



# Controlling the Attack Surface of Object-Oriented Refactorings

Sebastian Ruland<sup>1</sup>✉, Géza Kulcsár<sup>1</sup>, Erhan Leblebici<sup>1</sup>,  
Sven Peldszus<sup>2</sup>, and Malte Lochau<sup>1</sup>

<sup>1</sup> Real-Time Systems Lab, TU Darmstadt, Darmstadt, Germany  
{sebastian.ruland,geza.kulcsar,erhan.leblebici,  
malte.lochau}@es.tu-darmstadt.de

<sup>2</sup> Institute for Software Technology, University of Koblenz-Landau,  
Koblenz, Germany  
speldszus@uni-koblenz.de

**Abstract.** Refactorings constitute an effective means to improve quality and maintainability of evolving object-oriented programs. Search-based techniques have shown promising results in finding optimal sequences of behavior-preserving program transformations that (1) maximize code-quality metrics and (2) minimize the number of changes. However, the impact of refactorings on extra-functional properties like security has received little attention so far. To this end, we propose as a further objective to minimize the attack surface of programs (i.e., to maximize strictness of declared accessibility of class members). Minimizing the attack surface naturally competes with applicability of established *MoveMethod* refactorings for improving coupling/cohesion metrics. Our tool implementation is based on an EMF meta-model for Java-like programs and utilizes MOMoT, a search-based model-transformation framework. Our experimental results gained from a collection of real-world Java programs show the impact of attack surface minimization on design-improving refactorings by using different accessibility-control strategies. We further compare the results to those of existing refactoring tools.

## 1 Introduction

The essential activity in designing object-oriented programs is to identify class candidates and to assign *responsibility* (i.e., data and operations) to them. An appropriate solution to this *Class-Responsibility-Assignment (CRA)* problem, on the one hand, intuitively reflects the problem domain and, on the other hand, exhibits acceptable quality measures [4]. In this context, *refactoring* has become a key technique for agile software development: productive program-evolution phases are interleaved with behavior-preserving code transformations for updating CRA decisions, to proactively maintain, or even improve, code-quality metrics [13, 29]. Each refactoring pursues a trade-off between two major, and generally contradicting, objectives: (1) maximizing code-quality metrics, including fine-grained coupling/cohesion measures as well as coarse-grained anti-pattern

avoidance, and (2) minimizing the number of changes to preserve the initial program design as much as possible [8]. Manual search for refactorings sufficiently meeting both objectives becomes impracticable already for medium-size programs, as it requires to find optimal sequences of interdependent code transformations with complex constraints [10]. The very large search space and multiple competing objectives make the underlying optimization problem well-suited for search-based optimization [15] for which various semi-automated approaches for recommending refactorings have been recently proposed [18, 27, 28, 30, 34].

The validity of proposed refactorings is mostly concerned with purely *functional* behavior preservation [24], whereas their impact on *extra-functional* properties like program security has received little attention so far [22]. However, applying elaborated information-flow metrics for identifying security-preserving refactorings is computationally too expensive in practice [36]. As an alternative, we consider *attack-surface metrics* as a sufficiently reliable, yet easy-to-compute indicator for preservation of program security [20, 41]. *Attack surfaces* of programs comprise all conventional ways of entering a software by users/attackers (e.g., invoking API methods or inheriting from super-classes) such that an unnecessarily large surface increases the danger of exploiting vulnerabilities. Hence, the goal of a secure program design should be to grant least privileges to class members to reduce the extent to which data and operations are exposed to the world [41]. In JAVA-like languages, accessibility constraints by means of modifiers `public`, `private` and `protected` provide a built-in low-level mechanism for controlling and restricting information flow within and across classes, sub-classes and packages [38]. Accessibility constraints introduce compile-time security barriers protecting trusted system code from untrusted mobile code [19]. As a downside, restricted accessibility privileges naturally obstruct possibilities for refactorings, as CRA updates (e.g., moving members [34]) may be either rejected by those constraints, or they require to relax accessibility privileges, thus increasing the attack surface [35].

In this paper, we present a search-based technique to find optimal sequences of refactorings for object-oriented JAVA-like programs, by explicitly taking accessibility constraints into account. To this end, we do not propose novel refactoring operations, but rather apply established ones and control their impact on attack-surface metrics. We focus on *MoveMethod* refactorings which have been proven effective for improving CRA metrics [34], in combination with operations for on-demand strengthening and relaxing of accessibility declarations [38]. As objectives, we consider **(O1)** *elimination of design flaws*, particularly, **(O1a)** optimization of object-oriented coupling/cohesion metrics [5, 6] and **(O1b)** avoidance of anti-patterns, namely *The Blob*, **(O2)** *preservation of original program design* (i.e., minimizing the number of change operations), and **(O3)** *attack-surface minimization*. Our model-based tool implementation, called GOBLIN, represents individuals (i.e., intermediate refactoring results) as program-model instances complying to an EMF meta-model for JAVA-like programs [33]. Hence, instead of regenerating source code after every single refactoring step, we apply and evaluate sequences of refactoring operations, specified as model-transformation rules in HENSHIN [2], on the program model. To this end,

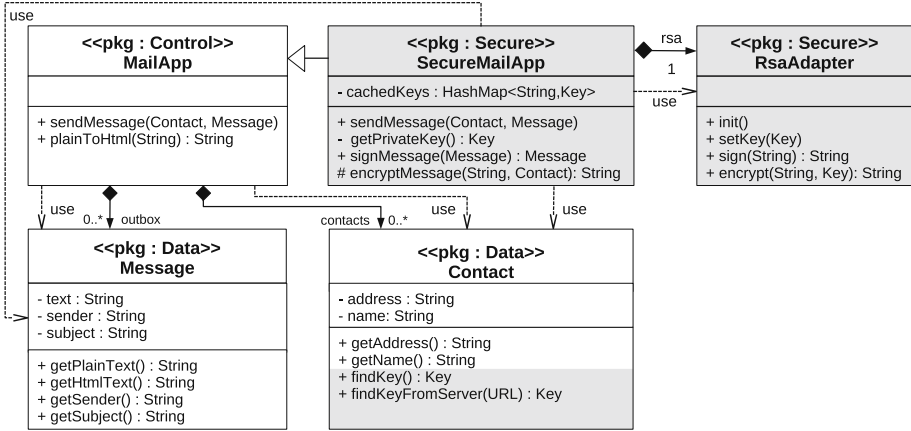


Fig. 1. UML class diagram of MAILAPP

we apply MOMoT [11], a generic framework for search-based model transformations. Our experimental evaluation results gained from applying GOBLIN as well as the recent tools JDEODORANT [12] and CODE-IMP [27] to a collection of real-world JAVA programs provide us with in-depth insights into the subtle interplay between traditional code-quality metrics and attack-surface metrics. Our tool and all experiment results are available on the GitHub site of the project<sup>1</sup>.

