independent runs to have an idea of the behavior of the algorithms. We execute 30 independent runs, which is a typically used value in the search-based research community.

We also compare the performance of the metaheuristics employed with random search to make sure that they can find better solutions than a pure random approach.

4.5. Results of the Experiment

This section presents and discusses the results of our experiment.

4.5.1. Individual task context versus accumulated task context

After applying their corresponding patch to each task snapshot, we perform floss refactoring (as described in Section4.4) and compare the count of anti-patterns before and after refactoring to assess the benefits of ReCon. We observe a small reduction in the number of anti-patterns as we can observe in Figure 8, where we present box plots of the 657 tasks of the Mylyn project. This result was expected because the number of relevant files for each task (*i.e.,* the developer's context) is small in general, and consequently, the number of refactoring opportunities too.

However, the accumulation of these small improvements (*i.e.,* reductions of anti-patterns occurrences) over a long period of time is likely to result in a significant improvement of the design quality of the system. To verify this

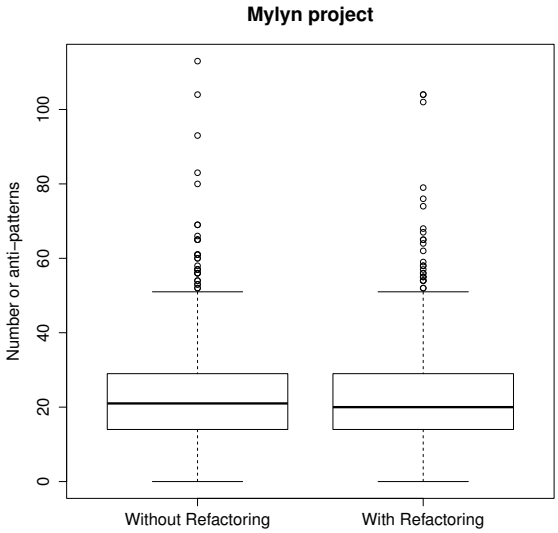

Figure 8: A comparison between the count of anti-patterns before and after applying floss-refactoring for each individual task using context.

hypothesis, we accumulate the contexts of all the individual tasks from the oldest to the most recent (ordered based on commit dates) and apply our automated floss refactoring approach using the accumulated context. This allows us to measure the accumulated impact of floss refactoring. We compute and compare the count of anti-patterns for (1) the source code without refactoring, (2) the source code after applying all the floss refactorings, and (3) the source code after performing a root-canal refactoring. In Figure 9, we present a comparison of the the anti-pattern's count for our three projects before and after refactoring (floss and root-canal). To verify if the observed differences (between the number of corrected anti-patterns for root-canal versus floss-refactoring) are statistically significant, we performed a Wilcoxon rank sum test [61] at 95% confidence level (*i.e.,* $\alpha = 5\%$). The test was statistically significant (*i.e.,* p-value<0.05), indicating that the distribution of the results is not the same for both groups. We also evaluated the magnitude of the difference by computing the Cohen's d effect size [62]. The results show that the difference is large for the three

projects ($d \geq 0.8$.).

Overall, we observe that our proposed automated floss refactoring approach can reduce approximately 50% of anti-patterns. This is a significant reduction considering the fact that it does not disrupt the developer's work flow, since it only recommends refactorings that affect files on which the developer is already working (i.e., files from the task context).

On the contrary, relying on root-canal refactoring is expensive. The number of refactoring opportunities detected go from 167 (Mylyn) to 2068 (Platform). However, applying floss refactoring with ReCon can alleviate this cost. From the individual tasks studied in this work we found that the tasks with more refactoring opportunities are: Mylyn task 87670, 34; PDE task 84503, 50; and Platform task 82540, 63. These number of refactorings, which might not be trivial to be generated manually, are feasible to be evaluated and applied for a developer with the help of our approach.

Nevertheless, after applying ReCon during the development and maintenance of a software system, developers can still perform a root-canal refactoring prior to the release of the system to remove the remaining anti-patterns. Figure 9 shows that a root-canal refactoring can be very effective at removing anti-patterns in a system. After the root-canal refactoring of Mylyn, PDE, and Platform, only respectively 1, 4, and 8 anti-patterns remained in the projects. We manually inspect these cases, and found that the anti-pattern remaining in Mylyn, that is a LP instance, was not removed because is inside an inner class for which our implementation of introduce parameter object is not suitable; in PDE two instances of lazy class could not be removed due to an issue with a missing package name; in Platform, half of the anti-patterns of SG type were not corrected because they refer to abstract classes belonging to external APIs (java.util, and java.io). Beside this drawbacks, we consider that the ReCon results are stable, and not biased towards any anti-pattern type.

Table 5: Count of anti-patterns after applying floss refactoring.

Anti-pattern	Original	GA	RS	SA	VNS
MYLYN					
SG	0	0	0	0	0
SC	0	0	0	0	0
LC	27	19	25	19	19
LP	140	49	94	49	49
Total	167	68	119	68	68
PDE					
SG	31	2	3	2	2
SC	0	0	0	0	0
LC	1205	180	193	180	180
LP	2276	1229	1320	1229	1229
Total	3512	1411	1516	1411	1411
PLATFORM					
SG	30	22	23	22	22
SC	0	0	0	0	0
LC	1242	336	341	336	336
LP	2286	1595	1651	1595	1595
Total	3558	1953	2015	1953	1953

In the following, we will analyze the performance of the metaheuristics employed in this work, and their corresponding resources consumption.

In Table 5 we present the average count of anti-patterns of the 30 independent runs for the three metaheuristics algorithms and random search in the accumulated floss refactoring scenario. As we can observe, the three metaheuristics are capable of removing the same number of anti-patterns, thought with some variations in the amount of memory and execution time required.

With respect to the instances of anti-patterns removed, there is little difference between the refactorings solutions found by each different metaheuristic, especially if we consider that the detection and generation of refactoring operations process is the same. However, the cpu time, and to some extent the memory consumption, that one algorithm takes to find the best combination of the refactorings is where we found more interesting differences. To corroborate this point, we manually compare the refactorings sequences and found that most of the differences are related to the position in which each metaheuristic

includes them in the sequence. This is true for the set of refactorings that are not conflicted, and do not required an specific order to be applied.

