Science of Computer Programming 191 (2020) 102418

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

# Mutating code annotations: An empirical evaluation on Java and C# programs ☆

Pedro Pinheiro [a], José Carlos Viana [a], Márcio Ribeiro [a,*], Leo Fernandes [b], Fabiano Ferrari [c], Rohit Gheyi [d], Baldoino Fonseca [a]

[a] Computing Institute, UFAL, Maceió-AL, Brazil
[b] Informatics Coord., IFAL, Maceió-AL, Brazil
[c] Computing Department, UFSCar, São Carlos-SP, Brazil
[d] Department of Computing and Systems, UFCG, Campina Grande-PB, Brazil

ARTICLE INFO

*Article history:*
Received 29 April 2019
Received in revised form 3 February 2020
Accepted 4 February 2020
Available online xxxx

*Keywords:*
Mutation testing
Code annotations
Mining bugs

ABSTRACT

Mutation testing injects code changes to check whether tests can detect them. Mutation testing tools use mutation operators that modify program elements such as operators, names, and entire statements. Most existing mutation operators focus on imperative and object-oriented language constructs. However, many current projects use meta-programming through code annotations. In a previous work, we have proposed nine mutation operators for code annotations focused on the Java programming language. In this article, we extend our previous work by mapping the operators to the C# language. Moreover, we enlarge the empirical evaluation. In particular, we mine Java and C# projects that make heavy use of annotations to identify annotation-related faults. We analyzed 200 faults and categorized them as "misuse," when the developer did not appear to know how to use the code annotations properly, and "wrong annotation parsing" when the developer incorrectly parsed annotation code (by using reflection, for example). Our operators mimic 95% of the 200 mined faults. In particular, three operators can mimic 82% of the faults in Java projects and 84% of the faults in C# projects. In addition, we provide an extended and improved repository hosted on GitHub with the 200 code annotation faults we analyzed. We organize the repository according to the type of errors made by the programmers while dealing with code annotations, and to the mutation operator that can mimic the faults. Last but not least, we also provide a mutation engine, based on these operators, which is publicly available and can be incorporated into existing or new mutation tools. The engine works for Java and C#. As implications for practice, our operators can help developers to improve test suites and parsers of annotated code.



☆ This work was partially funded by CNPq (421306/2018-1, 426005/2018-0, 309844/2018-5, 311442/2019-6, and 306310/2016-3), CAPES grants (175956 and 117875), and FAPESP (grant 2016/21251-0). This research was also partially funded by INES 2.0, CNPq grant 65614/2014-0.

* Corresponding author.
*E-mail addresses:* pmop@ic.ufal.br (P. Pinheiro), jcvf@ic.ufal.br (J.C. Viana), marcio@ic.ufal.br (M. Ribeiro), lfmo@cin.ufpe.br (L. Fernandes), fcferrari@ufscar.br (F. Ferrari), rohit@dsc.ufcg.edu.br (R. Gheyi), baldoino@ic.ufal.br (B. Fonseca).

https://doi.org/10.1016/j.scico.2020.102418
0167-6423/

## 1. Introduction

Mutation testing [1,19,28]—also known as *program mutation*, or simply *mutation*—is a fault-based testing criterion that has been investigated for more than four decades [42,46] and has two main goals: measuring the effectiveness of test suites [19,37], and evaluating the software itself (when faults are revealed during the test process) [37]. The criterion requires the creation of copies of the original program where each one contains a small program modification. The resulting programs are called *mutants*. In this context, the test suite needs to distinguish the behavior of the mutant from the behavior of the original program for at least one test case. If that happens, the mutant is said to be *killed*; otherwise, the mutant remains *alive* and must be analyzed. The analysis of a mutant may point out to a need for a new test case to kill the mutant, or may lead to the conclusion that the mutant is equivalent to the original program and, as such, must be discarded. The quality of a test suite regarding a set $M$ of mutants is given by the mutation score ($MS$), which represents a coverage measure for the mutation testing criterion. $MS$ is the ratio of killed mutants ($K$) with respect to $M$, except the set $E$ of equivalent mutants; that is, $MS = K/(M\text{-}E)$. $MS$ is a value in the interval $[0, 1]$; the closer to 1, the better the test set with respect to $M$.

Empirical studies have demonstrated that mutation testing is an effective technique in revealing faults [10,33]. The effectiveness of mutation testing largely depends on the transformations (*i.e.,* the applied to the original program to create the mutants) made by the mutation testing tool. To perform these transformations, the tools rely on specific rules known as *mutation operators*.

The majority of mutation operators have focused on imperative language constructs [30,40,42] and on object-oriented constructs [18,23,34]. However, many projects now make use of meta-programming through code annotations. Indeed, code annotations have become very popular and induced several important frameworks such as Hibernate [8], Spring [31], and JUnit [50] to redesign their interfaces. In this context, developers are facing bugs related to annotations, as demonstrated along this article. Such bugs can be categorized as *misuse* and *wrong annotation parsing* [44]. The former means that the developer does not know how to use the code annotation in a proper way; the latter means that the developer wrongly parses the annotated code, using reflection, for instance.

The lack of mutation operators for code annotations motivated us to propose and design these operators in a recent paper [44]. In that paper we proposed a set of mutation operators for Java programs to mimic code annotation-related faults. The set includes nine mutation operators, such as Add Annotation (ADA), Remove Annotation (RMA), and Remove Attribute (RMAT). Our operators can support developers in improving their test suites and avoiding code annotation-related faults. The initial evaluation of our mutation operators showed that they are able to mimic 95 (out of 100) code annotation-related faults mined from open source Java projects.

This article extends our prior work in several ways. Firstly, it maps the mutation operators to the C# language (Section 3). The contributions of such mapping are two-fold: (i) it improves the generality of our mutation operators by considering two languages used in projects that make heavy use of code annotations; and (ii) it demonstrates that the definitions of the operators are consistent with our prior research [44]. Secondly, this article reports on enlarged empirical evidence regarding the ability of the operators to mimic code annotation-related faults for the two targeted languages (Java and C#) (Section 4). The enlarged evaluation considers 200 code annotation-related faults. We also provide an extended and improved (when compared to a preliminary version reported in our prior work) repository[1] hosted on GitHub with the 200 code annotation faults we analyzed (Section 4.2.1). We organize the repository according to the type of errors made by the programmers while dealing with code annotations, and the mutation operator that can mimic the faults. The results of our enlarged evaluation shows that the operators can mimic 190 (out of 200) faults present in our repository, *i.e.*, 95% (Section 4.2.2). Last but not least, this article introduces a mutation engine[2] that automates the application of the operators to Java and C# programs (Section 5).

When presenting the operators in Section 3, we also discuss some mutations that may result in useless (that is, unproductive) mutants [25,43] (*e.g.*, redundant and equivalent mutants). These mutants may demand high effort to be identified [36,48]. We present and discuss simple strategies to avoid the generation of such mutants. We argue that implementing these strategies in the mutation engine is important to reduce costs. In fact, our engine provides configuration files so that developers can configure it to avoid the generation of these mutants as described in Section 5.

Before presenting the contributions, the next section describes a motivating example.

## 2. Background

In this section we present background on code annotations. Then, we present a motivating example to demonstrate the need for mutation operators for code annotations.

[1] The repository is available at <https://github.com/easy-software-ufal/annotations_repos/issues>.
[2] The mutation engine is publicly available at <https://github.com/easy-software-ufal/mutation-tool-for-annotations>.

### 2.1. Code annotations

Code annotations provide a powerful method to associate meta-data or declarative information with code. Annotations allow programmers to minimize the number of lines of code to express and implement a solution, which might reduce costs during the software development. In Java, frameworks widely used in practice such as Hibernate [8], Spring [31], and JUnit [50] quickly adopted annotations and had their interfaces redesigned by the developers. Similarly, C# frameworks such as NUnit [39] have been following the same trend. This way, users of these frameworks need to deal and have knowledge about several annotations.

Developers can use annotations in classes, fields, methods, and so forth. Code annotations are useful to:

- provide information to the compiler to detect errors (*e.g.*, `@Override`) and suppress warnings (*e.g.*, `@SuppressWarnings`);
- guide code generation and other artifacts (*e.g.*, `@Getter`, `@Setter`, `@AllArgsConstructor` from the *Lombok*[3] project);
- trigger runtime processing (*e.g.*, `@Test` from JUnit);
- apply characteristics to a declaration (*e.g.*, `[Serializable]` in a C# class);
- define program contracts (*e.g.*, `[ContractClassFor]` in Spec#);
- improve code inspection; etc.

When creating a new annotation, developers need to define the *target* to restrict to which elements the annotation can be applied. For example, targets for the `@NotNull` annotation include attributes and method parameters. The target information is important to design and implement mutation operators, as we shall see next.

### 2.2. Motivating example

When using mutation testing tools, we rely on mutation operators to create mutants. For example, widely used mutation testing tools such as MuJava [35], Major [32], and Pit [13] have operators to change boolean expressions (*e.g.*, `if (x == 0) {...}` is transformed to `if (x != 0) {...}`) and to replace literal values (*e.g.*, `x = x + 1` is transformed to `x = x + 0`).

In this sense, previous research claims that most mutation operators have been developed for procedural programs [11, 34]. As such, researchers extended the operators to also take into account object-oriented features such as inheritance and polymorphism. In a previous work, we followed this tendency of evolving mutation operators [44]. In particular, we presented operators for code annotations focused on the Java language.

To better explain the need for these operators, we now present a motivating example. Fig. 1 illustrates a domain class named `User` (at the top of the figure). This class contains annotations related to the Java Persistence API (JPA) and the `javax.validation` package to perform validations. Notice that the `name` field must be not null (see the `@NotNull` annotation). In addition, at the left-hand side, we have a test suite that contains a test to check the persistence of an `User` object by calling the `save` method. This original test suite passes **(1)**. If the developer accidentally removes the `@NotNull` annotation (see the bottom of the figure), the test still passes **(2)**. Thus, the test suite is not testing a situation where the `name` field is null. Thus, we need to improve our test suite to consider a new test case to deal with this situation. At the right-hand side, a new test case—`saveNullUser` method—tries to persist an `User` object with a null `name`. Then, the expected validation exception is thrown (`ConstraintViolationException`) and the test fails **(3)**.

It is important to highlight that this whole scenario matches the mutation testing process. Here, the mutant is a slightly modified version of the `User` class (particularly, without the `@NotNull` annotation) and the original version of our test suite passes; that is, the mutant survives. To kill this mutant, we need to improve the test suite.