## 2 Background and Motivation

We first introduce a running example to provide the necessary background and to motivate the proposed refactoring methodology.

**Running Example.** We consider a (simplified) e-mail client, called MAILAPP, implemented in JAVA. Figure 1 shows the UML class diagram of MAILAPP, where security-critical extensions (in gray) will be described below. We use stereotype `<<pkg : name>>` to annotate classes with package declarations. Central class **MailApp** is responsible for handling objects of classes **Message** and **Contact** both encapsulating application data and operations to access those attributes. The text of a message may be formatted as plain String, or it may be converted into HTML using method `plainToHtml()`.

**Design Flaws in Object-Oriented Programs.** The over-centralized architectural design of MAILAPP, consisting of a predominant *controller class* (**MailApp**) intensively accessing inactive *data classes* (**Message** and **Contact**), is frequently referred to as *The Blob* anti-pattern [7]. As a consequence, method `plainToHtml()` in class **MailApp** frequently calls method `getPlainText()` in class **Message** across

<sup>1</sup> <https://github.com/Echtzeitsysteme/goblin>.

class- and even package-boundaries. *The Blob* and other *design flaws* are widely considered harmful with respect to software quality in general and program maintainability in particular [7]. For instance, assume a developer to extend `MailApp` by (1) adding further classes `SecureMailApp` and `RsaAdapter` for encrypting and signing messages, and by (2) extending class `Contact` with public RSA key handling: method `findKey()` searches for public RSA keys of contacts by repeatedly calling method `findKeyFromServer()` with the URL of available key servers. This *program evolution* further decays the already flawed design of `MAILAPP` as class `SecureMailApp` may be considered as a second instance of *The Blob* anti-pattern: method `encryptMessage()` of class `SecureMailApp` intensively calls method `findKey()` in class `Contact`. This example illustrates a well-known dilemma of agile program development in an object-oriented world: *Class-Responsibility Assignment* decisions may become unbalanced over time, due to unforeseen changes crosscutting the initial program design [31]. As a result, a majority of object-oriented design flaws like *The Blob* anti-pattern is mainly caused by low cohesion/high coupling ratios within/among classes and their members [5,6].

**Refactoring of Object-Oriented Programs.** Object-oriented *refactorings* constitute an emerging and widely used counter-measure against design flaws [13]. Refactorings impose systematic, semantic-preserving program transformations for continuously improving code-quality measures of evolving source code. For instance, the *MoveMethod* refactoring is frequently used to update CRA decisions after program changes, by moving method implementations between classes [34]. Applied to our example, a developer may (manually) conduct two refactorings, **R1** and **R2**, to counteract the aforementioned design flaws:

- (**R1**) move method `plainToHtml()` from class `MailApp` to class `Message`, and
- (**R2**) move method `encryptMessage()` from class `SecureMailApp` to class `Contact`.

However, concerning programs of realistic size and complexity, tool support for (semi-)automated program refactorings becomes more and more inevitable. The major challenges in finding effective sequences of object-oriented refactoring operations consists in *detecting* flawed program parts to be refactored, as well as in *recommending* program transformations applied to those parts to obtain an improved, yet behaviorally equivalent program design. The complicated nature of the underlying optimization problem stems from several phenomena.

- **Very large search-space** due to the combinatorial explosion resulting from the many possible sequences of (potentially interdependent) refactoring-operation applications.
- **Multiple objectives** including various (inherently contradicting) refactoring goals (e.g., **O1–O3**).
- **Many invalid solutions** due to (generally very complicated) constraints to be imposed for ensuring behavior preservation.

Further research especially on the last phenomenon is required to understand to what extent a refactoring actually alters (in a potentially critical way) the

original program. For instance, for refactoring **R2** to yield a correct result, it requires to relax declared *accessibility constraints*: method `encryptMessage()` has to become `public` instead of `protected` after being moved into class `Contact` to remain accessible for method `sendMessage`, and, conversely, method `getPrivateKey()` has to become `public` instead of `private` to remain accessible for `encryptMessage()`. Although these small changes do not affect the functionality of the original program, it may have a negative impact on extra-functional properties like program security. Therefore, the amount of invalid solutions highly depends on the interaction between constraints and repair mechanisms.

**Attack Surface of Object-Oriented Programs.** The *attack surface* of a program comprises all conventional ways of entering a software from outside such that a larger surface increases the danger of exploiting vulnerabilities (either unintentionally by some user, or intentionally by an attacker) [20]. Concerning JAVA-like programs in particular, explicit restrictions of accessibility of class members provide an essential mechanism to control the attack surface. Hence, refactoring **R2** should be definitely blamed as harmful as the enforced relaxations of accessibility constraints, especially those of the indeed security-critical method `getPrivateKey()`, unnecessarily widen the attack surface of the original program. In contrast, refactoring **R1** should be appreciated as it even narrows the attack surface by setting method `plainToHtml()` from `public` to `private`.

**Challenges.** As illustrated by our example, the attack surface of a program is a crucial, but yet unexplored, factor when searching for reasonable object-oriented program refactorings. However, if not treated with special care, accessibility constraints may seriously obstruct program maintenance by eagerly suppressing any refactoring opportunity in advance. We therefore pursue a model-based methodology for automating the search for optimal sequences of program refactorings by explicitly taking accessibility constraints into account. We formulate the underlying problem as constrained multi-objective optimization problem (MOOP) incorporating explicit control and minimization of attack-surface metrics. This framework allows us to facilitate search-based model transformation capabilities for approximating optimal solutions.