We also observe that metaheuristics overcome random search in all the projects studied. To corroborate this result, we apply the same statistical test, Wilcoxon rank sum and Cohen's d effect size, and found that the results are statistically different (p-value<0.05), and that difference between the metaheuristics and random search is medium (d=0.07) in terms of anti-patterns correction.

The resources usage is depicted in Figure 10 for each metaheuristic. We can observe that SA has the fastest execution among the three metaheuristics followed close by GA. We corroborate this result applying Wilcoxon test and Cohen's d size effect, and found that this result is statistically significant in comparison with GA and VNS, and with a large difference (d = 1.09, 7.31). Concerning memory usage, the difference is also significant, but with a small difference for GA (d=0.037) and large for VNS (d=5.68).

In a scenario where developers are more interested in obtaining a solution fastest, SA is the recommended algorithm. GA consumes less memory but with more variability in the execution time. VNS report the highest values for memory consumption and execution time, given that it has to analyze many neighborhoods before finding an optimal solution. In any case the execution time required to perform floss refactoring using context in each individual task is less than 200 seconds in average (in case someone opts for VNS), which is acceptable when performing a coding task.

4.5.2. Performance of the algorithms

Finally, in Table 6 we present the resources usage for root-canal using SA metaheuristic, as it is the one to find solutions in the shortest time. As we can expect, the execution time and memory required to perform is bigger for root-canal refactoring, and these values increase proportionally to the number of classes in the studied project. This is expected since we look for refactoring opportunities in all the classes in the system in root-canal refactoring, while in floss refactoring we focus only on classes that are in the developer's context.

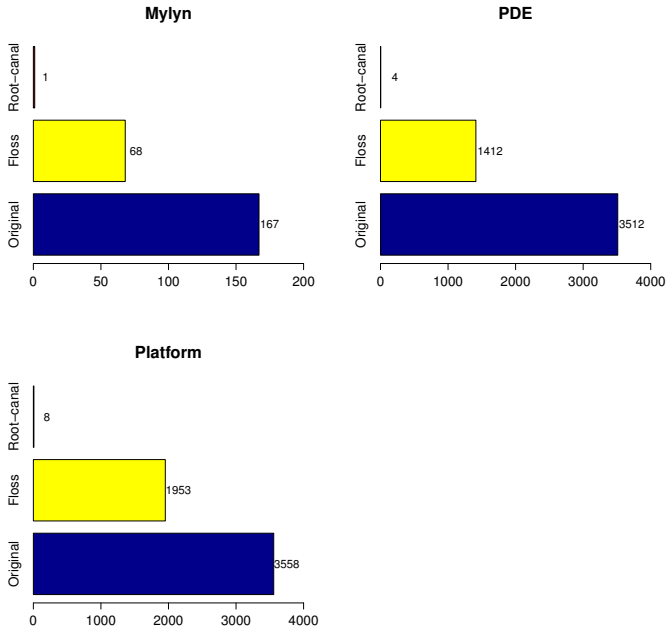

Figure 9: A comparison of anti-patterns occurrences after applying floss and root canal refactoring.

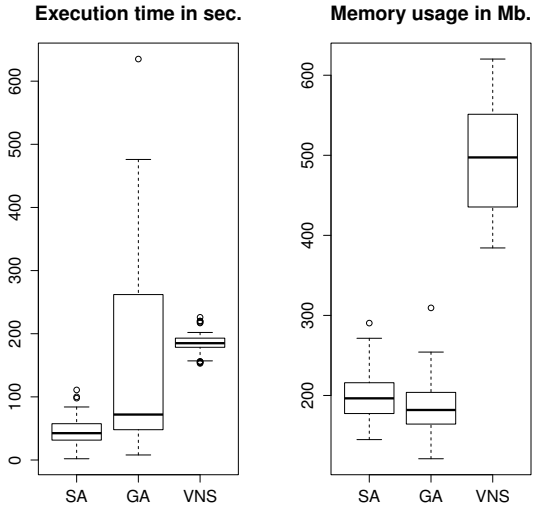

Figure 10: Resources consumption for each Algorithm when performing floss refactoring.

It is clear that there is a trade-off to make between the quality achieved and resources consumed, as the number of anti-patterns removed is less for floss refactoring.

Table 6: Resources usage for root-canal using SA.

Program	Memory usage (Mb).	Execution time (hh:mm:ss)
Mylyn	933.78	00:48:58
PDE	4505.83	10:44:15
Platform	5936.74	14:09:01

4.5.3. Quality evaluation

After analyzing our approach in terms of memory usage and execution time, we also consider important to assess the impact on the quality of the programs analyzed. For this purpose, we use the QMOOD (Quality Model for Object-Oriented Design) model [27] to evaluate the effect of the proposed refactoring sequences on certain quality attributes. The rational for selecting the QMOOD model is that previous studies have used it before to assess the effect of refactor-

ing [24, 25, 49], and it defines six desirable quality attributes (reusability, flexibility, understandability, functionality, effectiveness and extendibility) based on 11 object-oriented metrics. From these six quality attributes we only consider the following attributes:

- Reusability : the degree to which a software module or other work product can be used in more than one computer program or software system.
- Flexibility: the ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed.
- Understandability: the properties of designs that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure.
- Effectiveness: the design's ability to achieve desired functionality and behavior by using OO concepts.
- Extendibility: The degree to which a program can be modified to increase its storage or functional capacity.

The formulas to compute the aforementioned quality attributes are presented in Table 7, and the metrics in Table 8. We omit functionality, as by definition, refactoring is a behavior-preserving maintenance task, so we do not expect a raise in this quality function.

To compute the quality gain, we use the formula proposed in [63] where the total gain in quality G for each of the considered quality attributes q_i before and after refactoring is estimated as:

$$G_{qi} = q'_i - q_i, \tag{2}$$

where q'_i and q_i represents the value of the quality attribute i after and before refactoring.

Table 7: QMOOD evaluation functions.