Notice that we introduced the example presented in Fig. 1 just to explore a scenario of a missing annotation. However, faults like the one illustrated in Fig. 1 might be more frequent in case developers overuse annotations. Fig. 2 illustrates an example of such overuse. We extracted this code snippet from the social-network-spring project.[4] In the code snippet, there are several annotations that deal with persistence (*e.g.*, `@Entity` and `@Table`) and class-specific methods (*e.g.*, `@ToString`). In addition, there are annotations to generate code (*e.g.*, `@Getter` and `@NoArgsConstructor`). This way, the overuse of annotations might lead to code pollution, a problem referred as "annotation hell" [47]. Consequently, developers might find the code difficult to read and understand.

To mimic faults related to annotations and induce improvements in the test suites, in the next section we present a set of mutation operators for code annotations [44]. Our operators are able to mimic faults like the one we present in this section, *i.e.*, the removal of an annotation.

[3] https://projectlombok.org/ – accessed in February/2020.

[4] https://github.com/ASaunin/social-network-spring/blob/master/api/src/main/java/org.asaunin.socialnetwork/domain/Message.java – accessed in February/2020.

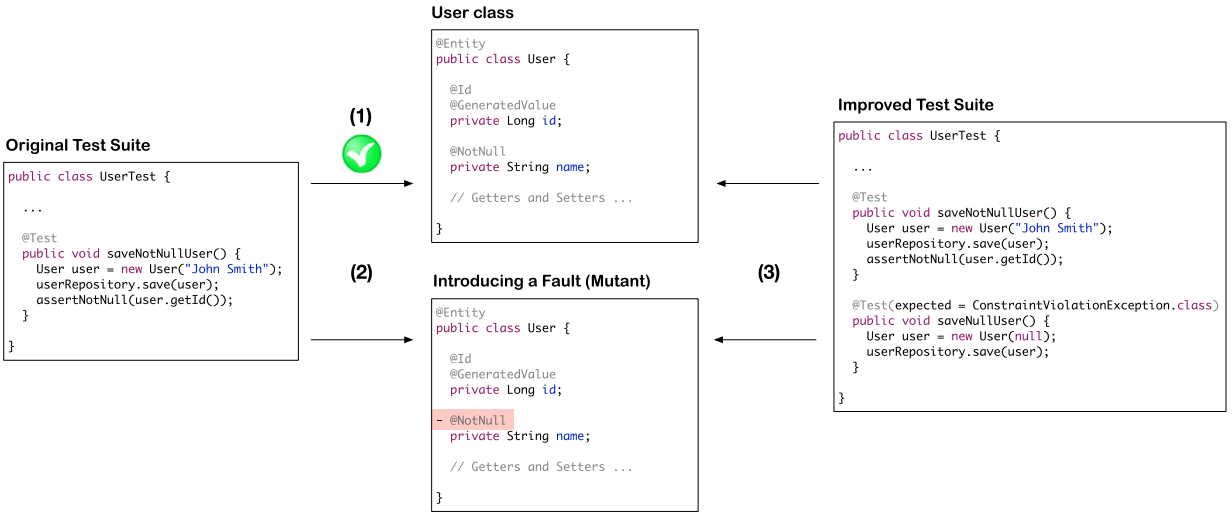


**Fig. 1.** Motivating example: the original test suite still passes against a mutant without the `@NotNull` annotation. This way, we need improve the test suite to kill the mutant, *i.e.*, we create a new test method: `saveNullUser`.

```
@Entity
@Table(name = "messages")
@Builder
@NoArgsConstructor
@AllArgsConstructor
@EqualsAndHashCode(of = {"sender", "recipient", "posted"})
@ToString(of = {"id", "body"})
public class Message implements Serializable {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Getter
  private Long id;

  @ManyToMany(fetch = FetchType.EAGER)
  @JoinColumn(name = "sender_id")
  @Getter
  @Setter
  private Person sender;

  ...
}
```

**Fig. 2.** Overuse of annotations.

## 3. Mutation operators for code annotations

In our previous paper [44], we introduced a set of mutation operators for code annotations and provided examples of their application to Java constructs. In comparison with our previous paper, here we map our operators to C# programs as well. For this, we improve the description of our operators and show how they can be applied to C# programs. Initially we present the operators and examples of how they work (Section 3.1). In this paper we also contribute with two examples—using two real-practice classes and their corresponding tests—thus demonstrating that our mutation operators can help developers to improve their test suites (Section 3.2).

### *3.1. Mutation operators*

A mutation operator is a rule for syntax transformations in a program [30]. Classical mutation operators modify the program by inserting, removing, and replacing operators (arithmetic, logical or relational), statements, variables, expressions, etc. Previously, we have proposed nine operators based on potential mistakes that a developer can make when dealing with code annotations [44]. The operators were designed by relying on (i) the authors' expertise in using annotations, as well as on (ii) possible—syntactically correct—misuse of programming constructs (namely, annotations). Notice that the latter case is a typical approach for mutation operator design, as done in past research [2,34]. To the best of our knowledge, this is the first set of mutation operators for code annotations in the literature.

We now proceed by presenting each operator. In each case, we present the acronym, name, and how the operator works. Explaining how they work is important to guide the development of a mutation engine that implements our operators, similar to the one we present in Section 5. Once again, we emphasize that the faults introduced by any mutation operator are intended to mimic typical mistakes made by programmers. The test suites should be sensitive enough to reveal those mimicked faults (*i.e.*, to detect the mutants), therefore increasing the confidence in both the program under test and the test suite. To present our operators, we follow the Java and C# notations for code annotations, *i.e.*, we use "`@`" and "`[]`", respectively. We also discuss some strategies to avoid the generation of useless mutants.

**ADA - Add Annotation** - The ADA operator adds a new code annotation to a valid target. For example, in the case where we add a code annotation to a class, we may change the way that frameworks and libraries "see" that class and the way the class works. For example, in C#, adding [Serializable][5] to a class indicates that instances of this class can be serialized. Notice that the number of application possibilities of this operator can grow fast.[6] In this way, this operator can create useless mutants and guide the construction of meaningless tests. Hence, a mutation engine should be aware of the annotations and their potential valid targets. For instance, the engine could avoid adding `@Cleanup`[7] to variables in which their corresponding objects do not have a `close()` method. Generic examples of how ADA can produce mutants for Java and C# programs are:

| Original Code (Java) | ADA Mutant |
|---|---|
| *class C {...}* | *@A class C {...}* |

| Original Code (C#) | ADA Mutant |
|---|---|
| *class C {...}* | *[A] class C {...}* |

**ADAT - Add Attribute** - The ADAT operator adds a valid attribute to the code annotation. To apply this operator, the mutation engine needs to know the annotation beforehand, *i.e.*, the attributes that the annotation supports, which ones can change the behavior if added, and potential values for the attributes to be added. For example, the `@CrossOrigin` annotation (provided by Spring.[8]) has multiple attributes (seven in total) such as `origins`, `maxAge`, and `allowCredentials`, allowing multiple configurations and potentially leading to several useless mutants. Generic examples of how ADAT can produce mutants for Java and C# programs are:

| Original Code (Java) | ADAT Mutant |
|---|---|
| *@A(x = y)* | *@A(x = y, a = b)* |

| Original Code (C#) | ADAT Mutant |
|---|---|
| *[A(x = y)]* | *[A(x = y, a = b)]* |

**CHODR - Change Order** - The CHODR operator changes the order of two code annotations. This operator is important to deal with cases in which developers are parsing annotated code. A different order might cause a different behavior when performing the parsing. Notice, however, that this operator may generate a high number of useless mutants. Generic examples of how CHODR can produce mutants for Java and C# programs are:

| Original Code (Java) | CHODR Mutant |
|---|---|
| *@A @B* | *@B @A* |

| Original Code (C#) | CHODR Mutant |
|---|---|
| *[A] [B]* | *[B] [A]* |

[5] https://docs.microsoft.com/en-us/dotnet/api/system.serializableattribute – accessed in February/2020.

[6] To create an approximation for the complexity of the ADA operator, consider that a mutation engine can use the operator to add only the $n$ annotations that are already present in the code. Given that the source code contains $m$ targets, the number of potential mutants is $O(n * m)$.

[7] https://projectlombok.org/features/Cleanup – accessed in February/2020.

[8] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/CrossOrigin.html – accessed in February/2020.

**RMA - Remove Annotation** - The RMA operator removes a code annotation. This operator is useful, for instance, to introduce the fault presented in Fig. 1. Generic examples of how RMA can produce mutants for Java and C# programs are:

| Original Code (Java) | RMA Mutant |
|---|---|
| *@A class C {...}* | *class C {...}* |

| Original Code (C#) | RMA Mutant |
|---|---|
| *[A] class C {...}* | *class C {...}* |

**RMAT - Remove Attribute** - The RMAT operator removes an attribute from a code annotation. Therefore, we can see this operator as the reverse transformation of the ADAT operator. To avoid useless mutants when applying this operator, the mutation engine must avoid the removal of attributes that do not change the observable behavior. Generic examples of how RMAT can produce mutants for Java and C# programs are:

| Original Code (Java) | RMAT Mutant |
|---|---|
| $@A(x = y,\ a = b)$ | $@A(x = y)$ |

| Original Code (C#) | RMAT Mutant |
|---|---|
| $[A(x = y,\ a = b)]$ | $[A(x = y)]$ |

**RPA - Replace Annotation** - The RPA operator replaces one code annotation by another. Notice that this operator suffers from the same problem as the ADA operator, *i.e.*, the number of annotations to be added is potentially very high. Therefore, mutation engines should select a meaningful annotation to replace the actual one, avoiding useless mutants. For example, instead of replacing `@NotNull` by `@Entity`[9] (which is meaningless), the engine could replace by `@Nullable`, *i.e.*, the annotation with the opposite behavior. Generic examples of how RPA can produce mutants for Java and C# programs are:

| Original Code (Java) | RPA Mutant |
|---|---|
| *@A* | *@B* |

| Original Code (C#) | RPA Mutant |
|---|---|
| *[A]* | *[B]* |

**RPAT - Replace Attribute** - The RPAT operator replaces a code annotation attribute by another. Like the RPA operator, the number of transformations available for this operator is high. In addition, to perform meaningful replacements, the engine should be aware of the attributes that the annotations support. For example, the `@EqualsAndHashCode` annotation of the *Lombok* project has the `of` and `exclude` attributes. This way, the mutation engine could replace `of` by `exclude`, thus changing the fields used to generate the `equals` and `hashCode` methods. The engine, on the other hand, must avoid replacing `of` by `name`, for example, since the latter does not exist in the `@EqualsAndHashCode` annotation. Generic examples of how RPAT can produce mutants for Java and C# programs are:

| Original Code (Java) | RPAT Mutant |
|---|---|
| $@A(x = y)$ | $@A(a = b)$ |

| Original Code (C#) | RPAT Mutant |
|---|---|
| $[A(x = y)]$ | $[A(a = b)]$ |

[9] `@Entity` is an annotation of the Java Persistence API.