### 3 Search-Based Program Refactorings with Attack-Surface Control

We now describe our model-based framework for identifying (presumably) optimal sequences of object-oriented refactoring operations. To explicitly control (and minimize) the impact of recommended refactorings on the attack surface, we extend an existing EMF meta-model for representing JAVA-like programs with accessibility information and respective constraints. Based on this model, refactoring operations are defined as model-transformation rules which allow us to apply search-based model-transformation techniques to effectively explore candidate solutions of the resulting MOOP.

### 3.1 Program Model

In the context of model-based program transformation, a *program model* serves as unified program representation (1) constituting an appropriate level of abstraction comprising only (syntactic) program entities being relevant for a given task, and (2) including additional (static semantic) information required for a given task [24]. Concerning program models for model-based object-oriented program refactorings in particular, the corresponding model-transformation operations are mostly applied at the level of classes and members, whereas more fine-grained source code details can be neglected. Instead, program elements are augmented with additional (static semantic) dependencies to other entities being crucial for refactoring operations to yield correct results [24–26]. Here, we employ and enhance the program model proposed by Peldszus et al. [33] for automatically detecting structural anti-patterns (cf. **O1b**) in JAVA programs. Their incremental detection process also includes evaluation of coupling and cohesion metrics (cf. **O1a**), and both metric values and the detected anti-patterns are added as additional information into the program model.

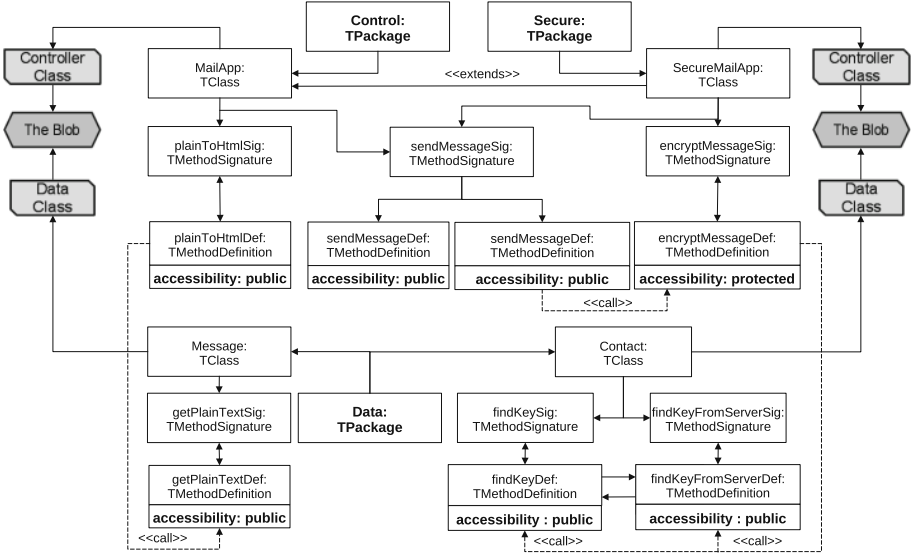
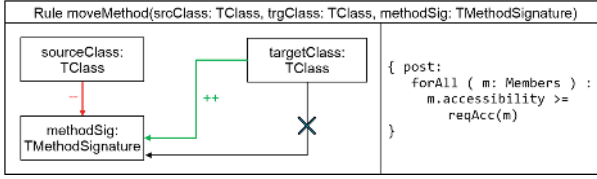


Fig. 2. Excerpt of the program-model representation of MailApp

Figure 2 shows an excerpt of the program-model representation for MailApp including the classes MailApp, Message, SecureMailApp, and Contact together with a selection of their method definitions. Each program element is represented by a white rectangle labeled with **name : type**. The available types of program entities and possible (syntactic and semantic) dependencies (represented by arrows) between respective program elements are defined by a *program meta-model*, serving as a template for valid program models [26, 37]. The program model comprises as first-class entities the classes (type TClass)



**Fig. 3.** Model-transformation rule for *MoveMethod* refactoring

together with their members as declared in the program. The representation of methods is split into signatures (type `TMethodSignature`) and definitions (type `TMethodDefinition`) to capture overloading/overriding dependencies among method declarations (e.g., overriding of method `sendMessage()` imposes one shared method signature, but two different method definitions). Solid arrows correspond to syntactic dependencies between program elements such as aggregation (unlabeled) and inheritance (label `extends`) and relations between method signatures and their definitions, whereas dashed arrows represent (static) semantic dependencies (e.g., arrows labeled with `call` denote caller-callee relations between methods).

**Design-Flaw Information.** The program model further incorporates information gained from *design-flaw detection* [33], to identify program parts to be refactored. In our example, design-flaw annotations (in gray) are attached to affected program elements, namely classes `Message` and `Contact` constitute *data classes* and classes `MailApp` and `SecureMailApp` constitute *controller classes*, which lead to two instances of the anti-pattern *The Blob*.

**Accessibility Information.** To reason about the impact of refactorings on the attack surface of programs, we extend the program model of Peldszus et al. by accessibility information. Our extensions include the attribute `accessibility` denoting the *declared accessibility* of entities as shown for method definitions in Fig. 2. In addition, our model comprises package declarations of classes (type `TPackage`) to reason about package-dependent accessibility constraints.

### 3.2 Model-Based Program Refactorings

Based on the program-model representation, refactoring operations by means of semantic-preserving program transformations can be concisely formalized in a declarative manner in terms of *model-transformation rules* [26]. A *model-transformation rule* specifies a generic change pattern consisting of a *left-hand side* pattern to be matched in an input model for applying the rule, and a *right-hand side* replacing the occurrence of the left-hand side to yield an output model. Here, we focus on (sequences of) *MoveMethod* refactorings as it has been shown in recent research that *MoveMethod* refactorings are considerably effective in improving CRA measures in flawed object-oriented program designs [34]. Figure 3 shows a (simplified) rule for *MoveMethod* refactorings defined on our program meta-model, using a compact visual notation superimposing the left- and right-hand

side. The rule takes a source class `srcClass`, a target class `trgClass` and a method signature `methodSig` as parameters, *deletes* the containment arrow between source class and signature (red arrow annotated with `--`) and *creates* a new containment arrow from the target class (green arrow annotated with `++`), only if such an arrow not already exists before rule application. The latter (*pre-condition*) is expressed by a *forbidden* (crossed-out) arrow. For a comprehensive list of all necessary pre-conditions (or, *pre-constraints*), we refer to [38].