Quality Factors	Quality Index Calculation
Reusability	-0.25 * DCC + 0.25 * CAM + 0.5 * CIS + 0.5 * DSC
Flexibility	0.25 * DAM - 0.25 * DCC + 0.5 * MOA +0.5 * NOP
Understandability	-0.33 * ANA + 0.33 * DAM - 0.33 * DCC + 0.33 * CAM -0.33 * NOP - 0.33 * NOM - 0.33 * DSC
Effectiveness	0.2 * ANA + 0.2 * DAM + 0.2 * MOA + 0.2 * MFA + 0.2 * NOP
Extendibility	0.5 * ANA -0.5 * DCC + 0.5 * MFA + 0.5 * NOP

Table 8: QMOOD quality metrics.

Design Property	Metric	Description
Design size	DSC	Design size in classes
Complexity	NOM	Number of methods
Coupling	DCC	Direct class coupling
Polymorphism	NOP	Number of polymorphic methods
Hierarchies	NOH	Number of hierarchies
Cohesion	CAM	Cohesion among methods in class
Abstraction	ANA	Average number of ancestors
Encapsulation	DAM	Data access metric
Composition	MOA	Measure of aggregation
Inheritance	MFA	Measure of functional abstraction
Messaging	CIS	Class interface size

In Figure 11, we can observe the quality gain obtained for each selected QMOOD attributes after applying root-canal and accumulated floss-refactoring using context. In both cases, the quality increases according to the five attributes. Reusability is the quality attribute that has the highest gain, while effectiveness has the lowest one (0.01,0.009), follow by flexibility (0.27, 0.19). We suggest that the negligible gain in effectiveness is due to the combination of metrics that does not penalize coupling like (DCC), that is impacted by the refactorings proposed in the case study. On the contrary, we observe that extendibility, which penalizes DCC with -0.5, show better results (0.46, 0.34). The low gain in flexibility is presumably due to the fact that a big portion of the weight of that quality attribute is on the *Number of polymorphism methods (NOP)* metric. This metric refers to methods that are overridden by one or more descendent classes. Since the refactorings applied on the programs do not override existing methods, as it is not required by the definition of the anti-patterns analyzed, the increment of this quality attribute is small. On the contrary, the reusability attribute which gives a high weighing to *Design Size* (Number of classes), and *Messaging* (communication between classes) metrics benefits from the decomposition on long parameter list, which is one of the most predominant anti-patterns in the three studied projects. Finally, the substantial increment in understandability reflects a drop in the complexity of the design structure. Understandability is one of the most desired attributes to achieve from the point of view of developers, as it eases the addition of new features and enhancements.

To summarize this section, we conclude that our proposed approach can successfully improve the quality of a software system, not only with respect to the number of anti-patterns corrected, but also in terms of reusability, understandability, and to a minor extent, flexibility.

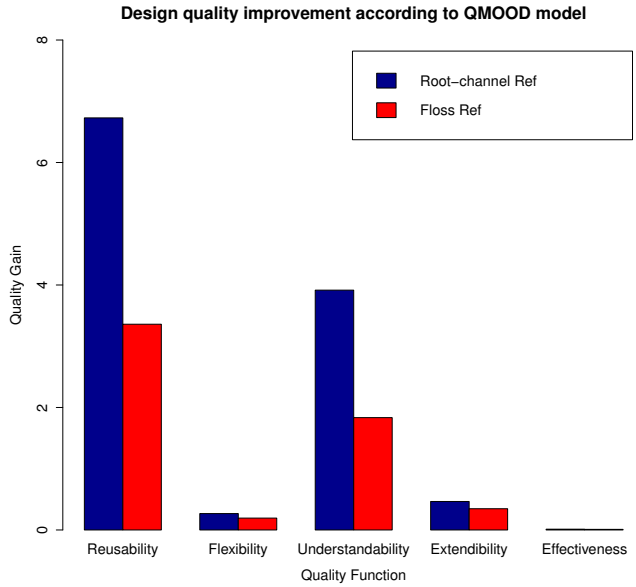

Figure 11: The impact of the best refactoring solutions on QMOOD quality attributes.

5. Discussion

Results from Section 4.5 show that our proposed approach ReCon is effective at correcting anti-patterns in software systems. ReCon can find refactoring solutions in a reasonable time using a reasonable amount of resources. The main contribution of ReCon is leveraging task context information to prioritize the refactoring of classes that undergo changes more often. This is especially convenient if we consider that the length of the sequence of refactorings is shorter in a floss scenario than a root canal one. The complexity of the scheduling of the refactorings is also simplified, as the number of possible conflicts is reduced, and finally it does not make too much sense to modify classes that do not change very often.

By contrasting the quality of the resulting design before and after refactoring using Recon in floss and root-canal scenarios give us an insight of the usefulness of the refactorings proposed not only in terms of anti-patterns correction, but in other quality attributes like coupling and design size.

Concerning to floss and root canal scenarios, one interesting finding is the distribution of anti-patterns among the classes that are touch by developers during a task context. For example, for Mylyn project, which has a total of 2365 classes, we covered 72% of them in the floss accumulated scenario (1697) and remove approximately 59% of anti-patterns; that means that 28% of the classes, which were not modified in our collected dataset, contain 41% of anti-patterns. The coverage of classes in PDE and platform is considerably less, 24% and 11%; however, the remaining anti-patterns in the untouched classes are 40% and 55% respectively. These results suggest that for PDE and platform, 60% of the anti-patterns studied are concentrated in a small portion of the system.

Finally, ReCon do not require any set of bad code examples to work like previous approaches [64, 30, 65, 63], so it can be used directly out of the box. Another advantage of ReCon is that the thresholds used for detecting the analyzed anti-patterns, can be easily modified according to the user needs through SAD and DETEX, without modifying any line of code in the implementation of ReCon.

6. Related Work

We report previous works related to anti-patterns, refactoring and the use of Mylyn context.