**RPAV - Replace Attribute Value** - The RPAV operator replaces a code annotation attribute value by another. To perform meaningful replacements and at the same time reduce costs, a mutation engine should be aware of the default and non-default values of an attribute. For example, the *Lombok* project has the `@RequiredArgsConstructor` code annotation responsible for generating constructors based on the class fields. This annotation has an optional attribute: the `access` attribute. The default value of this attribute is `AccessLevel.PUBLIC`. In this context, the engine could, for example, replace this value by non-default values, such as `AccessLevel.PROTECTED` and `AccessLevel.PRIVATE`. Generic examples of how RPAV can produce mutants for Java and C# programs are:

| Original Code (Java) | RPAV Mutant |
|---|---|
| $@A(x = y)$ | $@A(x = a)$ |

| Original Code (C#) | RPAV Mutant |
|---|---|
| $[A(x = y)]$ | $[A(x = a)]$ |

**SWTG - Switch Target** - The SWTG operator modifies the annotation target where a code annotation is inserted. For example, SWTG can move a code annotation from a field to a method. Generic examples of how SWTG can produce mutants for Java and C# programs are:

| Original Code (Java) | SWTG Mutant |
|---|---|
| $@A\ int\ v;$<br>$int\ getV()\{...\}$ | $int\ v;$<br>$@A\ int\ getV()\{...\}$ |

| Original Code (C#) | SWTG Mutant |
|---|---|
| $[A]\ int\ v;$<br>$int\ getV()\{...\}$ | $int\ v;$<br>$[A]\ int\ getV()\{...\}$ |

### 3.2. Usefulness of annotation-based mutation operators

To illustrate that our mutation operators are able to generate mutants that indeed would help developers to improve their test suites, we now present two examples based on two classes we selected from the projects we analyze in Section 4. To select the classes, we considered the following criteria:

- The classes must have associated tests;
- The classes must have several annotations in which it is possible to apply almost or all mutation operators we propose.

We searched for one class written in Java and one class written in C#. We have selected the first two classes we found that met the above mentioned criteria. The first class is from the *dropwizard* project and is written in Java. The second class is from the OchardCore project and is written in C#.

The class from the *dropwizard* project is `ServerPushFilterFactory` and its corresponding test class is `ServerPushFilterFactoryTest`. The latter contains JUnit tests to test the former, all created by the *dropwizard* developers. The `ServerPushFilterFactory` class has six fields, five getter methods, and five setter methods. In addition, there is a method named `addFilter`. Three fields and all getter and setter methods contain annotations. Fig. 3 illustrates the `ServerPushFilterFactory` class. The `ServerPushFilterFactoryTest` class has five test methods.

We executed the Pit tool [12,13], a well-known mutation testing tool for Java, to generate mutants of the `ServerPushFilterFactory` class. Pit generated 90 mutants. Then, we used our mutation operators to create mutants based on annotations. To do so, we applied each of our mutation operators to generate one mutant each. This resulted in eight mutants of the `ServerPushFilterFactory` class. We could not apply one operator (*i.e.*, RPAT) because no annotations of the class have attributes. Table 1 illustrates the mutations.

Afterwards, we executed the test methods of the `ServerPushFilterFactoryTest` class. The tests killed 78 mutants generated by the Pittool. This means that the eight mutants generated with our mutation operators for code annotations remained alive. This scenario illustrates that our operators might be useful to improve the test suite of a program, which is the most important objective of applying mutation testing. We then wrote six more test methods and our eight mutants were killed. This is important to mention because our tests demonstrate that our eight mutants are not equivalent.

As mentioned, we also performed a similar study on a class in C# (*i.e.*, `ApiController` from the OchardCore project). To select this class, we used the same criteria presented at the beginning of this section. Fig. 4 presents this class. We

```
public class ServerPushFilterFactory {

    private static final Joiner COMMA_JOINER = Joiner.on(",");
    private boolean enabled = false;

    @MinDuration(value = 1, unit = TimeUnit.MILLISECONDS)
    private Duration associatePeriod = Duration.seconds(4);

    @Min(1)
    private int maxAssociations = 16;

    @Nullable
    private List<String> refererHosts;

    @Nullable
    private List<Integer> refererPorts;

    @JsonProperty
    public boolean isEnabled() {
        return enabled;
    }

    ...

    @Nullable
    @JsonProperty
    public List<String> getRefererHosts() {
        return refererHosts;
    }

    @JsonProperty
    public void setRefererHosts(@Nullable List<String> refererHosts) {
        this.refererHosts = refererHosts;
    }

    @Nullable
    @JsonProperty
    public List<Integer> getRefererPorts() {
        return refererPorts;
    }

    @JsonProperty
    public void setRefererPorts(@Nullable List<Integer> refererPorts) {
        this.refererPorts = refererPorts;
    }

    public void addFilter(ServletContextHandler handler) { ... }

}
```

**Fig. 3.** `ServerPushFilterFactory` class from the dropwizard project.

**Table 1**
Mutations performed by eight of our mutation operators in the `ServerPushFilterFactory` class.

| Operator | Performed mutation |
|---|---|
| ADA | Add `@Nullable` to the `setRefererPorts` method |
| ADAT | Add `inclusive = false` to the `@MinDuration` annotation |
| CHODR | Change order of the annotations declared at the `getRefererPorts` method |
| RMA | Remove `@Nullable` from the `refererHosts` method |
| RMAT | Remove `TimeUnit.MILLISECONDS` from the `@MinDuration` annotation |
| RPA | Replace `@Nullable` by `@NotNull` at the `refererHosts` method |
| RPAV | Replace `1` by `5` at the attribute `value` of the `@MinDuration` annotation |
| SWTG | Moved `@JsonProperty` from the isEnabled method to the `enabled` field |

executed the VisualMutator tool [45] to generate mutants of the `ApiController` class. The tool generated 328 mutants. Afterwards, we applied our mutation operators for code annotations. Again, we applied our mutation operators to generate one mutant each. We created seven mutants. In particular, we could not apply two operators (*i.e.*, RPAT and SWTG). Table 2 summarizes the mutations we applied.

Then, we executed the test suite against the mutants. The tests killed 247 mutants, *i.e.*, 246 generated by the VisualMutator tool and one mutant based on code annotations. This particular mutant was generated using the RPAV operator. It changed the value of the `AuthenticationScheme` attribute from `[Authorize(AuthenticationSchemes = "Api")]` to `[Authorize(AuthenticationSchemes = "api")]`. This mutation implies that the server is prevented

```
[Route("api/content")]
[ApiController]
[Authorize(AuthenticationSchemes = "Api"), IgnoreAntiforgeryToken, AllowAnonymous]
public class ApiController : Controller
{
    private readonly IContentManager _contentManager;
    private readonly IAuthorizationService _authorizationService;

    public ApiController(
        IContentManager contentManager,
        IAuthorizationService authorizationService)
    {
        _authorizationService = authorizationService;
        _contentManager = contentManager;
    }

    [Route("{contentItemId}"), HttpGet]
    public async Task<IActionResult> Get(string contentItemId)
    {
        ...

        return Ok(contentItem);
    }

    [HttpDelete]
    [Route("{contentItemId}")]
    public async Task<IActionResult> Delete(string contentItemId)
    {
        ...

        return Ok(contentItem);
    }

    [HttpPost]
    public async Task<IActionResult> Post(ContentItem newContentItem, bool draft = false)
    {
        ...

        return Ok(contentItem);
    }
}
```

**Fig. 4.** `ApiController` class from the OchardCore project.

**Table 2**
Mutations performed by seven of our mutation operators in the `ApiController` class.

| Operator | Performed mutation |
|---|---|
| ADA | Add `[ProducesResponseType(StatusCodes.Status404NotFound)]` to `Post` method |
| ADAT | Add `Roles = "Admin"` to `[Authorize]` |
| CHODR | Change order of the annotations declared at `ApiController` class |
| RMA | Remove `[HttpDelete]` from `Delete` method |
| RMAT | Removes `AuthenticationSchemes = "Api"` from `[Authorize]` |
| RPA | Replace `[HttpGet]` for `[HttpPost]` at `Get` method |
| RPAV | Replace value for `AuthenticationSchemes`, at `[Authorize]`, from `"Api"` to `"api"` |

from authenticating clients, thus, sending a POST request to the *api/content* endpoint fails. An existing test case that focuses on authentication killed this mutant.

Again, to guarantee that our mutants are not equivalent, we wrote test cases to kill them.

## 4. Evaluation

In this section, we evaluate the mutation operators by investigating whether they can mimic realistic faults related to annotations. To do so, we analyze 200 code annotation-related faults, 100 in Java projects and 100 in C# projects. In what follows, we detail our evaluation. We first present the settings of our analysis (Section 4.1) and then the results and discussion (Section 4.2). Next, we present the threats to the validity of our study (Section 4.3). Last, we discuss the implications for the practical application of our operators (Section 4.4).

### 4.1. Settings

To evaluate the effectiveness of our mutation operators, we needed a comprehensive set of real faults. Thus, we searched for real-practice faults related to code annotations. In particular, we focused on Java and C# projects hosted on GitHub. To perform the search, we used the GitHub Search API [27]. The API helped us to select closed issues labeled as "bug" or "defect." In addition, at the issue title and body, we searched for the words "annotation" or "attribute," and for words that match the regular expressions `@[A-z]+` or `(\[\w+\])+`, since these expressions match the notations used by Java and C# for using code annotations, respectively.

To confirm whether the issues are indeed related to annotations, we performed a manual analysis. To do so, for each issue, we analyzed the fixing commit and the messages the developers posted regarding the issue. To make our manual analysis feasible, we stopped once we reached 200 confirmed code annotation-related faults. Two researchers individually analyzed the issues to confirm that the faults are indeed related to annotations. In case the researchers disagreed, we disregarded such a fault. For each fault, the researchers reviewed the code before and after the fixing commit. When provided, the researchers analyzed the regression tests that prevent the faults as well. The idea here was to follow the opposite path of the bug fix, *i.e.*, given the fixed code, is there any mutation operator (from Section 3) able to create a mutant that mimics the fault that has been fixed?