**Accessibility Post-constraints.** Besides pre-constraints, for refactoring operations to yield correct results, it must satisfy further *post-constraints* to be evaluated after rule application, especially concerning accessibility constraints as declared in the original program (i.e., member accesses like method calls in the original program must be preserved after refactoring [24]). As an example, a (simplified) post-constraint for the *MoveMethod* rule is shown on the right of Fig. 3 using OCL-like notation. `Members` refers to the collection of all class members in the program. The post-constraint utilizes helper-function `reqAcc(m)` to compute the *required access modifier* of class member `m` and checks whether the declared accessibility of `m` is at least as generous as required (based on the canonical ordering `private < default < protected < public`) [38].

For instance, if refactoring **R2** is applied to MAILAPP, method `encryptMessage()` violates this post-constraint, as the call from `sendMessage()` from another package requires accessibility `public`, whereas the declared accessibility is `protected`. Instead of immediately rejecting refactorings like **R2**, we introduce an *accessibility-repair operation* of the form `m.accessibility := reqAcc(m)` for each member violating the post-constraint which therefore causes a *relaxation* of the attack surface. However, this repair is not always possible as relaxations may lead to incorrect refactorings altering the original program semantics (e.g., due to method overriding/overloading [38]). In contrast, refactoring **R1** (i.e., moving `plainToHtml()` to class `Message`) satisfies the post-constraint as the required accessibility of `plainToHtml()` becomes `private`, whereas its declared accessibility is `public`. In those cases, we may also apply the operation `m.accessibility := reqAcc(m)`, now leading to a *reduction* of the attack surface. Different strategies for attack-surface reduction will be investigated in Sect. 4.

### 3.3 Optimization Objectives

We now describe the evaluation of objectives **(O1)**–**(O3)** on the program model, to serve as fitness values in a search-based setting.

**Coupling/Cohesion.** Concerning **(O1a)**, coupling and cohesion metrics are well-established quality measures for CRA decisions in object-oriented program design [4]. In our program model, *coupling* (**COU**) is related to the overall number of member accesses (e.g., *call-arrows*) across class boundaries [5], and for measuring *cohesion*, we adopt the well-known **LCOM5** metric to quantify *lack of cohesion* among members within classes [17]. While there are other metrics which indicate good CRA decisions, such as **Number of Children**, these metrics are not modifiable using *MoveMethod* refactorings and are therefore not used in

this paper [9]. Consequently, good CRA decisions exhibit low values for both **COU** and **LCOM5**. Hence, refactorings **R1** and **R2** both improve values of **COU** (i.e., by eliminating inter-class *call*-arrows) and **LCOM5** (i.e., by moving methods into classes where they are called).

**Anti-patterns.** Concerning **(O1b)**, we limit our considerations to occurrences of *The Blob* anti-pattern for convenience. We employ the detection-approach of Peldszus et al. [33] and consider as objective to minimize the number of *The Blob* instances (denoted **#BLOB**). For instance, for the original MailApp program (white parts in Fig. 1), we have **#BLOB** = 1, while for the extended version (white and gray parts), we have **#BLOB** = 2. Refactoring **R1** may help to remove the first occurrence and **R2** potentially removes the second one.

**Changes.** Concerning **(O2)**, real-life studies show that refactoring recommendations to be accepted by users must avoid a too large deviation from the original design [8]. Here, we consider the *number* of *MoveMethod* refactorings (denoted **#REF**) to be performed in a recommendation, as a further objective to be minimized. For example, solely applying **R1** results in **#REF** = 1, whereas a sequence of **R1** followed by **R2** most likely imposes more design changes (i.e., **#REF** = 2). In contrast, accessibility-repair operations do not affect the value **#REF**, but rather impact objective **(O3)**.

**Attack Surface.** Concerning **(O3)**, the guidelines for secure object-oriented programming encourages developers to grant as least access privileges as possible to any accessible program element to minimize the attack surface [19]. In our program model, the attack-surface metric (denoted **AS**) is measured as

$$\mathbf{AS} = \sum_{m \in \text{Members}} \omega(m.\text{accessibility}), \quad (1)$$

where weighting function  $\omega : \text{Mod} \rightarrow \mathbb{N}_0$  on the set *Mod* of accessibility modifiers may be, for instance, defined as  $\omega(\text{private}) = 0$ ,  $\omega(\text{default}) = 1$ ,  $\omega(\text{protected}) = 2$ ,  $\omega(\text{public}) = 3$ . Hence, a lower value corresponds to a smaller attack surface. For example, **R1** enables an attack-surface reduction by setting `plainToHtml()` from `public` to `private` which decreases **AS** by 3. In contrast, **R2** involves a repair step setting `encryptMessage()` from `protected` to `public` which increases **AS** by 1. Whether such negative impacts of refactorings on **(O3)** are outweighed by simultaneous improvements gained for other objectives depends, among others, on the actual weighting  $\omega$  applied. For instance, each further modifier `public` considerably opens the attack surface and should therefore be blamed by a higher weighting value, as compared to the other modifiers (cf. Sect. 4).

### 3.4 Search-Based Optimization Process

Our tool for recommending optimized object-oriented refactoring sequences, called GOBLIN<sup>2</sup>, is based on a combination of search-based multi-objective

<sup>2</sup> Goblin is supervillain and Head of National *Security* in the Marvel universe [3]. GOBLIN also means Generic Objective-Based Layout Improvements for Non-designs.

optimization techniques using genetic algorithms and model-transformations on the basis of the MOMoT framework [11]. Figure 4 shows an overview on GOBLIN. First, the input JAVA program is translated into our program model [33]. This *original program model* together with its objective values for (O1)–(O3) (i.e., its *fitness values*) serves as a baseline for evaluating the improvements obtained by candidate refactorings. The built-in genetic algorithm (NSGA-III) of MOMoT is initialized by an *initial population* of a fixed number of *individuals* serving as *generation 0*, where each individual constitutes a *sequence* of at least 1 up to a maximum number of *MoveMethod* rule applications (cf. Fig. 3) to the original program model. Thus, each individual corresponds to a refactored version of the original program model on which the resulting fitness values are evaluated. The refactored program model is obtained by applying the given sequence of refactorings to the original program model. Steps within a sequence not being applicable to an intermediate model (e.g., due to unsatisfied pre-conditions) are skipped, whereas steps producing infeasible results (e.g., due to unsatisfied and non-repairable post-conditions) cause the entire individual to become invalid (thus being removed from the population).