6.1. Anti-patterns

Anti-patterns such as those defined by Brown et al. [57] have been proposed to embody poor design choices. These anti-patterns stem from experienced software developers' expertise and are reported to negatively impact systems by making classes more change-prone [66] and defect-prone [8, 67]. They are opposite to design patterns [68], *i.e.,* they identify "poor" solutions to recurring design problems. For example, Brown et al. define 40 anti-patterns that describe the most common recurring pitfalls in the software industry [57]. Coplien and Harrison [12] described an anti-pattern as "something that looks like a good

idea, but which back-fires badly when applied". Khomh et al. [8] investigated MessageChains in ArgoUML, Eclipse, Mylyn, and Rhino and found them to be consistently related to high faults and change rates.

Concerning the detection of anti-patterns and code smells, we present the following representative works. Marinescu [69] proposed a metric-based approach to detect anti-patterns capturing deviations from "good design principles" through a set of rules comprised of metrics joined by set operators and relative thresholds. Munro [70] presented a similar rules metrics-based approach to detect code smells, and evaluated the choice of metrics and thresholds through an empirical study.

Moha et al., proposed a domain-specific language to characterize anti-patterns based on a literature review of existing work. They also proposed algorithms and a platform to automatically convert specifications into detections algorithms to apply in a software system. They achieved good precision and a perfect recall [52]. Our approach implements the detection rules defined by this work to identify anti-patterns in our studied systems.

Khomh et al., proposed a Bayesian approach to account for the uncertainty of the loosely specified definitions of anti-patterns. By computing the probability that a class participate in an anti-pattern, this approach allows quality analysts to prioritize the inspection of bad candidate classes [55]. These previous works has contributed significantly to the specification and automatic detection of antipatterns. However our approach aims to provide a mean to automatically remove the anti-patterns during maintenance sessions.

6.2. Refactoring

In an industrial setting, Rompaey et al. [21] found that refactoring can help to reduce over 50% of memory usage, and 33% startup time improvement in a telecommunication company. In a case study with several revisions of an open source project, Soetens and Demeyer [18] found that most refactorings tend to reduce the Cyclomatic complexity [71], especially when they target duplicate code. Du Bois et al. [20], performed an experiment with students and observed

that refactoring God classes improves the comprehensibility of the source code. In an industrial setting at Microsoft, Kim et al. [19] found that modules that underwent refactoring have less inter-module dependencies and less post-release faults.

Tsantalis et al., proposed different approaches to detect refactoring opportunities like extract method, move method, and remove non-trivial code smells like feature envy and type-checking, to improve the design quality of a system. They implemented their techniques as an Eclipse plug-in, named JDeodorant [4] allowing their evaluation on java source projects [72, 73, 74, 75]. Semi-automatic approaches provide an interesting compromise between fully automatic detection techniques and manual inspections. However, they require the developer to take decisions about the order of refactorings to be applied. On the contrary, our approach aims to relieve developers from the time-consuming task of selecting the best sequence of refactorings and evaluating their impact one by one.

6.3. Search-Based Refactoring

We present a sample of representative works in this category. O'Keeffe and Cinnéide [23] propose an approach that relies on the QMOOD model [27] to assess the quality of the candidate refactorings. They implement their approach using local search techniques, namely Simulated annealing (SA), and two versions of hill climbing. They found strong evidence that QMOOD flexibility and understandability attributes are the most suitable attributes to assess the quality of the refactoring solutions. The same authors extended this study in [49] by adding GA and compared the results of the four search techniques. They found Multiple-ascent hill climbing to be the most efficient search technique in terms of speed, quality obtained in different program inputs, and consistence for a different set of parameters; GA performs better with high values of cross-over and mutation; the effectiveness of simulated annealing varies in function of the

[4] http://www.jdeodorant.com

input program.

In this work we use GA and SA as a mean of comparison, because: GA is a global search algorithm and latest works on refactoring relied on variation of GA like Genetic Programming, NSGA-II, etc. SA is one of the algorithms that provide a strategy to escape local optima [48] and it has been applied to several combinatorial problems in combinatorial optimization (CO) like the Quadratic assignment problem (QAP) [76] and the Job Shop Scheduling problem (JSS) [77]. We use the same quality attributes to evaluate our approach, though our approach is anti-pattern-driven. In our approach the quality attribute with more gain is reusability, followed by understanding, and flexibility, while in their approach the order between reusability and understandability is swapped.

Seng et al. [24] propose an approach based on genetic algorithm, that aims to improve the cohesion of the entities through the implementation of the move method refactoring and evaluated the quality of the refactoring sequences with a fitness function that comprise coupling, cohesion, complexity and stability measurements.

Harman and Tratt [25] introduced a multi-objective approach for the problem of refactoring that allows to treat the refactoring problem as a multi-objective problem, where the goal is to find the *Pareto front*, *i.e.,* the set of solutions where there is no component that can be improved without decreasing the quality of another component. Thus, the outcome is not a single solution but a set of optimal solutions to be selected by the developer. Our work treat the problem of refactoring as a single objective, as we want to show that floss refactoring guided by the context is a good alternative to traditional automated refactoring approaches. We do not discard the use of multi-objective refactoring guided by context in the future.

Ouni et al. [26] propose a multi-objective evolutionary algorithm based on the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [78]. The two conflicting objectives of their approach are correcting the larger quantity of design defects, while preserving semantic coherence. For the first objective, they input

a set of rules to characterize design defects from the literature, and select the rules that detect the most design defects from a set of previous detected defects (example-based approach). The second objective is achieved by implementing two techniques to measure similarity among classes, when moving elements between them. The first technique evaluates the cosine similarity of the name of the constituents, *e.g.,* methods, fields, types. The second technique considers the dependencies between classes.

Moghadam and Cinnéide [40] propose an automated approach where the goal is to reach a desired design model, described as a UML diagram. The approach takes as input the source code of the *desired program* and the program to be improved; then, it abstracts the corresponding UML design models, and computes the differences between them. Next it maps the set of differences to source-level refactorings that will be applied in the code. The search problem consists in finding the larger sequence of refactorings that can be legally applied in the program. The metaheuristics algorithms used are the same as those implemented in [49]. The difference with our approach, is that the software designer need to provide a desired design, to allow the program to generate the refactoring sequences that are necessary to achieved this goal; however our approach do not require to provide this desired model and can work out of the box.