In this evaluation, we answer the following research questions:

- **RQ1:** How many code annotation-related faults are our mutation operators able to mimic?
- **RQ2:** Is there a set of operators that is able to mimic the great majority of the faults?
- **RQ3:** What are the operators that mimic few faults or even no faults at all?
- **RQ4:** Are there code annotation-related faults that our operators are not able to mimic?
- **RQ5:** Are the operators representative enough to mimic faults in both the Java and C# languages?

Answering **RQ1** is important to understand to what extent our operators are useful in practice. Answering **RQ2** and **RQ3** is important to guide potential mutation testing tools. For example, the tool could disregard operators that mimic few faults or no fault whatsoever, and consider operators responsible for simulating many faults instead. Answering **RQ4** is important to check whether new operators should be proposed. Finally, answering **RQ5** is important to check how generic our operators are with respect to different programming languages.

In what follows, we answer our research questions and discuss the results.

### 4.2. Results and discussion

After applying our search criteria in the GitHub issues, we ended up with 4,171 issues in 1,337 Java projects and 2,662 issues in 914 C# projects. We manually analyzed a subset of these issues in a random way to confirm that they are indeed related to code annotations. We stopped once we have reached 200 code annotation-related faults across Java and C# projects (*i.e.*, 100 for each language). To reach 200 faults, we have analyzed 5,508 out of 6,833 issues (80%). These faults are from 125 projects—66 written in Java and 59 written in C#. Tables 3 and 4 show the number of faults per project, for Java and C#, respectively.

#### 4.2.1. Repository of code annotation-related faults

To better organize the study and the results, during our manual analysis we have extended and improved our GitHub repository with the 200 annotation-related faults we analyzed.[10] We also provide the scripts we have used to mine code annotation-related fault candidates.[11] We have 200 issues in our repository, each one corresponding to one specific fault. Our issues contain the links to the original issues and to the commits that fixed them.

During our manual analysis, we classified the faults into two categories:

- ***Misuse:*** developers misunderstand how the code annotation works and they use it in a wrong way. For example, the developer removes a code annotation that could not be removed, forgets to include a code annotation attribute, or even combines two code annotations in which the first overrides the behavior of the second; and
- ***Wrong annotation parsing:*** developers commonly use techniques like Java reflection to check whether a class has a certain code annotation, for instance. Wrongly parsing the annotated code might lead to failures. Examples of wrong code annotation parsing include forgetting to check if the former has a certain attribute and forgetting to check whether the attribute has a certain value.

To indicate the category of each fault in our repository, we labeled each issue as *misuse* or *wrong annotation parsing*. In addition, we labeled each issue with the operator that can mimic it (*e.g.*, ADA, RMA, RPAV, etc). We use the label *no operator*

[10] https://github.com/easy-software-ufal/annotations repos/issues.
[11] https://github.com/easy-software-ufal/annotations_repos/tree/master/Miner.

**Table 3**
The 100 faults we analyzed, found in 66 Java projects.

| Project | Domain | Operators | Faults Frequency |
|---|---|---|---|
| abixen-platform | Microservice API. | RMA | 1 |
| achieve-lib-api | Multipurpose library. | RPAV | 1 |
| Achilles | Object mapping framework. | RPAV | 1 |
| actframework | Web server API. | CHODR, ADA | 6 |
| ametiste-metrics | Metric gathering library. | ADA | 1 |
| android-state | Utility library. | ADA | 1 |
| arquillian-cube | Container management. | ADA | 1 |
| BatooJPA | Persistence API. | SWTG, ADA | 3 |
| bazel | Build system. | RPAV | 1 |
| blaze-persistence | Criteria API. | ADA | 1 |
| cas | Authentication protocol. | RMA | 1 |
| cofoja | Contract programming framework. | ADA | 1 |
| concordion | Documentation tool. | ADA | 1 |
| crux | Web application framework. | RPAV, ADA | 2 |
| damapping | DA mapping framework. | ADA | 2 |
| docdoku-plm | Product lifecycle management. | RPAV | 1 |
| dropwizard | Web service API. | RMA, ADA | 5 |
| incubator-druid | Data store API. | RMA | 1 |
| eFapsApp-Commons | Generic software core library. | RPA | 1 |
| elasticsearch | Search engine. | RMA | 1 |
| elide | Web service API. | RMA | 1 |
| enunciate | Auxialiar code generation. | RMA, ADA | 3 |
| ff4j | Feature flags library. | ADA | 1 |
| flow | Web application framework. | ADA | 1 |
| framework | Web application framework. | ADA | 4 |
| fscrawler | File system crawler. | ADA | 1 |
| gatk | Genome analysis toolkit. | ADA | 1 |
| genie | Distributed Big Data Orchestration Service. | RPAV | 2 |
| guava | Core libraries. | ADA | 1 |
| gwtquery | Web development API. | RMA, RPAV | 2 |
| hibernate-hydrate | Duct-tape fix library. | - | 1 |
| jDTO-Binder | DTO framework. | ADAT, ADA | 2 |
| LoganSquare | Serialization library. | ADA | 2 |
| mapstruct | Annotation processor. | ADA | 4 |
| material-dialogs | Dialogs API. | RPAV | 1 |
| needle | Unit testing framework. | RMA | 1 |
| netty | Network application framework. | RMA | 1 |
| NotRetrofit | REST client. | ADA | 1 |
| org.parallelj | Parallel computing runtime. | RPAV | 1 |
| Pericopist | Message catalogs generation tool. | RPAV | 2 |
| podam | Mocking library. | ADA | 1 |
| presto | SQL query engine. | RMA | 2 |
| qbit | Microservice library. | ADA | 1 |
| riak-java-client | Database client. | ADA | 2 |
| RoboSpock | Testing framework. | CHODR | 1 |
| smarthome | Development framework. | RPA | 1 |
| sniffy | Profiler. | ADAT | 1 |
| vaadin/spring | Glue library. | ADA | 1 |
| spring-boot | Spring Boot. | RMAT, RPAV | 2 |
| spring-cloud-config | Configuration library. | - | 1 |
| spring-cloud-consul | Spring Cloud Consul. | ADA | 1 |
| spring-data-jest | Spring Data Implementation for Jest. | ADA | 1 |
| spring-security | Spring Security. | RPAV | 1 |
| springlets | Utilities library. | ADA | 1 |
| stag-java | Speedy Type Adapter Generation. | ADA | 1 |
| swagger4spring-web | Glue library. | ADAT | 3 |
| teasy | UI automation testing framework. | ADA | 1 |
| thindeck | Web Hosting. | RMA | 2 |
| transfuse | Dependency Injection framework. | ADA | 1 |
| vertx-mongo-client | Mongo Client. | RMA | 1 |
| video-recorder-java | UI testing video recording Library. | RPAV | 1 |
| vraptor | Web framework. | ADA | 1 |
| vraptor4 | Web and CDI framework. | ADA | 1 |
| wildfly-camel | Glue code library. | ADA | 1 |
| wisdom | Web framework. | ADA | 2 |
| xstream | Serialization library. | ADAT, ADA | 2 |
| **TOTAL** | | | 100 |

**Table 4**
The 100 faults we analyzed, found in 59 C# projects.

| Project | Domain | Operators | Faults Frequency |
|---|---|---|---|
| AgodaAnalyzers | Software development utility. | ADA | 1 |
| aspnet-api-versioning | Web development library. | ADA | 2 |
| Autofac | Software development utilities. | RMA, ADA | 4 |
| azure-iot-protocol-gateway | Framework. | RMA | 1 |
| azure-webjobs-sdk | Framework. | RMAT, ADA | 4 |
| BenchmarkDotNet | Software development utility. | RPAV, ADAT, ADA | 6 |
| BetterCMS | CMS. | RMA | 1 |
| BinanceDotNet | REST API client. | RMA | 1 |
| BootstrapTagHelpers | Web development library. | ADA | 1 |
| Cake.OctoDeploy | Software development utility. | RMA | 1 |
| Caliburn.Micro | Application development framework. | RMA | 1 |
| ClearCanvas | Software development libraries. | ADA | 1 |
| clipr | Software development library. | ADA | 1 |
| CodeOnlyStoredProcedures | Software development library. | ADA | 1 |
| CommandLineUtils | Software development library. | RPAV | 1 |
| Cosmonaut | SDK. | ADA | 1 |
| couchbase-lite-net | Database engine. | RMA | 1 |
| csharp-api-sdk | SDK. | ADA | 1 |
| duality | Game development framework. | ADA | 1 |
| EDI.Net | Software library. | RPAV | 1 |
| elasticsearch-net | Search engine. | ADAT, RMA, ADA | 5 |
| ReversePoco | Software development utility. | RMA | 1 |
| bridgedotnet/Frameworks | Software development library. | RMA | 2 |
| Hangfire.RecurringJobExtensions | Software development library. | RPAV | 1 |
| JsonApiDotNetCore | Framework. | ADA | 2 |
| JsonPatch | Software development library. | RMAT | 1 |
| language-ext | Software development library. | CHODR | 1 |
| QuantConnect/Lean | SDK. | RMA | 1 |
| MetaWear-SDK-CSharp | SDK. | RMA | 1 |
| SqlTableDependency | Software development library. | ADAT | 1 |
| aspnet/Mvc | Web development framework. | RPAV, RMA, RPA, ADA | 11 |
| NakedObjectsFramework | Software development framework. | RMA, ADA | 3 |
| Neo4jClient | Database client. | ADA | 1 |
| NServiceBus | Software library. | ADA | 2 |
| octokit.net | REST API client library | RPAV | 1 |
| opencover | Software development utility. | ADA | 1 |
| OrchardCore | CMS. | ADA | 1 |
| poshtools | Software development utility. | RMAT | 1 |
| PowerArgs | Software library. | ADA | 1 |
| PyriteServer | Software application. | RMA | 1 |
| QCVOC | Software application. | RMAT | 1 |
| RefactoringEssentials | Software development utility. | ADA | 1 |
| RestSharp | Software library. | ADA | 1 |
| Rosetta | Software development utility. | RPAV | 1 |
| roslyn | Compiler. | ADA | 5 |
| aspnet/Routing | Web development. | RPAV | 2 |
| Scrutor | Software development utility. | NaN | 1 |
| aspnet/Security | Web development. | RPAV, ADA | 2 |
| SHFB | Software development utility. | RPAV | 1 |
| SkiaSharp | Graphics API. | RPAV | 1 |
| SpecFlow | Software development utility. | RMAT | 1 |
| spmeta2 | Software development utility. | RMA | 1 |
| sqlite-net | Database client. | RMA, ADA | 4 |
| Starcounter.Authorization | Software development library. | ADA | 1 |
| TheBelt | Private software project. | RPAV | 1 |
| Microsoft/Tx | Software library. | ADA | 1 |
| VaraniumSharp | Software library. | RMA | 2 |
| VRTK.Unity.Core | Software development utility. | SWTG | 1 |
| xamarin-macios | SDK. | ADA | 1 |
| **TOTAL** | | | 100 |