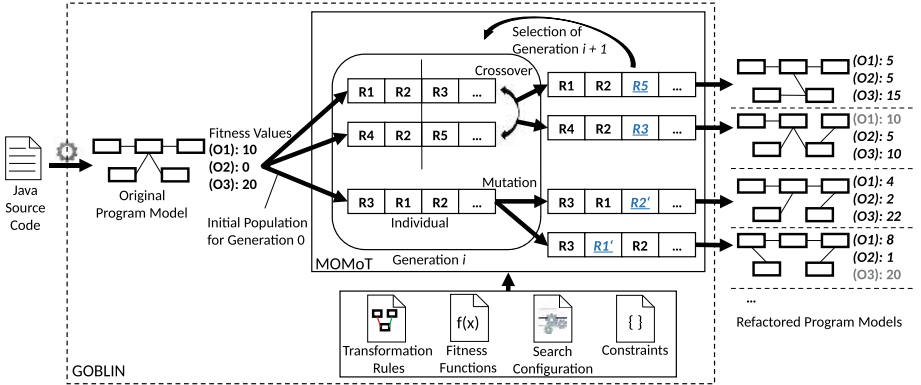


Fig. 4. Architecture of the GOBLIN tool

For deriving generation  $i + 1$  from generation  $i$ , NSGA-III first creates a set of new individuals using random *crossover* and *mutation* operators. As indicated in Fig. 4, a crossover splits and recombines two individuals into a new one, while a mutation generates a new individual by injecting small changes into an existing one. Afterwards, in the *selection* phase, individuals from the overall population (the original and newly created individuals) are selected into the next generation, depending on their *fitness* values. For more details on NSGA-III, we refer to [15, 28]. The search-process terminates when a maximum number of generations (or, individuals, respectively) has been reached, resulting in a Pareto-front of non-dominated individuals, each constituting a refactoring recommendation [11].

## 4 Experimental Evaluation

We now present experimental evaluation results gained from applying GOBLIN to a collection of JAVA programs. First, to investigate the impact of *attack-surface reduction* on the resulting refactoring recommendations, we consider the following *reduction strategies*, differing in when to perform attack-surface reduction during search-space exploration (where step means a refactoring step):

- **Strategy 1: *A priori* reduction.** Before the first and after the last step.
- **Strategy 2: *A posteriori* reduction.** Only after the last step.
- **Strategy 3: *Continuous* reduction.** After every refactoring step.

We are interested in the impact of each strategy on the trade-off between *attack-surface* metrics and design-quality metrics (i.e., do the recommended refactoring sequences tend to optimize more the attack surface aspect or the program design?). We quantify *attack-surface impact* (**ASI**) and *design impact* (**DI**) of a refactoring recommendation *rr* as follows:

$$\text{ASI}(\text{rr}) = \frac{\text{AS}(\text{rr}) - \text{AS}(\text{orig})}{\text{AS}(\text{orig})} \quad (2)$$

$$\text{DI}(\text{rr}) = \frac{\text{COU}(\text{rr}) - \text{COU}(\text{orig})}{\text{COU}(\text{orig})} + \frac{\text{LCOM5}(\text{rr}) - \text{LCOM5}(\text{orig})}{\text{LCOM5}(\text{orig})} \quad (3)$$

where *orig* refers to the original program. Second, we consider the impact of different weightings  $\omega$  on attack-surface metric **AS**. As modifier **public** has a considerably negative influence on the attack surface, we study the impact of increasing the penalty for **public** in  $\omega$ , as compared to the other modifiers. We are interested especially in whether there exists a threshold for which any design-improving refactoring would be rejected as security-critical. Finally, we compare GOBLIN to the recent refactoring tools JDEODORANT and CODE-IMP, which both do not explicitly consider attack-surface metrics as optimization objective so far. To summarize, we aim to answer the following research questions:

- **(RQ1: Objective Trade-Off)** Which attack-surface reduction strategy offers the best trade-off between attack-surface impact and design impact when taking the original program as a baseline?
- **(RQ2: Weighting of Attack Surface)** Which weighting of **public** in the attack-surface metric constitutes a critical threshold obstructing any design-improving refactorings?
- **(RQ3: Tool Comparison)** Which tool provides the best trade-off between attack-surface impact and design impact in refactoring recommendations?

### 4.1 Experiment Setup and Results

We conducted our experiments on an established corpus of real-life open-source JAVA programs of various size [33, 39] as listed in Table 1 (with lines of code

LOC, number of packages  $\#P$ , number of classes  $\#C$  and number of methods  $\#M$ ). For a compact presentation, we divide the corpus into three program-size categories (*small*, *mid-sized*, *large*), indicated by horizontal lines in Table 1. All experiments have been executed on a Windows-Server-2016 machine with a 2.4 GHz quad-core CPU, 32 GB RAM and JRE 1.8. We used the default genetic-algorithm configuration of MOMoT in all our experiments [11]: termination after 10,000 individual evaluations, population size of 100, and each individual consisting of at most 10 refactorings. We applied the metrics for (O1)–(O3) (cf. Sect. 3.3) to compute fitness values. GOBLIN requires 25 min to compute a set of refactoring recommendations for the smallest program, up to several hours in the case of large programs, which is acceptable for a search-based (off-line) optimization approach. We selected a representative set of computed recommendations which were manually checked for program correctness and impact.