Mkaouer et al. [79] propose an extension of the work presented in [26], by allowing the user to interact with the candidate solutions found by the multi-objective genetic algorithm. Their approach consists in the following steps: (1) a NSGA-II algorithm proposes a set of refactoring sequences that satisfies three conflicting objectives, *i.e.,* improving software quality (based on QMOOD model), minimizing the number of refactorings and preserving semantic coherence; (2) an algorithm ranks the candidates solutions, and presents them to the user, according to the candidates features (number of occurrences in the Pareto front, candidates' order, and user's feedback); (3) a local-search algorithm updates the set of solutions after several iterations with the user, or when several program changes have been applied.

Our proposed approach applies the same metaheuristics used in the aforementioned works. However, it differs in the following points: (1) while all of these approaches implemented genetic algorithm variations (single and multi-objective, *i.e.,* NSGA-II), our approach suggests to change the focus of research to floss refactoring guided by developer's context. We consider it to be more suitable for the refactoring problem because: the maintenance activities can be interspersed with refactoring suggestions, similar to "quick fixes" provide by current developers IDEs, like Eclipse, IntelliJ IDEA, etc. Hence the developer can select among a reasonable number of candidate refactorings while working, and improve the quality of the system incrementally. In addition to this 4 out of 7 existing refactoring approaches require the user to input a set of defects examples to generate the detection rules, however, in practice it is not feasible to ask such a bothering task to the users, especially since the first motivation of an automated approach is to relieve the burden of refactoring activities on users, by suppressing manual error-prone activities.

6.4. Usage of Mylyn Context

Mylyn context have been used in two different ways: (1) to assist developers during task resolutions *i.e.,* the context is collected and directly used during the resolution of the current task, and (2) to understand developers' activities.

Kersten and Murphy [80] used the task context to reduce information overhead by filtering and keeping in the developers' environment (*i.e.,* package explorer in the IDE) only the program entities relevant to the developer's task. This prevents the developer from searching for relevant information in a large information space; improving the developer's productivity. Robbes and Lanza [81] also used developer's previously collected contexts to build a code completion tool that reduces developers' scrolling effort. Users found their proposed tool to be more accurate than previous tools. Recently, Lee et al. [82] proposed an approach (named MI) to recommend relevant entities to developers. They used both view and selection activities on the entities from the developer's context and mine association rules to identify relevant entities.

Among the studies that used Mylyn context to examine developers' activities is the work of Sanchez et al. [83], who studied developers interruptions and found that work fragmentation is correlated with lower productivity. Ying and Robillard [84] and Zhang et al. [85] studied how developers perform editing activities. Ying and Robillard [84] defined file editing styles (edit-first, edit-last, and edit-throughout) and found that enhancement tasks are associated with a high fraction of edit events at the beginning of the programming session(*i.e.,* edit-first). Zhang et al. [84] characterize how several developers concurrently edit a file and derive concurrent, parallel, extended, and interrupted file editing patterns. They found these file editing patterns to be related to future faults. Soh et al. [86] used Mylyn context to study how developers' navigate through program entities. They found that developers spend more effort on tasks when they exhibit unreferenced exploration (*i.e.,* program entities are almost equally revisited) compared to reference exploration (*i.e.,* revisitation of a set of entities).

To the best of our knowledge, none of the previous works that used developers' context aimed to perform automated software refactoring.

7. Threats to validity

We now discuss the threats to validity of our study following common guidelines for empirical studies [87].

Construct validity threats concern the relation between theory and observation. Our modeling approach assumes that each anti-pattern is of equal importance, when in reality, this may not be the case.

Threats to internal validity concern our selection of subject systems, tools, and analysis method. The accuracy of DECOR impacts our results. DECOR is an academic tool which has been reported to achieve high recall and reasonable precision [52]. However, other anti-pattern detection techniques and tools may provide different results. The rational behind using Mylyn's interaction histories is that Mylyn plug-in is the only tool that has been applied to several

open-source projects to gather developers' interactions and these are publicly available. Note that the projects analyzed are the top-three open-source projects with more interaction histories.

Conclusion validity threats are related to the violation of the assumptions of the statistical tests and the diversity of our dataset. We used non-parametric tests (Wilcoxon rank sum) that make no assertion about the distribution of the data. We used data from three open-source projects that have different sizes and involve many developers.

External validity threats relate to the generalization of our results. Because our subject projects are open-source and because we used a particular yet representative subset of anti-patterns as proxy for software design quality, we cannot guarantee that the findings of this study can generalize to proprietary software projects and other open-source projects. In the future, we plan to analyze more projects, including proprietary projects and projects written in different programming languages, to draw more general conclusions.

Reliability validity threats concern the possibility of replicating this study. All the raw data used in this paper are available in Eclipse Bugzilla. The projects studied in this paper are also available online for the public.

8. Conclusion

In this paper, we propose a novel approach to solve the problem of correcting anti-patterns. Previous approaches from the literature recommend refactoring opportunities to developers without considering their coding tasks, even though studies (*e.g.,* [22]) have found that developers prefer refactoring suggestions that can be applied to files that are active in their workspace. This lack of consideration for developers' context may explain the poor adoption of automated refactoring approaches in industry. In addition to this, many of the existing approaches require that developers input a set of bad code examples, to generate detection rules, or a desired model to generate the corresponding refactoring solution. These requirements put extra work on developers, slowing the refactor-

ing process, and even rendering it impractical in certain cases. To address these issues, we propose ReCon, an automated refactoring approach that leverages developer's context and metaheuristic techniques to compute the best sequence of refactoring that affects only entities in the developer's context. We performed a case study using three open-source project and found that ReCon can successfully correct more than 50% of anti-patterns in a project using less resources than the traditional approaches from the literature. More importantly, ReCon does not disrupt the developer's work flow, since it only recommends refactorings that affect files on which the developer is already working (*i.e.,* files from the task context).

We also assess the quality of our subject projects before and after applying ReCon, using five quality attributes defined in the QMOOD model [27]. Results show that ReCon can achieve a significant quality improvement in terms of reusability, understandibility, extendibility and to some extent flexibility, while effectiveness reports a negligible increment.