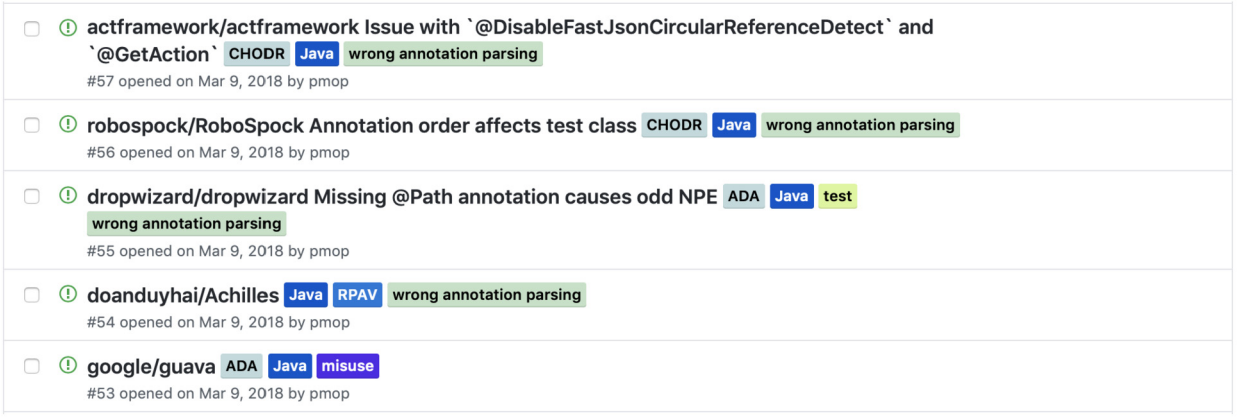


**Fig. 5.** Screenshot of part of our GitHub repository.

**Table 5**
Distribution of the faults according to the annotations parts.

| **Fault** | Frequency | |
|---|---|---|
| | **Java** | **C#** |
| Annotation | 73 | 72 |
| Attribute | 7 | 8 |
| Attribute Value | 15 | 15 |
| None | 5 | 5 |
| **TOTAL** | 100 | 100 |

if none of our operators is able to mimic the issue. In case the developer fixed the issue and at the same time introduced a test to prevent the issue from happening again, we labeled the issue with the *test* label. Fig. 5 illustrates five issues of our repository. Notice that four of them fit the *wrong annotation parsing* category and only one fits the *misuse* category (namely, in the *guava* project). To fix the issue of the *dropwizard* project, the developer added a test to prevent this issue in the future.

To the best of our knowledge, we created the first repository that stores faults related to code annotations in two programming languages. This repository may support further studies not only on the mutation testing topic but also on other research initiatives related to, for example, automatic code repair and, more generally, related to code annotations.

*4.2.2. Answering the research questions*

From our analysis of 200 code annotation faults, we found 51 in the *misuse* category and 149 in the *wrong annotation parsing* category. We found that our operators are able to mimic 190 out of the 200 faults we analyzed. Table 5 distributes the faults according to the parts of a code annotation, *i.e.*, the annotation itself, attribute, and attribute value. The majority of the faults occurs at the code annotation itself. Table 6 presents the number of faults mimicked by each operator.

Table 7 shows the mutation operators and the two categories of faults we discuss in this article. The majority of the mimicked faults are related to the *wrong annotation parsing* category, *i.e.*, 74.5%. The ADA and RPAV operators are more likely to expose faults of such category. On the other hand, RMA is the one more likely to expose faults of the *misuse* category, *i.e.*, 79.48%.

**Answer to RQ1:** Our operators are able to mimic 190 out of the 200 faults (95%) we have analyzed.

Regarding **RQ2**, we found that three mutation operators (out of nine) are able to mimic up to 83% of all faults we analyzed, namely: Add Annotation (ADA), Remove Annotation (RMA), and Replace Attribute Value (RPAV).

Fig. 6 presents an example from the *thindeck* project.[12] The original code (before the commit) does not have the `@Immutable` annotation. This example fits into the *misuse* category, *i.e.*, the developer forgot to annotate the classes. To fix this problem, the developer added such an annotation into four classes. The developer also created a test case to check immutability for these classes.

[12] https://github.com/piotrkot/thindeck/commit/95bb424a38c7116b0868a16c4bfba855e6de041a – accessed in February/2020.

**Table 6**
Number of faults mimicked by each operator.

| Operator | Description | Frequency | |
|---|---|---|---|
| | | **Java** | **C#** |
| ADA | Adds annotation to valid target | 51 | 46 |
| ADAT | Adds valid attributes to annotation | 6 | 3 |
| CHODR | Changes the order of two annotations | 3 | 1 |
| RMA | Removes an annotation | 16 | 23 |
| RMAT | Removes an attribute from annotation | 1 | 5 |
| RPA | Replaces an annotation | 2 | 1 |
| RPAT | Replaces an attribute by another | 0 | 0 |
| RPAV | Replaces an annotation value for another | 15 | 15 |
| SWTG | Switches annotation to another applicable target | 1 | 1 |
| No operator | Faults not mimicked by any operator | 5 | 5 |
| **TOTAL** | | 100 | 100 |

**Table 7**
Mutation operators mapped to faults, classified as *Misuse* and *Wrong annotation parsing* categories.

| Operator | Misuse | Wrong Parsing | Total faults |
|---|---|---|---|
| ADA | 6.20% | 93.81% | 97 |
| ADAT | 0% | 100% | 9 |
| CHODR | 0% | 100% | 4 |
| RMA | 79.48% | 20.51% | 39 |
| RMAT | 50% | 50% | 6 |
| RPA | 66.66% | 33.33% | 3 |
| RPAT | 0 | 0 | 0 |
| RPAV | 30% | 70% | 30 |
| SWTG | 0 | 100% | 2 |
| No operator | 0 | 100% | 10 |
| **TOTAL** | 25.5% | 74.5% | 200 |

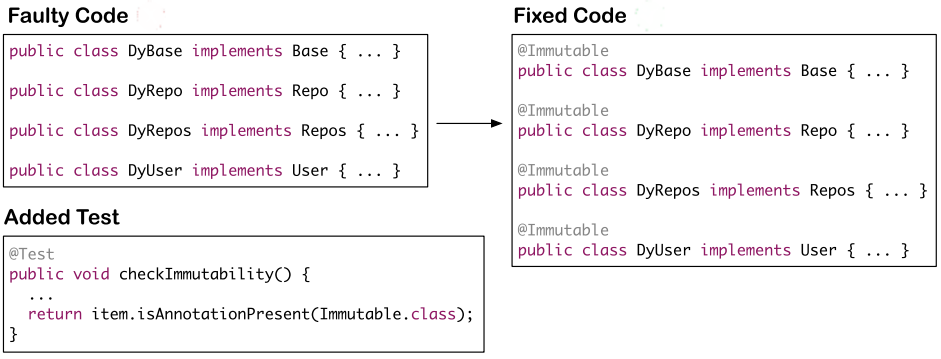


**Fig. 6.** Fault from the *thindeck* project (Java) that could be mimicked by our Remove Annotation (RMA) operator.

Notice that this scenario fits into the mutation testing approach: there is no test to check immutability; to apply our operators, we follow the mutating testing assumption: the original code must be close to the correct (cf. the Competent Programmer Hypothesis [19]); our Remove Annotation (RMA) operator injects a fault, *i.e.*, it removes the `@Immutable` annotation; the test suite still passes and the mutant without the annotation survives; and finally the developer creates a new test to check immutability and kill the mutant. This way, our operator helps to improve the test suite.

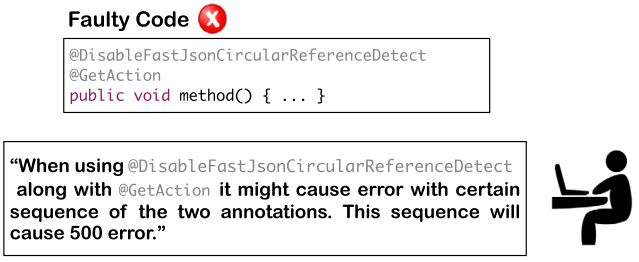


**Fig. 7.** *Wrong annotation parsing* example from the *actframework* project (Java). Our Change Order (CHODR) operator can mimic this fault.

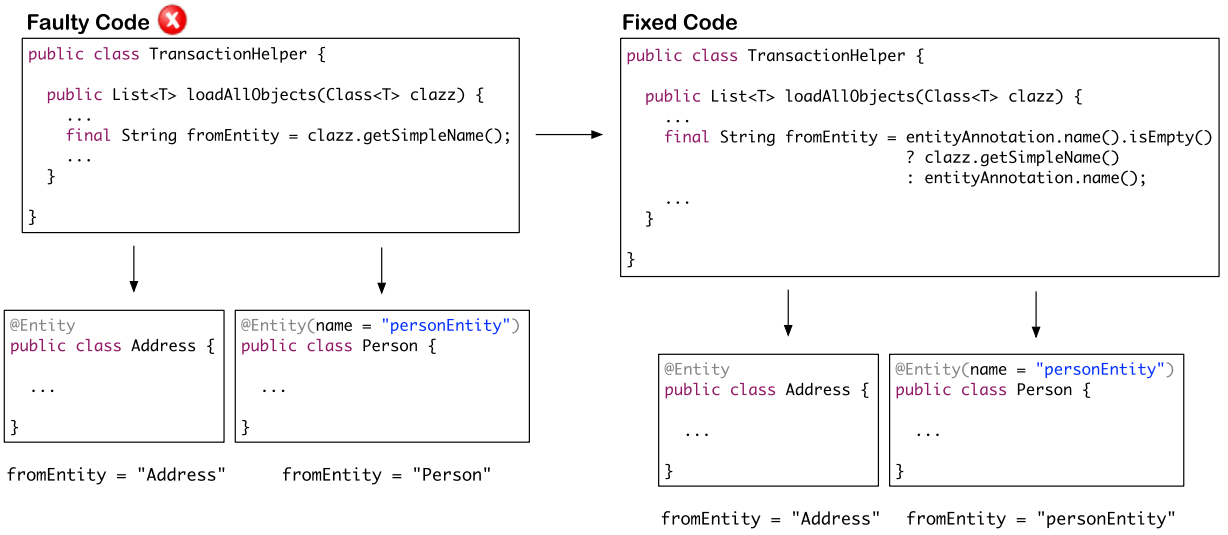


**Fig. 8.** *Wrong annotation parsing* example from the *needle* project (Java). Our operator Add Attribute (ADAT) is able to mimic this fault.