For (RQ1), we measured **ASI** and **DI** values for two runs of GOBLIN (cf. Figs. 6a, b, c, d, e and f). Figures 6a and b (first row, side by side) show a box-plot for each Strategy (1–3) for *small* programs of our corpus ( $\#iSj$  referring to the program number  $i$  in Table 1 and Strategy  $j$ ). The box-plots show the distribution of **ASI** (Fig. 6a) and **DI** (Fig. 6b) values for each refactoring recommendation of GOBLIN. The figure-pairs 6c–6d and 6e–6f show the same data for *mid-sized* and *large* programs, respectively. For (RQ2), we used Strategy 3 from (RQ2) and varied function  $\omega$  to study different penalties for modifier **public**. Figure 5 plots the (minimal) values of **ASI** and **DI** depending on  $\omega(\text{public})$  (from 3 up to 100). Regarding (RQ3), we compare the results of GOBLIN to those of state-of-the-art refactoring recommender tools, JDEODORANT [12] and CODE-IMP [27]. Refactorings proposed by JDEODORANT have as singleton optimization objective to eliminate specific anti-patterns through heuristic refactoring strategies. In particular, JDEODORANT employs *ExtractClass* [13] to eliminate *The Blob* (also called *GodClass*), by separating parts from the controller-class into a freshly created class. Thus, each recommendation of JDEODORANT subsumes multiple *MoveMethod* refactorings (into the fresh target class). In contrast, CODE-IMP pursues a search-based approach, including a variety of

Program	Version	LOC	#P	#C	#M
1: QuickUML	2001	2,667	1	19	175
2: JSciCalc	2.1.0	5,437	3	121	563
3: JUnit	3.8.2	5,780	11	105	841
4: Gantt	1.10.2	21,331	28	256	1,925
5: Nutch	0.9	21,437	24	273	1,750
6: Lucene	1.4.3	25,472	15	276	1,750
7: log4j	1.2.17	31,429	35	394	3,240
8: JHotDraw	7.6	31,434	24	312	3,781

Table 1. Evaluation corpus

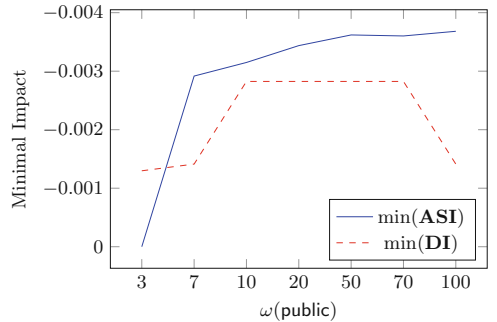
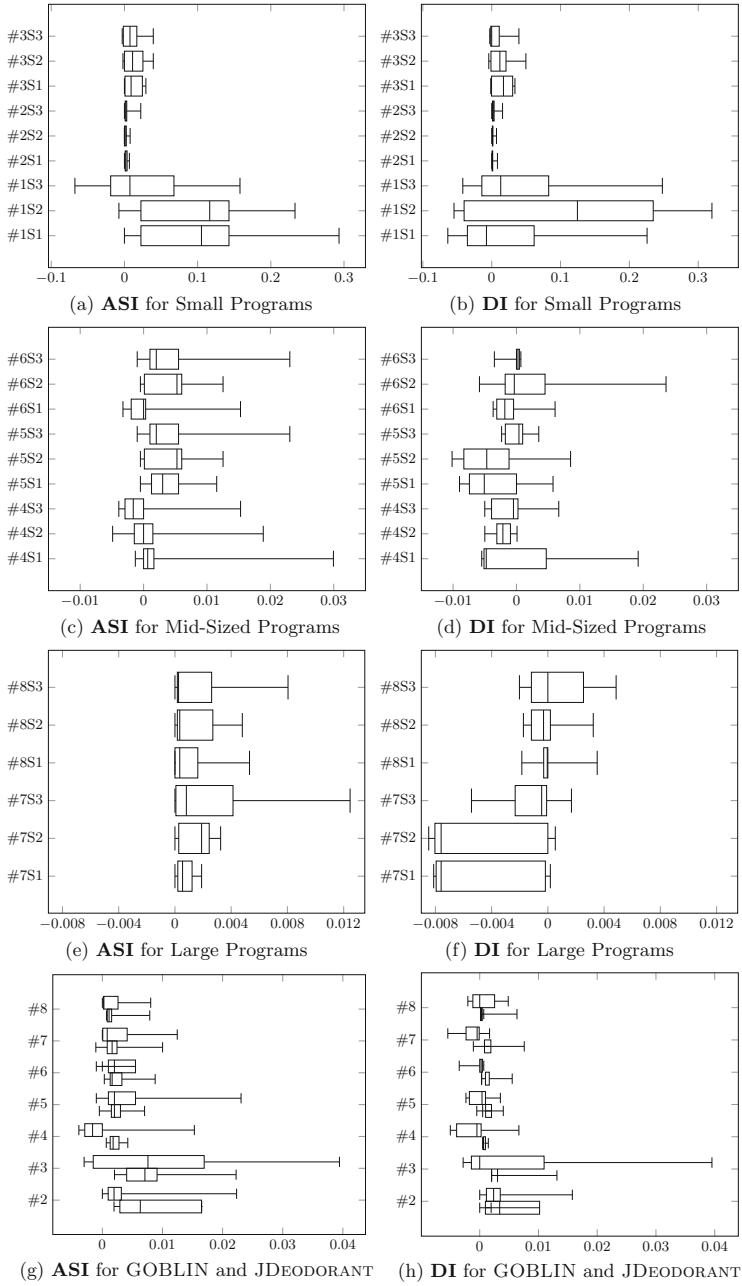


Fig. 5. Minimal **ASI** and **DI** values for different weightings of **public**

**Fig. 6.** Measurement results

refactoring operations and design-quality metrics. For a comparison to GOBLIN, we used the *MoveMethod* refactoring of CODE-IMP which produces one sequence of *MoveMethod* refactorings per run. Figures 6g and h contain comparisons of **ASI** and **DI** values, respectively, for our corpus (excluding QUICKUML due to relatively very high variations). For each program, the upper box-plot shows the results for GOBLIN and the lower one for JDEODORANT, respectively. CODE-IMP only successfully produced results for QUICKUML and JUNIT (10 runs each) while terminating without any result for the others.