As a future work, we plan to extend our approach ReCon to include the correction of more object-oriented anti-patterns. We also plan to add more objective functions to capture other quality aspects in the search of more human-like refactoring operations. For example, we will use historical information and relationships between classes apart from context metrics, like Mylyn's Degree of Interest (DOI), to guide the search of new refactoring opportunities.

References

[1] M. E. Fagan, Design and code inspections to reduce errors in program development, IBM Systems Journal 15 (3) (1976) 182–211.

[2] W. S. Humphrey, T. R. Snyder, R. R. Willis, Software Process Improvement at Hughes Aircraft, IEEE Software 8 (4) (1999) 11–23.

[3] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz, What we have learned about fighting

defects, in: Proc. of the 8th IEEE Symposium on Software Metrics, 2002, pp. 249–258.

[4] S. McConnell, Code Complete, Microsoft, 2004.

[5] M. M. Lehman, Feedback in the software evolution process, Information and Software Technology 38 (11) (1996) 681–686.

[6] C. Izurieta, Decay and grime buildup in evolving object oriented design patterns, Ph.D. thesis (2009).

[7] S. Vaucher, F. Khomh, N. Moha, Y. Gueheneuc, Tracking design smells: Lessons from a study of god classes, in: Reverse Engineering, 2009. WCRE '09. 16th Working Conference on, 2009, pp. 145–154.

[8] F. Khomh, M. D. Penta, Y.-G. Gueheneuc, G. Antoniol, An exploratory study of the impact of antipatterns on class change- and fault-proneness, Empirical Softw. Engg. 17 (3) (2012) 243–275.

[9] W. Cunningham, The wycash portfolio management system (1992).

[10] D. L. Parnas, Software aging, in: ICSE '94: Proc. of the 16th Int'l conference on Software engineering, IEEE Computer Society Press, 1994, pp. 279–287.

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

[12] J. O. Coplien, N. B. Harrison, Organizational Patterns of Agile Software Development, 1^{st} Edition, Prentice-Hall, Upper Saddle River, NJ (2005), 2005.

[13] A. J. Riel, Object-oriented design heuristics, Vol. 335, Addison-Wesley Reading, 1996.

[14] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, Do they really smell bad? a study on developers' perception of bad code smells, in:

Software Maintenance and Evolution (ICSME), 2014 IEEE Int'l Conference on, IEEE, 2014, pp. 101–110.

[15] M. Abbes, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension, in: Software Maintenance and Reengineering (CSMR), 2011 15th European Conf. on, 2011, pp. 181–190.

[16] S. Vaucher, F. Khomh, N. Moha, Y.-G. Gueheneuc, Tracking design smells: Lessons from a study of god classes, in: Reverse Engineering, 2009. WCRE'09. 16th Working Conf. on, IEEE, 2009, pp. 145–154.

[17] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, in: Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. on the, IEEE, 2010, pp. 106–115.

[18] Q. D. Soetens, S. Demeyer, Studying the effect of refactorings: A complexity metrics perspective, in: Quality of Information and Communications Technology (QUATIC), 2010 7th Int'l Conf. On the, 2010, pp. 313–318.

[19] M. Kim, T. Zimmermann, N. Nagappan, A field study of refactoring challenges and benefits, in: Proc. of the ACM SIGSOFT 20th Int'l Symposium on the Foundations of Softw. Eng., FSE '12, ACM, 2012, pp. 50:1–50:11.

[20] B. D. Bois, S. Demeyer, J. Verelst, T. Mens, M. Temmerman, Does god class decomposition affect comprehensibility?, in: IASTED Conf. on Software Engineering, 2006, pp. 346–355.

[21] B. van Rompaey, B. Du Bois, S. Demeyer, J. Pleunis, R. Putman, K. Meijfroidt, J. C. Dueas, B. Garcia, Serious: Software evolution, refactoring, improvement of operational and usable systems, in: Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conf. On, 2009, pp. 277–280.

[22] E. Murphy-Hill, C. Parnin, A. P. Black, How we refactor, and how we know it, Software Engineering, IEEE Transactions On 38 (1) (2012) 5–18.

[23] M. O'Keeffe, M. O. Cinneide, Search-based software maintenance, in: Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on, 2006, pp. 10 pp.–260. doi:10.1109/CSMR.2006.49.

[24] O. Seng, J. Stammel, D. Burkhart, Search-based determination of refactorings for improving the class structure of object-oriented systems, GECCO 2006: Genetic and Evolutionary Computation Conference, Vol 1 and 2 (2006) 1909–1916.

[25] M. Harman, L. Tratt, Pareto optimal search based refactoring at the design level, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, 2007, pp. 1106–1113. doi:10.1145/1276958.1277176.

[26] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, Search-based refactoring: Towards semantics preservation, in: Software Maintenance (ICSM), 2012 28th IEEE International Conference on, IEEE, 2012, pp. 347–356.

[27] J. Bansiya, C. G. Davis, A hierarchical model for object-oriented design quality assessment, Software Engineering, IEEE Transactions on 28 (1) (2002) 4–17.

[28] T. Mens, T. Tourwé, A survey of software refactoring, Software Engineering, IEEE Transactions on 30 (2) (2004) 126–139.

[29] T. Mens, G. Taentzer, O. Runge, Analysing refactoring dependencies using graph transformation, Software & Systems Modeling 6 (3) (2007) 269–285.

[30] A. Ouni, M. Kessentini, H. Sahraoui, M. Boukadoum, Maintainability defects detection and correction: a multi-objective approach, Automated Software Engineering 20 (1) (2013) 47–79.

[31] E. Murphy-Hill, A. P. Black, Refactoring tools: Fitness for purpose, Software, IEEE 25 (5) (2008) 38–44.

[32] A. Ouni, M. Kessentini, H. Sahraoui, M. S. Hamdi, The use of development history in software refactoring using a multi-objective evolutionary algorithm, in: Proceedings of the 15th annual conference on Genetic and evolutionary computation, ACM, 2013, pp. 1461–1468. doi:10.1145/2463372.2463554.