**Answer to RQ2:** Yes. Three operators are able to mimic the majority of the faults (*i.e.*, 83%). The Add Annotation (ADA) mimics 97 faults, Remove Annotation (RMA) mimics 39 faults, and Replace Annotation Value (RPAV) mimics 30 faults.

Regarding **RQ3**, only one operator could not mimic any fault: the Replace Attribute (RPAT) operator. The operators Add Attribute (ADAT), Change Order (CHODR), Remove Attribute (RMAT), Replace Annotation (RPA), and Switch Target (SWTG) mimicked altogether a very low number of faults: only 24 (12%). Fig. 7 illustrates a *wrong annotation parsing* fault example from the *actframework* project. Because the developer declared the annotations in a specific order, the code responsible for parsing the annotated code did not behave as expected and raised an error.[13] Fig. 7 also shows the developer's comment with respect to the fault. Our Change Order (CHODR) operator could mimic this situation: after changing the order, the parsing of the annotated code would not work as expected.

Fig. 8 illustrates an example in which the Add Attribute (ADAT) operator can mimic the fault. This example fits into the *wrong annotation parsing* category. The faulty code (left-hand side) parses the annotated code of the *needle* project[14] and assigns the class name to the `fromEntity` variable (*e.g.*, `Address` and `Person`) regardless of the `name` attribute defined at the `@Entity` annotation. The fixed code (right-hand side), on the other hand, takes the `name` attribute into account. In this way, `fromEntity` receives the class name in case the annotation attribute is not defined (*e.g.*, `Address`). Otherwise, `fromEntity` receives the attribute value (*e.g.*, `personEntity`).

**Answer to RQ3:** Only one operator could not mimic any fault: the Replace Attribute (RPAT) operator. Five operators, *i.e.*, Add Attribute (ADAT), Change Order (CHODR), Remove Attribute (RMAT), Replace Annotation (RPA), and Switch Target (SWTG), mimicked altogether a very low number of faults: 24 (12%).

[13] https://github.com/actframework/actframework/issues/260 – accessed in February/2020.
[14] https://github.com/akquinet/needle/commit/bcd709b4c4f9a74d3d5af31d024695721a2ab033 – accessed in February/2020.

**Faulty Code**

```
@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)
public class SimpleGrantedAuthorityMixin {

  public SimpleGrantedAuthorityMixin(
        @JsonProperty("role") String role) {
    ...
  }

}
```

**Fixed Code**

```
@JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.NONE)
public class SimpleGrantedAuthorityMixin {

  public SimpleGrantedAuthorityMixin(
        @JsonProperty("authority") String role) {
    ...
  }

}
```

**Fig. 9.** None of our operators could mimic this example (Java). In this scenario, we would need higher-order mutation operators to change two different code locations (*i.e.*, `ANY` to `NONE` at the `JsonAutoDetect` annotation; and `role` to `authority` at the `JsonProperty` annotation). In this way, the mutant would have two different syntactical mutations at the code annotations.

**Faulty Code**

```
[ProvideLanguageService(…
,EnableCommenting = true)]
```

**Fixed Code**

```
[ProvideLanguageService(…
    ,EnableCommenting = true
+ ,RequestStockColors = true)]
```

**Fig. 10.** *Misuse* example from the *Visual Studio 2017 SDK* project (C#). Our operator Remove Annotation Attribute (RMAT) is able to mimic this fault.

**Faulty Code**

```
[AuthorizeAttribute(Roles="user, admin")]
public class MyController : Controller {
 ...
}
```

**Users with the admin role will never be authorised because the role name will be evaluated to " admin" instead of "admin".**

**Fig. 11.** *Wrong annotation parsing* example from the *ASP.NET MVC* project (C#). Notice that there is one space character before the word in the first string and that there are no space characters in the second string. Our operator Replace Attribute Value (RPAV) is able to mimic this fault.

Regarding **RQ4**, our operators could not mimic 10 faults (five in Java and five in C#). To mimic them, we need higher-order mutation operators [29,36], *i.e.*, operators that change two or more code locations, such as removing two annotations or removing an annotation and changing an attribute to another. Fig. 9 illustrates an example from the *spring-security* project.[15] Notice that the developer changed annotation attributes of two different annotations (`@JsonAutoDetect`, at the class declaration; and `@JsonProperty`, at the constructor parameter).

**Answer to RQ4:** Ten faults need higher-order mutation operators. Thus, we could not mimic them by using our operators.

Fig. 10 illustrates a fault from the *Visual Studio 2017 SDK* project.[16] The annotation `[ProvideLanguageService]` informs *Visual Studio* that a *VSPackage* provides a language service, such as brace matching. In this example, the developer forgot to set `RequestStockColors` to `true`. As a result, `RequestStockColors` is set to `false` by default. This indicates that the language service will supply custom color-able items, which the developer did not have implemented nor intended to. We are able to mimic this fault by applying the Remove Annotation Attribute (RMAT) operator.

Fig. 11 illustrates a fault that our operator Replace Annotation Value (RPAV) is able to mimic. `[AuthorizeAttribute]` is a code annotation from the *ASP.NET MVC 5.2*.[17] It specifies that accesses to a controller or action method is restricted to users who meet the authorization requirement. In this example, the developer sets `Role` to the value `"user, admin"`. Notice that there is a space after the comma. The code responsible for parsing the `[AuthorizeAttribute]` annotation did not trim spaces originally, which in turn led to an error when authorizing users with the `admin` role.

[15] https://github.com/spring-projects/spring-security/commit/c2d8ea92d0cd1f3a5b78c30d76dd6bc820b27e93 – accessed in February/2020.

[16] https://docs.microsoft.com/en-us/dotnet/api/microsoft.visualstudio.shell.providelanguageserviceattribute?view=visualstudiosdk-2017 – accessed in February/2020.

[17] https://docs.microsoft.com/en-us/dotnet/api/system.web.mvc.authorizeattribute?view=aspnet-mvc-5.2 – accessed in February/2020.

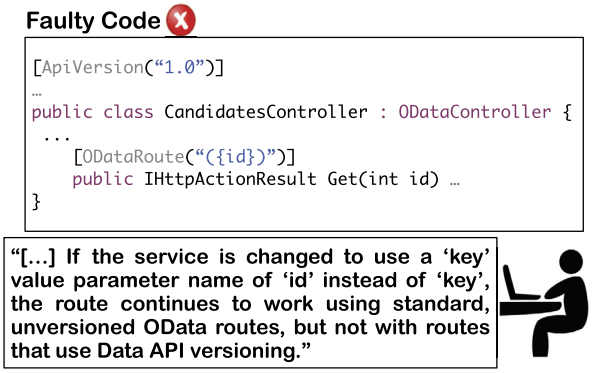


**Fig. 12.** *Wrong annotation parsing* example from the *ASP.NET Versioning* project (C#). Our operator Add Annotation (ADA) is able to mimic this fault.

Fig. 12 illustrates a fault that our operator Add Annotation (ADA) operator is able to mimic. `[ODataRoute]` is a code annotation from the *OData v4 Web API*.[18] We can use it when implementing *routing*, *i.e.,* defining a route that matches a URI to an action. In this example, the developer sets the key value parameter as `id`. If the service is changed to use the key value parameter name of `id` instead of `key`, the route continues to work using standard, unversioned `OData` routes, but not with routes that use OData API versioning. Our operator ADA is able to mimic this fault by applying `[ApiVersion]` on such a context.

After presenting examples of faults in Java and C# that our operators are able to mimic, we now answer **RQ5**.

> **Answer to RQ5:** The set of mutation operators we propose are representative in the sense we are able to mimic the majority of the faults in projects written in two languages, *i.e.*, Java and C#. By mapping our operators to a new language (C#), we improved their generality.

### 4.3. Threats to validity

We now present the threats to the validity of our evaluation. We follow the convention presented by Wohlin et al. [51].

The set of projects we used represents a threat to external validity. To alleviate this threat, we selected projects with heavy use of Java and C# annotations. Like the projects, the faults we analyzed may not be representative. In this context, the projects are from different domains, sizes, and development periods, which minimizes this threat. Nevertheless, we acknowledge that a larger set of faults would better evaluate our operators and even lead to new operators since we were not exhaustive on this front.

The manual task to filter out the faults related to annotations represents a threat to conclusion validity. This manual analysis may lead to false positives, which may change our results and conclusions. To minimize this threat, two researchers analyzed all faults individually. In case they disagreed when analyzing one specific fault, we disregarded it from our dataset. Also, the manual analysis poses a threat to internal validity, *i.e.*, when mapping the faults into each operator. Again, we minimize this threat because of the detailed analyses of the two researchers.

### 4.4. Implications for the practical application of our operators

- **Improving test suites:** previous research has reported that developers are overusing annotations [47], which might lead to problems of code understanding and maintainability. In such a case, the number of faults related to code annotations is likely to increase and more tests may have to be implemented to protect the software systems from these faults. In this scenario, our mutation operators might be important to induce developers to improve their test suites and, consequently, avoid annotation-related faults.
- **Improving parsers of annotated code:** our mutation operators are capable of simulating faults in code responsible for parsing annotated code. This way, when using our operators, developers might implement better parsers and improve the quality of their projects.

[18] http://odata.github.io/WebApi/03-03-attrribute-routing/ – accessed in February/2020.

```
{
  "annotation": "@Entity",
  "targets": ["type"],
  "attributes": [{
    "name": "name",
    "type": "string",
    "validValues": ["person"]
  }]
}
```

**Listing 1.** Example of configuration file loaded in our engine. This configuration file describes the @Entity target as `"type"`, which includes class, interface, or enum declaration.