## 4.2 Discussion

Concerning **(RQ1)**, Strategy 3 leads to the best attack-surface impact for *small* programs (under neglectible execution-time overhead), while even slightly improving the design impact. Although this clear advantage dissolves for *mid-sized* and *large* programs, it still contributes to a reasonable trade-off, while attack-surface reductions tend to hamper design improvements as expected. Calculating the Pearson correlation [32] between **ASI** and **DI** shows that (1) the strategy does not influence the correlation and (2) for *small* programs, GOBLIN finds refactorings which are beneficial for both attack surface and program design.

Concerning **(RQ2)**, Fig. 5 shows that a higher value for  $\omega(\text{public})$  leads to a better attack-surface impact, as attack-surface-critical refactorings are less likely to survive throughout generations. The increase in **ASI** is remarkably steep from  $\omega(\text{public}) = 3$  to  $\omega(\text{public}) = 7$ , but exhibits slow linear growth for higher values. Regarding the design impact, up to  $\omega(\text{public}) = 10$ , the best achieved **DI** also grows linearly, but afterwards, no more **DI** improvements emerge. In higher value ranges ( $>70$ ), **DI** reaches a threshold, and degrades afterwards.

Regarding **(RQ3)**, the *The Blob* elimination strategy of JDEODORANT necessarily increases attack surfaces, as calls to extracted methods have to access the new class, thus necessarily increasing accessibility at least up to **default**. As also shown in Fig. 6g, there are almost no refactorings proposed by JDEODORANT with a positive attack-surface impact. Surprisingly, JDEODORANT also achieves a less beneficial design impact than GOBLIN, with a strong correlation between **ASI** and **DI**. Our unfortunately very limited set of observations for CODE-IMP shows that, due to the similar search technique, the refactorings found by CODE-IMP and GOBLIN are quite similar. Nevertheless, due to the different focus of objectives, CODE-IMP tends to increase attack surfaces. Although, the differences in metrics definitions forbid any definite conclusions, however, CODE-IMP does not achieve any design improvements according to our metrics.

To summarize, our experimental results demonstrate that attack-surface impacts of refactorings clearly deserve more attention in the context of refactoring recommendations, revealing a practically relevant trade-off (or, even contradiction) between traditional design-improvement efforts and extra-functional (particularly, security) aspects. Our experiments further uncover that existing tools are mostly unaware of attack-surface impacts of recommended refactorings.

## 5 Related Work

**Automating Design-Flaw Detection and Refactorings.** Marinescu proposes a metric-based design-flaw detection approach similar to Peldszus et al. in [33], which is used in our work. However, both works do not deal with elimination of detected flaws [21]. In contrast, the DECOR framework also includes recommendations for eliminating anti-patterns, whereas, in contrast to our work, those recommendations remain rather atomic and local. More related to our approach, Fokaefs et al. [12] and Tsantalis et al. [40] consider (semi-)automatic refactorings to eliminate anti-patterns like *The Blob* in the tool JDEODORANT. Nevertheless, they focus on optimizing one single objective and do not consider multiple, esp. extra-functional, aspects like security metrics as in our approach.

**Multi-objective Search-Based Refactorings.** O’Keeffe and Ó Cinnéide use search-based refactorings in their tool CODE-IMP [28] including various standard refactoring operations and different quality metrics as objectives [27]. Seng et al. consider a search-based setting, where, similar to our approach, compound refactoring recommendations comprise atomic *MoveMethod* operations. Harman and Tratt also investigate a Pareto-front of refactoring recommendations including various design objectives [16], and more recently, Ouni et al. conducted a large-scale real-world study on multi-objective search-based refactoring recommendations [30]. However, neither of the approaches investigates the impact of refactorings on security-relevant metrics as in our approach.

**Security-Aware Refactorings.** Steimann and Thies were the first to propose a comprehensive set of accessibility constraints for refactorings covering full JAVA [38]. Although their constraints are formally founded, they do not consider software metrics to quantify the attack surface impact of (sequences of) refactorings. Alshammari et al. propose an extensive catalogue of software metrics for evaluating the impact of refactorings on program security of object-oriented programs [1]. Similarly, Maruyama and Omori propose a technique [22] and tool [23] for checking if a refactoring operation raises security issues. However, all these approaches are concerned with security and accessibility constraints of specific refactorings, but they do not investigate those aspects in a multi-objective program optimization setting. The problem of measuring attack surfaces serving as a metric for evaluating secure object-oriented programming policies has been investigated by Zoller and Schmolitzky [41] and Manadhata and Wing [20], respectively. Nevertheless, those and similar metrics have not yet been utilized as optimization objective for program refactoring. Finally, Ghaith and Ó Cinnéide consider a catalogue of security-relevant metrics to recommend refactorings using CODE-IMP, but they also consider security as single objective [14].

## 6 Conclusion

We presented a search-based approach to recommend sequences of refactorings for object-oriented JAVA-like programs by taking the attack surface as additional optimization objective into account. Our model-based methodology, implemented in the tool GOBLIN, utilizes the MOMoT framework including the genetic algorithm NSGA-III for search-space exploration. Our experimental results gained from applying GOBLIN to real-world Java programs provides us with detailed insights into the impact of attack-surface metrics on fitness values of refactorings and the resulting trade-off with competing design-quality objectives. As a future work, we plan to incorporate additional domain knowledge about critical code parts to further control security-aware refactorings.

**Acknowledgements.** This work was partially funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project as well as by the German Research Foundation (DFG) in the Priority Programme SPP 1593: Design For Future - Managed Software Evolution (LO 2198/2-1, JU 2734/2-1).