[33] Mylyn wiki, `http://wiki.eclipse.org/Mylyn`, accessed: 2016-02-23.

[34] L. M. Layman, L. A. Williams, R. St Amant, Mimec: intelligent user notification of faults in the eclipse ide, in: Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering, ACM, 2008, pp. 73–76.

[35] Mylyn task-focused interface, `http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.mylyn.help.ui%2FMylyn%2FUser_Guide%2FTask-Focused-Interface.html`, accessed: 2016-02-23.

[36] S. Kirkpatrick, Optimization by simulated annealing: Quantitative studies, Journal of statistical physics 34 (5-6) (1984) 975–986.

[37] D. Whitley, A genetic algorithm tutorial, Statistics and computing 4 (2) (1994) 65–85.

[38] D. Whitley, An overview of evolutionary algorithms: practical issues and common pitfalls, Information and software technology 43 (14) (2001) 817–831.

[39] N. Mladenović, P. Hansen, Variable neighborhood search, Computers & Operations Research 24 (11) (1997) 1097–1100.

[40] I. H. Moghadam, M. O. Cinneide, Code-imp: A tool for automated search-based refactoring, in: Proceedings of the 4th Workshop on Refactoring Tools, IEEE Computer Society, 2011, pp. 41–44.

[41] M. Amoui, S. Mirarab, S. Ansari, C. Lucas, A genetic algorithm approach to design evolution using design pattern transformation, International Journal of Information Technology and Intelligent Computing 1 (2) (2006) 235–244.

[42] S. Bouktif, G. Antoniol, E. Merlo, M. Neteler, A novel approach to optimize clone refactoring activity, in: Proceedings of the 8th annual conference on Genetic and evolutionary computation, ACM, 2006, pp. 1885–1892.

[43] M. O'Keeffe, M. Ó. Cinnéide, Search-based refactoring: an empirical study, Journal of Software Maintenance and Evolution: Research and Practice 20 (5) (2008) 345–364.

[44] T. G. Crainic, M. Gendreau, P. Hansen, N. Mladenović, Cooperative parallel variable neighborhood search for the p-median, Journal of Heuristics 10 (3) (2004) 293–314.

[45] P. Hansen, N. Mladenović, D. Urošević, Variable neighborhood search and local branching, Computers & Operations Research 33 (10) (2006) 3034–3045.

[46] V. C. Hemmelmayr, K. F. Doerner, R. F. Hartl, A variable neighborhood search heuristic for periodic routing problems, European Journal of Operational Research 195 (3) (2009) 791–802.

[47] C. H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[48] C. Blum, A. Roli, Metaheuristics in combinatorial optimization: Overview and conceptual comparison, ACM Comput. Surv. 35 (3) (2003) 268–308. doi:10.1145/937503.937505.
URL http://doi.acm.org/10.1145/937503.937505

[49] M. O'Keeffe, M. O. Cinneide, Getting the most from search-based refactoring, Gecco 2007: Genetic and Evolutionary Computation Conference, Vol 1 and 2 (2007) 1114–1120doi:10.1145/1276958.1277177.

[50] D. Romano, S. Raemaekers, M. Pinzger, Refactoring fat interfaces using a genetic algorithm, Tech. rep., Delft University of Technology, Software Engineering Research Group (2014).

[51] Z. Soh, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, Towards understanding how developers spend their effort during maintenance activities, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, 2013, pp. 152–161.

[52] N. Moha, Y.-G. Gueheneuc, L. Duchien, A. Le Meur, Decor: A method for the specification and detection of code and design smells, Software Engineering, IEEE Transactions on 36 (1) (2010) 20–36.

[53] M. Fowler, Refactoring – Improving the Design of Existing Code, 1^{st} Edition, Addison-Wesley, 1999.

[54] E. Van Emden, L. Moonen, Java quality assurance by detecting code smells, in: Reverse Engineering, 2002. Proceedings. Ninth Working Conference on, IEEE, 2002, pp. 97–106.

[55] F. Khomh, S. Vaucher, Y.-G. Gueheneuc, H. Sahraoui, A bayesian approach for the detection of code and design smells, in: Quality Software, 2009. QSIC'09. 9th International Conference on, IEEE, 2009, pp. 305–314.

[56] M. Fowler, Refactoring: improving the design of existing code, Pearson Education India, 1999.

[57] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, T. J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, 1^{st} Edition, John Wiley and Sons, 1998.

[58] W. F. Opdyke, Refactoring object-oriented frameworks, Ph.D. thesis, University of Illinois at Urbana-Champaign (1992).

[59] M. Lanza, R. Marinescu, Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems, Springer Science & Business Media, 2007.

[60] Inner class example, `https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html`, accessed: 2015-06-03.

[61] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.

[62] J. Cohen, Statistical power analysis for the behavioral sciences (rev, Lawrence Erlbaum Associates, Inc, 1977.

[63] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, M. S. Hamdi, Improving multi-objective code-smells correction using development history, Journal of Systems and Software 105 (0) (2015) 18 – 39.

[64] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, A. Ouni, Design defects detection and correction by example, in: Program Comprehension (ICPC), 2011 IEEE 19th International Conference on, IEEE, 2011, pp. 81–90.

[65] I. H. Moghadam, M. O. Cinneide, Automated refactoring using design differencing, in: Software Maintenance and Reengineering (CSMR), 16th European Conference on, Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR, IEEE Computer Society, 2012, pp. 43 – 52.

[66] F. Khomh, M. Di Penta, Y. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, in: Reverse Engineering, 2009. WCRE '09. 16th Working Conference on, 2009, pp. 75–84.

[67] S. E. S. Taba, F. Khomh, Y. Zou, A. E. Hassan, M. Nagappan, Predicting bugs using antipatterns, in: Proc. of the 29th Int'l Conference on Software Maintenance, 2013, pp. 270–279.

[68] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, 1^{st} Edition, Addison-Wesley, 1994.

[69] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: IEEE Int'l Conference on Software Maintenance, ICSM, IEEE Computer Society, 2004, pp. 350–359.