## 5. Mutation engine

Based on the proposed nine mutation operators, we designed a mutation engine for Java and C#. The engine implements all nine mutation operators we present and provides means (*i.e.,* configuration files) to apply strategies to avoid the generation of useless mutants, as discussed in Section 3.

Six of our operators require additional context information to be applied, namely: ADA, ADAT, RPA, RPAT, RPAV, SWTG. For example, we can place certain annotations (*e.g.*, `@Entity`) only at classes, interfaces, and enum declarations. Notice that this context information is important to avoid useless mutants. In other words, placing annotations like `@Entity` at field and method declarations does not make sense and can even cause the program to not compile.

To avoid these scenarios, our engine relies on JSON configuration files to guide the generation of the mutants. Listing 1 illustrates an example of such a file considering the ADA operator and the `@Entity` annotation. To define the places to which it is possible to add the `@Entity` annotation, developers should use the `targets` list. Notice that the list contains only one element, *i.e.*, `type`. This guides our ADA operator to only place the `@Entity` annotation into classes, interfaces, and enum declarations. In addition, notice that we can define information regarding annotation attributes. For example, the `@Entity` annotation may have a `name` attribute. The `name` type is `string` and the `validValues` represents potential valid values for the `name` attribute. This way, given the JSON configuration file presented in Listing 1 and the `@Entity` annotation in the original program, the ADAT operator is able to create the following mutant: `@Entity(name = "person")`.

All operators that perform replacement tasks (*i.e.*, RPA, RPAT, and RPAV) need additional context information. For example, as mentioned in Section 3, replacing `@NotNull` by `@Entity` is meaningless and yields a useless mutant. The engine could replace `@NotNull` by `@Nullable` instead. To guide the engine to avoid meaningless replacements, we can use the `replaceableBy` key of our JSON configuration file. For instance, Listing 2 shows that the RPA operator can replace `@Consumes` by `@Produces`, and vice-versa. We can also configure the RPAT and RPAV operators. For example, `@Consumes` and `@Produces` annotations have the `value` attribute. We can assign a list of strings to it. Listing 2 also shows examples of valid values for the `value` attribute. This way, given the annotation `@Consumes(value = "image/jpeg")` and using the RPAV operator, the engine is able to create the following mutant `@Consumes(value = "image/png")`, for instance. In this way, notice that depending on how the developer configures the JSON file, the engine is able to generate application-specific mutants.

After presenting two configuration files targeting the Java language, we now illustrate an example for C# (Listing 3). Notice that our JSON configuration file has the same structure regardless of the language (*i.e.*, Java or C#). In this example, we are guiding the tool to add the `[Required]` annotation to C# fields. According to the official language documentation,[19] this annotation "*specifies that when a field on a form is validated, the field must contain a value.*" In case the field is null or empty, an exception is raised and the message defined in the annotation attribute, *i.e.*, `ErrorMessage`, is shown. To avoid meaningless mutants, developers should not declare (in the configuration file) "method" as a `type`, for example, since the tool would place the `[Required]` annotation at methods, which does not make sense.

To implement the SWTG operator, we need to know in advance to which targets we can move a certain annotation. We can define this in our JSON configuration file. For example, Listing 2 describes that the `@Consumes` valid targets are `type` and `method`. This way, if the `@Consumes` annotation is placed at a method in the original program, the engine can move it to a class declaration and vice-versa.

Three operators do not need context information to be applied: RMA, RMAT, and CHODR. We implemented RMA and RMAT by simply removing the code annotation itself or part of it. When considering the CHODR operator, we only change the order by using permutations (*i.e.*, $n!, n =$ *number of annotations*). In summary, these three operators do not need any configuration to work properly.

[19] https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations.requiredattribute?view=netframework-4.8 – accessed in February/2020.

```
[
  {
    "annotation": "@Consumes",
    "replaceableBy": ["@Produces"],
    "targets": ["type", "method"],
    "attributes": [{
      "name": "value",
      "type": "string[]",
      "validValues": ["image/jpeg", "image/png"]
    }]
  },
  {
    "annotation": "@Produces",
    "replaceableBy": ["@Consumes"],
    "targets": ["type", "method"],
    "attributes": [{
      "name": "value",
      "type": "string[]",
      "validValues": ["image/jpeg", "image/png"]
    }]
  }
]
```

**Listing 2.** Example of configuration file loaded in our engine. This configuration file describes context information about the @Consumes and @Produces annotations.

```
{
  "annotation": "System.ComponentModel.DataAnnotations.Required",
  "replaceableBy": [],
  "targets": ["field"],
  "attributes": [
    {
      "name": "ErrorMessage",
      "type": "String",
      "validValues": ["\"Error\""]
    }
  ]
}
```

**Listing 3.** Example of configuration file loaded in our engine. This configuration file describes the [Required] target as `"field"`.

The engine is written in Kotlin.[20] Kotlin is a general-purpose programming language, developed by JetBrains. Kotlin is statically typed, with type inference, and designed to interoperate fully with Java. Our engine is available online.[21]

As a last note regarding our mutation engine, we reinforce the fact that mutation of code annotations is a very recent topic; to date, we are aware of only one study that addressed it, performed by ourselves [44]. Consequently, little is known about representative types of mutants regarding code annotations, and only empirical studies will reveal the representativeness of the rules embedded in the configuration files. Our initial set of configuration files (some of them described in this section) is not intended to be comprehensive and definite, since annotations may come from standard APIs or be user defined. Due to this characteristic, our strategy of parameterizing (through external configuration files) the design of mutation operators gives the tool engineer (or even the tool user) the freedom to customize the definition of mutations for programs that employ specific types of annotations. In other words, the six operators (ADA, ADAT, RPA, RPAT, RPAV, SWTG) that require contextual information may be implemented with fine-tuned configuration files based on contextual information.

[20] http://kotlinlang.org/ – accessed in February/2020.
[21] https://github.com/easy-software-ufal/mutation-tool-for-annotations – accessed in February/2020.

## 6. Related work

While mutation testing was heavily studied in the context of imperative languages [49], recent research also followed the trend of dealing with different paradigms, frameworks, and other language constructs. For example, Hajjaji et al. [3] proposed a set of mutation operators for configurable systems in C programs. The operators can add, remove, and change `#ifdefs` and even elements related to the variability model. In this context, they identified that traditional C mutation operators were not able to mimic variability faults. Estero-Botaro et al. [24] designed a set of mutation operators for the WS-BPEL 2.0 (Web Services Business Process Execution Language). Ferrari et al. [26] proposed a suitable set of mutation operators for AspectJ-based programs. Deng et al. [20] defined mutation operators specific to the characteristics of Android apps, such as the event handler and the activity lifecycle. Previous work also proposed other Java-specific mutation operation sets related to object-oriented programming [34] and concurrency mechanisms [9,17].

Regarding C#, Derezińska [21] proposed a set of mutation operators that targets specific features of the language, extending a previous work by Baudry et al. [7]. These operators can, for example, change a delegated method, delete member variable initialization, and even remove exception handling. In this context, Derezińska identified that some mutation operators are not appropriate for C# programs and that some operators require specific pre- and post-conditions to be applied effectively, taking into account not only local but also structural information of the program [22]. With that in mind, our mutation operators for code annotations also require specific pre- and post-conditions to be applied effectively.

Addressing the faults raised by code annotations has been tackled before. Sanchez et al. [14] observed that the Java native support to construct annotations is very poor, thus limiting the ability to specify the elements where an annotation can be placed and further correctness conditions. This way, they describe *Ann*, a modeling language for the design and validation of Java annotations. *Ann* makes use of a constraint solver over models to detect whether the constraints posed by a set of annotations are unsatisfiable. Another piece of work describes a new tool (named *AnnaBot*) to make assertions about how annotations are used [16]. *AnnaBot* introduces a Domain-Specific Language for declarative specification of additional checking in Java annotations. It works by reading this specification and verifying the compliance of the annotated file with respect to the specification. These tools extend the static checking for code annotations, but cannot prevent failures caused by dynamic behavior. For example, they do not deal with issues caused by wrong annotation parsing (such as the fault presented in Fig. 8).

Another initiative to statically check code annotations follows the idea that annotations should describe the way in which they should be validated [38]. The approach is based on meta-annotations (`@Validators`) for the validation of annotations in Java applications. These meta-annotations describe the rules of use of domain level annotations. Differently, from our mutation testing operators, these meta-annotations cannot support developers in testing the dynamic behavior of parsing annotated code.

Araújo et al. [6] proposed an approach to identify mutation operators that mimic faults that an automatic static analyzer is not capable of detecting. In this way, to better evaluate our operators, we can perform a study also based on static analyzers to better understand and prioritize our operators.

## 7. Concluding remarks and future work

Many projects nowadays make use of meta-programming through code annotations. In a previous work, we presented the first set of mutation operators for code annotations [44]. We proposed nine mutation operators and evaluated whether they can mimic realistic faults. We extended this prior work in several ways. First, we mapped our mutation operators to a different programming language (*i.e.*, C#), improving the generality of our operators. Second, our enlarged evaluation using 200 issues of open-source projects (100 in Java and 100 in C#) improves confidence on the ability of our operators to mimic code annotation-related faults. To better structure our evaluation, we extended and improved our code-annotation issue repository. Each issue of the repository is labeled with information like the type of the error that the developer committed and the mutation operator able to mimic the fault. The repository now contains 200 code annotation faults and may support not only further mutation testing research, but also general software engineering studies on code annotations.

The results of our evaluation showed that our operators are able to mimic 190 out of 200 issues (*i.e.*, 95%). Curiously, we achieved the same ratio for both the Java and C# languages. We also found that three (out of nine) operators were responsible for simulating 83% of all faults: Add Annotation (ADA), Remove Annotation (RMA), and Replace Annotation Value (RPAV). In addition, one operator was not able to mimic any fault, *i.e.*, Replace Attribute (RPAT). We also identified faults that need more complex code transformations to be mimicked. In this case, we could use the idea of high-order mutation [29,36]. We also discuss scenarios to avoid meaningless mutants. For example, the Replace Annotation (RPA) operator should select a meaningful annotation to replace the actual one (*e.g.*, `@NotNull` by `Nullable`).

Moreover, this article introduced a mutation engine for code annotations. The engine is available for download and automates the application of the nine operators to Java and C# programs. Developers can configure the engine to avoid meaningless mutants (such as replacing `@NotNull` by `Entity`) by using the engine configuration files.