## References

1. Alshammari, B., Fidge, C., Corney, D.: Assessing the impact of refactoring on security-critical object-oriented designs. In: Proceedings of APSEC, pp. 186–195 (2010)
2. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: advanced concepts and tools for in-place EMF model transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010). [https://doi.org/10.1007/978-3-642-16145-2\\_9](https://doi.org/10.1007/978-3-642-16145-2_9)
3. Bendis, B.M.: Secret Invasion, vol. 1–8. Marvel, New York (2009)
4. Bowman, M., Briand, L.C., Labiche, Y.: Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. IEEE Trans. Softw. Eng. **36**(6), 817–837 (2010)
5. Briand, L.C., Daly, J.W., Wust, J.K.: A unified framework for coupling measurement in object-oriented systems. IEEE Trans. Softw. Eng. **25**(1), 91–121 (1999)
6. Briand, L.C., Daly, J.W., Wüst, J.: A unified framework for cohesion measurement in object-oriented systems. Empir. Softw. Eng. **3**(1), 65–117 (1998)
7. Brown, W.J., Malveau, R.C., McCormick III, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley, New York (1998)
8. Candela, I., Bavota, G., Russo, B., Oliveto, R.: Using cohesion and coupling for software remodularization: is it enough? ACM Trans. Softw. Eng. Methodol. **25**(3), 24:1–24:28 (2016)
9. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
10. Van Eetvelde, N., Janssens, D.: Extending graph rewriting for refactoring. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 399–415. Springer, Heidelberg (2004). [https://doi.org/10.1007/978-3-540-30203-2\\_28](https://doi.org/10.1007/978-3-540-30203-2_28)

11. Fleck, M., Troya, J., Wimmer, M.: Search-based model transformations with MOMoT. In: Van Gorp, P., Engels, G. (eds.) ICMT 2016. LNCS, vol. 9765, pp. 79–87. Springer, Cham (2016). [https://doi.org/10.1007/978-3-319-42064-6\\_6](https://doi.org/10.1007/978-3-319-42064-6_6)
12. Fokaefs, M., Tsantalis, N., Stroulia, E., Chatzigeorgiou, A.: JDeodorant: identification and application of extract class refactorings. In: Proceedings of ICSE, pp. 1037–1039 (2011)
13. Fowler, R.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (2000)
14. Ghaith, S., Ó Cinnéide, M.: Improving software security using search-based refactoring. In: Fraser, G., Teixeira de Souza, J. (eds.) SSBSE 2012. LNCS, vol. 7515, pp. 121–135. Springer, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-33119-0\\_10](https://doi.org/10.1007/978-3-642-33119-0_10)
15. Harman, M., Mansouri, S.A., Zhang, Y.: Search based software engineering: a comprehensive analysis and review of trends techniques and applications (2009)
16. Harman, M., Tratt, L.: Pareto optimal search based refactoring at the design level. In: Proceedings of GECCO, pp. 1106–1113. ACM (2007)
17. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice-Hall Inc., Upper Saddle River (1996)
18. Kessentini, M., Sahraoui, H., Boukadoum, M., Wimmer, M.: Search-based design defects detection by example. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 401–415. Springer, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19811-3\\_28](https://doi.org/10.1007/978-3-642-19811-3_28)
19. Long, F., Mohindra, D., Seacord, R.C., Sutherland, D.F., Svoboda, D.: The CERT Oracle Secure Coding Standard for Java. Addison-Wesley Professional, Boston (2011)
20. Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Trans. Softw. Eng. **37**(3), 371–386 (2011)
21. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws, pp. 350–359. IEEE (2004)
22. Maruyama, K., Omori, T.: Security-aware refactoring alerting its impact on code vulnerabilities. In: APSEC, pp. 445–451. IEEE (2008)
23. Maruyama, K., Omori, T.: A security-aware refactoring tool for Java programs. In: Proceedings of WRT, pp. 22–28. ACM (2011)
24. Mens, T., Demeyer, S., Janssens, D.: Formalising behaviour preserving program transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002). [https://doi.org/10.1007/3-540-45832-8\\_22](https://doi.org/10.1007/3-540-45832-8_22)
25. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. SOSYM **6**(3), 269–285 (2007)
26. Mens, T., Van Eetvelde, N., Demeyer, S., Janssens, D.: Formalizing refactorings with graph transformations. J. Softw. Evol. Process **17**(4), 247–276 (2005)
27. Moghadam, I.H., Ó Cinnéide, M.: Code-Imp: a tool for automated search-based refactoring. In: Proceedings of WRT, pp. 41–44. ACM (2011)
28. O’Keefe, M., Ó Cinnéide, M.: Search-based refactoring: an empirical study. J. Softw. Maint. Evol. Res. Pract. **20**(5), 345–364 (2008)
29. Opdyke, W.: Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois (1992)
30. Ouni, A., Kessentini, M., Sahraoui, H.A., Inoue, K., Deb, K.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. ACM Trans. Softw. Eng. Methodol. **25**(3), 23:1–23:53 (2016)

31. Parnas, D.L.: Software aging, pp. 279–287. IEEE (1994)
32. Pearson, K.: VII. Mathematical contributions to the theory of evolution.—III. regression, heredity, and panmixia. *Philos. Trans. R. Soc. Lond. Math. Phys. Eng. Sci.* **187**, 253–318 (1896)
33. Peldszus, S., Kulcsár, G., Lochau, M., Schulze, S.: Continuous detection of design flaws in evolving object-oriented programs using incremental multi-pattern matching. In: *Proceedings of ASE*, pp. 578–589 (2016)
34. Seng, O., Stammel, J., Burkhart, D.: Search-based determination of refactorings for improving the class structure of object-oriented systems. In: *Proceedings of GECCO*, pp. 1909–1916 (2006)
35. Shin, Y., Williams, L.: Is complexity really the enemy of software security? In: *QoP*, pp. 47–50 (2008)
36. Smith, S.F., Thober, M.: Refactoring programs to secure information flows, pp. 75–83. ACM (2006)
37. Stahl, T., Völter, M.: *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, Chichester (2006)
38. Steimann, F., Thies, A.: From public to private to absent: refactoring JAVA programs under constrained accessibility. In: Drossopoulou, S. (ed.) *ECOOP 2009*. LNCS, vol. 5653, pp. 419–443. Springer, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-03013-0\\_19](https://doi.org/10.1007/978-3-642-03013-0_19)
39. Tempero, E., Anslow, C., Dietrich, J., Han, T., Li, J., Lumpe, M., Melton, H., Noble, J.: The Qualitas Corpus: a curated collection of Java code for empirical studies. In: *Asia Pacific Software Engineering Conference*, pp. 336–345 (2010)
40. Tsantalos, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.* **35**(3), 347–367 (2009)
41. Zoller, C., Schmolitzky, A.: Measuring inappropriate generosity with access modifiers in Java systems. In: *Proceedings of IWSM-MENSURA*, pp. 43–52 (2012)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