[70] M. Munro, Product metrics for automatic identification of "bad smell" design problems in java source-code, in: Software Metrics, 2005. 11th IEEE International Symposium, 2005, pp. 15–15. doi:10.1109/METRICS.2005.38.

[71] T. McCabe, A complexity measure, IEEE Trans. Softw. Eng. SE-2 (4) (1976) 308–320.

[72] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of type-checking bad smells, in: Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on, IEEE, 2008, pp. 329–331.

[73] M. Fokaefs, N. Tsantalis, E. Stroulia, A. Chatzigeorgiou, Jdeodorant: identification and application of extract class refactorings, in: Proceedings of the 33rd International Conference on Software Engineering, ACM, 2011, pp. 1037–1039.

[74] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, Jdeodorant: Identification and removal of feature envy bad smells, in: Software Maintenance, 2007. ICSM 2007. IEEE International Conference on, 2007, pp. 519–520.

[75] N. Tsantalis, A. Chatzigeorgiou, Identification of extract method refactoring opportunities for the decomposition of methods, Journal of Systems and Software 84 (10) (2011) 1757–1782.

[76] D. T. Connolly, An improved annealing scheme for the qap, European Journal of Operational Research 46 (1) (1990) 93–100.

[77] P. J. Van Laarhoven, E. H. Aarts, J. K. Lenstra, Job shop scheduling by simulated annealing, Operations research 40 (1) (1992) 113–125.

[78] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, Evolutionary Computation, IEEE Transactions on 6 (2) (2002) 182–197.

[79] M. W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, M. Ó Cinnéide, Recommendation system for software refactoring using innovization and interactive dynamic optimization, in: Proceedings of the 29th ACM/IEEE Int'l Conf. on Automated software engineering, ACM, 2014, pp. 331–336.

[80] M. Kersten, G. C. Murphy, Using task context to improve programmer productivity, in: Proceedings of the 14th ACM SIGSOFT/FSE, 2006, pp. 1–11.

[81] R. Robbes, M. Lanza, Improving code completion with program history, Automated Software Engineering 17 (2) (2010) 181–212.

[82] S. Lee, S. Kang, S. Kim, M. Staats, The impact of view histories on edit recommendations, Software Engineering, IEEE Transactions on 41 (3) (2015) 314–330.

[83] H. Sanchez, R. Robbes, V. M. Gonzalez, An empirical study of work fragmentation in software evolution tasks, in: Proceedings SANER, 2015, pp. 251–260.

[84] A. Ying, M. Robillard, The influence of the task on programmer behaviour, in: Proceedings ICPC, 2011, pp. 31–40.

[85] F. Zhang, F. Khomh, Y. Zou, A. E. Hassan, An empirical study of the effect of file editing patterns on software quality, in: Proceedings WCRE, 2012, pp. 456–465.

[86] Z. Soh, F. Khomh, Y.-G. Gueheneuc, G. Antoniol, B. Adams, On the effect of program exploration on maintenance tasks, in: Reverse Engineering (WCRE), 2013 20th Working Conference on, 2013, pp. 391–400.

[87] R. K. Yin, Case Study Research: Design and Methods - Third Edition, 3rd Edition, SAGE Publications, 2002.

Biography

Rodrigo Morales is a Ph.D. candidate at Polytechnique Montreal. He earned his Bsc. degree in computer science in 2005 from Polytechnic of Mexico. And in 2008, he earned his Msc. in computer technology from the same University, where he also worked as a Professor in the computer Science department for five years. He has also worked in the bank industry as a software developer for more than three years. He is currently supervised by Foutse Khomh, Giuliano Antoniol (Poly Montreal), and Francisco Chicano (ETS Spain). His research interests are software design quality, anti-patterns and automated-refactoring.

Zephyrin Soh is a postdoc in the GIGL Department of the École Polytechnique, Canada. He received a Ph.D in Software Engineering from the same

University in 2015. His primary research interests are interaction traces, Eye-tracking, Program Exploration, and software comprehension.

Foutse Khomh is an assistant professor at the École Polytechnique de Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM, MSR, SANER, ICWS, HPCC, IPCCC, JSS, JSP, and EMSE. He has served on the program committees of several international conferences including ICSM, SANER, MSR, ICPC, SCAM, ESEM and has reviewed for top international journals such as SQJ, EMSE, TSE and TOSEM. He is on the Review Board of EMSE. He is program chair for Satellite Events at SANER 2015 and program co-chair for SCAM 2015. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and guest editor for a special issue on Release Engineering in the IEEE Software magazine.

Giuliano Antoniol (Giulio) received his Laurea degree in electronic engineering from the Universita' di Padova, Italy, in 1982. In 2004 he received his PhD in Electrical Engineering at the Ecole Polytechnique de Montreal. He worked in companies, research institutions and universities. In 2005 he was awarded the Canada Research Chair Tier I in Software Change and Evolution. He has participated in the program and organization committees of numerous IEEE-sponsored international conferences. He served as program chair, industrial chair, tutorial, and general chair of international conferences and workshops. He is a member of the editorial boards of four journals: the Journal of Software Testing Verification & Reliability, the Journal of Empirical Software Engineering and the Software Quality Journal and the Journal of Software Maintenance and Evolution: Research and Practice. Dr Giuliano Antoniol served as Deputy Chair of the Steering Committee for the IEEE International Conference on Software Maintenance. He contributed to the program committees of more than 30 IEEE and ACM conferences and workshops, and he acts as referee for all major software engineering journals. He is currently Full Professor at the Ecole Polytechnique de Montreal, where he works in the area of software evolution, software traceability, search based software engineering, software testing and software maintenance.

Francisco Chicano holds a PhD in Computer Science from the University of Malaga and a Degree in Physics from the National Distance Education University. Since 2008 he is with the Department of Languages and Computing Sciences of the University of Malaga. His research interests include the application of search techniques to Software Engineering problems. In particular, he contributed to the domains of software testing, model checking, software project scheduling, and requirements engineering. He is also interested in the application of theoretical results to efficiently solve combinatorial optimization problems. He is in the editorial board of several international journals and has been programme chair in international events.