As future work, we intend to further study whether specific frameworks (*e.g.*, *Hibernate*) need specific operators. Additionally, we intend to provide a broader analysis regarding the applicability of our operators to other languages that also support code annotations such as Python, C++, Ruby, and Haskell. In addition, we intend to evaluate the quality of the mutants generated by our operators. In particular, after applying our mutation engine, we will check the number of equivalent

and duplicated mutants [19,41] and even assess the number of redundant mutants [4]. Moreover, we intend to evaluate the correlation of our mutants with real faults related to code annotations, similarly to what has been done in previous studies with similar intent [5,15,33]. We also intend to provide rules in terms of a JSON-like scheme to validate the configuration files and warn developers with respect to potential wrong configurations, which would lead to meaningless mutants. Last but not least, we intend to perform an extensive empirical study focusing on scenarios such as the ones we presented in Section 3.2, *i.e.*, we will create code annotation mutants and execute the tests to check to what extent these mutants help with improving the test suites.

## References


[1] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, F.G. Sayward, Mutation analysis, Technical Report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, USA, 1979.
[2] H. Agrawal, R.A. DeMillo, R. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, E.H. Spafford, Design of mutant operators for the C programming language, Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, USA, 1989.
[3] M. Al-Hajjaji, F. Benduhn, T. Thüm, T. Leich, G. Saake, Mutation operators for preprocessor-based variability, in: Proceedings of the 10th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), Salvador, BA, Brazil, ACM Press, New York, 2016, pp. 81–88.
[4] P. Ammann, M.E. Delamaro, J. Offutt, Establishing theoretical minimal sets of mutants, in: Proceedings of the International Conference on Software Testing, Verification, and Validation (ICST), Cleveland, OH, USA, IEEE, 2014, pp. 21–30.
[5] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? in: Proceedings of the 27th International Conference on Software Engineering (ICSE), St. Louis, MO, USA, ACM Press, 2005, pp. 402–411.
[6] C.A. Araújo, M.E. Delamaro, J.C. Maldonado, A.M.R. Vincenzi, Correlating automatic static analysis and mutation testing: towards incremental strategies, J. Softw. Eng. Res. Dev. 4 (2016).
[7] B. Baudry, F. Fleurey, J.-M. Jézéquel, Y. Le Traon, From genetic to bacteriological algorithms for mutation-based testing, Softw. Test. Verif. Reliab. 15 (2) (2005) 73–96.
[8] C. Bauer, G. King, Java Persistence with Hibernate, Manning Publications Co., Shelter Island, NY, USA, 2006.
[9] J. Bradbury, J. Cordy, J. Dingel, Mutation operators for concurrent Java (J2SE 5.0), in: Proceedings of the 2nd Workshop on Mutation Analysis (Mutation), Raleigh, NC, USA, Kluwer Academic Publishers, 2006.
[10] T.T. Chekam, M. Papadakis, Y. Le Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, IEEE, 2017, pp. 597–608.
[11] P. Chevalley, Applying mutation analysis for object-oriented programs using a reflective approach, in: Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC), Macao, China, IEEE, 2001, pp. 267–270.
[12] H. Coles, PITest mutation testing tool for Java, http://pitest.org/, 2019. (Accessed November 2019).
[13] H. Coles, T. Laurent, C. Henard, M. Papadakis, A. Ventresque, PIT: a practical mutation testing tool for Java (demo), in: Proceedings of the 25th International Symposium on Software Testing and Analysis. ISSTA 2016, Saarbrücken, Germany, ACM Press, 2016, pp. 449–452.
[14] I. Córdoba-Sánchez, J. de Lara, Ann: a domain-specific language for the effective design and validation of Java annotations, Comput. Lang. Syst. Struct. 45 (2016) 164–190.
[15] M. Daran, P. Thévenod-Fosse, Software error analysis: a real case study involving real faults and mutations, in: Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), San Diego, CA, USA, ACM Press, 1996, pp. 158–171.
[16] I. Darwin, AnnaBot: a static verifier for Java annotation usage, Adv. Softw. Eng. (2010) 1–7.
[17] M.E. Delamaro, M. Pezzè, A.M.R. Vincenzi, J.C. Maldonado, Mutant operator for testing concurrent Java programs, in: Proceedings of the 15th Brazilian Symposium on Software Engineering (SBES), Brazilian Computer Society, Rio de Janeiro, RJ, Brasil, 2001, pp. 386–391.
[18] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, J. Domínguez-Jiménez, Assessment of class mutation operators for C++ with the MuCPP mutation system, Inf. Softw. Technol. 81 (2017) 169–184.
[19] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, IEEE Comput. 11 (4) (1978) 34–43.
[20] L. Deng, N. Mirzaei, P. Ammann, J. Offutt, Towards mutation analysis of Android apps, in: Proceedings of the 10th International Workshop on Mutation Analysis (Mutation), Graz, Austria, IEEE, 2015, pp. 1–10.
[21] A. Derezińska, Advanced mutation operators applicable in C# programs, in: K. Sacha (Ed.), Software Engineering Techniques: Design for Quality, Springer-Verlag, Boston, MA, 2006, pp. 283–288.
[22] A. Derezińska, Quality assessment of mutation operators dedicated for C# programs, in: Proceedings of the International Conference on Quality Software (QSIC), Beijing, China, IEEE, 2006, pp. 227–234.
[23] A. Derezinska, K. Kowalski, Object-oriented mutation applied in common intermediate language programs originated from C#, in: Proceedings of the 6th International Workshop on Mutation Analysis (Mutation), Berlin, Germany, IEEE, 2011, pp. 342–350.
[24] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Quantitative evaluation of mutation operators for WS-BPEL compositions, in: Proceedings of the 5th International Workshop on Mutation Analysis (Mutation), Paris, France, IEEE, 2010, pp. 142–150.
[25] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, J. Maldonado, Avoiding useless mutants, in: Proceedings of the 16th International Conference on Generative Programming: Concepts & Experience (GPCE), Vancouver, BC, Canada, ACM Press, 2017, pp. 187–198.
[26] F.C. Ferrari, J.C. Maldonado, A. Rashid, Mutation testing for aspect-oriented programs, in: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST), Lillehammer, Norway, IEEE, April 2008, pp. 52–61.
[27] GitHub Inc., Github search REST API v3, https://developer.github.com/v3/search/, 2019. (Accessed November 2019).
[28] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Trans. Softw. Eng. 3 (4) (Jul. 1977) 279–290.
[29] Y. Jia, M. Harman, Higher order mutation testing, Inf. Softw. Technol. J. 51 (10) (2009) 1379–1393.
[30] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.
[31] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, R. Evans, Spring - Java/J2EE application framework, Reference Manual Version 2.0.2, Interface21 Ltd., 2006.
[32] R. Just, The MAJOR mutation framework: efficient and scalable mutation analysis for Java, in: Proceedings of the 23rd International Symposium on Software Testing and Analysis (ISSTA), San Jose, CA, USA, ACM Press, 2014, pp. 433–436.
[33] R. Just, D. Jalali, L. Inozemtseva, M. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing? in: Proceedings of the 22nd SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Hong Kong, China, ACM Press, 2014, pp. 654–665.
[34] Y.-S. Ma, Y.-R. Kwon, A.J. Offutt, Inter-class mutation operators for Java, in: Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE), Annapolis, MD, USA, IEEE, 2002, pp. 352–366.
[35] Y.-S. Ma, A.J. Offutt, Y.-R. Kwon, MuJava: an automated class mutation system, Softw. Test. Verif. Reliab. 15 (2) (2005) 97–133.

[36] L. Madeyski, W. Orzeszyna, R. Torkar, M. Jozala, Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation, IEEE Trans. Softw. Eng. 40 (1) (2014) 23–42.
[37] A.P. Mathur, Foundations of Software Testing, 1st edition, Addison-Wesley, Toronto, ON, Canada, 2007.
[38] C. Noguera, R. Pawlak, Aval: an extensible attribute-oriented programming validator for Java, J. Softw. Maint. Evol. 19 (4) (2007) 253–275.
[39] NUnit Team, http://nunit.org/, 2018. (Accessed November 2019).
[40] A.J. Offutt, R.H. Untch, Mutation 2000: uniting the orthogonal, in: Proceedings of the Mutation 2000 Symposium, San Jose, CA, USA, Kluwer Academic Publishers, 2000, pp. 34–44.
[41] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon, Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in: Proceedings of the 37th International Conference on Software Engineering (ICSE), Florence, Italy, ACM Press, 2015, pp. 936–946.
[42] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: A.M. Memon (Ed.), Advances in Computers, vol.112, Elsevier, Amsterdam, the Netherlands, 2019, pp. 275–378.
[43] G. Petrović, M. Ivanković, B. Kurtz, P. Ammann, R. Just, An industrial application of mutation testing: lessons, challenges, and research directions, in: Proceedings of the 13th International Workshop on Mutation Analysis (Mutation), Västerås, Sweden, IEEE, 2018, pp. 47–53.
[44] P. Pinheiro, J.C. Viana, L. Fernandes, M. Ribeiro, F.C. Ferrari, B. Fonseca, R. Gheyi, Mutation operators for code annotations, in: Proceedings of the 3rd Brazilian Symposium on Systematic and Automated Software Testing (SAST), São Carlos, SP Brazil, ACM Press, 2018, pp. 77–86.
[45] Piotr Trzpil, VisualMutator - NET mutation testing, https://visualmutator.github.io/web/, 2019. (Accessed November 2019).
[46] A.V. Pizzoleto, F.C. Ferrari, A.J. Offutt, L. Fernandes, M. Ribeiro, A systematic literature review of techniques and metrics to reduce the cost of mutation testing, J. Syst. Softw. 157 (2019).
[47] H. Rocha, M.T. Valente, How annotations are used in Java: an empirical study, in: Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE), Knowledge Systems Institute Graduate School, Miami Beach, FL, USA, 2011, pp. 426–431.
[48] D. Schuler, A. Zeller, Covering and uncovering equivalent mutants, Softw. Test. Verif. Reliab. 23 (5) (2013) 353–374.
[49] A. Sullivan, K. Wang, R.N. Zaeem, S. Khurshid, Automated test generation and mutation testing for Alloy, in: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, IEEE, 2017, pp. 264–275.
[50] P. Tahchiev, F. Leme, V. Massol, G. Gregory, JUnit in Action, 2nd edition, Manning Publications Co., Shelter Island, NY, USA, 2010.
[51] C. Wohlin, P. Runeson, M. Host, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering: An Introduction, Kluwer Academic Publishers, 2000.